Redbooks
ibm.com/redbooks

Modernizing IBM i Applications

Tim Rowe
Brad Bentley
Steve Bradshaw
Jim Buck
Andrew Clark
Andrew Youens
Scott Forstie
Ludovic Gasc
Shankar Giduturi
Wim Jongman
Sébastien Julliand
Gaurav Kumar
Marius le Roux
Benoit Marolleau
Kent Milligan
Ryan Moeller
Rahul Pareek
Het Patel
Michael Pavlak
Pascal Polverini
Marina Schwenk
Vikas Sonwal
Todd Stewart
Jeff Ticker
Khushi Wadhwa
Carol Woodbury
Jia Tian Zhong
George Van Eaton

IBM

An Introduction to Modernizing IBM i Applications

What This Covers

This IBM Redbooks® publication addresses modernization at all levels of the operating system. It covers development concepts that are in themselves modern and, when used in aggregate, will allow you to build applications that are functional, intuitive, performant and secure.

Whether you are evolving existing workloads or building new solutions, this book provides the foundation and tools you need to modernize with confidence and deliver better software, faster. You will see practical techniques to accelerate development, strengthen testing practices, and simplify integration with other applications, regardless of whether they run on IBM i.

Authors and Contributors

Brad Bentley is a Principal in Technology Expert Labs at IBM, based in Rochester, MN, USA. He has 27 years of experience in application modernization and transformation on IBM Power systems and focuses on optimizing existing workloads, leveraging cloud-native technologies, DevOps practices, and AI-driven monitoring solutions. Brad holds a Bachelor of Science in Computer Science and has worked with IBM i's largest customers across finance, manufacturing, and healthcare industries, helping them navigate their modernization journeys. He actively shapes industry best practices through customer engagements, conference presentations, and technical publications. Brad contributed to this IBM Redbooks publication during a residency on IBM i Interfacing and RPG best practices.

Steve Bradshaw is the CEO & Friendly Techie at RowtonIT.com, a UK IBM Business Partner founded in 2008 specializing in supporting Small to Medium-sized Enterprises. He has four decades of experience working on IBM i Systems since 1988. For over 15 years, Steve has been the Technical Director of the i-UG IBM i user group i and a member of the Common Europe Advisory Council, where he works closely with IBM to help shape the future of IBM i. He was made an IBM Champion in 2018 and spends much of his time teaching modern IBM i Best Practices at Universities in the UK, IBM i conferences around the world, as well as posting much of his material online.

Jim Buck is president and founder of imPower Technologies, based in Kenosha, Wisconsin. He provides professional IBM i training focused on modern ILE RPG and CLLE. With 38 years of experience in education, manufacturing, and healthcare, Jim co-authored several widely used IBM i textbooks with Bryan Meyers, including Programming in ILE RPG, 5th Edition, and the CL book. These titles are used by companies and colleges worldwide. He is a long-time IBM Champion for Power Systems and has served on multiple teams developing IBM and COMMON certification tests.

Andrew Clark is the Senior Principal Software Engineer for IBM i at Rocket Software. Andrew has over 35 years of experience on IBM i beginning with the "secret Mankato Project" on a pre-release version of OS/400, and an internship at IBM Rochester working on the Query team. He has development expertise in multiple programming languages including RPG, Java, and C++; a background in green-screen, Windows, mobile and web development; proficient at everything DB2/SQL on IBM i, has extensive experience in the entire DevSecOps lifecycle, a frequent speaker at Common and other events, is an AWS certified Solutions Architect and IBM Champion.

Scott Forstie is a Senior Technical Staff Member and Db2 for i Architect at IBM, where he has contributed to operating system development since 1989. He leads the Db2 for i development team and is a frequently published author and speaker at industry events worldwide. Scott actively shares SQL expertise through multiple channels including the Db2 for i SQL Tutor, the iSee Video blog, and GitHub, making advanced database concepts accessible to the IBM i community.

Ludovic Gasc has been passionate about Open Source, bioinformatics, and hardware for over 20 years. He has worked with various start-ups and scale-ups in Telecom and IoT, and has contributed to several Open Source software projects within the Python, Java, and JavaScript ecosystem. Moreover, he has worked as a DevOps/SRE

specialist to deploy and maintain a real-time communication solution in a multi-cloud environment (4 clouds) with the help of Ansible. Currently, he specializes in modernizing the IBM Power platform using the AI and Open Source/ RedHat stack.

Shankar Giduturi is a Solution Architect and Enterprise Architect at Republic Services, based in Phoenix, Arizona, United States. He has 20+ years of experience in enterprise technology, with deep expertise in Order to Cash platforms and IBM i modernization. His focus includes practical architecture, DevOps adoption, integration design, and delivery reliability for large-scale business systems. Shankar contributed to IBM Redbooks content across stored procedures, Jenkins CI/CD, data queues, Integrated Web Services, SQL-based web service consumption, fully free-format RPG, SQL procedure testing, embedded SQL, and JSON processing in RPGLE, emphasizing resilient modernization and maintainable solutions.

Wim Jongman is CTO and co-owner of Remain Software, founded in 1992 to provide DevOps solutions for IBM i. Wim has worked with IBM i and its predecessors since the platform's introduction in 1988 and has remained a strong advocate for its continued evolution. He focuses on modernizing IBM i development through DevOps, automation, integration, and continuous delivery. With decades of hands-on experience, Wim combines deep platform knowledge with a practical view on how enterprise software teams can move faster without losing reliability. His credo says it all: "To the Batmobile. Let's go!"

Sébastien "Seb" Julliand is the R&D Technical Lead at i and me - Training and Consulting GmbH, based in Annecy, France. He has been working on IBM i for 20 years, and his field of expertise ranges from application development and modernization to DevOps. Seb has been an IBM Champion since 2024 and an active member of the IBM i community, both as a speaker and as one of the main and most active contributors to the Code for IBM i VS Code extension and other open-source projects related to IBM i. He contributed to this IBM Redbook focused on application modernization by sharing his expertise on VS Code, Code for IBM i, Git and Jenkins.

Gaurav Kumar is a Senior Systems Administrator at HSBC, specializing in IBM i modernization, code development and DevOps practices. He has 6 years of experience working on infrastructure automation using Ansible, implementing CI/CD pipelines, enhancing system observability (using OTEL, Prometheus, Manzan) as well as IBM i Integrated Web Services. He contributed to this IBM Redbooks publication covering modern IDE tools and productivity features, benefits of source control and free-form RPG, and Ansible-based automation for IBM i.

Marius le Roux is an IBM i Consultant and owner of MLR Consulting, specializing in modernization, integration, and enterprise architecture on IBM i. He has over 20 years of experience across operations, development, and system administration. He is a co-author of multiple IBM Redbooks publications, including IBM i 7.6 Features and Functions, and contributes regularly through technical articles and community forums such as IBM i groups on TechXchange.

Benoit Marolleau is a Solutions Architect and AI Engineer in IBM Client Engineering EMEA at IBM, based in Montpellier, France. He has 20 years of experience in application and infrastructure modernization, with specialized expertise in IBM i platform integration and AI solutions. Benoit holds engineering degrees in data, computer science, and management. He assists enterprise clients with digital transformation initiatives, focusing on modernizing legacy systems and implementing Cloud, Open source and AI-driven architectures. A regular speaker and part-time educator, he is passionate about knowledge sharing and diversity in technology. He contributed to this IBM Redbooks publication on application modernization concepts, AI and Web technologies.

Kent Milligan is a Senior Db2 for i Consultant in IBM Technology Expert Labs based in Rochester, Minnesota. He has over 30 years of experience as a Db2 for IBM i consultant and developer in the IBM Rochester Lab. Kent is a frequently published author and speaker on Db2 topics at IBM i industry events. He joined the IBM Redbooks residency to contribute expertise in the modernization of Db2 for i databases.

Ryan Moeller is a Staff Software Developer at IBM Rochester where he has contributed to the SQL Query Engine team since 2019, working on SQL optimization and database tooling. Ryan is an experienced speaker and has presented on various Db2 for i topics at events around the globe.

Rahul Pareek is a Technical Architect – IBM i at Programmers.io, based in Pune, India. He has 15 years of experience in IBM i application development and specializes in establishing modern source control workflows, enabling DevOps adoption, and transforming traditional IBM i development into agile, CI/CD-driven delivery models.

He contributed to this IBM Redbooks publication during a residency on Git adoption on IBM i and addressing common implementation challenges.

Het Patel is an AI Software Development Engineer at IBM with approximately 8 years of experience in software development and artificial intelligence. He specializes in IBM Bob™, an AI-powered assistant, and development tools for IBM i and database engineering by contributing to Db2 for i, where he led the FVT team for several features. Het contributed to this IBM Redbooks publication during a residency on AI, Database Engineering, Java Modernization, and DevOps Practices for IBM i Modernization. He recently started a speaker journey for the IBM i community in areas of AI and Database by COMMON.

Pascal Polverini is a digital transformation expert specializing in IBM i modernization, test automation, and AI governance. As CTO of Polverini & Partners, a tooling and automation solutions provider, he brings over 30 years of experience in software architecture, development strategy, and cross-technology integration. A recognized industry voice and IBM Redbooks author, he regularly publishes and speaks at international events. Pascal joined this residency to contribute his expertise in automated testing, AI validation, and globalization.

Tim Rowe is a Senior Technical Staff Member and Business Architect at IBM for Application Development, where he has contributed to operating system development since 1989. He leads the application development & systems management teams and is a frequently published author and speaker at industry events worldwide. Tim actively shares best practices for development and system management through the iSee Video blog, speaks at countless events. When not talking about modernization on IBM i, he can be found on the Soccer pitch where he is a certified official.

Marina Schwenk is a highly accomplished and recognized leader in the IBM i community, distinguished by a rapid and impactful career. Since launching her career in the field in 2015 at Everbrite in Greenfield WI as a programmer analyst, she now holds the position of Senior Programmer analyst and IBM i Admin for Innovative Software Solutions inc. She was recognized early when she was named an IBM Fresh Face in 2019, four-year recognition as an IBM Champion, COMMON Americas John Earl Scholarship in 2020 and the Al Barsa Scholarship in 2022. Schwenk actively serves in several key leadership roles. She is the President of the Wisconsin Midrange Computer Professional Association (WMCPA), COMMON Americas Advisory Council (CAAC), and COMMON Americas Board of Directors, helping to shape the future of the world's largest community of IBM Power Systems users.

Vikas Sonwal is a Technical Architect in the Java Team at Programmers.io, based in Jaipur, India. He has over 18 years of experience in Java, API, Web, and Mobile technologies and focuses on a range of responsibilities spanning project management, delivery management, and technical architecture across different projects. Vikas holds a Master's in Computer Application and has worked on notable solutions including Green2Glass, a product from Programmers.io. He contributed to this IBM Redbooks publication during a residency on UI Modernization, sharing approaches to developing mobile applications with IBM i as a backend, using modern technologies such as Flutter, React Native, Angular, and Node.js.

Todd Stewart is an Application Architect at Heartland Cooperative, Iowa's second largest agriculture co-op. He has over 35 years of experience in IBM midrange computing, developing applications from the System/34 to IBM i on Power. Todd has held roles including Developer, Analyst, Project Lead, Solutions Delivery Manager, and Application Development Director in the retail and bonding/insurance industries. He is a 2026 IBM Champion, a member of the COMMON Americas Advisory Council, COMMON Board of Directors, and holds a DevOps Foundation Certification. Todd contributed to this IBM Redbooks publication during a residency on IBM i modernization and DevOps practices.

Jeff Tickner is a Senior Consultant and Workflow Architect at Arcad Software, where he has contributed to Application Life Cycle Management projects since 2008. He designs and implements highly automated workflow for customers using both traditional CM and Git based open systems tools for customers of all sizes with all kinds of business requirements. Jeff actively shares best practices for development workflow with Git and he was part of the team that ported Subversion server to run on IBM i back in 2005. When not talking about Git on IBM i, he can be found on the ski hill where he is a Volunteer Ski Patrol.

George VanEaton is a Manager of Product Development at MEDHOST, based in Tennessee, United States. He has over 30 years of experience in IBM i application development, specializing in RPG, Db2 for i, and enterprise healthcare systems. George focuses on modernizing legacy architectures through data services, stored procedures,

and AI-assisted development using tools such as IBM Bob™ and GitHub Copilot. He also shares insights on IBM i modernization and leadership through his professional platform, The RPG Blend. He leads cross-functional teams delivering regulatory initiatives, including Price Transparency. He contributed to this IBM Redbooks publication during a residency on IBM i modernization and AI-assisted development practices.

Khushi Wadhwa is an IBM i developer at Programmers.io, based in Jaipur, India. She has 4 years of experience in IBM i application development & modernization. Her experience spans API integration, web application development, and the adoption of DevOps practices, with a focus on implementing modern authentication standards, building secure web solutions, and driving git version control. She contributed to this IBM Redbooks publication during a residency on web application development on IBM i.

Carol Woodbury is Senior Advisor and IBM i Security SME at Kisco Systems. Carol has over 30 years' experience in IBM i Security beginning as Team Leader and Chief Engineering Manager for Security at IBM in Rochester, MN. Since leaving IBM, she has helped organizations - large and small - strengthen their security posture by using the integrated features of the operating system. Carol is a frequent speaker at user groups around the world and has written many articles on IBM i Security. She has two current books - IBM i Security Administration and Compliance, third edition and Mastering IBM i Security, both available from amazon.com. Carol holds her CISSP and CRISC security certifications and has been an IBM Champion since 2018.

Andy Youens is the Managing Director of UK based FormaServe Systems, an IBM ISV since 1990. Andy is a seasoned IBM i professional with over 45 years of experience in the field. He has a deep expertise in IBM i systems, including system administration, application development and modernization. Andy's extensive knowledge and hands-on experience have made him a sought-after expert in the IBM i community, both at home and internationally. As an active contributor to the IBM i community, Andy frequently shares his insights and expertise through speaking engagements, webinars and technical articles. An IBM champion, who co-wrote the IBM i developer certification, the latest IBM Developer video & exam and Redbook on Power Modernization. He is the owner of PowerWire, the monthly newsletter and articles on IBM Power. Prior to IT, Andy is proud to have served in the UK's armed forces, the Royal Navy.

Justin(Jia Tian) Zhong is a Brand Technical Specialist for IBM Power at IBM, with nearly three decades of experience on IBM i. For more than 17 years, he has served as a Senior Software Engineer specializing in Db2 for i and IBM i performance at the IBM China Systems Technology Laboratory (CSTL). Justin is also an active contributor to the IBM i community, sharing his expertise across China, Hong Kong, and Taiwan.

Understanding Modernization Fundamentals

What does modernization mean for IBM i?

What It Is

Modernization comes in many forms; it is not just simply the process of updating a user interface to make it look contemporary. In this Redbook we will show you how to modernize your applications from the ground up, taking advantage of new IBM i specific operating system features and functions, as well as harnessing the power of open-source tools and languages.

We discuss different development languages, how to select and combine them. We demonstrate how to take advantage of Db2 for i to deliver faster, more robust and more secure data to your applications.

Covering a wide range of User Interfaces, Development Tools and Integration techniques, you will learn how to create modern, modular applications that are easier to maintain and future proof against the ever-changing demands of modern solutions.

You'll also learn to use Artificial Intelligence (AI) in your development, testing modern applications as well as integrating AI into your application to make it more functionally rich.

Finally, we ensure that we keep security in mind throughout so as to create "Secure by Design" applications that are less likely to have exploits and will be easier to remediate in the event that one is found.

Why It Matters

Few things stand still in this world, and software applications are no exception. If you do not continuously enhance them, then they wither and eventually die. It is vital that any application that has users that depend upon it be well designed, maintained and supported. In addition to making applications more functionally rich, we also need to consider how they will be maintained in the future. It is not uncommon for a good application to have a life that spans many decades. Therefore it is important that it consistently evolves with the times to meet the current needs and enable future needs to be added without reintroducing old bugs or new security flaws.

How It Works

Modernization is not a single process that is completed and then forgotten about. You deliver true modernization by constantly reviewing how things are done, embracing new tools, protocols and working practices.

This doesn't mean you must constantly rewrite entire applications. Instead, constantly evolve your development, testing and support processes to take advantage of these new ideas as you carry out your everyday work.

For example, should you need to introduce a new database table, consider taking advantage of new functionality from day one. Adding functionality like:

- Temporal Support to improve Business Intelligence
- Database Constraints to add business logic to your data
- Row Column Access Control to secure your data by design
- Journaling to improve auditing
- Commitment Control to improve data integrity

Key Takeaways

- Modernization can be applied at all layers of an application solution, from operating system to user interface.
- Modernization is an iterative process that constantly evolves but does not require that you throw out all your previous work to implement it.
- Modernization makes your applications better, more secure, and easier to maintain.
- Modernization helps you maintain your relevance and reputation as a developer in a world that is increasingly taking advantage of AI and Automation.
- Instead of feeling threatened by modernization, we demonstrate how to embrace it and how it can make your life easier, more productive and hopefully more interesting.

What Approaches to Modernization Exist?

Opening Summary When organizations decide to modernize their IBM i applications, they quickly discover that "modernization" encompasses a vast range of methodologies. Some teams focus on transforming user interfaces while preserving existing business logic. Others prioritize architectural changes that enable better integration and scalability. Still others concentrate on adopting modern development practices and tools that improve productivity and code quality.

The reality is that modernization is not a single path but rather a multifaceted journey that can touch every aspect of your application ecosystem. Covering hardware, operating system version, database, business logic, user interfaces, development practices, even deployment and ongoing security practices. Understanding these various dimensions and how they interconnect is essential for crafting a modernization strategy that delivers real business value rather than simply checking boxes on a technology trend list.

Why It Matters

The approach you choose for modernization determines far more than just the technical outcome. It shapes your timeline, budget, risk profile, and the skills your team needs to develop or acquire. More importantly, different approaches address different business problems. Modernizing your user interface improves user experience but doesn't necessarily make your applications easier to integrate with other systems. Adopting modern development tools increases developer productivity but doesn't automatically improve application performance or security.

Understanding the full spectrum of modernization approaches allows you to make strategic decisions that align with your specific business objectives rather than pursuing modernization for its own sake. Organizations that succeed in modernization typically don't choose a single approach but rather combine multiple complementary strategies. They might modernize interfaces for user-facing applications while simultaneously refactoring their database design and adopting DevOps practices for faster, more reliable deployments. The key is understanding what each approach offers and how different approaches can work together to achieve your broader goals.

Key Takeaways

1. **Modern Architecture and Development Practices Transform How You Build** Moving to modern architecture represents one of the most fundamental shifts in IBM i modernization. Service-oriented architecture (SOA) and microservices approaches break monolithic applications into smaller, more manageable components that communicate through well-defined interfaces.

   This architectural shift enables you to expose IBM i business logic as APIs or web services that can be consumed by web applications, mobile apps, or partner systems. This will also enable easier and productive integration with the next generation of AI assistants.

   The benefits extend beyond integration. Modular architectures make applications easier to maintain, test, and evolve over time.

   Modern development tools and practices amplify these architectural benefits. Integrated development environments like RDi (Rational Developer for i), VS Code with IBM i and IBM Bob™, extend to provide features that developers expect: intelligent code completion, integrated debugging, source control integration, and refactoring tools. These tools dramatically improve developer productivity compared to traditional SEU-based development.

   Adopting modern development practices like CI/CD, DevOps, and automated testing is equally important. These practices reduce deployment risks and enable faster delivery of new features. When combined with modern source control systems like Git, these practices provide better collaboration, change tracking, and the ability to roll back problematic changes quickly.

2. **Interface Modernization and Programming Style Evolution Go Hand-in-Hand** Modern interfaces represent the most visible aspect of modernization, transforming green screen 5250 applications into responsive web and mobile experiences. Tools for screen modernization can wrap existing programs with contemporary interfaces, providing quick wins with minimal code changes. However, deeper interface modernization often involves rethinking the entire user experience. Moving from screen-by-screen navigation to more intuitive, task-oriented workflows.

   This interface evolution naturally leads to changes in programming style. Modern applications increasingly adopt data-centric development approaches where business logic focuses on data manipulation and validation, while presentation logic is handled separately by web frameworks or mobile applications. This separation of business logic and interface makes an application more maintainable and enables the same business logic to serve multiple interface types: web, mobile, B2B, and even traditional green screens.

3. **Embrace Open Source Where Appropriate** The shift toward modern programming styles also includes embracing open-source languages and libraries. Many IBM i organizations now use Node.js, Python, PHP, or Java alongside traditional RPG and COBOL, leveraging the rich ecosystems of open-source libraries for tasks like communication, data transformation and or integration with third-party services.

   This approach allows you to use the right tool for each job while maintaining your investment in proven IBM i business logic. Modern RPG itself has evolved significantly, supporting free-format syntax, embedded SQL, and modern programming constructs that make code more readable and maintainable.

4. **Database Re-engineering and Integration Enable True Transformation** Database modernization often receives less attention than interface changes but can deliver profound benefits. Many IBM i applications use physical files with embedded business logic in trigger programs or rely on logical files for data access.

   Re-engineering these structures to use SQL-based views, stored procedures, and modern database features improves performance, simplifies maintenance, and enables better integration with reporting and analytics tools.

Database modernization also facilitates data-centric development by creating clear, well-documented data models that can be accessed through multiple channels.

5. **Opening Your Application to Third-party Integration** Web and application integration represent another critical dimension of modernization. Modern applications rarely exist in isolation, they need to exchange data with e-commerce platforms, CRM systems, payment processors, and countless other services.

   Enabling this integration requires exposing IBM i functionality through REST APIs, consuming external web services, and implementing robust data exchange mechanisms. The IBM i platform provides excellent tools for this integration, from built-in web services support to open-source integration frameworks.

6. **Create Secure Applications from the Start** Security modernization must accompany these integration efforts. As applications become more connected and accessible from more locations, security requirements intensify. Modern security practices include implementing OAuth or JWT-based authentication for APIs, encrypting data in transit and at rest, implementing role-based access control with fine-grained permissions, and maintaining comprehensive audit trails. Security can no longer be an afterthought—it must be integrated into every aspect of your modernization strategy.

7. **Embrace DevOps** DevOps practices tie everything together by enabling rapid, reliable delivery of changes. Automated deployment pipelines, infrastructure-as-code and comprehensive monitoring transform how you deliver and maintain applications. DevOps isn't just about speed. It reduces risk through automation, improves quality through continuous testing, and enables faster response to business needs.

Bringing It All Together

Successful modernization rarely means pursuing all these approaches simultaneously. Instead, it requires understanding your specific business drivers and choosing the combination of approaches that addresses your most pressing needs while building toward your long-term vision.

For example, you might start with interface modernization to improve user experience, then adopt modern development tools to increase productivity, followed by database re-engineering to enable better analytics, and finally implement comprehensive DevOps practices to accelerate delivery.

The key is recognizing that these approaches complement each other; modern interfaces work better with well-designed databases, modern development tools enable better security practices, and DevOps makes it safer to adopt new technologies.

Why Modernize? Key Drivers and Benefits

Why modernize the database layer?

What This Covers

This topic delves into the reasons and possible benefits to be realized by choosing to modernize the data model.

What It Is

Database modernization on IBM i transforms legacy DDS-based files into modern SQL DDL tables, enabling advanced Db2 for i capabilities that improve performance, security, maintainability, and integration with modern tools.

Why It Matters

Here are some common scenarios that can benefit from database modernization:

- The database must be modernized to make it more accessible to users and to ensure that the use of reporting and analytics software does not adversely affect the performance of the system.
- A new application is written over an existing database. The implementation of the new application would benefit from Db2 features that are only supported by SQL (Structured Query Language) such as auto-generated columns.
- An existing application is rewritten. Modernizing the database reduces the complexity of the programs in the rewritten application.

- Data integrity and business rule enforcement is embedded within the application program logic. As a result, the integrity of the database is regularly compromised by interfaces that directly update the database without using the existing programs. A modernized database would enable data integrity and simple business rules to be enforced at the database layer, so data is protected at all times.

Key Takeaways

Benefit Category	Key Improvements
Talent	Easier to find and retain talent on your team
Integration	Standard SQL, REST APIs, modern tool compatibility
Maintainability	Centralized logic, reduced code complexity
Security	RCAC, encryption
Features	More data types, Temporal tables, JSON/XML, generated columns
Cost	Lower TCO, faster development, better ROI

Why modernize application interfaces?

What This Covers

The green screen interface has been the face of IBM i applications for generations. It's familiar, efficient for trained users, and has powered countless business operations reliably for decades. Yet, when a new employee encounters a 5250-terminal session for the first time, their reaction often ranges from confusion to disbelief.

This disconnect between what your applications can do and how users experience them represents one of the most critical challenges facing IBM i organizations today. Modernizing application interfaces isn't about chasing trends or abandoning proven technology, it's about ensuring your valuable business logic remains accessible, usable, and relevant in a world where user expectations have fundamentally changed.

Why It Matters

Your application interface is the bridge between your business logic and the people who depend on it. When that bridge is difficult to cross, the consequences ripple throughout your organization. Productivity suffers as users struggle with interfaces that require extensive training and memorization. Error rates can increase when cryptic codes replace intuitive visual cues. Recruitment becomes harder when talented professionals view legacy interfaces as a career dead-end rather than an opportunity.

The business case for interface modernization extends beyond user satisfaction as modern interfaces enable mobile access, allowing your workforce to be productive from anywhere. They make your existing IBM i applications active participants in digital transformation rather than isolated islands.

Perhaps most importantly, a modern interface protects your investment in IBM i by making your applications more attractive to use and easier to maintain, extending their viable lifespan and maintaining their relevance.

Key Takeaways

1. User Experience Directly Impacts Business Performance Every keystroke, every screen navigation, and every moment of confusion adds up across your organization. When users must memorize function key combinations, navigate through multiple screens for simple tasks, or manually transcribe information between systems, you're paying a hidden tax on productivity.

Modern interfaces reduce this friction through intuitive design, contextual information, and streamlined workflows. The impact is measurable: faster transaction processing, reduced training time for new employees, and fewer errors that require correction. Organizations that modernize their interfaces often report significant and tangible productivity improvements. This is not because the underlying business logic changed, but simply because users can work more naturally and efficiently.

2. A Modern Workforce Expects a Modern Interface The workforce entering your organization today grew up with smartphones, web applications, and intuitive interfaces. When you ask them to work with green screen applications, you're not just presenting a learning curve, you're sending a message about your organization's commitment to modern practices.

   Talented developers and business users increasingly view legacy interfaces as a red flag, questioning whether the organization values innovation and employee experience. Conversely, modernizing your interfaces signals that you're investing in the future, making your organization more attractive to the talent you need to compete. This isn't about superficial appearances; it's about creating an environment where skilled professionals want to build their careers.

3. Modern Interfaces Enable New Business Capabilities Interface modernization opens doors that green screens simply cannot. Mobile access allows field service technicians to update work orders in real-time, sales representatives to check inventory while meeting with customers, and managers to approve requests from anywhere.

   Web-based interfaces enable customer and supplier self-service portals, reducing administrative overhead while improving service levels. Modern interfaces also facilitate integration with analytics tools, allowing users to visualize data and gain insights that would be invisible in traditional character-based displays. These are not just "nice-to-have" features, they're competitive advantages that can differentiate your business in the marketplace.

Why modernize program code?

What This Covers

Justifying the resource, cost, and risk of modernizing code is important. To convince your business that a modernization project is a sound investment, you need to convey the benefits of modernization.

Why It Matters

Modernization projects can deliver significant value to the business, including:

1. Faster velocity for satisfying business requirements
2. Lower maintenance costs
3. Easier for the business to acquire and on-board talent
4. Improved ability to retain staff
5. Doors open to using CI/CD tools, test automation and other DevOps methodologies
6. Elimination of duplicated code
7. Easier transition to CI/CD, test automation and other DevOps methodologies
8. Ability to modernize the UI/UX and database

Developer Productivity and Talent

Modern code is easier for developers to learn and maintain.

- **Free-form syntax** (RPG, COBOL) is more readable and familiar to developers with experience in other languages
- **Standard programming constructs** (IF/ELSE, DO/WHILE) replace legacy operation codes
- **Easier onboarding** for new developers reduces training time and costs
- **Talent acquisition** is significantly easier when using modern coding practices
- **Developer satisfaction** improves, leading to better retention

Example:

```
// Modern free-form RPG
if customerStatus = 'ACTIVE';
  processOrder(orderId);
endif;

// vs. Legacy fixed-format
C     CUSTSTAT  IFEQ      'ACTIVE'
C                   CALL      'PROCORD'
C                   PARM                    ORDERID
C                   ENDIF
```

Key Takeaways

- The cost and time needed to modernize an application's code will have significant ROI through reduced maintenance costs, improved developer productivity, and enhanced system capabilities
- Future maintenance costs decrease dramatically with modern, modular code
- Staffing challenges are reduced as modern code attracts and retains talent
- Developer efficiency increases through code reuse, better tools, and clearer structure
- Business agility improves with faster feature delivery and easier integration
- Technical debt is reduced, preventing future costly rewrites
- Competitive advantage is gained through faster time-to-market and innovation

Why Enable Cloud and Mobile Access?

What is it?

The traditional model of business computing (employees sitting at desks, connected to systems through local trusted workstations and networks) has fundamentally changed. Today's workforce expects to access information and complete tasks from anywhere, whether at their desk, roaming the warehouse, working from home, at a customer site, or even traveling between locations. The new mantra has to be making our applications online, all the time.

Meanwhile, cloud computing has transformed how organizations think about infrastructure, offering flexibility and capabilities that were once available only to the largest enterprises. For IBM i organizations, the question is no longer whether to enable cloud and mobile access, but how quickly can you do so without compromising the security and reliability that make your platform valuable.

Why It Matters

Business doesn't happen only at desks anymore. Your sales team needs inventory information while meeting with customers. Your service technicians require work order details at job sites. Your managers need to approve requests and review dashboards regardless of their location. When your IBM i applications are accessible only from office workstations, you're forcing your business to operate within artificial constraints that competitors without legacy systems don't face.

The implications extend beyond convenience. Organizations that cannot provide mobile and cloud access face real competitive disadvantages. They risk losing deals because sales representatives cannot access real-time information during customer meetings. They experience delays because approvals must wait until someone returns to the office. They struggle with business continuity when circumstances prevent employees from reaching physical workstations.

Cloud and mobile access are not a passing fad, they are technologies that remove barriers that limit an organization's productivity, agility and resilience in an increasingly demanding business environment.

Key Takeaways

1. Mobile Access Transforms How Work Gets Done: Enabling mobile access to IBM i applications fundamentally changes the relationship between work and location. Field service technicians can update job status, record parts usage, and capture customer signatures on-site, eliminating paperwork and reducing the time between service completion and billing. Warehouse staff can perform inventory transactions using mobile devices, improving

accuracy and efficiency. Executives can review key metrics and approve critical decisions from anywhere, preventing bottlenecks that slow business operations.

The value is not just about doing the same work in different places; it is about enabling workflows that weren't previously possible. When information flows in real-time between mobile workers and your IBM i systems, you eliminate delays, reduce errors from manual data entry, and create a more responsive organization.

2. Cloud Capabilities Provide Flexibility Without Sacrificing Control: Cloud access to IBM i applications can offer benefits that extend far beyond remote connectivity. It enables you to scale resources dynamically based on demand, rather than provisioning for peak capacity that sits idle most of the time. It facilitates disaster recovery and business continuity by providing geographic redundancy without the cost of maintaining multiple physical data centers.

   Cloud deployment can also simplify partner and customer access, you can provide secure, controlled access to specific applications without requiring VPN configurations or complex network arrangements.

   Importantly, cloud access doesn't mean abandoning your IBM i platform or migrating everything off premises. Many organizations adopt hybrid approaches, keeping core systems on premises while using cloud services to extend access and capabilities. This flexibility allows you to modernize at your own pace while maintaining security and maximizing flexibility.

3. Remote Access Enables Business Continuity and Attracts Talent: Recent years have demonstrated that organizations capable of operating remotely possess a critical competitive advantage. When circumstances prevent employees from reaching physical offices, whether due to weather, health concerns, or personal situations, mobile access ensures business operations continue uninterrupted. This resilience protects revenue, maintains customer service levels, and provides peace of mind for both leadership and employees.

   Additionally, enabling remote access expands your talent pool dramatically. You are no longer limited to hiring within commuting distance of your office(s); you can recruit the best person for the job regardless of their location.

   In an increasingly global market, where attracting and keeping the best talent is increasingly challenging, this flexibility can be the difference between filling critical positions and leaving them vacant.

What are the business benefits of modernization?

There are several business benefits to modernizing your IBM i environment. Modernization is not just limited to the code base, database or OS version you are running. Modernization encompasses your entire environment.

Benefit #1: Operations

Modernization directly reduces your overall operational risk. At the application layer, you eliminate the catastrophic exposure of relying on a single person who "knows it all" ensuring your code is manageable, maintainable, and transferable to any qualified resource for years to come. A modernized database is accessible, scalable, and understandable, removing the bottleneck of tribal knowledge and making your data a business asset rather than a liability. At the system level, hardening your security posture reduces your exposure to both internal and external threats — protecting the business from risks that legacy configurations were never designed to address.

Benefit #2: Total Cost of Ownership

Modernization is not an expense. It stops the hemorrhaging of IT costs and lowers your overall budget over time. By modernizing the code base, database and system you are lowering your day to day, year by year costs by ensuring you have a scalable, secure foundation that your business can grow from.

Benefit #3: Talent Acquisition and Retention

An unmodernized system, code base and database comes with an expectation that the skillset needed to maintain the existing system is more advanced than new IT professionals can handle. By modernizing your system, code base and database you are eliminating the risk of not finding the resources you need to maintain your existing system. Developers, system architects, admins all want to work on a modern system; they don't want to lose any of the skills they have retained by experience and formal training. The newer generations are more likely to jump ship if they are expected to work on system/36 code, multi member files and no possibility of using newer modern tools.

Benefit #4: Scalability

When you have a monolithic code base, outdated system and database the business struggles to scale to new heights. New requests take a lot of time to implement causing businesses to struggle to scale. Over time the delay in implementing new business requests has a bigger impact to the business than expected. With Modernization you see business requests implemented in a quick and orderly fashion, enabling your business to scale to new heights.

Benefit #5: Business Growth

On the same token as scalability, when you modernize your codebase, database and system you find that you are able to take a good hard look at business opportunities that help your business grow. Imagine having System/36 code, multi member files and trying to acquire another company and have that company supported by your existing set up. It would take months or years to fully onboard the new company where the ROI shrinks in the amount of time it takes for the request to be fulfilled.

Conclusion

Overall the IBM i is your core business system and it should stay that way. Its scalability, security capabilities and ability to interconnect with various systems allows your opportunities to be endless you just have to take the step and modernize what you have and take advantage of everything IBM i is offering you.

Planning Your Modernization Journey

Where should I start my modernization journey?

What This Covers

Modernizing an IBM i application can feel like standing at the base of a mountain, looking up at the peak shrouded in clouds. The path forward isn't always clear, and the sheer scope of possibilities can be overwhelming. So not feeling certain about where to start is common and you should not let this concern you in any way. But that does not mean it is not important to consider where to start.

What It Is

Many of the most successful modernization journeys don't begin with grand technical blueprints or a massive budget. They start with a simple question: "What problem are we trying to solve?"

Whether you're a developer eager to leverage modern tools or a manager balancing business needs with technical debt, understanding where to begin is the difference between a transformation that benefits your organization and one that antagonises it.

Why It Matters

The IBM i platform has proven its reliability and value for decades, but the world around it has changed dramatically. Your users expect responsive, intuitive interfaces; your business demands reliability and integration with new technologies; and both parties often take it for granted that your ability to quickly respond to changes, without impacting them, is not an easy balance to maintain.

How It Works

Starting in the right place matters because resources are finite. Every organization has constraints on time, budget, and expertise. A poorly chosen starting point can consume resources without delivering visible value, eroding confidence in the entire modernization initiative. Conversely, beginning with the right focus creates momentum, demonstrates quick wins, and builds the organizational support needed for sustained transformation.

Key Takeaways

1. Start with User Pain Points, Not Technology: The most effective modernization journeys begin by listening to your users, your customers, and your business stakeholders. Find out what frustrates them most about the current system?

   Is it the green screen interface that requires extensive training? The inability to access information on mobile devices? Slow response times during peak hours? Inability to integrate with key partner systems? By identifying

the most pressing pain points, you create a clear target for your initial efforts. This approach ensures your modernization delivers immediate, tangible value that people can see and appreciate.

2. Assess Before You Act: You cannot effectively modernize what you don't understand. Before making any changes, invest time in documenting your current application, as a minimum, ensure you understand:
    - What programs are actively used versus those that are remnants of how you used to work.
    - Which business processes depend on which applications.
    - Where are the integration points with other systems.
    - Which users use which parts of these applications.

    This assessment doesn't need to be exhaustive or perfect, but it should give you enough visibility to make informed decisions. Many organizations discover during this phase that significant portions of their codebase are no longer used, immediately simplifying the modernization scope. Others identify critical dependencies that must be preserved, preventing costly mistakes down the road.

3. Think Big, Start Small, Scale Fast: Successful modernization requires balancing ambition with pragmatism. Develop a vision for where you want to get to but don't try to achieve it all at once. Instead, identify a small, well-defined project that can demonstrate value quickly, ideally within days or weeks rather than months or years.

    This might be modernizing the user interface for a single, frequently used application, or exposing a critical business function as a web service, or making it easier for users to access key performance data.

    The goal is to prove the concept, learn from the experience, and build confidence. Once you've achieved that first success, you can scale the approach across more applications, refining your methods and building expertise along the way.

What roadblocks should I anticipate?

What This Covers

Modernization often involves adopting new technologies, redesigning processes, and encouraging cultural change within an organization. However, while the benefits of modernization can be significant, organizations frequently face several roadblocks during the journey. These obstacles can slow progress, increase costs, or even cause projects to fail if they are not anticipated and managed effectively.

What It Is

Common roadblocks include resistance to change, legacy systems, lack of clear strategy, limited skills within the workforce, and budget constraints. At the same time, employees may feel uncertain or uncomfortable about changes to familiar tools and workflows.

Without careful planning and communication, these challenges can stall progress and reduce the overall effectiveness of modernization efforts. Understanding these potential barriers early allows organizations to plan more effectively and build strategies that reduce risk while maintaining momentum.

Why It Matters

Recognizing roadblocks in modernization is important because when obstacles arise unexpectedly, they can delay implementation, increase costs, and create frustration among teams. In some cases, poorly managed modernization projects can disrupt operations or fail to deliver the expected benefits.

By anticipating challenges, you can design more realistic plans and allocate resources more effectively. For example, identifying skills gaps early allows organizations to provide training or hire specialists before major systems are implemented.

Addressing roadblocks also improves employee engagement. When organizations communicate openly about challenges and involve employees in the process, they build trust and encourage collaboration. This helps create a culture that supports innovation and continuous improvement.

How It Works

Successfully navigating modernization roadblocks requires a proactive and structured approach.

First, conduct a thorough assessment of their current systems, processes, and capabilities. This helps identify potential risks such as outdated technology, complex integrations, or operational dependencies that could complicate modernization efforts.

Second, create a clear and realistic strategy. A well-defined roadmap with measurable goals helps teams understand priorities and ensures that modernization initiatives remain aligned with broader business objectives. Breaking large transformations into smaller phases can also make the process more manageable and reduce disruption.

Third, investing in people is essential. Training programs, workshops, and open communication help employees understand the purpose of modernization and develop the skills needed to work with new technologies and processes. When employees feel supported, they are more likely to embrace change rather than resist it.

Finally, organizations should adopt flexible implementation methods such as iterative or agile approaches. These allow teams to test solutions, gather feedback, and adjust plans before full-scale deployment. Continuous evaluation ensures that problems are identified early and addressed quickly.

Key Takeaways

Resistance and legacy systems are major barriers. Many modernization efforts struggle because of outdated infrastructure and reluctance to change.

Planning reduces risk. Identifying potential roadblocks early allows organizations to prepare solutions before problems escalate.

People are critical to success. Training, communication, and strong leadership help teams adapt and support long-term transformation.

Don't let perfection get in the way of progress. Getting started can be daunting and so if in doubt, start small, demonstrate success and then use this as part of your justification for future modernization projects including any investments and support you may need.

What Problem Are We Actually Trying to Solve?

What is it?

Ironically, one of the most dangerous moments in any modernization initiative comes right at the beginning, when enthusiasm is high and the path forward seems clear. Teams gather around whiteboards, vendors present compelling demonstrations, and everyone agrees that "we need to modernize." Yet beneath this apparent consensus often lies a fundamental disconnect, with different stakeholders imagining different problems and different solutions.

For example: An IT director might worry about the available resources, the CFO might want a solution that lowers ongoing running costs, the sales team might want real-time access to inventory information on mobile devices and an operations manager wants a more intuitive interface to make his team more productive.

Each of these is a real problem, but they require different solutions. Rushing into modernization without clearly defining which problems you are trying to solve is like setting out on a journey without knowing your destination, you might end up somewhere, but probably not where you need to be.

Why It Matters

Problem definition isn't just an academic exercise or a box to check before getting to the "real work" of modernization. It's the foundation that determines whether your modernization initiative will deliver business value or white elephant that fails to meet expectations.

When you don't clearly define the problem, you cannot measure success, you cannot make informed decision between competing approaches, and you cannot effectively communicate with stakeholders about what you are doing and why it matters. These misalignments don't just waste money, they erode confidence in IT, create organizational fatigue toward change and leave the real problems unsolved.

How it Works

User Experience Problems Versus Technical Debt Problems One of the most common sources of confusion in modernization initiatives is conflating user experience problems with technical debt problems. These are fundamentally different challenges requiring different solutions.

- User experience problems: These manifest in how people interact with your applications: difficult navigation, cryptic error messages, inability to access systems from mobile devices, workflows requiring too many steps, or interfaces demanding extensive training. These problems directly impact productivity, error rates, and user satisfaction. They're highly visible to business stakeholders and create urgency for change.
- Technical debt problems: These live beneath the surface, often stemming from poorly structured code that's difficult to maintain, a lack of automated testing that makes changes risky, monolithic architectures preventing independent deployment, or database designs making reporting difficult. These problems primarily impact IT's ability to respond to business needs quickly and reliably. They're often invisible to business stakeholders until they manifest as slow delivery of new features or frequent production issues.

User Experience and Technical Debt problems generally require different solutions and have different business cases. Screen modernization tools can address user experience problems quickly with relatively low risk, but they don't resolve technical debt. Conversely, refactoring code to improve maintainability doesn't immediately improve user experience.

Many organizations need to address both, but trying to solve them simultaneously in a single initiative often leads to scope creep, extended timelines, and diluted focus. Understanding which problem is more urgent for your business or recognizing that you need parallel efforts to address both is essential for effective planning.

Key Takeaways

Integration and Accessibility Challenges in a Connected World Modern business operates in an ecosystem of interconnected systems. Your IBM i applications do not exist in isolation, they need to exchange data with other platforms, share information with mobile applications, integrate with partner systems, and feed data to analytics tools.

When stakeholders say, "we need to modernize," they're often really saying "we need our IBM i applications to participate more effectively in our broader technology ecosystem." Good modernization strategy will generally involve a design that allows for easier integration with other applications, you must resist the temptation to try and do everything inside the IBM i environment.

Defining Your Problem Clearly The path to successful modernization begins with honest, specific problem definition. Instead of saying "we need to modernize," articulate exactly what's not working. For example, "Our sales team loses deals because they can't check inventory while meeting with customers," or "We can't deliver new features quickly because our code is difficult to test and deploy," or "We're losing experienced developers and can't recruit replacements."

These specific problem statements lead to focused solutions, measurable outcomes, and stakeholder alignment. They also help you avoid the trap of modernizing for modernization's sake, pursuing technology changes that do not actually address your business challenges.

Take the time to clearly define what problem you're solving, and you'll dramatically increase your chances of modernization success. If a problem becomes too complex to clearly express, then break it down into a series of smaller problems and describe success for each of them individually as well as the group as a whole.

What happens if we do nothing?

What This Covers

When you consider the consequences of doing nothing, it is important to consider not only the functionality and aesthetics of your applications but the wider implications to the applications' long term viability in your business.

What It Is

Doing nothing can often seem like a safe and sensible strategy, as change can often bring additional work, cost and resistance from colleagues, management, users and other stakeholders.

It can feel like a poor proposition with the IT team getting the majority of blame for the pain of change and the minority of the recognition for improved productivity, security and long-term cost savings that such enhancement deliver.

This do-nothing attitude is often expressed via the now infamous mantra "if it ain't broke then don't fix it".

Why It Matters

The problem with this attitude is that doing nothing for prolonged periods of time means that when something does need to be done, it becomes much more difficult to achieve, this is for several reasons:

- Atrophy of application knowledge
- Less able to deliver new functionality quickly and cost effectively
- Longer, more complex development processes
- Loss of access to application developers and harder to train replacements
- More difficulty when integrating with third party APIs and tooling
- Application no longer suits the way the business operates
- Higher risk of business interruption as changes are implemented
- Harder to implement and audit more application and data security

Any one of these bring can bring the overall value of the application into question, leading business management to consider replacement rather than enhancement and this replacement may include replacing you and your colleagues.

How It Works

Nothing stands still in the IT world, so if you are not moving forward then you are falling behind. This is often the key reason for technical debt accumulating over the years.

It is often the result when business leaders ask more from their existing teams than application support and enhancement. It is made worse when companies do not replace existing IT resources as they move on.

In some cases, businesses have chosen to outsource their IT support and development in the true belief that they can improve productivity and reduce costs. Unfortunately, all too often they find that such short-term savings lead to higher longer-term costs, costs not just measured financially but also in modification timescales, data risk and lost opportunity.

Key Takeaways

One such example is that applications designed in years gone by focused on data priority in their core design but gave little to no thought about data security. The modern IT environment means that such systems run a much higher risk of unauthorized data access, exfiltration or even malicious corruption.

The key takeaway is that Modernization needs to be a constant factor that is considered by all IT professionals and their management. As a failure to invest in an application that is key to the way a business operates, is a risk to the long-term success of that business.

Strategic Considerations

Is modernization about technology, process, or culture?

What This Covers

Modernization is often interpreted as simply adopting the latest technology. While new tools and platforms play an important role, true modernization is broader and requires a combination of all three elements: technology, process, and culture.

What It Is

Modernization is not just about upgrading systems, replacing or updating software. It is a strategic transformation, that when done right, improves how an organization operates, delivers value, and adapts to change. Technology such has hardware, software and programming languages continuously evolve and an agile modernization strategy will

embrace the functionality that these new technologies bring in order to enhance the overall solution in the most efficient, cost-effective and secure manner.

Processes ensure that these capabilities are planned, tested and used efficiently through improved throughput. Culture is often overlooked and is a key factor as to whether people within an organization are willing and able to embrace the changes that this modernization delivers. Therefore, the most successful modernization strategies require a balance of all three elements Technology, Process and Culture.

Why It Matters

Modernization matters because organizations operate in rapidly changing environments. Customer expectations evolve quickly, competition increases, and technological advances constantly reshape industries. Organizations that fail to modernize risk inefficiency, loss of competitiveness and in extreme case even data loss through corruption or security breaches. Additionally, modernization helps attract and retain talent. Employees increasingly expect modern tools, flexible ways of working, and organizations that encourage continuous productivity improvements.

How It Works

Implementing modernization requires a structured but adaptable approach. Ideally, organizations should establish a clear vision for modernization. Leadership must define what modernization means for the organization, including the outcomes they want to achieve, such as improved customer experience, faster product development, or operational efficiency.

Technology investments should align with these desired outcomes, not just purchases made because there is newer version available. For example, adopting new hardware or an operating system upgrade but just as important is deciding when to adopt new programming styles, programming languages and development tools. Thought should be given to adopting new technologies such Artificial Intelligence, automation tooling and integration with cloud services.

Processes must evolve to take advantage of new capabilities. For example, introducing automation should also involve revising workflows to remove redundant steps. Agile or iterative approaches can help organizations adapt processes continuously as they learn what works best.

Cultural change is essential. For example, modernizing your training, communication, and collaboration with other employees and business partners.

Together this helps understand why modernization matters and how they can contribute to it and when people feel supported and involved, modernization initiatives are far more likely to succeed.

Key Takeaways

Modernization requires balance. Technology, process, and culture must evolve together for meaningful transformation.

Modernization requires cooperation. Even the best technology and processes fail if people are unwilling to adopt new ways of working.

Modernization is continuous. It is not a one-time project but an ongoing effort to adapt, learn, and improve over time.

How can modernizing an IBM i application make it more secure?

What This Covers

YES! Without a shadow of a doubt, the best way to make an application more secure is to modernize it. And as you modernize you do so with both data integrity and data security in mind.

For more information, see IBM Security and Privacy by Design.

What It Is

IBM Security by design strategy is a streamlined and agile set of focused security and privacy practices. These practices aim to reflect their commitment to improving security and privacy when designing generally available products and services from IBM. it is influenced by the United States National Institute of Standards and Technology Secure Software Development Framework.

Why It Matters

Most businesses operating today must adhere to one or more compliance standards. Common amongst these are:

- CCPA – California Consumer Privacy Act & GDPR – General Data Protection Regulation: Legal regulations that govern how organizations collect, process, store, and protect personal data of individuals in their jurisdictions, that apply to companies that may reside outside.
- HIPAA – Health Insurance Portability and Accountability Act: A U.S. law that establishes national standards for protecting sensitive patient health information and securing electronic healthcare data.
- PCI DSS – Payment Card Industry Data Security Standard: A global security standard that requires businesses handling credit card data to maintain secure systems, protect cardholder data, and regularly test their security controls.
- SOX – Sarbanes-Oxley Act: A U.S. federal law that mandates strict financial reporting, internal controls, and corporate governance requirements for publicly traded companies.
- ISO/IEC 27001 – Information Security Management System Standard: An international standard that specifies requirements for establishing, implementing, maintaining, and continually improving an information security management system (ISMS).

All of the above have strict requirements for compliance and significant legal and financial consequences for failing to comply.

How It Works

This process does not mean that you undertake one single project to just focus on security of your application; instead add security as a key design factor to the changes you plan to make to your system as you modernize it.

One way to do this is by following IBM's secure by design strategy. To do this there are three steps that you must build into every change you make to any part of your application:

- Threat assessments: Perform both cyber and privacy threat assessments to help ensure data minimization and adequate data protection are in place from the beginning.
- Security testing: Perform System, code and application security tests; in addition consider using tools or third parties to perform penetration tests and manual ethical hacks. Such testing should be integrated and automated into DevOps pipelines, supporting agile practices and continuous deployment.
- Release review: Carry out a final product review of changes prior to release, check that key foundational security requirements have been or will be addressed to the satisfaction of your business's standards.

Key Takeaways

If you include "Secure by Design" thinking when making the changes to your system, over time you will consistently improve the security of your application alongside its functionality.

While there is an overhead in adding security to your modernization process, this is tiny compared with the effort of trying to retrofit security to an existing application on mass. The cost is also offset by the reduction in risk of large financial penalties that your company might be liable for in the event of a breach, as most regulators will reduce or even waive the fine if you can demonstrate the due diligence you give to security when making changes to your systems.

Trademarks

IBM, IBM i, Db2, WebSphere, Rational, and IBM Bob are trademarks of International Business Machines Corporation, registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available at ibm.com/trademark.

Visual Studio Code is a trademark of Microsoft Corporation.

Git is a trademark of Software Freedom Conservancy.

All other trademarks are the property of their respective owners.

The Path to Modernization

Assessment and Planning

How do I assess my current state?

Where Are You Starting

Every modernization journey begins the same way: with an honest pause and a single, critical question: where are you today? Before new platforms, new tools, or new architectures, there is this unavoidable question: where are you today? Not where strategy decks suggest you should be, not where competitors appear to be—but where your systems, teams, and practices truly stand. Your applications may be working. They may have worked reliably for years, quietly supporting the business through growth, change, and uncertainty. The real question is whether your software can adapt, scale, and evolve at the pace your organization now requires.

Understanding Your Starting Point: Build an inventory of your core applications

Your application landscape is not just a collection of systems. It is a record of past decisions: business priorities, technical constraints, shortcuts taken under pressure, and innovations that once felt bold. Even if your software is working, it might need to be modernized. There are important software quality attributes to consider beyond the functionality, such as maintainability, flexibility, usability, and performance. For this evaluation, consider the following steps:

Build an inventory of your core applications:

– **Business purpose:** Define the business purpose of the application from a business perspective. Understand which business processes are supported by the application and how critical each one is. – **Primary stakeholder:** Determine who the stakeholders are that you should consider in an eventual modernization project. This list reminds you how important this project is for different people, not just the IT experts. – **Technical expert:** You must clearly understand who the technical personnel are that maintain the applications. These experts know many small details about the application that can be critical in the project.

Leverage AI-Assisted Development Tools:

Modern AI-assisted development tools like IBM Bob™ can significantly accelerate your assessment process. These tools can analyze your existing programs, extract embedded business rules, and provide insights into code structure and dependencies that might take weeks to document manually. By understanding both the technical implementation and business logic within your applications, AI tools help you build a more complete inventory faster and with greater accuracy.

Evaluate each application against the following criteria:

– User interface flexibility – Maintainability – Performance – Ease of integration – Available skills – Business value

Define your modernization goals:

– Better maintainability – New user interface – Improved code readability – Better database access methods

Prioritize the inventory according to the goals and understand the scope of change

As you assess your starting point, one reality quickly becomes clear: modernization is never confined to a single layer. Each modernization effort touches multiple dimensions:

- database architecture
- user interfaces

- system integration
- business logic
- operational tooling
- development practices

These dimensions evolve together. A change in one inevitably affects the others. A modern interface still depends on stable business logic. Automation depends on testable code. Improved operations require visibility built into the application itself. Successful modernization recognizes these dependencies early. You are just starting, so do not attempt to plan the entire modernization project now. Your modernization plans change over time and are refined along the way. Find a small piece that you can start with. Focus on small pieces that over time can lead to significant things.

What Are Your Modernization Goals

There is no universal blueprint for modernization. The path you choose depends entirely on where you are starting and where you need to go. Your business constraints, technical debt, available skills, and organizational readiness all shape which approach makes sense. Depending on your modernization goals, there are several approaches that are available.

What you must keep in mind is that there is not a "one size fits all" approach to the modernization process. This section introduces some of the most common approaches in application modernization, which are easily adaptable with most modernization goals.

This section introduces the most common modernization approaches. Each has distinct trade-offs. Each serves different goals. Understanding these options helps you make informed decisions about your modernization journey.

Modernization Approaches and Strategies

Modernization Approaches

The following modernization approaches are detailed in the rest of this chapter:

- **Replacement**: Discarding the existing system entirely and building or buying something new
- **Reengineering**: Transforming the original application into a new form while preserving its core value
- **Refacing**: Modernizing the user interface without restructuring underlying business logic
- **Refactoring**: Improving internal code structure without changing external behavior

Each approach serves different modernization goals and carries different levels of risk, effort, and benefit. Review each approach carefully to determine which best fits your specific situation and objectives.

Replacement

Replacement means discarding the existing system entirely and building or buying something new. The new system might be custom-developed or a commercial off-the-shelf product configured for your needs.

There can be many risks in this approach that you should consider. For example: There is no guarantee that the new system contains the same functions or performance as the old system, which might have a long history of changes to make it customized for the business needs. This situation probably is going to cause some degradation of the service at some level. The IT personnel that are involved in maintenance of the old system need training to learn about the new system. This training is definitely not optional. New applications can change your backup and recovery methodologies and must be considered.

When replacement makes sense:

- Modernization costs exceed the value of preserving existing code
- The current application cannot meet fundamental business requirements
- Previous modernization attempts have failed

- The technology stack is obsolete beyond practical recovery

The risks are real:

- **Lost institutional knowledge**: The old system embodies years of business rules, edge cases, and customizations. These are rarely documented completely. The new system may not replicate this accumulated wisdom, leading to service degradation or missing functionality.
- **Skills gap**: Your team knows the old system intimately. They will need substantial training on the new system. This is not optional—it is a critical success factor.
- **Operational disruption**: New applications often require different backup strategies, monitoring approaches, and recovery procedures. These changes ripple through your entire operations.

Replacement is the highest-risk approach. It should be chosen deliberately, not as a default when modernization seems difficult. The use of AI-Assisted development tools makes reengineering a good alternative for modernizing your applications on IBM i.

Reengineering: Transforming What Exists

Reengineering is the modernization approach that transforms the original application into a new form to improve its original characteristics, such as functionality, performance, flexibility, UI, and maintainability with less risk than replacement. It transforms your existing application into a new form while preserving its core value. Unlike replacement, reengineering treats the existing system as an asset, not a liability. You improve functionality, performance, flexibility, and maintainability while managing risk through incremental change.

Reengineering typically follows a two-phase process:

- Reverse engineering
- Forward engineering

Reverse Engineering: Understanding What You Have

Reverse engineering extracts high-level processes and structures from existing code, creating representations that humans can understand and reason about. Given the complexity of many legacy applications and the frequent absence of documentation, this phase is challenging but essential.

Key activities:

1. **Generate structural representations** that make the code comprehensible
2. **Map code structures to business functions**, identifying reusable business rules that can be extracted and modularized
3. **Construct architectural views** that reveal how the application works as a whole

Reverse engineering is not a one-time activity. You will return to it repeatedly as your understanding deepens. Early confusion is normal. As modernization progresses, clarity emerges. *AI-assisted coding tools can dramatically accelerate reverse engineering by automatically analyzing code structure, identifying patterns, and generating documentation that would traditionally take weeks or months to create manually.*

Forward Engineering: Building What You Need

Forward engineering—sometimes called refinement—creates the new system from the high-level understanding gained through reverse engineering.

Key activities:

1. **Design the target architecture** for the modernized application
2. **Define functional transformations** from old to new
3. **Implement the new code** using modern practices, patterns, and tools

This is where you apply contemporary design principles, leverage modern development tools, and incorporate best practices. Later chapters in this book provide specific techniques, tools, and patterns to guide this work. *AI tools like IBM Bob can accelerate implementation by generating boilerplate code, suggesting modern patterns, and ensuring consistency across the codebase.*

Refacing: Modernizing the User Experience

Refacing focuses exclusively on the user interface. The goal is to improve usability and flexibility from the user's perspective without restructuring underlying business logic or data access code.

Refacing has been available for decades, with solutions varying widely in capability and sophistication. Its primary advantage is speed: you can potentially transform a 5250 green screen application into a web or mobile interface in days, not months.

Refacing is not an endpoint. It can be a valuable first step—a way to demonstrate progress quickly while planning deeper modernization. It buys time and builds momentum.

Types of Refacing

Screen Scraping: Screen scraping converts terminal-based applications into window-based, web, or mobile interfaces by emulating a user at the terminal. The scraper intercepts the 5250 data stream, extracts data, and generates a new interface. The user sees modern UI; the scraper handles translation.

This approach works even when source code is unavailable or inaccessible. It is purely external and the application itself remains unchanged.

Open Access (OA) Based Refacing: IBM Rational Open Access (OA): RPG Edition operates at a different level. Instead of intercepting the 5250 data stream, OA directly accesses the RPG program's I/O buffer. You can process data by field name or through data structures for each format.

This provides more control and flexibility than screen scraping. You can modify UI behavior directly from RPG code and pass data between your new interface and existing business logic.

> **Note:** While Open Access is mentioned here for completeness, it is not considered a strategic modernization approach. It remains tightly coupled to the RPG program structure and does not address the underlying maintainability challenges of the application. It is best viewed as a tactical option rather than a foundation for long-term modernization.

Web Technology Based Refacing: A third refacing approach involves re-developing selected screens using modern front-end technologies and open source frameworks such as React, Angular, or Vue.js. Rather than translating the 5250 data stream, developers build new UI components from scratch—guided by the existing screen layouts and field definitions. AI-assisted tools can accelerate this process by reading 5250 screen descriptions and generating corresponding front-end components automatically.

These new web-based components generally integrate with the IBM i backend through REST APIs and web services, decoupling the presentation layer from the underlying programs. This approach gives organizations access to the full ecosystem of modern web development: component libraries, responsive design, accessibility standards, and rich user interactions. Please refer to the chapter dedicated to front-end technologies on IBM i.

However, this approach alone shares a fundamental limitation with other refacing techniques: the backend programs remain unchanged. The IBM i business logic, data access patterns, and program structure stay as they are—potentially still hard to maintain and difficult to evolve. Web technology refacing modernizes the front-end experience without addressing the underlying technical debt. For organizations seeking end-to-end improvement, this approach should be seen as a starting point rather than a destination, with refactoring of the backend as the natural next step.

Benefits of Refacing

- **Rapid results**: Start and finish quickly without understanding complex business logic

- **Focused scope**: Change only the UI layer, reducing risk to stable business logic
- **Stepping stone**: Create momentum for more comprehensive modernization
- **Visible improvement**: Demonstrate progress to stakeholders quickly

The Limitations Are Significant

Refacing makes applications look modern. It does not make them modern.

The underlying code remains inflexible and difficult to maintain. Performance may suffer because the application was not designed for the new interface patterns. A green screen workflow translated directly to web or mobile often creates awkward user experiences—menu selections that should be dropdowns, sequential screens that should be single forms.

Refacing is tactical, not strategic. It addresses symptoms, not causes. Make this clear to management. Without deeper modernization, the application remains fundamentally unchanged beneath its new surface.

Refactoring: Improving Internal Structure

Refactoring restructures code to improve its internal quality without changing external behavior. Applications that interact with the refactored system continue working without modification. This is the "do no harm" principle.

The "do no harm" philosophy depends on comprehensive testing. Tests must be automated wherever possible and must cover all aspects of application behavior. Without thorough testing, refactoring becomes risky—you cannot verify that external behavior remains unchanged or that you have not introduced new defects. *AI-assisted tools excel at refactoring by automatically identifying code smells, suggesting improvements, generating comprehensive unit tests, and safely applying transformations while preserving behavior.*

When Refactoring Is the Right Choice

Choose refactoring when your goals include:

- **Reducing maintenance costs** through clearer, more maintainable code
- **Increasing code clarity** so developers can understand and modify it confidently
- **Enabling adaptation** so the application can evolve with business needs
- **Minimizing disruption** to other systems and users
- **Preserving platform and language** to leverage existing infrastructure and skills
- **Utilizing current team expertise** rather than requiring wholesale retraining
- **Enabling UI flexibility** by separating business logic from presentation, making it easier to support multiple interfaces

Refactoring as a Foundation

Some degree of refactoring is necessary in almost every modernization approach. Even when replacing or reengineering, you often need to refactor portions of the system to understand them or to create clean integration points.

The modernization model described later in this book includes activities closely aligned with refactoring principles. Evaluate which activities your specific project requires. Not every modernization needs every technique, but understanding refactoring principles benefits all approaches. *With AI-assisted coding tools like IBM Bob, refactoring becomes more systematic and less resource-intensive, making it a practical choice for organizations that previously found it too daunting or time-consuming.*

Modernization Model

This section presents the modernization model proposed in this book, introducing the principles that guide the approach and a prescriptive set of phases you can follow to achieve your modernization goals.

What is Modernization?

Modernization is a discovery process and you must learn a lot. But even so, it is important to follow a model that can guide your way. There will be more than just one modernization project ahead, so prepare to record your journey so others can repeat your success and avoid your mistakes. This section provides you with a model that can help you understand what you must do to modernize an application. There are some general principles that are applicable not only to this model but to any custom process that you define.

A Model or a Methodology

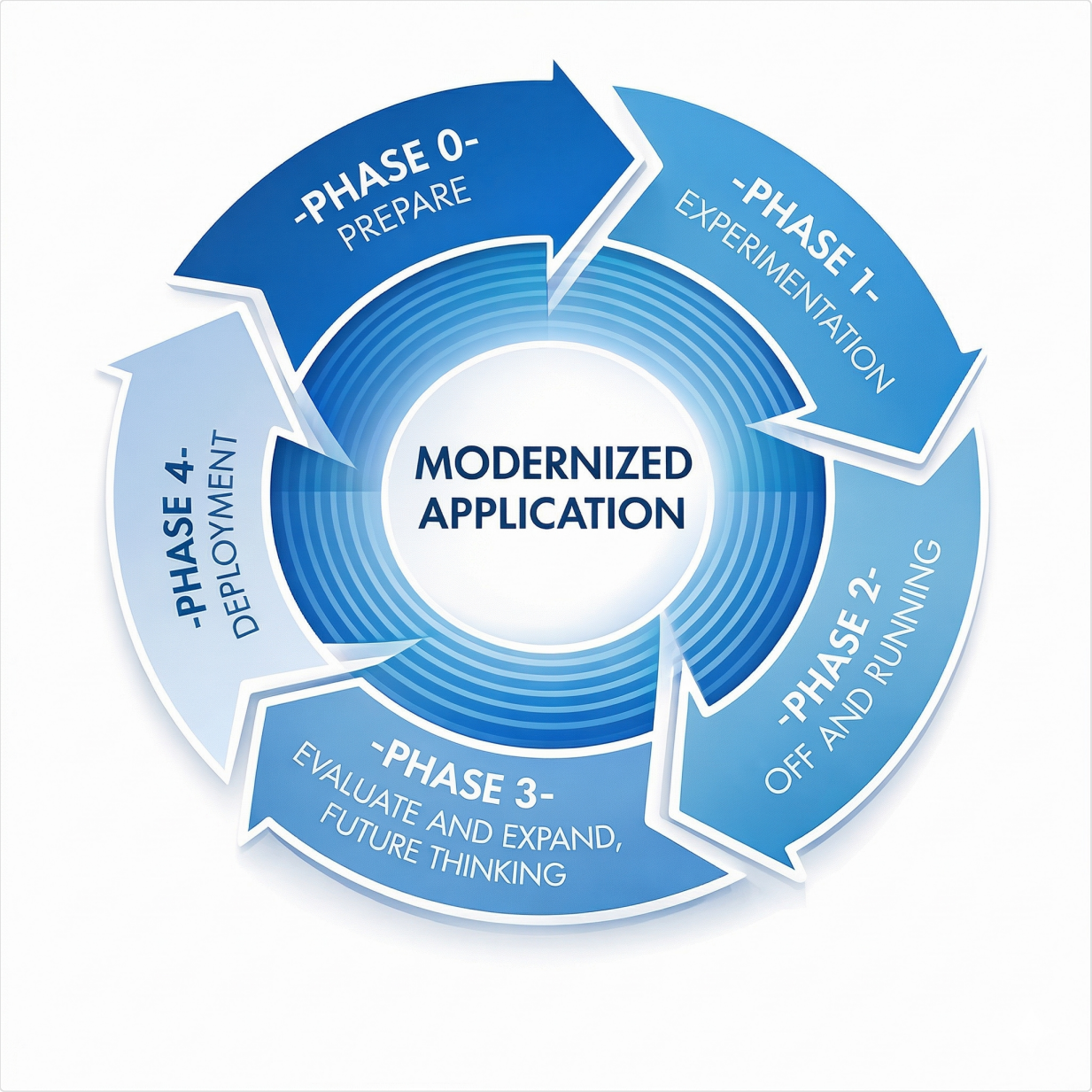
Before learning about the modernization model, it is necessary to understand the distinction between a model and a methodology. These terms are sometimes mistakenly used as synonyms. A model defines an approach—a scheme for doing things. It defines what you need to do, not how to do it. A methodology, on the other hand, defines every activity that must be performed to achieve each step of the model. A methodology can be thought of as the implementation of a model.

This book defines a modernization model rather than a methodology. Modernization can be approached in different ways, and tracking every new technique can be challenging. However, a model can be time-proof. It is your responsibility to adapt this model to your specific situation while keeping in mind the principles, guidelines, and phases that every modernization effort should follow.

The Model

Modernizing applications is more than following steps and performing activities. It requires a set of principles that you must remember. The modernization model proposed here defines this set of principles along with a set of phases to achieve your modernization goals.

The modernization model (see the following) defines an iterative modernization cycle that is repeated as many times as needed. Every iteration contributes to the main objective: building a modern application.



The Modernization Model diagram showing iterative phases: Prepare, Experimentation, Off and Running, Evaluate and Expand, and Deployment

After every iteration, you should pause and reassess the situation. You can change your modernization approach or any specific technology you selected. The key point is that you can see results early and stop as soon as your modernization goals are achieved.

The Phases

The model consists of five phases that guide the modernization process:

Phase 0: Prepare This phase addresses the planning needed for each iteration. You must select the part of the application you are going to modernize and define how you will measure your progress. Every iteration must include planning activities so you can effectively adapt to the immediate situation.

Phase 1: Experimentation You must try new things in each iteration. You must understand the technologies available in the market and determine whether they fit your situation. This phase is about experimenting with approaches, tools, and techniques before committing to them in production.

Phase 2: Off and Running This phase is about doing the real work and applying what you experimented with to achieve the goals of the iteration. Here you build your sandbox environment, understand the application through various analysis techniques, and perform the actual modernization work.

Phase 3: Evaluate and Expand, Future Thinking This phase involves analyzing the work done so far, including all previous iterations. Here you can see how to introduce new technologies to enable greater flexibility in the modernized application. This phase is dedicated to analyzing how to expand the current functions of the application and optimize its architecture.

Phase 4: Deployment You must deliver the software as soon as possible. Every cycle ends with working software. You must carefully select the deployment strategy to reduce risks, whether using a big-bang, incremental, or evolutionary approach.

The model is iterative and incremental. After completing Phase 4, you return to Phase 0 to begin the next iteration, continuously improving the application until your modernization goals are fully achieved. Not all phases require the same effort in every iteration—early iterations may require more preparation and experimentation, while later iterations can move more quickly through these phases as your understanding deepens.

What is responsive design and when to use it?

Responsive design is a web development approach that enables user interfaces to adapt automatically to different screen sizes and devices—from desktop monitors to tablets and smartphones. Rather than building separate web interfaces for each device type, responsive design uses flexible layouts, fluid grids, and CSS media queries to adjust content presentation based on the viewport. The goal is to provide usable and accessible web interfaces across a range of devices while maintaining a single codebase.

Responsive design is most appropriate when modernized IBM i applications are delivered through browser-based interfaces that must support users across multiple device types. This commonly occurs when refacing legacy 5250 interfaces with modern web applications or when building new front-ends using frameworks such as React or Angular. In these cases, responsive design allows business users to access operational screens from desktop workstations, field personnel to use tablets for data entry, and managers to review dashboards from mobile devices without requiring separate web implementations for each form factor.

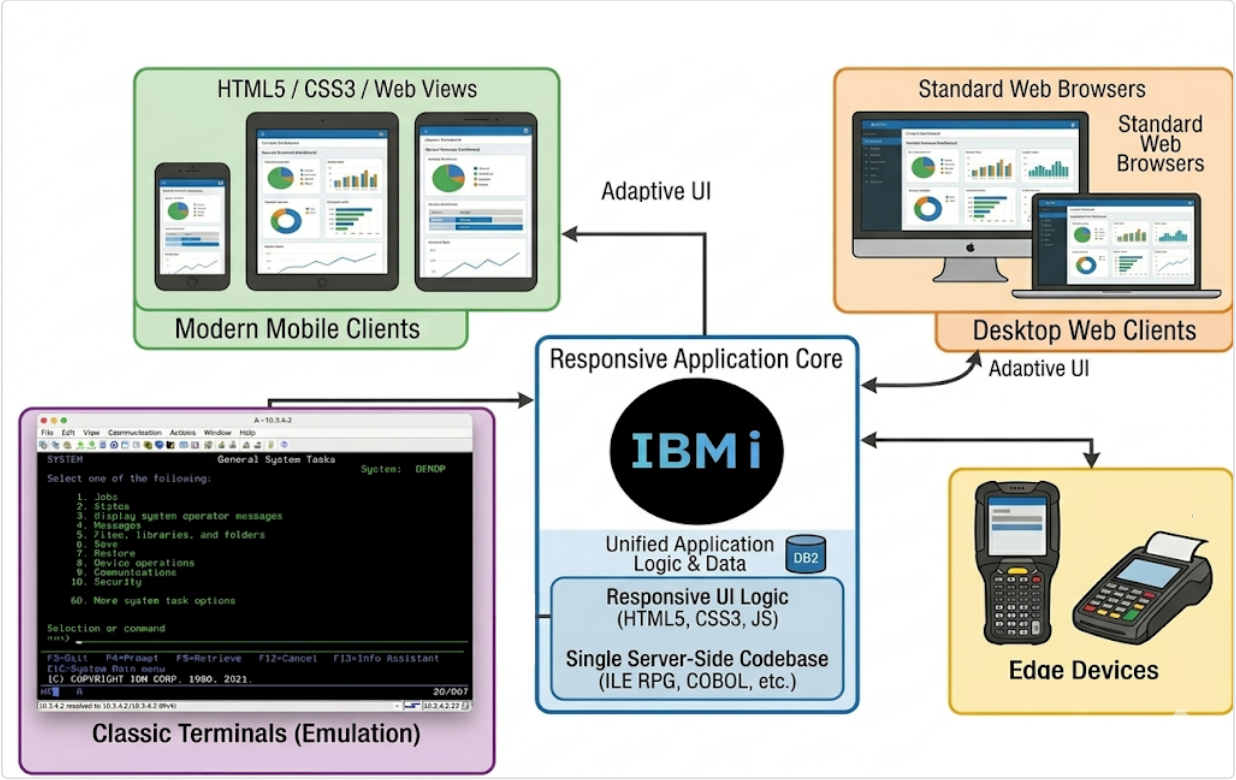


Figure: Responsive design illustration showing how a single web interface adapts to desktop, tablet, and mobile screen sizes

In IBM i modernization projects, responsive design applies primarily to the presentation layer of applications that consume services exposed by the platform. Business logic implemented in programs or other components may be exposed through APIs, integration services, or messaging systems, which can then be consumed by different client applications. When a web application is one of those clients, responsive design can simplify front-end development by enabling a single interface to adapt to desktop and mobile browsers. While this approach can reduce the need to maintain separate web interfaces and may lower development and maintenance overhead, it does not replace all client strategies. Some use cases—such as specialized mobile workflows or offline capabilities—may still require dedicated mobile applications.

IBM i also supports responsive data retrieval through SQL pagination using the OFFSET and LIMIT (FETCH) clauses, allowing applications to adapt result set sizes based on device capabilities. This enables efficient data handling where mobile devices can request smaller page sizes while desktop applications retrieve larger datasets, optimizing both network usage and user experience across different form factors.

This section provides a quick introduction to responsive design. Throughout this book, you will explore the technologies and architectural practices required to implement responsive design on the IBM i platform.

General Principles for Modernization

Successful modernization requires adherence to fundamental principles that guide decision-making and ensure sustainable progress. These principles apply regardless of your chosen modernization approach.

Iterative and Incremental Process

Modernization is both iterative and incremental. Break the entire project into smaller, manageable iterations that are easier to complete and faster to deliver. Each iteration makes the application more modern, one component at a time. This agile approach prevents the overwhelm and frustration that often accompany "big-bang" attempts. Remember: divide and conquer.

Continuous Learning and Research

Technology evolves rapidly. What is cutting-edge today may be obsolete in months. Make it a habit to read and research modernization theory and technologies. Attend seminars and conferences. Talk to colleagues about their successes and failures with modernization approaches and technologies. Modernization affects many companies, platforms, and languages—you are not alone. Find time to experiment with technologies and create proof of concepts to validate them in your environment.

Automation

When you find yourself repeating the same task, automate it. There are always better ways to accomplish repetitive work. Look for commercial modernization tools or create your own automation scripts. Automation reduces errors, saves time, and allows your team to focus on higher-value activities. For detailed guidance on automation and CI/CD practices, see DevOps Practices for IBM i.

Testing

Testing is the only way to ensure you are not breaking existing functionality. Implement comprehensive testing strategies:

Regression tests compare the modernized application's results against the original baseline. These tests do not require deep system understanding, making them valuable when starting modernization with incomplete code comprehension.

Automated unit tests validate small application components. Unlike regression tests, unit tests require more code understanding but provide faster feedback. Design unit tests to be self-configuring and fast-running—you should be able to execute them after every compilation.

These approaches complement each other. Automate your tests and run them frequently. This is the only way to feel confident while changing code.

Avoid Overanalysis

Modernization is about experimentation. Do not let fear derail your efforts. Start working as soon as you can. Legacy code will surprise you occasionally, no matter how much you analyze it. Remember the iterative and incremental nature of the process—plan and analyze in the same way.

Agile Principles

Modernization projects must be agile. Focus on avoiding activities and documentation that do not add value:

Deliver working software frequently. Plan iterations to deliver working software fast and frequently. Your executives must believe in the value of modernization, so deliver value early. Break work into manageable iterations —typically two weeks—and deliver progress continually. However, do not obsess over breaking things into the smallest possible units. The purpose is flexibility and incremental updates.

Focus on software over documentation. Avoid documentation that does not add value. Look for dynamic documentation that can be easily maintained and reused. Content management systems enable collaborative documentation where the user community can contribute.

Simplicity is essential. Keep things as simple as possible. Resist the temptation to add features that might be used in the future but are not needed immediately. Unnecessary complexity introduces bugs that can damage your process.

Give continuous attention to technical excellence. Modernization is about improvement. Focus on applying better practices and technologies. Become an advocate of excellence. Promote best practices and transmit your knowledge. Help others understand the importance of doing things correctly.

Business people and developers must work together. Build good communication lines between developers and business experts. The application you are modernizing supports the business. Do not underestimate the contributions that non-IT personnel can make to the project.

Use Modern Tools and Standards

Modern tools are key to success

Stop using obsolete tools! Modern tools help you accomplish tasks faster and better. While you may be more productive with familiar tools, do not mistake familiarity for superiority. Changing tools requires an adjustment period, but modern tools ultimately enable greater productivity. Attracting new talent requires tools that resemble what developers use elsewhere. "Green screen" development tools give the impression of obsolete technology. Many developers see 20-50% productivity improvements by switching from green-screen development environments to modern integrated development environments (IDEs).

Development and IDEs: VS Code for i extensions, RDi (Eclipse-based IDE) with IBM and third-party features/ extensions for accelerating your development and modernization efforts. **Database and Management**: SQL and database tooling, Access Client Solutions (ACS) for system administration. **DevOps and Lifecycle Management**: Testing frameworks like RPG-Unit, Git, Jenkins or other DevOps pipeline for CI/CD automation for ILE and Open source developments can accelerate your modernization efforts.

You'll find more details in the following chapters.

AI-Assisted Development Partners

We could not cover modern tooling without mentioning AI-assisted development partners! AI-assisted development tools can significantly enhance productivity during IBM i modernization. Tools like **IBM Bob** can further boost productivity by 20-30% by helping you understand legacy code, extract business rules, generate documentation, and accelerate refactoring efforts. These tools analyze your existing applications to reveal hidden dependencies, suggest improvements, and guide modernization decisions with data-driven insights. For comprehensive coverage of AI-assisted modernization techniques and tools, see AI and IBM Bob for IBM i Development.

Adopting modern tools—both IDEs and AI-assisted development platforms—is a critical step in modernizing your IBM i applications.

Leverage Existing Standards

To conclude this general principles section: do not reinvent the wheel! If a good standard exists in the industry, analyze it. If it fits your needs, adopt it. You are not the only one modernizing applications. Many solutions you need have already been researched and implemented by others. Learn from the broader community and apply proven practices to your IBM i modernization efforts.

Implementation Phases

How do I prepare for modernization? (Phase 0)

Applications evolve into complex ecosystems. Over time, business processes, organizational structures, and integrated systems form dependencies around them. Modernization extends beyond code—it requires comprehensive preparation across technical, organizational, and business dimensions.

Phase 0 establishes the foundation for successful modernization. The objective is to gain a clear, comprehensive understanding of what the application means to all stakeholders and to secure the organizational commitment needed to sustain the effort.

Building the Business Case

Before launching a modernization project, you need executive support and funding. A well-constructed business case translates technical needs into business value, enabling decision-makers to evaluate the project on cost-effectiveness and strategic alignment.

A complete business case includes:

Problem Statement

Document the current application's inadequacies, inefficiencies, and limitations. Use quantifiable metrics—system downtime, maintenance costs, time-to-market delays, user productivity losses—to illustrate the business impact. Include metrics related to deployment frequency, lead time for changes, and mean time to recovery (MTTR) to highlight the need for modern DevOps practices. Decision-makers need concrete data to understand why modernization is necessary now.

Solution

This is the high-level description of the solution for the problem. You should include the following items: – Current state-of-the-art technologies. – An overview of the proposed solution. – A preliminary cost and schedule proposal. You can also include a brief description of the modernization model that is going to guide the project.

Risks

Conduct an honest risk assessment. Identifying potential obstacles early builds credibility and enables proactive mitigation planning. Common risks include:

- Incomplete documentation or loss of institutional knowledge
- Resistance to change from users or IT staff
- Integration challenges with dependent systems
- Resource constraints (budget, personnel, time)
- Technology learning curves
- Lack of automated testing infrastructure, increasing regression risk
- Insufficient DevOps tooling and practices, slowing deployment velocity

Benefits

Quantify expected benefits across multiple scenarios:

- **Pessimistic scenario**: Minimum expected benefits if risks materialize. Show how mitigation strategies ensure value even under adverse conditions.
- **Optimistic scenario**: Maximum potential benefits if the project proceeds smoothly. Keep projections realistic to maintain credibility.

Express benefits in business terms: reduced maintenance costs, faster feature delivery, improved user productivity, enhanced security posture, reduced technical risk. Highlight DevOps benefits such as faster deployment cycles, reduced deployment failures, and improved mean time to recovery.

Metrics

You must define how you are going to measure your progress. Each iteration should be evaluated against these metrics. This is not only something that is important to your manager; it also helps you adjust your plan. Regular measurement against these metrics enables course correction and demonstrates progress to stakeholders.

Stakeholders

Identify everyone affected by or contributing to the modernization:

- **Executive sponsors**: Provide funding and organizational support
- **IT leadership**: Allocate resources and align with IT strategy
- **Development team**: Execute the technical work

- **DevOps team**: Establish CI/CD pipelines, automation infrastructure, and deployment processes
- **Testing/QA team**: Define testing strategy, create automated tests, and validate functionality
- **Business users**: Provide domain expertise and validate functionality
- **Operations team**: Support deployment and production operations
- **Security team**: Ensure compliance and security requirements are met

In small projects or organizations, some of these roles may overlap or be absent. Adjust the list based on your specific context.

Proof of Concept

If possible, demonstrate a small-scale success before requesting full project funding. A proof of concept validates the chosen modernization approach, builds confidence among stakeholders, identifies unforeseen challenges early.

The business case is essential for the modernization project. Without it, it is doubtful that you can obtain the funding that is required and keep it throughout the project's life. It is vital that the business case is complete. Otherwise, your modernization effort can be hampered even before you change a single line of code.

Building the Team

Modernization requires diverse skills and perspectives. Assemble a cross-functional team early and define clear roles and responsibilities. Modern IBM i development demands collaboration between traditional IBM i developers, DevOps engineers, and testing specialists. Besides the developers that do the actual work, the modernization team is also composed of non-IT personnel (for example, the users). Do not underestimate their contributions or commitment to the project.

Core Roles

Modernization Team Lead

The team lead directs the modernization effort and serves as the primary point of contact for stakeholders. This person must be a modernization advocate who understands both legacy and modern technologies. It is a pragmatic and open-minded professional willing to adapt the approach as needed. This person must be skilled at managing change and addressing resistance, and must be passionate about continuous improvement.

Modernization Engineers

These developers execute the technical modernization work. Select engineers who:

- Have experience with IBM i technologies and modern development practices
- Understand the importance of iterative progress over attempting to fix everything at once
- Are willing to learn new technologies and approaches
- Write testable code and embrace test-driven development (TDD) practices

DevOps Engineer

Operations rely more and more on DevOps engineers who establish the automation infrastructure that enables rapid, reliable deployments. Ideally, a modernization project should include a person who designs and implements CI/CD pipelines for IBM i applications, automates build, test, and deployment processes. A DevOps engineer also knows how to configure monitoring and observability tools, create and maintain automated test suites (QA Testing), bridging the gap between development and operations teams.

Business Logic Expert

The business logic expert is usually a non-IT person. They can be final users or someone whose business process is supported by the application. This role is essential for the project. They can give you guidance about understanding the reasons behind specific functionalities of the application.

Application Maintenance Personnel

The current maintainers possess invaluable knowledge about the application's history, design decisions, and undocumented behaviors. They:

- Explain why the code is structured as it is
- Identify areas of technical debt and known issues
- Understand dependencies and integration points
- Can validate that modernized code preserves critical functionality
- Help create regression tests based on known edge cases

Important: Maintenance personnel may feel threatened by modernization. Address concerns directly, emphasize their value to the project, and involve them as partners in the effort.

Developing the Plan

Do not try to anticipate every little detail. Things most likely will change, so define a high-level roadmap of the project. Then, do a detailed plan for only one or two iterations. Use the roadmap as you perform your plan to keep your project on track.

This plan should define your immediate objectives for the iteration and the tasks that are needed to accomplish them. Make sure that your plan includes a deliverable at the end of the iteration.

Developing Standards and Processes

At first, you experiment frequently with the modernization process, but remember to document what does and does not work for you. During this process, you must define standards to ensure that the modernization team does things in a predictable way. As soon as possible, define a set of rules and standards that aim for quality. Be prepared to modify your standards as you and your team find improved techniques.

Key Standards to Define

- **Coding standards**: Naming conventions, formatting, documentation requirements
- **Testing standards**: Unit test coverage requirements, integration testing approach, test naming conventions
- **Code review process**: Who reviews, what to look for, approval criteria
- **Version control practices**: Branching strategy, commit message format, pull request workflow
- **Documentation requirements**: What to document, where to store it, how to keep it current
- **CI/CD pipeline standards**: Build triggers, automated test execution, deployment approval gates
- **Deployment procedures**: Automated deployment scripts, rollback procedures, deployment validation
- **Monitoring and observability**: What to monitor, alerting thresholds, logging standards

Leverage Existing Standards

Don't reinvent the wheel. If there is a good standard in the industry, analyze it. If it fits your needs, adopt it. You are not the only one trying to modernize applications. Many things that you need already are researched and implemented by someone else.

Phase 1: Experimentation

Experimentation is where modernization theory meets reality! This phase validates your assumptions, tests new technologies, and builds the knowledge foundation for successful implementation. The objective of this phase is to reduce risk by identifying obstacles early, when they are cheapest to address.

Proof of Concept Development

Validate technologies through working prototypes, not vendor presentations. Create proof of concepts (PoCs) that demonstrate feasibility in your specific environment:

- **Technology validation**: Test modern frameworks (Node.js, Python, Java) alongside traditional IBM i technologies
- **Integration testing**: Verify that new components can consume IBM i services (REST APIs, database access, message queues)
- **Performance baseline**: Measure response times, throughput, batch processing time, and resource utilization under realistic load
- **Developer experience**: Assess tooling, debugging capabilities, and learning curve

If evaluating commercial solutions, leverage free trials to build working prototypes. Marketing materials promise capabilities; PoCs prove them. Document findings: what works, what doesn't, and why, to inform future decisions.

Involve stakeholders (see Phase 0 : Prepare) in PoC reviews. Their feedback ensures you are solving the right problems and building confidence in the modernization approach.

Start with Simpler Applications

While learning the modernization process, avoid the most complex or critical applications. Select applications that:

- Deliver business value
- Have manageable scope (hundreds of programs, not thousands)
- Possess adequate documentation or knowledgeable maintainers. This is where AI-assisted tools like IBM Bob can help.
- Represent common patterns you will encounter in larger applications

Experiment Freely

Change code, redesign UIs, refactor architecture—without fear of breaking production. The ideas and experiences that you gain in this activity help you work better on the new application and to understand the underlying code.

- **Try multiple approaches**: Compare REST APIs vs. message queues for integration; evaluate React vs. Angular for UI
- **Test automation tools**: Experiment with CI/CD pipelines, automated testing frameworks, and deployment automation
- **Explore modern patterns**: Microservices, event-driven architecture, hybrid with containerization: understand their fit for IBM i workloads
- **Validate AI-assisted development**: Test code generation, refactoring suggestions, and documentation tools

Reassess Continuously

Experimentation broadens perspective and challenges assumptions. Be prepared to adjust your modernization approach based on what you learn: technology choice, architecture decisions, tooling, and scope. Flexibility is strength. Do not be afraid to reassess your modernization approach if you think that there is a better way to do it.

Define Measurement and Success Criteria

As you gain application knowledge, establish metrics to track progress:

- **Code quality**: Cyclomatic complexity, test coverage, technical debt reduction
- **Performance**: Response time improvements, resource utilization, throughput gains
- **Business value**: Time-to-market reduction, defect rate improvements, user satisfaction etc.

These metrics provide objective progress indicators for stakeholders.

Phase 1 transforms uncertainty into knowledge. The insights gained here prevent costly mistakes during implementation and deployment.

How do I scale modernization efforts? (Phase 2-3)

After experimentation validates your approach, Phase 2 executes the modernization work while Phase 3 evaluates progress and optimizes architecture. AI-assisted tools like IBM Bob and other LLM-based agentic solutions can accelerate these phases by automating code analysis, generating documentation, suggesting refactoring patterns, and identifying optimization opportunities. Agentic AI can now optimize and refactor code, convert applications between languages and architectures, generate unit tests, and create deployment scripts—significantly reducing manual effort in these steps. The key is to balance automation with human oversight, ensuring that AI suggestions are validated and that critical business logic remains intact.

Phase 2: Implementation

This phase focuses on doing real work. Now that you have a panoramic view of the available technologies and techniques, start modernizing the application. Keep in mind the iterative and incremental nature of the modernization process, so start step-by-step in a way that makes sense for your situation. Divide the application into small units that you can modernize one at a time, but do not lose the whole view of the application. The following sections describe some of the activities that you should consider in this phase.

Build Your Sandbox Environment

Establish an isolated development environment with complete application objects, test data with automated refresh capabilities, and comprehensive test suites. Integrate CI/CD pipelines to automate builds, testing, and deployment. Always journal development data for debugging. This infrastructure becomes the foundation for sustainable modernization.

- **Objects**: Copy all of the application objects to your sandbox, including, at the least, data areas, programs, service programs, user spaces, data queues, and any other object that your application is going to need to run in an isolated environment. To help you build the sandbox, create the application structure diagrams, which can be generated with automated tools.
- **Data**: In addition to the executable objects, make sure to have all of the tables that interact with your application. Isolate the data to allow changes to be made without affecting other projects. You should develop a method for restoring your sandbox data to its initial state in an automated fashion, which helps you run repeated test scenarios and automate testing. Always journal your sandbox data. Journals are indispensable when debugging your changes.
- **Tests**: You need a complete set of tests to ensure that your modernization work is going in the planned direction. Your sandbox should contain a test set that you can run multiple times and easily. Without tests, any change can be unrecoverable. In the first iterations, you must have a set of regression tests. As the project advances, your understanding of the application increases, enabling you to write specific unit tests.

Understand the Application

Gather existing documentation

Gather documentation, even if incomplete. Read code collaboratively to identify patterns and business rules. It is likely that there is no documentation of the original application. Do your best to compile every piece of documentation that is available, and use modern tools to generate missing documentation or dependency maps.

AI-assisted tools like IBM Bob work hand in hand with third-party application discovery tools to provide comprehensive application understanding. While specialized discovery tools create detailed blueprints and dependency maps of complex applications, agentic AI tools analyze code to extract business rules, generate documentation, and suggest refactoring opportunities. This combination accelerates understanding by providing both architectural views and deep code-level insights.

Static Analysis

This is the analysis of the source code or the compiled object without running it. Usually, the static analysis has the following components:

- **Usage of code analysis tools**: There are several tool providers that have specialized tools to review and help you analyze your monolithic programs. If you are looking for specialized tools for code analysis, there are many ISV's that have great specialized tools as well as AI based support such as IBM Bob that can help with this.
- **Extraction of business rules**: Business rules refer to the business logic that is embedded in the code. Many tools help you extract this type of data from the code, which can help you understand the functionality that is implemented in the program.
- **Dependency analysis between components**: Refers to the identification of relationships between the modules of the application. The diagram that is generated can help you understand how one component integrates with the whole application. Deterministic rule-based tools from software vendors, complemented by generative AI solutions are very good at identifying such relationships.
- **Internal structure diagram**: Most legacy applications are large, monolithic programs. To easily understand the flow inside these programs, some tools can generate call diagrams between internal subroutines and subprocedures.
- **Complexity and maintainability metrics**: Complexity and maintainability metrics can help you diagnose the section of code on which you are working and measure your progress. They can also help you estimate the amount of work that is needed to modernize.

Dynamic Analysis

This type of analysis is performed by tracing the running of the programs in a specific environment. There are many uses of the results of the dynamic analysis:

- **Code coverage**: Traditional applications usually contain "dead code". This is code that is not run but is still in the program. To identify code that is no longer needed, you can use a tool that helps you calculate how much code is covered with a set of transactions and how much code is possibly dead.
- **Dynamic call diagrams**: The preferred way to understand how the program works is by monitoring its behavior while running. Dynamic analysis tools usually provide data that depicts the execution flow of a program, subroutines, and subprocedures.

To be effective, the dynamic analysis requires a good set of tests to produce an interesting result. Again, AI-assisted tools like IBM Bob can accelerate this process by analyzing code coverage reports and automatically generating test cases to exercise uncovered code paths.

Modernize the Code

When your sandbox is prepared and you have a better understanding of the application, you can start doing modernization work:

Cleaning the code: Remove elements that make code hard to understand—eliminate duplicated code, convert to more readable forms (RPG free format, modern SQL), and remove unused variables and procedures.

Restructuring the code: Extract related business logic into service programs that can be reused. Encapsulate global variables, convert subroutines to procedures, rename variables and subprocedures to meaningful names, and write self-documenting code.

Repackaging: Organize code by single responsibility. Every module must contain only functionally related subprocedures, and every service program must contain only functionally related modules. Separate monolithic modules into focused components.

Modernizing the database: Apply modern database practices (covered in Chapter 9)—use SQL tables instead of physical files, add constraints and triggers, implement proper indexing strategies.

API development: Expose business logic as REST or SOAP services using IWS, Liberty, or open source frameworks (Node.js, Python).

Test Continuously

Run your complete test suite after each change. If tests fail, stop and fix before proceeding. Automated testing in CI/CD pipelines provides immediate feedback, ensuring you are not introducing regressions.

Phase 3: Evaluate and Expand

The main objective of this phase is to maximize the benefits of current state-of-the-art technologies applied to your application. You have prepared your environment and modernized the application. Now evaluate your progress. If evaluation indicates you should step back, do not worry—you might repeat this step multiple times. If you have accomplished your goals, start expanding the application's value and prepare it for the future.

Measuring

Determine whether you are achieving modernization objectives. If your main goal is to reduce complexity, measurements must reflect complexity reduction (cyclomatic complexity, code duplication metrics). If improving user experience, measure it (response time, user satisfaction scores). Keep control of objectives and progress toward goals. Your measurements should indicate adjustments needed for the project.

Improving the Architecture

Consider optimizing the application architecture. Separate the application into layers or tiers (presentation, business logic, data access) and include new components such as a UI replacement layer. This step focuses on the application as a whole.

Integrating

A common modernization goal is integrating the application into enterprise architecture. Enable the application for modern integration patterns—expose REST APIs using IWS or Liberty, integrate with enterprise messaging systems (Kafka, MQ, MQTT), provide JDBC/ODBC access for external systems.

Bringing in New Technologies

If new technology can improve the application, include it carefully to avoid future obstacles. Always aim for standard tools, technologies, and well-tested solutions (React, Angular for UI; Node.js, Python for services). Evaluate the user community and official support. New technologies can become obsolete, so wrap them in abstraction layers that enable easy replacement if necessary.

Prepare for Continuous Modernization

Modernization is ongoing practice. Establish DevOps culture on IBM i as on every other platform! Automate everything, measure everything, improve continuously. Maintain test coverage as the application evolves. Monitor and address technical debt before it accumulates. Keep dependencies, frameworks, and tools current. Document architectural decisions for future reference.

How do I deploy modernized applications? (Phase 4)

It's time to deliver the results of your work. However, delivered components should not be considered final products. With each iteration, you can improve software even after delivery. The key is to deploy as soon as possible to

demonstrate the value of your modernization effort. **AI-assisted tools** can generate deployment scripts, automate documentation creation, and streamline the deployment process.

Documenting the Application

The modernized application is likely different from the original. Create effective documentation that explains the new application to maintenance personnel, users, and stakeholders.

Take advantage of available resources. Use diagrams (class, sequence, deployment) to help people understand the application. While UML diagrams are designed for object-oriented languages, they adapt easily to IBM i applications. Standard diagrams make you appear more professional and are widely understood.

Your documentation must add value—avoid documentation that goes directly to the shelf. Modern documentation platforms (Confluence, Notion, GitBook, GitHub/GitLab wikis, ReadTheDocs) help create simple, collaborative documentation. Documentation must be easily maintained. Many tools can generate documentation from code or compiled objects (Swagger/OpenAPI for APIs, JSDoc for JavaScript, Javadoc for Java, automated documentation generators). Use them or create them if possible. AI tools can now generate comprehensive documentation from code, including architecture diagrams, API references, and user guides.

Training the People

Depending on your modernization approach, you might need to train users, application maintenance personnel, or both.

Use the documentation you built as the primary reference for trainees. Ensure documentation is easily accessible and contains all necessary information for users and IT experts.

If the new application is more difficult to understand than the original, this indicates the modernization plan needs improvement. Evaluate the new system from different perspectives.

Defining Your Deployment Strategy

Just as there are many modernization approaches, there are multiple ways to deploy the new application. Ensure your deployment strategy aligns with your modernization approach:

Big-bang approach: Replaces the entire application at once. Used when modernization projects must solve immediate problems affecting critical services. However, associated risk is higher and must be carefully managed. When following this approach, maintain the old and new applications in parallel. Changes in the new system must be reflected in the old one and vice versa, leading to significant work and effort.

Incremental approach: Also known as "phased-out," sections of the application are re-engineered and added incrementally, allowing you to deliver results and identify errors more quickly. You change only one component at a time without considering the application as a whole. The modernization process might take longer to finish, but risk is lower.

Evolutionary approach: Similar to incremental, but sections to replace are selected based on functions rather than structure. Developers must identify functional parts that are separated in the original application. This requires extra effort but delivers business value more directly.

Deployment Approach Comparison

Component	Big-bang	Incremental	Evolutionary
Replacement	All at once	Structure	Functionality
Overhead	Low	High	High
Risk	High	Low	Medium
Time to deploy	Fast	Slow	Slow

Modern deployment practices: Leverage CI/CD pipelines to automate deployment. Use blue-green deployments or canary releases to minimize risk when it makes sense. Implement automated rollback procedures. Monitor deployments with observability tools (application performance monitoring, log aggregation, error tracking), these technologies are available on IBM i!

Repeat as Necessary

After finishing the process, repeat it as many times as needed. In every iteration, a new component is modernized and you move closer to your goals.

Not all phases require the same effort in every iteration. Initially, invest more effort in preparation and experimentation phases. As the project advances, these phases become shorter. The phases adapt during successive iterations.

Evaluate your progress in every iteration and adjust your path accordingly. While going through all phases, adhere to modernization principles and adjust the process to your situation.

Security and Quality Assurance

How do I integrate DevSecOps?

DevSecOps integrates security throughout the development and operations lifecycle. For IBM i modernization projects, this means embedding security into every phase while leveraging IBM i's inherent security capabilities.

Integrating DevSecOps across the application lifecycle requires organizational commitment and process changes:

- **Development**: Establish secure coding standards, include security in code reviews, automate vulnerability scanning on your code (RPG, COBOL, Java, open source).
- **Build and Test**: Integrate automated security testing into CI/CD pipelines—static analysis, dependency scanning, configuration validation.
- **Deployment**: Automatically verify security configurations post-deployment. Maintain automated rollback capability for security issues.
- **Operations**: Monitor for security events and establish incident response procedures.Track security metrics to drive continuous improvement.

For detailed implementation guidance of DevSecOps on IBM i.

What testing considerations should I make when considering path to modernize?

When planning a **modernization of IBM i applications**, testing becomes a **critical factor**—modernization often changes the **architecture, interfaces, or deployment**, which can introduce subtle defects. Proper testing ensures **functionality, performance, and business continuity** are preserved.

Here’s a structured approach to the key considerations:

1. Understand the Scope of Modernization

Modernization can involve:

- Moving from **RPG/CL to service programs or APIs**
- Introducing **web or mobile interfaces**
- Refactoring code for **modular or cloud-ready architecture**
- Updating **databases or moving to SQL-centric logic**

Each type of change has testing implications. Ask:

- Which modules are changing?
- Are interfaces being altered?
- Will batch jobs behave differently?

2. Test Implementation

Refer to chapter **Test Driven Development with Automation / How do I implement tests?** and emphasis particularly on the **New silent test cases** topic.

3. Preserve Business Logic (Regression Testing)

- Most IBM i systems contain **years of tested business rules**.
- Regression testing ensures that **modernized code produces identical results**.

Consider:

- Automated regression tests for RPG programs, batch jobs, and CL scripts
- Database validation tests for **Db2 for i** operations
- End-to-end testing for workflows spanning multiple programs

Tip: Tools might be highly valuable here. (Ref. chapter **Test Driven Development with Automation / What testing tools work with IBM i?**)

4. Integration Testing

Modernization often introduces:

- APIs, web services, or new front-end clients
- External system connections

Testing should verify:

- New interfaces correctly call existing logic
- Data flows properly between old and new modules
- Transactions remain consistent across services

5. Unit Testing for Refactored Modules

If you refactor RPG or CL programs into **modular service programs**:

- Introduce unit tests for each module
- Automate tests to verify **inputs/outputs**
- Ensure test coverage of **all code paths**

This is essential to **detect subtle logic changes** during refactoring.

6. Performance & Workload Testing

Modernization can affect performance:

- Web or API layers may introduce **latency**

- Database schema changes can affect **query performance**
- Batch jobs may need tuning in new environments

Include:

- Workload tests simulating expected traffic
- Stress tests for peak scenarios
- Monitoring tools to track response times and resource usage

7. End-to-End Testing

Modernized applications often change the **user experience**:

- Green-screen → web interfaces
- Batch output → dashboards or reports

Ensure E2E tests validate:

- Complete workflows from user input to database updates
- Reporting and output remain accurate
- All integrated systems behave as expected

8. Data Migration Testing

Modernization may involve **moving or transforming data**:

- Validate that **all records migrate correctly**
- Verify **historical reports and business logic** still produce correct results
- Test edge cases such as **nulls, special characters, or large datasets**

9. Security & Compliance Testing

Modernization may expose systems in new ways:

- APIs accessible externally
- Web or mobile front-ends

Test for:

- Authentication and authorization
- Input validation and injection vulnerabilities
- Data privacy compliance (GDPR, HIPAA, etc.)

10. Automation Considerations

- Automate regression, integration, and end-to-end tests wherever possible
- Use CI/CD pipelines to run tests on each build
- Ensure automated tests cover **both legacy and modernized code paths**

11. Continuous Monitoring Post-Modernization

Even after modernization:

- Monitor production for anomalies
- Track performance metrics
- Validate that **business rules continue to operate correctly**

Modernization is rarely a one-off; continuous testing ensures reliability over time.

Summary

When modernizing IBM i applications, testing should:

1. Preserve existing **business logic** with regression tests
2. Verify **integration points** and data flows
3. Add **unit tests** for refactored modules
4. Perform **performance, workload, and stress testing**
5. Validate **end-to-end business workflows**
6. Ensure **data migration is accurate**
7. Check **security and compliance**
8. Automate tests wherever possible
9. Monitor the system **post-modernization**

IBM i Architecture Modernization

Foundational Concepts

What are symptoms of poor architecture?

Often, a company does not even realize that their existing applications have poor architecture; they just know that it is hard to make changes and add new functionality. There are several symptoms that can indicate that your application might need to be modernized:

- **Tight coupling**: When components of an application are tightly coupled, it means that they are highly dependent on each other. This can make it difficult to change one component without affecting others, leading to a fragile codebase.
- **Lack of modularity**: When an application lacks modularity, it means that it is not organized into separate, independent modules. This can make it difficult to understand and maintain the code, as well as to reuse components.
- **Poor separation of code**: In monolithic code, business logic is often mixed with database access and user interface code. This can make it difficult to understand and maintain the code, as well as to reuse components.

Legacy code and technical debt

From an IBM i perspective, many (if not most) home-grown applications have code in them that was written 20 or 30 years ago; the tools and designs of that time (RPG OPM, flat files, record-level access, display files, etc.) were simply not designed for modern application development, and lacked the basic building blocks like modules, procedures and SQL access, that modern ILE application design allows.

It is a testament to the power of the IBM i platform that code written in the 90s continues to run mission-critical applications across the world, but the unfortunate reality is that the same "don't fix it if it ain't broken" attitude has also created an enormous technical debt that modernization seeks to overcome.

Designing for today and tomorrow

Beyond the architectural limitations of the original code, the design of the entire system is typically rooted in the traditional "green screen" 5250 interface; modern applications demand modern, web, mobile, AI and other interfaces that did not even exist when the application was first designed.

The whole mindset of modern application design revolves around a complete mind-shift of design to data-centric design and loosely coupled components that are "interface-agnostic", not only to the needs of today, but open to new interfaces and ideas that we might not even be able to imagine.

Modularization and Business Objects

Why Is Modularization Important?

What This Covers

Modularization is a foundational architectural principle on any software project written in any programming language, but there are unique considerations when applied to modern IBM i application design. This section will help you learn what modularization means, why it matters, and how it enables maintainability, testing, scalability, and modern integration patterns and how to apply those principles to your existing IBM i applications and systems.

What It Is

Modularization is the practice of organizing an application into small, cohesive, and loosely coupled units. Instead of building large, monolithic programs that combine user interface, business rules, and data access, modularization separates these responsibilities into clearly defined components.

On IBM i, modularization typically includes:

- Splitting monolithic programs into *procedures*
- Placing reusable logic in *service programs*
- Exposing functionality through well-defined *interfaces* (procedures, APIs, or services)
- Decoupling layers such as UI, business logic, and data access

In simple terms: **modularization breaks a large system into understandable parts**, each responsible for one thing, and provides predictable ways for those parts to interact.

Why It Matters

Modularization directly impacts the long-term health and adaptability of an IBM i application. Its importance grows as organizations pursue modernization, cloud integration, automation, and continuous delivery.

1. Maintainability and Reduced Risk

Large monolithic programs are hard to change because everything is interconnected. A small update in one area may have unintended consequences in another. Modular code reduces this risk by limiting the impact of a change; when components are isolated, developers can safely enhance or replace a module without destabilizing the entire system.

2. Improved Testability

Testing a monolithic program is difficult because logic is interwoven. Modularization enables:

- Unit testing of individual procedures
- Automated testing with frameworks
- Faster detection of defects
- Simplified regression testing When business rules live in discrete procedures or service programs, they can be validated independently from the UI or database access.

It is possible to write external programs, instead of procedures, modules and service programs, that follow these same architectural guidelines and "theoretically" could be used instead; but from a practical perspective, existing unit test tooling (discussed in other chapters) assumes/requires that procedures be used, and there are considerable performance/functional advantages to using procedures over external programs, so it is strongly advised that you follow this same pattern.

3. Team Scalability and Parallel Development

When everything resides in one or two giant programs, developers are forced to “wait their turn,” often causing merge conflicts and slow delivery cycles. Modularization allows team members to work on different components in parallel. This leads to:

- Faster development cycles
- Reduced bottlenecks
- Improved productivity across both experienced and new developers

4. Reuse and Standardization

Modular components can be reused across batch jobs, interactive applications, services, or APIs. For example, a pricing routine implemented as a service program can be reused by green-screen applications, web services, and mobile apps without duplication. This reduces inconsistency and eliminates redundant maintenance work.

5. Enabler for Modern Architecture

Modularization is not just a “nice-to-have”—it is a prerequisite for adopting modern patterns such as:

- **Service orientation**
- **Microservices**
- **API-first design**
- **Event-driven architectures**
- **DevOps and CI/CD**
- **Automated deployments**

Organizations cannot successfully adopt these models if their applications remain monolithic. Modularization is the bridge between traditional IBM i programming and modern application architecture.

How It Works

Modularization works by applying proven engineering principles:

Single Responsibility Principle

Each module should focus on one job. This makes code simpler, more understandable, and easier to evolve.

Encapsulation

Modules hide internal details and present a clean interface. In IBM i terms, a service program exposes procedures with stable signatures while hiding its internal workings.

Separation of Concerns

UI, business logic, and data access should be separated so that each can evolve independently. A modernized IBM i architecture typically isolates:

- UI layer (5250, web, or mobile)
- Business logic layer (ILE programs and service programs)
- Data access layer (SQL modules or services)

Contract-Based Interfaces

Modules interact through defined contracts—procedure signatures, APIs, or messages. As long as the contract stays stable, the implementation can change freely without disrupting callers.

Applying these principles results in cleaner design, safer changes, and long-term architectural flexibility.

Key Takeaways

- Modularization is essential for maintainable and adaptable IBM i applications.
- It reduces risk, simplifies testing, and increases development velocity.
- It enables parallel development and supports reuse across applications.
- It is the foundation for service orientation, APIs, microservices, and DevOps.
- Modularization transforms monolithic IBM i codebases into architectures that support modern business needs.
- Modularization is a prerequisite for later approaches such as business objects, service orientation, and API-first design discussed in subsequent sections

How Do I Apply Modularization Principles?

What This Covers

This section provides practical guidance on applying modularization principles to IBM i applications. It explains both general software engineering concepts and IBM i–specific techniques such as procedures, modules, and service programs. You will learn how to restructure monolithic applications into layered, maintainable, testable components and how incremental modularization supports modernization goals including APIs, DevOps, and service-oriented design.

What It Is

Applying modularization principles means transforming tightly coupled, monolithic code into smaller, cohesive components with clearly defined responsibilities. These principles are universal in modern software engineering—such as separation of concerns, encapsulation, and contract-based design—and are implemented on IBM i using constructs like ILE modules, procedures, and service programs. The goal is to create stable interfaces, isolate logic, and promote reuse across applications and services.

Why It Matters

Modularization is essential for evolving legacy IBM i applications into modern architectures. It reduces complexity, improves maintainability, and enables parallel development across teams. By isolating business logic, organizations gain flexibility to expose functionality as APIs, integrate with external systems, or shift workloads to new architectural patterns. Modularization is also foundational for automated testing and CI/CD pipelines, which require deterministic, independent units of logic.

How It Works

1. Identify Modularization Candidates

Begin by locating natural seams in existing programs. Look for areas where UI, business logic, and data access are mixed together. Repeated logic, large subroutines, and routines with many responsibilities are strong candidates for extraction.

2. Define Clear Interfaces

Before extracting logic, establish well-defined interfaces. For IBM i, this means designing stable procedure signatures that describe inputs, outputs, and expected behavior. Interfaces should remain stable even as implementations evolve.

3. Extract Logic into Procedures and Modules

Convert monolithic routines and subroutines into standalone procedures within ILE modules. Apply the Single Responsibility Principle so each procedure performs one task. Ensure each procedure has predictable outcomes to support unit testing and reuse.

4. Create Service Programs for Reuse

Group related procedures into service programs to promote reuse across UI layers, batch jobs, and APIs. Use internal procedures for implementation details and export only the public interface procedures. Bind business logic service programs to your application programs to centralize logic.

5. Separate Application Layers

Refactor the application into distinct layers:

- **UI layer**: 5250, web, or mobile interfaces
- **Business logic layer**: ILE modules and service programs
- **Data access layer**: SQL-based modules or I/O service modules

This separation ensures that each layer evolves independently and reduces duplication.

6. Implement Data Access Abstraction

Move data access into separate modules, preferably using SQL. This supports data-centric programming and allows business logic to be insulated from physical database structures. SQL modules provide a clean and consistent way to manage queries and updates.

7. Consider Higher Level Abstraction

Procedures, modules and service programs can be considered the "low level" building blocks of a modern application on IBM i, but it may be easier to consume if you provide, additional, "higher level" interfaces to those functions.

- **Actions**: typically implemented as a command (*CMD), for functions that do not return results (other than pass/fail) especially where default values can be assumed for some/many parameters
- **Lists**: returns tabular results, typically implemented as an SQL view or User-Defined Table Function (UDTF)
- **Methods**: may or may not return results (Result Set or scalar), typically implemented as an SQL procedure or User-Defined Function (UDF).

We discuss each of these subjects in more detail in IBMI-ARCH-PROCEDURE-035, and your lower-level design may be influenced by any higher-level abstraction that you choose to implement, either now or in the future.

8. Add Tests as You Modularize

Modularization enables unit testing. As you create new procedures and service programs, write tests that validate behavior. Modern tooling discussed in other chapters relies on procedure-based logic, making this step easier and more effective.

9. Refactor Iteratively

Avoid large rewrites. Instead, modularize incrementally—focus first on high-change areas or routines with significant reuse potential. Over time, the system becomes more maintainable and aligned with modern best practices.

Key Takeaways

- Modularization is implemented by isolating logic into procedures, modules, and service programs.
- Clear interfaces and contract-based design ensure long-term stability.
- Layered architecture—UI, business logic, and data access—supports cleaner design and modernization.
- SQL-based data access modules promote data-centric architecture.
- Incremental modularization reduces risk and enables testing, DevOps, APIs, and service-oriented patterns.

What Is the Business Object Model?

What This Covers

This topic explains what a business object model is, why it matters when modernizing IBM i applications, and how it fits into a modern application architecture. It introduces the concept at an architectural level, independent of any specific programming language or implementation technique.

What It Is

A business object model is a way of structuring an application around meaningful business concepts rather than around programs, files, or user interfaces. In a business object model, the primary building blocks of the system are objects that represent things the business understands, such as customers, orders, invoices, products, or accounts.

A business object represents:

- A distinct business concept
- The state associated with that concept
- The business rules that govern it
- The operations that are allowed to act upon it

A business object model describes how these objects relate to one another, what responsibilities each object owns, and how business logic is partitioned across the system.

It is important to understand what a business object is *not*. A business object is not a database table, not a program, and not a screen. While business objects may use tables internally and may be implemented by procedures or services, the object itself is a logical construct that exists above those physical elements.

Why It Matters

Traditional IBM i applications are often structured around technical artifacts: programs call other programs, programs read and write to files, and business logic is scattered across many entry points. As applications grow and interfaces expand to include web services, APIs, batch jobs, and external consumers, this approach leads to duplication of logic and increasing architectural fragility.

Without a business object model:

- Business rules are repeated across multiple programs
- Changes require coordinated updates in many places
- Interfaces tend to expose table structures rather than intent
- Modernization efforts become more complex and risky

A business object model provides a stabilizing layer; by explicitly defining business objects and assigning ownership of rules and behavior to them, the application gains clearer structure. Business logic is centralized and consistently applied, and technical details become implementation concerns rather than architectural drivers.

This is especially important for modernization; APIs, services, and integration points become easier to design when they operate on well-defined business objects, rather than on raw database records or program calls.

How It Works

At a conceptual level, a business object model works by aligning application structure with business responsibility. Each business object has a clearly defined purpose and set of behaviors. Operations that affect an object's state flow through the object, ensuring that rules are enforced consistently.

The business object model is independent of programming language choice. Object-oriented programming languages often provide native constructs for modeling business objects, but the concept is equally applicable in environments that use procedural languages like RPG, COBOL or even CL. On IBM i, business objects are commonly implemented using service programs, cohesive procedure groups, SQL procedures, and SQL functions that together represent a single business concept.

In practice, a business object often spans multiple tables and interacts with other objects. For example, an order object may coordinate customer information, pricing rules, inventory availability, and validation logic. The business object model defines these relationships explicitly and ensures that business logic is expressed in terms of business responsibilities rather than being buried inside individual programs.

Sample Business Object Model

The following diagram provides a simplified, conceptual view of a business object model. It focuses on relationships and responsibilities rather than implementation details such as files, tables, or programs.

```
@startuml
title Sample Business Object Model

class Customer {
  customerNumber
  name
  status
}

class Order {
  orderNumber
  orderDate
  amount
  --
  placeOrder()
  cancelOrder()
}

class OrderLine {
  quantity
  unitPrice
}

class Product {
  productCode
  description
}

class PricingRules {
  calculatePrice()
}

class Inventory {
  checkAvailability()
}

Customer "1" -- "0..*" Order : places
Order "1" -- "1..*" OrderLine : contains
OrderLine "*" -- "1" Product : references

Order ..> PricingRules : uses
Order ..> Inventory : checks

@enduml
```

This UML diagram illustrates several important ideas:

- The *Order* business object is the central coordinator for order-related behavior.
- Supporting concerns such as pricing and inventory are treated as distinct business responsibilities.
- Relationships are expressed in business terms, not technical dependencies.

- The model avoids exposing implementation details such as database tables or user interfaces.

Relationship to Modern Architecture

The business object model sits at the center of a modern architecture and connects several other modernization concepts.

Modularization provides the technical mechanisms to decompose code into reusable components. The business object model provides the semantic structure that determines how those modules are grouped.

Data-centric programming moves data integrity and validation into the database layer. Business objects coordinate and contextualize that data logic, ensuring it aligns with business intent.

Service orientation and API-first design build naturally on a business object model. Services become expressions of business capabilities, and APIs expose operations on business objects rather than low-level data access.

Because of this, the business object model is often the missing link between incremental code cleanup and a coherent modern architecture.

Key Takeaways

- A business object model structures applications around business concepts rather than technical artifacts.
- Business objects encapsulate state, rules, and allowed operations.
- The model is conceptual and independent of programming language or database design.
- On IBM i, business objects are commonly implemented using service programs and SQL logic.
- A clear business object model simplifies modernization, integration, and future architectural evolution.
- Business objects form the internal foundation for later service-oriented and API-first designs, where these same Customer and Order capabilities are exposed through explicit interfaces.

How Do I Implement Business Objects?

What This Covers

This section explains how to implement business objects in a modern IBM i application. Building directly on the business object model introduced earlier, it describes how to translate business concepts into concrete IBM i implementation structures using service programs, procedures, SQL-based logic, and data-centric enforcement. The focus is on architectural patterns rather than language syntax.

This topic assumes the existence of a conceptual business object model. For an overview of that model and the relationships it describes, see: Sample Business Object Model UML.

Overview: From Model to Implementation

A business object model defines the business concepts that exist in a system and the responsibilities assigned to each of those concepts. Implementing business objects is the process of turning that conceptual model into executable structures that enforce business behavior consistently.

There is intentionally no one-to-one mapping between a business object and a technical artifact. A business object is not a program, not a table, and not a single procedure. On IBM i, a business object is typically implemented as a cohesive set of artifacts that together represent a single business responsibility.

The goal of implementation is not to recreate the model verbatim in code, but to ensure that business logic, rule ownership, and access patterns align with the business object boundaries defined by the model.

What a Business Object Maps To on IBM i

In a modern IBM i architecture, a business object is commonly implemented using:

- A **service program** that defines the object's logical boundary
- One or more **public procedures** that represent business operations
- Supporting **private procedures** for internal behavior
- SQL-based data access logic such as SQL procedures, views, UDFs, or UDTFs

- Database-enforced rules including constraints and triggers

The service program acts as the object container. Procedures within that service program form the object's public interface and internal implementation, while Db2 enforces fundamental data rules.

Step 1: Identify the Business Object Boundary

The first step in implementation is to define the scope of each business object. This boundary determines which responsibilities are owned by the object and which belong elsewhere.

When identifying a business object boundary, consider:

- What business concept the object represents
- Which operations are meaningful on that concept
- Which rules must always be enforced when the object changes state
- Which data the object owns versus data it references

For example, an *Order* business object typically owns order creation and cancellation, validation of order state, and coordination with pricing and inventory rules. Customer and product data may be referenced but are not owned by the order object itself. For more information, see Sample Business Object Model UML.

Clear boundaries prevent business logic from being duplicated or scattered across unrelated programs.

Step 2: Define the Public Interface

The public interface defines how other parts of the system interact with a business object. These interfaces should express business intent rather than technical detail.

Well-designed business object interfaces are:

- Named after business actions such as *CreateOrder* or *CancelOrder*
- Stable over time
- Minimal in terms of required parameters
- Independent of physical table layouts

On IBM i, these interfaces are typically implemented as exported procedures within a service program. Application programs, batch jobs, APIs, and other consumers interact with the business object by calling these procedures rather than accessing tables directly.

```
**free
//-------------------------------------------------------------------
// Order Business Object - Public Interface
//-------------------------------------------------------------------

// Create a new order for a customer.
// Returns *ON for success, *OFF for failure.
dcl-pr CreateOrder ind;
  customerNumber packed(6:0);
  orderAmount    packed(11:2);
end-pr;


// Cancel an existing order.
// Returns *ON if the order was successfully cancelled.
dcl-pr CancelOrder ind;
  orderNumber packed(10:0);
end-pr;
```

Step 3: Implement Internal Logic Behind the Interface

Within the service program, public procedures coordinate the work required to perform a business operation. This may include:

- Calling SQL procedures, views, or functions
- Coordinating multiple database updates
- Managing workflow or sequencing of actions
- Invoking private helper procedures

Internal structure is intentionally hidden. Callers should not depend on or be aware of how the business object is implemented. This separation allows the implementation to evolve without impacting consumers.

Private procedures inside the service program are commonly used to organize complex logic while preserving a clean external interface.

Step 4: Integrate Data-Centric Rules

Business objects complement data-centric programming rather than replacing it.

Rules related to data validity and integrity should be enforced at the database level using constraints, referential integrity, and triggers. Business object procedures should assume that these rules exist and should not duplicate them.

Instead, business objects focus on business workflow and intent, while Db2 ensures that data remains in a valid and consistent state regardless of how it is accessed.

For more information, see What is Data-centric Programming?.

Step 5: Control Access Through the Business Object

Once implemented, business objects should become the primary access path for application logic.

This means:

- Application programs call business object procedures
- Services and APIs map to business object operations
- Direct table access is limited to internal logic and database enforcement

By funnelling access through business objects, change becomes easier to manage and business rules remain centralized and consistent.

Relationship to Services and APIs

Business objects are internal architectural constructs. Services and APIs are external representations of business capabilities.

Services built on top of business objects are typically thin layers that translate external requests into business object operations. This approach ensures that APIs expose business intent rather than low-level CRUD semantics and supports long-term evolution of both interfaces and implementations.

Common Pitfalls

When implementing business objects, common pitfalls include:

- Treating business objects as renamed database tables
- Allowing callers to bypass business object interfaces
- Exposing internal SQL structures through procedure signatures
- Over-fragmenting logic across unrelated service programs
- Attempting to force strict object-oriented patterns where they do not add value

The objective is architectural consistency and clarity, not adherence to a specific programming paradigm.

Key Takeaways

- Business objects are implemented as cohesive service-level boundaries.
- Service programs are the natural IBM i construct for business object implementation.
- Public procedures express business intent, while internal logic remains encapsulated.
- Data-centric enforcement and business object logic work together.
- Implementing business objects simplifies testing, services, and modernization efforts.
- In later sections, these **Customer** and **Order** business objects are exposed through APIs and, optionally, decomposed into microservices.

Data-Centric Programming

What Is Data-Centric Programming?

What This Covers

This section explains the principles of data-centric programming in general, and specifically how they apply to IBM i modernization - describing how shifting the business rules and data processing logic into the database layer (DB2 for i) improves consistency, integrity, and efficiency across all applications.

What It Is

Data-centric programming is an architectural approach in which data — along with its rules, relationships, and integrity — is treated as the foundation of the system. Instead of relying on individual applications to enforce business logic, the database itself manages these responsibilities. This includes constraints, relationships, validation logic, and controlled access paths.

In general software engineering terms, data-centric programming emphasizes:

- Declarative logic rather than procedural logic
- Set-based processing instead of record-at-a-time operations
- Centralized enforcement of business rules
- Clear separation between data storage and application behavior

On IBM i, data-centric programming leverages native Db2 for i capabilities such as constraints, triggers, SQL views, and SQL-based data access, allowing the database to become an intelligent, active component instead of a passive storage layer.

Why It Matters

Traditional IBM i applications frequently use application-centric design. Business rules are scattered across RPG programs, embedded in native record-level I/O, and duplicated across modules and interfaces. This makes modernization difficult and risky.

Data-centric programming matters because it:

- Provides a single, consistent source of business rules
- Eliminates duplication of logic across programs
- Improves data integrity by enforcing rules at the database layer
- Supports modern integration patterns such as APIs, microservices, and cloud services
- Makes analytics and reporting more consistent and reliable
- Reduces the cost of long-term maintenance

By moving validation, relationships, and structure into Db2 for i, multiple applications,— 5250, web, mobile, APIs, or batch processes — automatically gain consistent behavior without needing to reimplement rules.

How It Works

General Principles

Data-centric architecture is built on widely accepted engineering practices:

- **Use SQL as the primary interface** to data for clarity, stability, and flexibility.
- **Favor set-based operations** for efficiency and declarative logic.
- **Centralize validation and integrity rules** rather than duplicating them in applications.
- **Abstract physical database structures** using views or service modules.

These principles help ensure stable, predictable behavior no matter how many applications interact with the data.

IBM i–Specific Implementation

Db2 for i provides mature, high-performance features to support data-centric architecture:

- **Constraints** (primary key, foreign key, unique, check) enforce integrity without application code.
- **Triggers** centralize auditing, validation, and transformation logic.
- **SQL views** provide abstraction layers decoupling applications from physical schema changes.
- **SQL procedures, functions, and UDF/UDTFs** provide consistent access patterns.
- **RCAC (Row and Column Access Control)** allows fine-grained security without embedding logic in programs.

While DDL-defined tables are recommended over DDS-described files for development (especially when features like identity columns or advanced constraints are required), many DDS-based files can still be used within a data-centric architecture; the key is not the file definition method alone, but **where business rules and relationships are enforced**.

Key Takeaways

- Data-centric programming makes the database, not individual programs, the authority for business rules and integrity.
- SQL, constraints, triggers, and views enable consistent behavior across all applications.
- IBM i supports both DDS- and DDL-based tables, though DDL offers more capabilities.
- A data-centric approach reduces duplication, improves data quality, and supports modern architectures.
- Moving to data-centric design is essential for long-term IBM i modernization and integration readiness.

How Do I Move to Data-Centric Programming?

What This Covers

This section walks through a practical, example-driven procedure for converting an application-centric IBM i design into a fully data-centric architecture. You will see how to move business rules, validation, and data access from RPG programs into Db2 for i using SQL tables, constraints, triggers, SQL-based access modules, and optional enhancements such as views, UDF/UDTFs, and RCAC.

Before You Start

To implement the RCAC examples below, you'll need to be authorized to the `QIBM_DB_SECADM` function usage identifier.

```
CHGFCNUSG FCNID(QIBM_DB_SECADM) USER(User_Profile_Name) USAGE(*ALLOWED)
```

1. Identify the Existing Application-Centric Pattern

Business rules like *“Credit limit must be >= order amount”* and *“Status must be A=Active or H=Hold”* are embedded directly inside RPG logic. External tools writing to these files must also replicate the same rules, which is risky.

By moving to a data-centric architecture, Db2 for i becomes the single enforcement point for all data rules.

DDS Files (Original Physical Files)

```
A          R CUSTREC
A            CUSTNO        6S 0
A            NAME         30A
A            CREDITLIM     9S 2
A            STATUS         1A
A          K CUSTNO
```

```
A          R ORDHDR
A            ORDERNO       10S 0
A            CUSTNO         6S 0
A            ORDERDATE       L
A            AMOUNT        11S 2
A          K ORDERNO
```

RPG Program (Monolithic, Application-Centric)

```
CHAIN (CustNo) CUST;
IF NOT %FOUND(CUST)
   OR (CUST.CREDITLIM < OrderAmt)
   OR (CUST.STATUS <> 'A' AND CUST.STATUS <> 'H')
   OR (OrderAmt < 0);
   RETURN *OFF;
ENDIF;
WRITE ORDHDR;
RETURN *ON;
```

2. Convert DDS Files to SQL Tables With Constraints

SQL Table Definitions

```
CREATE OR REPLACE TABLE CUST (
  CUSTOMER_NUMBER   FOR CUSTNO    DECIMAL(6,0) PRIMARY KEY,
  NAME                            VARCHAR(30),
  CREDIT_LIMIT      FOR CREDITLIM DECIMAL(9,2) NOT NULL,
  STATUS                          CHAR(1) CHECK (STATUS IN ('A','H'))
);

CREATE OR REPLACE TABLE ORDER_HEADER FOR SYSTEM NAME ORDHDR (
  ORDER_NUMBER      FOR ORDERNO   DECIMAL(10,0) PRIMARY KEY,
  CUSTOMER_NUMBER   FOR CUSTNO    DECIMAL(6,0),
  ORDER_DATE        FOR ORDERDATE DATE NOT NULL,
  AMOUNT                          DECIMAL(11,2) CHECK (AMOUNT >= 0),
  FOREIGN KEY (CUSTOMER_NUMBER) REFERENCES CUST(CUSTOMER_NUMBER)
);
```

> **Note:** we add new, "long" SQL names e.g. ORDHDR→ORDER_HEADER, CUSTNO→CUSTOMER_NUMBER, but system names are preserved to allow existing queries, RPG programs etc. to still refer to them.

3. Add a Trigger to Enforce Business Rules Centrally

```
CREATE OR REPLACE TRIGGER ORDER_HEADER_VAL
BEFORE INSERT ON ORDER_HEADER
REFERENCING NEW ROW AS N
FOR EACH ROW
BEGIN
  DECLARE CustLimit DECIMAL(9,2);
  SELECT CREDIT_LIMIT INTO CustLimit
    FROM CUST
    WHERE CUSTOMER_NUMBER = N.CUSTOMER_NUMBER;
  IF N.AMOUNT > CustLimit THEN
     SIGNAL SQLSTATE '75001'
       SET MESSAGE_TEXT = 'Order exceeds credit limit';
  END IF;
END;
```

4. Add an SQL-Based Access Module

```
CREATE OR REPLACE SEQUENCE ORDER_SEQUENCE
  FOR SYSTEM NAME ORDERSEQ
  START WITH 50004;
```

```
CREATE OR REPLACE PROCEDURE INSERT_ORDER (
   IN pCUSTOMER_NUMBER DECIMAL(6,0),
   IN pAMOUNT          DECIMAL(11,2)
)
LANGUAGE SQL
BEGIN
  INSERT INTO ORDER_HEADER (ORDER_NUMBER, CUSTOMER_NUMBER, ORDER_DATE, AMOUNT)
  VALUES (NEXT VALUE FOR ORDER_SEQUENCE,
          pCUSTOMER_NUMBER,
          CURRENT_DATE,
          pAMOUNT);
END;
```

5. Update RPG to Use SQL Instead of Native I/O

```
EXEC SQL
  SELECT NAME, CREDIT_LIMIT
    INTO :Name, :Limit
    FROM CUST
    WHERE CUSTOMER_NUMBER = :CustNo;
EXEC SQL
  CALL INSERT_ORDER(:CustNo, :OrderAmt);
```

Additional Modernization Enhancements (Optional)

A1. Create a View for Abstraction

```
CREATE OR REPLACE VIEW ORDER_SUMMARY AS
SELECT  O.ORDER_NUMBER,
        O.CUSTOMER_NUMBER,
        C.NAME,
        O.ORDER_DATE,
        O.AMOUNT
FROM ORDER_HEADER O
JOIN CUST C
  ON O.CUSTOMER_NUMBER = C.CUSTOMER_NUMBER;
```

A2. Add a UDF or UDTF for Reusable Business Logic

```
CREATE OR REPLACE FUNCTION OPEN_ORDERS(CustNo DECIMAL(6,0))
RETURNS TABLE (ORDER_NUMBER DECIMAL(10,0), AMOUNT DECIMAL(11,2))
LANGUAGE SQL
RETURN
  SELECT ORDER_NUMBER, AMOUNT
    FROM ORDER_HEADER
    WHERE CUSTOMER_NUMBER = CustNo;
```

A3. Apply RCAC for Centralized Data Security

```
CREATE OR REPLACE MASK CUSTOMERS_NAME_MASK
ON CUST
FOR COLUMN NAME
RETURN CASE WHEN SESSION_USER IN ('MGR1','MGR2') THEN NAME
            ELSE '***REDACTED***' END
ENABLE;

ALTER TABLE CUST ACTIVATE COLUMN ACCESS CONTROL;
```

Verify It Worked

Insert sample customers

```
INSERT INTO CUST (CUSTOMER_NUMBER, NAME, CREDIT_LIMIT, STATUS)
VALUES
  (100001, 'Acme Industries', 5000.00, 'A'),
  (100002, 'Global Manufacturing', 12000.00, 'H'),
  (100003, 'Sunrise Retail', 8000.00, 'A');
```

Insert sample orders

```
INSERT INTO ORDER_HEADER (ORDER_NUMBER, CUSTOMER_NUMBER, ORDER_DATE, AMOUNT)
VALUES
  (50001, 100001, CURRENT_DATE - 5 DAYS, 1500.00),
  (50002, 100003, CURRENT_DATE - 2 DAYS, 3000.00),
  (50003, 100001, CURRENT_DATE, 450.00);
```

Add a new order

```
CALL INSERT_ORDER(100001, 1000.00);
```

Display Order Summary

```
SELECT * FROM ORDER_SUMMARY;
```

> **Note:** `***REDACTED***` will be displayed for `NAME` when you are using a profile other than `MGR1` or `MGR2` as enforced by the RCAC definition above.

Display Open Orders

```
SELECT * FROM TABLE(OPEN_ORDERS(100001)) AS OpenOrders;
```

Verify It Does Not Work

> **Note:** all of these are expected to **fail**

Attempt to add a customer with an invalid status

```
INSERT INTO CUST (CUSTOMER_NUMBER, CREDIT_LIMIT, STATUS)
VALUES (200001, 500, 'X');
```

Should fail with `[SQL0545] INSERT, UPDATE, or MERGE not allowed by CHECK constraint`

Attempt to add an order without a corresponding customer

```
CALL INSERT_ORDER(100004, 1000.00) -- should fail because CUSTNO=100004 does not exist
```

Should fail with `[SQL0530] Operation not allowed by referential constraint`

Attempt to add an order that exceeds the credit limit for the customer

```
CALL INSERT_ORDER(100001, 7500.00); -- should fail because pAMOUNT=7500 exceeds CREDIT_LIMIT=5000 for CUST
```

Should fail with `[SQL0723] SQL trigger ORDER_HEADER_VAL failed... and the message is Order exceeds credit limit`

Key Takeaways

- Db2 for i enforces business rules centrally via constraints, triggers, and RCAC.
- Views and UDTFs create reusable business logic and stable interfaces.
- Data-centric programming simplifies modernization and eliminates duplicated logic.
- The **Customer** and **Order** example used here is revisited in later sections to illustrate service-oriented, API-first, and microservices designs.

What Is Service Orientation?

What This Covers

This topic introduces service orientation as an architectural style and explains what it means to design systems around services rather than programs, files, or components. It clarifies the core ideas behind service-oriented architecture and positions service orientation as a complementary concept to the business object model described earlier.

While **business objects** focus on organizing internal logic around *business concepts*, **service orientation** focuses on *how those capabilities are exposed, consumed, and evolved* through well-defined service boundaries and contracts. Together, these concepts form the foundation for APIs, microservices, and modern integration patterns.

What It Is

Service orientation is an architectural approach in which application functionality is organized as a set of services that expose well-defined business capabilities through explicit interfaces.

A service represents a discrete unit of business functionality that can be invoked by other parts of the system or by external consumers. Services are defined by what they do, not by how they are implemented internally. Each service exposes a contract that describes the operations it supports, the inputs it expects, and the outcomes it produces.

Service orientation is not a specific *technology* like Web Services or REST (although many services are actually implemented this way) and does not require a particular programming language or runtime; it is a *design philosophy* that emphasizes clear boundaries, explicit contracts, and independence between service consumers and service implementations.

Why It Matters

Many traditional systems evolve as tightly coupled collections of programs that depend directly on each other's internal behavior and data structures. As systems grow, this coupling increases the risk and cost of change. Small modifications can ripple across applications, and introducing new interfaces often requires duplicating logic or exposing internal details.

Service orientation addresses these challenges by introducing an abstraction layer. By defining functionality as services with stable contracts, systems become easier to extend, integrate, and evolve. Consumers depend on what a service promises to do, rather than how it does it.

This approach supports modernization by enabling:

- Clear separation between internal implementation and external use
- Safer integration with new applications and platforms
- Incremental evolution of legacy systems without large rewrites

Service orientation also provides the conceptual foundation for API-first design and microservices architecture, both of which rely on the same core principles.

How It Works

In a service-oriented system, functionality is grouped into services based on business responsibility. Each service owns its behavior and rules and is accessed through a clearly defined interface. Services communicate with each other by invoking operations rather than by directly sharing data structures or internal logic.

Service interactions are designed to minimize assumptions. A service consumer does not need to know how a service is implemented, where its data is stored, or what internal components it uses. This loose coupling allows services to change internally without breaking consumers, as long as the contract remains stable.

Service orientation applies at multiple levels. A system can be service-oriented even if it is deployed as a single application. Similarly, services do not need to be remotely deployed or independently scaled to qualify as service-oriented. The defining characteristics are the boundaries, contracts, and interaction patterns, not the deployment model.

Key Takeaways

- Service orientation is an architectural style that organizes systems around services.
- Services expose business capabilities through explicit and stable contracts.
- Service orientation focuses on interaction boundaries rather than internal implementation.
- It complements the business object model by governing how capabilities are accessed.
- Service orientation forms the foundation for APIs and microservices.
- Service orientation provides the organizational structure behind APIs and microservices discussed later in this chapter.

Service-Oriented and Microservices Architecture

What Are Service Properties and Patterns?

What This Covers

This reference describes the common properties that characterize well-designed services and the recurring architectural patterns used in service-oriented systems. It provides shared vocabulary and definitions that are referenced by other service-oriented, microservices, and API-related topics.

This section is not intended to prescribe a single architecture, but to document the characteristics and patterns that commonly appear in service-oriented designs.

Service Properties

Service properties describe the fundamental characteristics that distinguish services from other architectural building blocks such as programs, modules, or components.

Loose Coupling

Services are loosely coupled, meaning that service consumers depend only on the service contract, not on its internal implementation. Changes to service internals do not affect consumers as long as the contract remains stable.

Explicit Contracts

Each service exposes an explicit contract that defines:

- Available operations
- Required inputs
- Expected outputs
- Error conditions

The contract represents a clear agreement between the service and its consumers.

Autonomy

A service controls its own logic and data. It makes decisions independently and does not rely on other services to manage its internal state. This autonomy supports independent evolution and simplifies reasoning about behavior.

Encapsulation

Internal logic, data access, and implementation details are hidden from consumers. Services expose behavior, not internal structures.

Discoverability

Services are designed so consumers can identify what functionality they provide and how to interact with them. Discoverability may be supported through documentation, registries, or standardized naming and interface conventions.

Statelessness (Where Appropriate)

Many services are designed to be stateless across requests, meaning that they do not retain conversational state between calls. Stateless designs simplify scaling and improve reliability, though some services may legitimately manage state as part of their responsibility.

Reusability

Services are defined around business capabilities rather than specific use cases, allowing them to be reused across applications, channels, and interfaces.

Common Service Patterns

Service patterns describe recurring ways that services are structured and composed to solve common architectural concerns.

Facade Service

A façade service provides a simplified interface over a set of underlying services or components. It shields consumers from complexity and reduces the number of interactions required to perform a task.

Orchestration

In orchestration, a central service coordinates a sequence of service calls to perform a business process. The orchestration logic explicitly controls the flow of execution.

Choreography

In choreography, services collaborate through events or messages without a central coordinator. Each service reacts to events and performs its role independently.

Coarse-Grained vs. Fine-Grained Services

- **Coarse-grained services** expose higher-level business operations, reducing chatty communication.
- **Fine-grained services** expose smaller operations that may offer greater flexibility but require careful coordination.

Synchronous vs. Asynchronous Interaction

- **Synchronous services** require immediate responses and are often used for request/response interactions.
- **Asynchronous services** decouple senders and receivers through messaging, improving resilience and scalability.

Shared-Nothing Services

Shared-nothing services avoid sharing databases or internal state with other services. This pattern supports autonomy and reduces coupling between service implementations.

API Gateway

An API gateway acts as a single entry point for external consumers. It may provide routing, security, rate limiting, and protocol adaptation while delegating business logic to underlying services.

How These Are Used

Not every service-oriented system will use all properties or patterns. Architectural choices depend on system scale, organizational maturity, performance requirements, and operational constraints.

Service properties and patterns should be treated as design tools rather than strict rules. Understanding them allows architects and developers to reason more clearly about trade-offs as systems evolve.

Key Takeaways

- Service properties define the essential characteristics of services.
- Service patterns describe recurring structural and interaction approaches.
- Not all properties or patterns are required in every system.
- These concepts provide shared vocabulary for service-oriented design.
- They are foundational to microservices and API-first architectures.

How do containers fit IBM i architecture?

As containerisation becomes the dominant model for deploying modern applications, a natural question arises: where do containers fit in a platform that was never designed with them in mind?

Understanding IBM i's Architecture

As we know, IBM i is a single-level storage system built around a proprietary object model. Every resource is an object managed by the operating system. Applications run in a tightly integrated environment.

This architecture is precisely why IBM i is so reliable. The tradeoff is that it operates on fundamentally different assumptions from the open, layered design that containers were built for.

So even though many containers run on the same machine, each one believes

that it is the only thing running. This is what makes containers feel isolated, even though they’re sharing the same kernel.

Open Source and the Chroot Approach

IBM has invested significantly in bringing open-source tooling to IBM i itself through the IBM i Open Source Software initiative, delivered via RPM packages through ACS (Access Client Solutions). This has brought Python, Node.js, PHP, Git and many other tools to the platform natively.

While these tools do not run in containers, they do allow IBM i to host modern application runtimes directly.

Some shops use chroot environments on IBM i as a lightweight alternative to containers, creating isolated file system trees that partition open-source workloads from the rest of the system.

It is not containerisation in the Docker sense, but it offers a degree of isolation and repeatability that suits many use cases.

Hybrid Integration Patterns

In practice, many organisations are adopting a hybrid model: business logic and data remain on IBM i, while containerised services run on adjacent Linux partitions or in the cloud. These communicate with the IBM i via REST APIs, MQ messaging, or direct database calls through ODBC/JDBC.

This pattern makes considerable sense. IBM i's integrated Db2 database, with its decades of transactional data, is not something you simply migrate away from. Instead, it becomes a stable backend, while containers handle the dynamic, scalable front-end layer.

An e-commerce platform, for example, might run its order management system entirely on IBM i while deploying containerised product recommendation engines and web-sites in Docker.

IBM Cloud and the Managed Services Path

IBM has also made it easier to connect IBM i environments to containerised workloads running in IBM Cloud. Services like IBM Cloud Kubernetes Service and Red Hat OpenShift on IBM Cloud can act as the container orchestration layer for organisations whose IBM i systems sit on-premises, linked through VPN or IBM's dedicated connectivity options.

This is particularly valuable for shops looking to extend their IBM i investment into a cloud-native architecture without undertaking a full rewrite.

Conclusion

Containers do not run natively on IBM i and trying to force that model onto the platform would be at odds with what makes it valuable.

The smarter approach is to treat IBM i and containers as complementary rather than competing technologies. IBM i excels at what it has always excelled at, reliability, integrated data management and decades of proven business logic.

Containers excel at scalability, portability and rapid deployment of modern applications.

The organisations getting the best results are the ones who don’t try to put every single workload into a container. Instead, they step back and ask a smarter question: which parts of the system should stay on IBM i and which are better suited to containers?

Making that choice wisely is what lets IBM i keep delivering its stability while still giving the business the speed and flexibility that modern development needs.

What Is Microservices Architecture?

What This Covers

This topic explains what microservices architecture is, how it relates to service-oriented design, and why it represents a more specialized architectural style rather than a universal solution. It clarifies what distinguishes microservices from general service orientation and provides context for when and why microservices are used.

What It Is

Microservices architecture is an architectural style in which an application is composed of a set of small, independently deployable services, each representing a distinct business capability and owning its own logic and data.

Microservices build on service-oriented principles such as explicit contracts, loose coupling, and autonomy. What distinguishes microservices is the emphasis on independence at runtime and deployment time. Each microservice is designed to be developed, deployed, scaled, and versioned independently of other services.

A microservice is not defined by technology choice or implementation language. It is defined by its scope, ownership, and independence. Microservices communicate with other services through well-defined interfaces and avoid sharing internal state or databases.

Why It Matters

As systems grow in size and complexity, large, tightly coupled applications become difficult to evolve. Independent teams may struggle to deploy changes safely, small improvements may require system-wide releases, and operational risk increases over time.

Microservices architecture addresses these challenges by introducing strong boundaries that align software structure with organizational responsibility. Teams can evolve services independently, deploy updates without coordinating global releases, and scale parts of the system according to actual demand.

However, microservices also introduce new complexity. Distributed systems must deal with network latency, partial failures, operational overhead, and increased observability requirements. For these reasons, microservices are most effective when organizational maturity, tooling, and operational practices are ready to support them.

Understanding what microservices architecture is—and what it is not—helps avoid adopting it prematurely or for the wrong reasons.

How It Works

In a microservices architecture, each service encapsulates a specific business capability and owns the data required to support that capability. Services interact through well-defined, technology-agnostic interfaces and avoid direct access to each other's data stores.

Services are designed to be independently deployable. This means that changes to one service do not require rebuilding or redeploying other services, provided interfaces remain stable. Communication between services is typically handled through synchronous APIs, asynchronous messaging, or a combination of both.

Microservices architecture places a strong emphasis on operational considerations. Logging, monitoring, error handling, versioning, and security become integral parts of the design. The architectural focus extends beyond code structure to include deployment pipelines, runtime environments, and failure management strategies.

Importantly, a system does not need to consist entirely of microservices to benefit from service-oriented thinking. Many architectures combine microservices with modular monoliths or service-oriented subsystems, evolving incrementally based on business and organizational needs.

Key Takeaways

- Microservices architecture is a specialized form of service-oriented architecture.
- Microservices emphasize independent deployment, scalability, and ownership.
- Each microservice represents a discrete business capability and owns its data.
- Microservices introduce operational complexity alongside architectural benefits; some systems or applications might not benefit from this level of decomposition, so remember that *microservices may not be absolutely required in a modernization project*.
- Microservices are most effective when applied deliberately rather than universally.

What Is API-First Design?

What This Covers

This topic explains what API-first design means in modern application architecture and how it applies in IBM i environments that contain significant existing systems and long-lived business logic. It introduces API-first design as a discipline for defining and managing external interfaces deliberately, rather than as an expectation that systems be built from scratch.

The focus of this section is conceptual: understanding what API-first design is, why it matters, and how it fits alongside service-oriented and legacy architectures commonly found on IBM i.

What It Is

API-first design is an architectural approach that treats APIs as stable, intentional products with clearly defined contracts, rather than as accidental by-products of internal implementations.

In IBM i systems, APIs almost never replace existing programs, files, or business logic. Instead, APIs typically sit *on top of* existing RPG programs, service programs, and Db2 for i logic, acting as controlled entry points into the system. API-first design means that once an API exists, its contract is treated as more stable and more carefully governed than the internal RPG or SQL code behind it.

The term "first" does not imply that APIs must be implemented before any internal logic exists; in long-lived IBM i applications, internal logic almost always exists first. API-first design means that when functionality is exposed outside the system—whether to a web application, mobile client, partner system, or cloud service—the API contract is designed consciously and owned explicitly.

API-first design is compatible with monolithic applications, modular systems, and service-oriented architectures on IBM i. It is a design discipline, not a development sequence or deployment model.

In this chapter, API-first design is applied to the same **Customer** and **Order** business objects introduced earlier in this chapter, without requiring redesign of internal logic.

Why It Matters

IBM i systems often evolve over decades, accumulating business rules spread across interactive programs, batch jobs, database triggers, and ad-hoc interfaces. Over time, programs and database objects become de facto integration points simply because other systems begin to depend on them.

When such systems are exposed externally without a clear API strategy, risk increases dramatically. External consumers may begin to depend directly on program interfaces, parameter lists, file layouts, or table structures—making internal change difficult or dangerous.

API-first design addresses this problem by making interfaces explicit and intentional. APIs become the supported, documented way to access business capabilities, while internal programs and data structures remain implementation details. This allows IBM i teams to modernize incrementally, improving internals over time without breaking external consumers.

API-first design also enables IBM i systems to participate safely in modern integration patterns—supporting web applications, REST clients, event-driven systems, and cloud services—without rewriting existing business logic.

How It Works

In practice, API-first design introduces a clear boundary between external consumers and internal implementation. APIs act as façades that translate external requests into calls to existing business objects, service programs, SQL procedures, or legacy workflows.

For example, an API that creates an order does not expose Db2 table layouts or call low-level RPG programs directly. Instead, it invokes a higher-level business operation implemented in a service program, which may internally coordinate multiple RPG procedures, SQL statements, and data-centric rules.

The API contract becomes the primary point of coordination for change. Internal RPG programs, SQL procedures, or file structures can be refactored as needed, provided the API contract remains stable. Versioning strategies allow APIs to evolve when change is unavoidable, without forcing immediate updates on all consumers.

API-first design does not require exposing the entire system. Many IBM i teams start by identifying a small number of high-value integration points—such as order entry, customer lookup, or inventory availability—and applying API-first discipline there. Additional APIs can be introduced gradually as needs evolve.

Importantly, API-first design applies equally to modular monoliths, service-oriented systems, and microservices. It defines how capabilities are exposed externally, not how internal logic must be structured.

Key Takeaways

- API-first design prioritizes stable, intentional external contracts.
- “First” refers to contractual discipline, not development order.
- API-first design is compatible with long-lived IBM i systems and monoliths.
- APIs should expose business capabilities, not RPG programs or database tables.
- API-first design reduces integration risk and supports gradual modernization.

How Do I Design Microservices for IBM i?

What This Covers

This topic explains how to design microservices for IBM i systems in a deliberate and practical way. It builds on earlier concepts such as business objects, data-centric programming, service orientation, and API-first design, and shows how those patterns can be extended into a microservices architecture when appropriate.

This procedure uses the same *Customer* and *Order* domain introduced earlier to illustrate how an existing IBM i application can be selectively decomposed into microservices without rewriting the entire system.

Before You Start

Before designing microservices, ensure that the following foundations are in place:

- ☐ Customer and Order business logic is already modularized
- ☐ Business objects are implemented behind stable service program interfaces
- ☐ Data integrity is enforced using Db2 for i (constraints, triggers, referential integrity)
- ☐ APIs are used as stable external contracts
- ☐ There is a clear business or organizational reason to adopt microservices

Microservices introduce operational complexity and should be adopted intentionally, not as a default modernization step.

Steps

1. Decide Whether Microservices Are Actually Needed

The first step is not technical - it is architectural.

Microservices are most appropriate when:

- Teams need to deploy independently
- Parts of the system change at different rates
- Scaling requirements differ significantly across capabilities
- Ownership boundaries align with business domains

If the business object logic--**Customer** and **Order** in this example--can evolve together and be deployed as a single unit, a modular monolith or service-oriented design may be sufficient.

> **Note:** Microservices should be chosen because they solve a problem, not because they represent architectural maturity.

2. Identify Clear Service Boundaries

In a microservices architecture, each service represents a **distinct business capability** and owns its behavior and data.

Using the existing domain:

- **Customer service** owns customer identity, status, and lifecycle rules
- **Order service** owns order creation, cancellation, and order state

Each service must have:

- A clear purpose
- A defined boundary
- No responsibility for another service's data or rules

Avoid splitting services along technical lines such as batch vs interactive or RPG vs SQL.

3. Assign Data Ownership Explicitly

A defining characteristic of microservices is **independent data ownership**.

For Customer and Order:

- The Customer service owns customer tables
- The Order service owns order tables
- No shared tables across services

Shared-database designs undermine service autonomy and should be avoided. Instead:

- Services communicate through APIs
- Cross-service data is referenced by identifiers (for example, `customerNumber`)
- Referential integrity across services is enforced at the application level, not the database level

On IBM i, this often means creating schemas or libraries that are owned by a specific service.

4. Design Service APIs as the Only Integration Point

Each microservice exposes functionality exclusively through its API.

For the Order service, this includes operations such as:

- `CreateOrder`
- `CancelOrder`
- `GetOrder`

Internal service programs, RPG procedures, and Db2 tables are not accessed directly by other services.

APIs should:

- Express business intent
- Be versioned deliberately
- Remain stable even as internal implementation changes

This builds directly on the API-first patterns described in What is API-first design?.

Example: Order Microservice RPGLE Skeleton

This example shows a minimal RPGLE service-program skeleton representing an **Order microservice**. It illustrates how a microservice on IBM i can delegate persistence and rule enforcement to Db2 for i by calling an existing SQL procedure.

This example builds directly on the `INSERT_ORDER()` procedure introduced in How do I move to data-centric programming?.

> **Note:** This is a logical skeleton only. Error handling, logging, authorization, and transport-specific concerns are intentionally omitted.

```
**free
//-------------------------------------------------------------
// Order Microservice – RPGLE Skeleton
//-------------------------------------------------------------
ctl-opt dftactgrp(*no)
        actgrp(*caller)
        option(*srcstmt:*nodebugio);

//-------------------------------------------------------------
// CreateOrder : Public business operation
//-------------------------------------------------------------
dcl-proc CreateOrder export;
  dcl-pi *n ind;
    customerNumber packed(6:0);
    orderAmount    packed(11:2);
  end-pi;

  //-------------------------------------------------
  // Delegate validation, persistence and validation to Db2
  //-------------------------------------------------
  exec sql
     CALL INSERT_ORDER(:customerNumber,
                       :orderAmount);

  if sqlcode < 0;
     return *off;
  endif;

  return *on;

end-proc;
```

This example reinforces that:

- The microservice owns business behavior, not persistence details
- Db2 for i enforces data-centric rules
- The service depends on SQL procedures, not tables
- Internal implementation can evolve without changing API contracts

5. Decouple Services Through Messaging Where Appropriate

Not all service interactions need to be synchronous.

For example:

- Order creation may publish an event
- Downstream systems may react asynchronously

Asynchronous communication:

- Reduces runtime coupling
- Improves resilience
- Simplifies independent evolution

On IBM i, this can be implemented using messaging or queue-based patterns without adopting an entirely new platform.

6. Plan Deployment and Operational Boundaries

Microservices require more than code separation.

Each service should:

- Be deployable independently
- Have its own deployment pipeline
- Log and report errors independently
- Be monitored as a separate unit

On IBM i, this does not require abandoning the platform. Microservices can coexist with:

- ILE service programs
- SQL procedures
- PASE-based runtimes or containers

The key requirement is operational independence, not a specific technology stack.

7. Migrate Incrementally

Few IBM i systems can—or should—be decomposed all at once.

A common and effective approach is:

- Keep the core system as a modular monolith
- Extract one service at a time
- Begin with a clear domain such as Order
- Allow hybrid architectures during transition

Incremental migration reduces risk and preserves business continuity.

Verify It Worked

You can validate a successful microservices design if:

- Customer and Order services can be deployed independently
- Each service owns its data and business rules
- Services communicate only through APIs or messaging
- Internal changes in one service do not require changes in others
- Operational issues are isolated to individual services

Success is measured by independence and resilience, not by service count.

Key Takeaways

- Microservices are an architectural choice, not a requirement
- Business capability boundaries define microservice boundaries
- Independent data ownership is essential
- API-first design is a prerequisite, not an afterthought
- IBM i systems can adopt microservices incrementally and pragmatically

How Do I Implement API-First Patterns?

What This Covers

This topic explains how to apply API-first design patterns in an existing IBM i environment using a realistic, incremental approach. It shows how to expose **Customer** and **Order** capabilities that we've used in other topics in this chapter through stable APIs while preserving existing business logic, service programs, and Db2 for i enforcement.

This procedure does not assume greenfield development or a microservices architecture; instead, it demonstrates how API-first discipline can be layered on top of a long-lived, internally complex system.

The examples in this procedure build directly on the **Customer** and **Order** business objects and service programs introduced earlier in this chapter.

Before You Start

Before applying API-first patterns, you should have:

- ☐ Existing Customer and Order business logic (RPG programs, service programs, SQL)
- ☐ Business object interfaces for operations such as CreateOrder and GetCustomer
- ☐ A service-oriented internal structure (even if the system is still monolithic)
- ☐ Agreement that APIs will be treated as stable, long-lived contracts

The goal is not to redesign the system, but to control how it is accessed.

Steps

1. Identify the Customer and Order Capabilities to Expose

Begin by identifying *business capabilities* that external consumers need, not internal programs or tables.

For the Customer and Order domain, good candidates include:

- CreateOrder
- CancelOrder
- GetOrder
- GetCustomer
- UpdateCustomerStatus

Avoid exposing:

- Db2 tables directly
- Record-level access programs
- Screen-oriented RPG programs
- Parameter lists designed for interactive use

At this stage, API-first design is a filtering exercise: deciding which behaviors are safe and valuable to expose.

---

2. Ensure Each Capability Is Implemented Behind a Service Program

API-first patterns rely on stable internal interfaces. Each exposed capability should already exist—or be wrapped—inside a service program representing the Customer or Order business object.

For example:

- Order operations are implemented in an ORDER service program
- Customer operations are implemented in a CUSTOMER service program

If legacy logic exists only in old RPG programs, wrap that logic rather than rewriting it. The service program becomes the internal contract that the API layer depends on.

This step ensures that APIs do not call internal programs directly and that business rules remain centralized.

---

3. Design API Contracts Independently of RPG and Db2 Structures

Design API contracts that express **business intent**, not internal structure.

For example:

- An Order API accepts `customerNumber` and `orderAmount`
- It does not expose ORDER_HEADER columns
- It does not reflect RPG parameter lists
- It does not mirror Db2 table layouts

API request and response models should be stable, descriptive, and decoupled from physical storage. This allows internal refactoring (program changes, table changes, constraints) without breaking consumers.

At this point, the API contract is considered a first-class architectural artifact.

4. Implement the API as a Façade Over Business Objects

Implement each API endpoint as a façade that:

- Validates external input
- Translates request formats into internal calls
- Invokes Customer or Order service procedures
- Formats responses for external consumers

For example:

- A CreateOrder API calls the CreateOrder procedure in the ORDER service program
- That procedure enforces business rules and relies on Db2 constraints and triggers
- The API layer does not re-implement business rules

The façade pattern allows internal logic to remain unchanged while providing clean external access.

Example: API Interfaces for Customer and Order Capabilities

API-first design describes *how interfaces are designed and governed*, not the protocol used to expose them. While HTTP and RESTful APIs are common, they are not required. The same principles apply to message-based interfaces, internal procedure calls, or other transport mechanisms.

The examples below use HTTP-style URLs simply because they are familiar and widely understood.

Create Order (HTTP Example)

```
POST /api/v1/orders
Content-Type: application/json

{
  "customerNumber": 100001,
  "orderAmount": 1500.00
}
```

This API represents the **CreateOrder** business capability. Internally, it invokes the `CreateOrder` procedure in the Order service program.

Get Order (HTTP Example)

```
GET /api/v1/orders/50001
```

Example response:

```
{
  "orderNumber": 50001,
  "customerNumber": 100001,
  "orderDate": "2026-03-15",
  "amount": 1500.00,
  "status": "OPEN"
}
```

Cancel Order (HTTP Example)

```
POST /api/v1/orders/50001/cancel
```

This maps directly to the **CancelOrder** business operation.

Get Customer (HTTP Example)

```
GET /api/v1/customers/100001
```

Example response:

```
{
  "customerNumber": 100001,
  "name": "ACME Industries",
  "status": "ACTIVE"
}
```

Important Note: APIs Are Not the Same as "Web"

Although these examples use HTTP-style endpoints, API-first design does **not** require web technology, REST, or JSON.

The same API-first patterns apply when:

- APIs are implemented as internal procedures with well-defined contracts
- Messages are exchanged using queues or events
- Existing enterprise protocols are used

HTTP and REST are simply common *delivery mechanisms*. The core of API-first design is the **contract**, not the transport.

5. Version APIs, Not Internal Programs

API-first patterns require deliberate versioning.

Version:

- API endpoints
- API request and response models
- API semantics

Do not version:

- RPG service programs
- Internal procedures
- Db2 tables or fields

For the Customer and Order APIs, this means:

- Introducing new API versions when breaking change is required
- Keeping older API versions stable for existing consumers
- Allowing internal code to evolve independently

For example, if we added a **version 2** version of the version one interface above, it might appear something like this:

```
GET /api/v2/customers/100001
```

6. Prevent Accidental APIs

Once APIs exist, it is critical to prevent new accidental integration points from emerging.

This includes:

- Discouraging direct table access by external systems
- Avoiding documentation that points consumers to internal programs
- Treating APIs as the only supported external contract

Over time, API-first discipline shifts integration pressure away from legacy internals and toward intentional interfaces.

Verify It Worked

You can confirm that API-first patterns are working if:

- External consumers use APIs instead of internal programs or tables
- Customer and Order logic can change without breaking consumers
- New integrations start with API contracts, not program calls
- API versions are documented and managed deliberately

Success is measured by reduced coupling, not by the absence of legacy code.

Key Takeaways

- API-first patterns can be applied incrementally in IBM i systems.
- APIs should sit on top of business objects (for example, **Customer** and **Order**).
- Façade and versioning patterns protect internal complexity.
- Legacy RPG and Db2 logic can be preserved while exposure is modernized.
- API-first design improves long-term evolvability without forcing rewrites.

Architecture Patterns and Security

Which Architecture Pattern Should I Choose?

What This Covers

This topic helps you decide which architecture patterns discussed in this chapter are appropriate for your system. Rather than recommending a single “best” architecture, it explains how different patterns address different problems and how they can be combined to fit real-world constraints.

This discussion is grounded in the realities of long-lived IBM i systems, where core business logic is already in production. Throughout the chapter, specific business object examples (such as **Customer** and **Order**) have been used to illustrate concepts; here, those examples are generalized into a broader decision framework you can apply to any business domain.

How to Think About Architecture Choices

Architecture is not a linear maturity path. Systems do not “graduate” from one pattern to another, and newer patterns are not inherently better than older ones.

Instead, architecture is a set of **deliberate trade-offs** made to balance:

- Business change frequency
- Integration needs
- Operational complexity
- Organizational structure
- Risk tolerance

Patterns such as modularization, business objects, data-centric design, service orientation, API-first design, and microservices are **tools**, not destinations. They can be adopted independently or together, incrementally or selectively. The presence of one pattern does not require adoption of the next.

The goal is not to maximize architectural sophistication, but to choose patterns that reduce friction and enable safe evolution.

The Architecture Patterns Available to You

This chapter introduced several architectural patterns, each addressing a different concern. Where helpful, concrete examples from earlier procedures are referenced to reinforce how these patterns look in practice.

- **Modular Monolith** – Internally modularized code deployed as a single unit. Explored in: How do I apply modularization principles?
- **Business Object–Centric Architecture** – Logic organized around business objects such as **Customer** and **Order**, with clear ownership of behavior and rules. See the UML model in What is the business object model? and the service-program implementation in How do I implement business objects?
- **Data-Centric Architecture** – Data integrity and core business rules enforced directly in Db2 for i using constraints, triggers, and SQL procedures. For example, the `INSERT_ORDER()` procedure shown in How do I move to data-centric programming?
- **Service-Oriented Architecture** – Business capabilities organized into services with explicit boundaries and contracts. Introduced in What is service orientation? and summarized in What are service properties and patterns?
- **API-First Architecture** – External access governed by stable, intentional APIs that act as façades over internal business logic. Concrete examples such as `GET /api/v1/orders/50001` appear in How do I implement API-first patterns?

- **Microservices Architecture** – Independently deployed services with autonomous data ownership and APIs as the only integration point. An Order microservice skeleton using `dcl-proc CreateOrder() export` is shown in How do I design microservices for IBM i?

Each pattern answers a different architectural question. None replaces the others.

Applying Patterns to the Same Business Capabilities

To make these differences concrete, consider the same set of business capabilities—such as managing **Customer**, **Order**, product, or invoice data—implemented using different architectural patterns.

Business Capabilities as a Modular Monolith

In a modular monolith, all business capabilities are deployed together. Internal modular boundaries exist, but there are no external APIs or runtime separation.

This approach is effective when:

- Change is coordinated or infrequent
- One team owns the full application
- Simplicity and operational stability are priorities

A modular monolith can be well-architected without being distributed.

Business Objects with Data-Centric Design

In this model, business capabilities are represented as explicit business objects with clear ownership of behavior. Db2 for i enforces data integrity using constraints, triggers, and referential integrity.

Deployment may still be unified, but internal clarity improves significantly. Business rules are centralized, duplication is reduced, and refactoring becomes safer.

For many IBM i systems, this is a strong and sufficient architectural end state.

Service-Oriented Business Capabilities

Service-oriented architecture organizes business capabilities into services with clearly defined boundaries and interfaces. Services communicate through explicit contracts rather than shared logic.

This pattern is useful when:

- The system is increasing in complexity
- Multiple consumers exist
- Future external exposure is anticipated

Service orientation improves structure without forcing independent deployment.

Business Capabilities with API-First Exposure

API-first design introduces stable external contracts for business capabilities while keeping internal implementation flexible. APIs act as façades over business objects and services.

External consumers depend on APIs, not internal programs or database structures. Internal logic can evolve without breaking integrations.

This pattern is especially effective when:

- External integration is growing
- Internal modernization must be incremental
- Contract stability is critical

Business Capabilities as Microservices

In a microservices architecture, business capabilities are split into independently deployed services with autonomous data ownership.

This approach offers maximum flexibility but introduces operational complexity. It is most appropriate when:

- Teams require independent deployment
- Scaling needs differ significantly
- Organizational ownership aligns with service boundaries

Microservices solve specific problems; they are not a universal goal.

Decision Factors to Consider

Choosing among these patterns requires weighing multiple factors:

Change and Deployment

Do business capabilities change together, or can **Customer** and **Order** evolve independently? Independent change favors decoupled architectures; coordinated change favors simpler deployment.

Integration Needs

Are consumers increasing in number or diversity? If so, API-first patterns may provide stability without structural disruption.

Organizational Structure

Architecture should reflect team boundaries. Distributed designs are harder to sustain without clear ownership.

Operational Readiness

More distributed patterns require mature monitoring, logging, deployment, and recovery practices. Choose patterns your organization can operate reliably.

Common but Costly Mistakes

Architecture decisions often fail when:

- Microservices are adopted simply to appear modern
- APIs expose tables or internal structures instead of **Customer** or **Order** behaviors
- Internal boundaries are unclear before distribution
- Patterns are applied as mandatory steps rather than tools

These mistakes add complexity without reducing risk.

Valid End States

All of the following can represent a well-architected IBM i system:

- A clean, modular monolith
- A business object–centric application with data-centric enforcement
- A service-oriented system with API-first exposure
- A hybrid system with selective microservices

There is no required final form. The right architecture depends on context.

How This Chapter Fits Together

This chapter presented architecture as a layered set of choices:

- Modularization improves internal structure
- Business objects clarify responsibility

- Data-centric design enforces integrity
- Services define boundaries
- APIs control exposure
- Microservices enable independent evolution when needed

The same **Customer** and **Order** capabilities were viewed through these different lenses to show that architecture is about perspective and intent, not replacement.

Key Takeaways

- Architecture is about trade-offs, not trends.
- Patterns are composable and incremental.
- API-first and microservices are choices, not requirements.
- **Customer** and **Order** capabilities can be modernized safely without rewrites.
- The best architecture is one you can evolve deliberately.

How do I implement Secure by Design principle?

How can modernizing an IBM i application make it more secure?

What This Covers

YES! Without a shadow of a doubt, the best way to make an application more secure is to modernize it. And as you modernize you do so with both data integrity and data security in mind.

For more information, see IBM Security and Privacy by Design.

What It Is

IBM Security by Design strategy is a streamlined and agile set of focused security and privacy practices. These practices aim to reflect their commitment to improving security and privacy when designing generally available products and services from IBM. It is influenced by the United States National Institute of Standards and Technology Secure Software Development Framework.

Why It Matters

Most businesses operating today must adhere to one or more compliance standards. Common amongst these are:

- CCPA – California Consumer Privacy Act & GDPR – General Data Protection Regulation

  Legal regulations that govern how organizations collect, process, store, and protect personal data of individuals in their jurisdictions, that apply to companies that may reside outside.

- HIPAA – Health Insurance Portability and Accountability Act

  A U.S. law that establishes national standards for protecting sensitive patient health information and securing electronic healthcare data.

- PCI DSS – Payment Card Industry Data Security Standard

  A global security standard that requires businesses handling credit card data to maintain secure systems, protect cardholder data, and regularly test their security controls.

- SOX – Sarbanes-Oxley Act

  A U.S. federal law that mandates strict financial reporting, internal controls, and corporate governance requirements for publicly traded companies.

- ISO/IEC 27001 – Information Security Management System Standard

  An international standard that specifies requirements for establishing, implementing, maintaining, and continually improving an information security management system (ISMS).

All of the above have strict requirements for compliance and significant legal and financial consequences for failing to comply.

How It Works

This process does not mean that you undertake one single project to just focus on security of your application, instead add security as a key design factor to the changes you plan to make to your system as you modernize it.

One way to do this is by following IBM's secure by design strategy. To do this there are three steps that you must build into every change you make to any part of your application:

- Threat assessments: Perform both cyber and privacy threat assessments to help ensure data minimization and adequate data protection are in place from the beginning.
- Security testing: Perform system, code and application security tests, in addition consider using tools or third parties to perform penetration tests and manual ethical hacks. Such testing should be integrated and automated into DevOps pipelines, supporting agile practices and continuous deployment.
- Release review: Carry out a final product review of changes prior to release, check that key foundational security requirements have been or will be addressed to the satisfaction of your business's standards.

Key Takeaways

If you include "Secure by Design" thinking when making the changes to your system, over time you will consistently improve the security of your application alongside its functionality.

While there is an overhead in adding security to your modernization process, this is tiny compared with the effort of trying to retrofit security to an existing application on mass. The cost is also offset by the reduction in risk of large financial penalties that your company might be liable for in the event of a breach, as most regulators will reduce or even waive the fine if you can demonstrate the due diligence you give to security when making changes to your systems.

Modern Development Tools for IBM i

Modern IDE Fundamentals

Why use modern IDEs over PDM/SEU?

> "It is tempting, if the only tool you have is a hammer, to treat everything as if it were a nail." - Abraham Maslow, 1966

The above quote is known as the Law of the instrument and it is the perfect reply to the question raised by this topic. Let's see why it matters to replace a hammer with a toolbox.

Why It Matters

Modernizing an application means refreshing both the source code and the development tools you use to build that application.

SEU is part of the Application Development ToolSet (ADTS) that was introduced in 1988, when AS/400 was launched. And it was a great editor that fitted the 5250 terminals used back then.

Code Navigation

Today, modern monitors display far more than the 24 or 27 lines available on a screen to navigate the source code of an application. We all have one or two wide monitors sitting on our desks that can display way more information than

a terminal. Working on legacy applications means looking at huge code bases. A modern editor will allow to open multiple tabs, organize them, compare them, etc - it lets users organize their workspace as they please.

Being able to locate where a variable is defined and used, or to display the body of a procedure at a glance is one click away in an IDE. Modernizing means having a global view of an application and it's exactly what modern IDEs can help with.

Basic editing features

Undo/Redo, Copy/Paste, Save As, code snippets, content assist...these are the basic features everyone needs hourly when working on code. SEU barely offers these if not at all. Any mainstream code editor will allow to use these very basic features.

New language features adoption

Since 2008, ADTS is officially categorized as legacy tools - which means that since IBM i V6R1, it has not received new updates, including support for new RPGLE language features. Using recent RPGLE built-in functions in SEU will result in syntax checking complaining about unknown keywords whereas they are perfectly valid RPGLE keywords, which defeats the purpose of syntax checking entirely.

Moving to the IFS and using Git

Physical Source Files have a limited width and don't let the code be organized in multiple levels. They also prevent the use of Git for source control management. Moving the source code to the IFS is the first step toward a more secure and flexible workspace. RDi and VS Code have no problem working with IFS files whereas SEU is limited to the QSYS file system.

Customization/Extensibility

Modern IDEs are meant to be extensible, in an easy way. SEU allows customization to some extent but it's very limited compared to what VS Code has to offer. Its extension marketplace is the host to thousands and thousands of extension that will let user change any aspect of the editor and add more features to it.

Work offline

At the end of the day, compiling an IBM i program still needs to be done on an IBM i. But unlike SEU that requires to be connected all the time in order to edit source code, VS Code and RDi allows to work on source code locally, disconnected from IBM i, and synchronize the changes later on (with or without Git). It gives more flexibility to the developers and prevents loss of changes if the connection drops unexpectedly.

Attracting new talents

> You can catch more flies with honey than with vinegar.

Beyond code maintenance and application architecture, using a modern IDE also means coding on IBM i is no different from coding on Linux or MacOS. Modern tools are well known by the new generation of coders. Putting freshly graduated software developers on SEU is guaranteed to drive them away. Have them use VS Code, and they will feel right at home.

AI Integration

AI development tools are getting more and more integrated into the development process. But these tools will NEVER be integrated or be accessible from SEU. Using SEU for your developments means missing out on powerful and intelligent features meant to empower the developers.

Shifting the mindset

Adopting a new IDE often comes with a period of adaptation and the learning curve can be steep. But it's worth it! Access to tools that make developers' lives easier helps shifting their mindset, facilitating the adoption of new paradigm like procedure-oriented RPGLE.

Key Takeaways

If you are going to build the next Ferrari, a rusty hammer and a bended screwdriver won't help. You'll need modern tools to achieve this goal efficiently.

- Modern IDEs helps enforce modern practices
- They empower the developer
- New developers are more familiar with modern IDEs
- SEU is frozen in time
- Modern IDEs are the keys to open the modernization gates

What security considerations should I make when considering development tools?

Development tools, like any other piece of software that connects to your IBM i, require specific security attention. These tools will communicate over the network, sensitive information will travel through it so these communications need to be secured.

They are also softwares that need to be regularly updated to keep the risk of vulnerabilities at bay.

Keep the communication secure

Securing a communication line means encrypting all data transmitted over it. It should be encrypted to prevent any threat actors from sniffing the communication and syphoning sensitive information like credentials or just source code.

RDi

RDi connects to IBM i using the JTOpen - IBM Toolbox for Java library.

In RDi, To make a connection secure in the Remote System Explorer, make sure to check the `Use secure communication methods` checkbox. This will force the use of SSL/TLS to secure the connection and encrypt the communication.

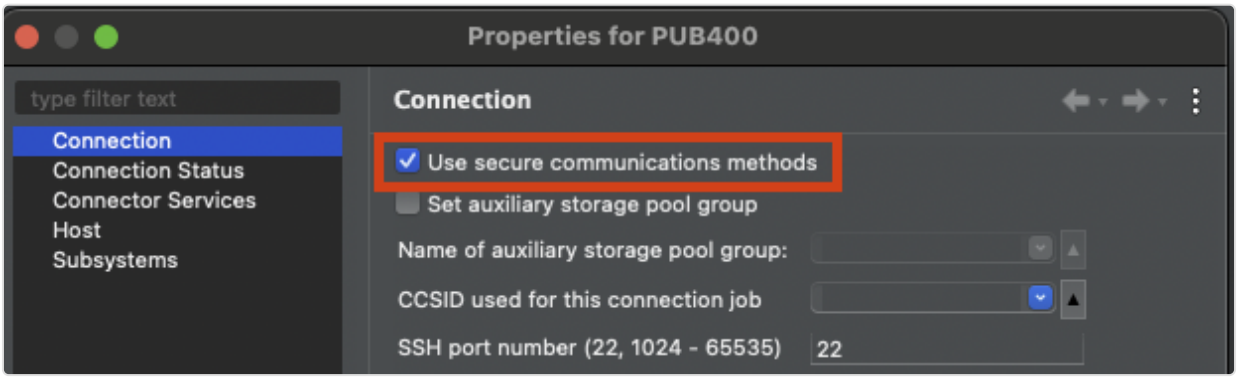


Figure. IBM i Tools Secure Communication Configuration

For this to work, the remote IBM i must be configured to accept secure connections. This involves several complex tasks that are covered in the dedicated RDi documentation section.

Once the IBM i is configured to accept secure connections, the next step is to forbid unsecure connections and close the unsecure TCP ports used by JTOpen.

Code for IBM i

Code for IBM i connects to IBM i over SSH, which means the connection is inherently encrypted. But SSH itself is more or less secure depending on its configuration.

By default, SSH will allow users to connect using their IBM i profile's password. Use key-based authentication as a better practice, ensuring that no credentials are exchanged and if a key is compromised, it can easily be removed from the list of authorized keys to keep the system secure. Code for IBM i fully supports passphrase protected keys.

Encourage or require keys protected with a passphrase. In this case, if a malicious actor obtains the key, they'll need the passphrase to be able to use it which reduces the risk of having the key to be used before it's removed from the authorized keys.

Below is a list of recommended values to keep the connection secure and harden the SSH configuration. The SSH server configuration is located in the `/QOpenSys/QIBM/ProdData/SC1/OpenSSH/etc/sshd_config` file on the IFS.

Key	Value	Explanation
Port	not 22	While not making it more secure, changing the default port reduce the number of automated attacks and makes the server less visible to casual scanning.
Protocol	2	Only SSH version 2 should be used. Version 1 is obsolete and vulnerable.
PermitRootLogin	no	The yes value allows QSECOFR to log into the system through SSH. Setting it to no limits the risk of targeted attack on this login.
PasswordAuthentication	no	Disables password authentication and force the use of key based authentication.
ChallengeResponseAuthentication	no	Disables interactive authentication methods susceptible to phishing attacks.
PermitTunnel	no	Disables tunneling to limit risks of abuse.
X11Forwarding	no	Disables X11 session transfer, rarely used and a possible vector of attack.
MaxAuthTries	3	Limits the number of unsuccessful connection attempts before disconnection.

ACS

ACS connects to IBM i using a JDBC connection using the JDBC driver provided by the JTOpen - IBM Toolbox for Java library.

The same principles as RDi's apply in this case to secure the connection. Once the IBM i host is ready to accept secure connections, turn on the `Use SSL for connection` option in the properties of the connections defined under the `System Configuration` section.

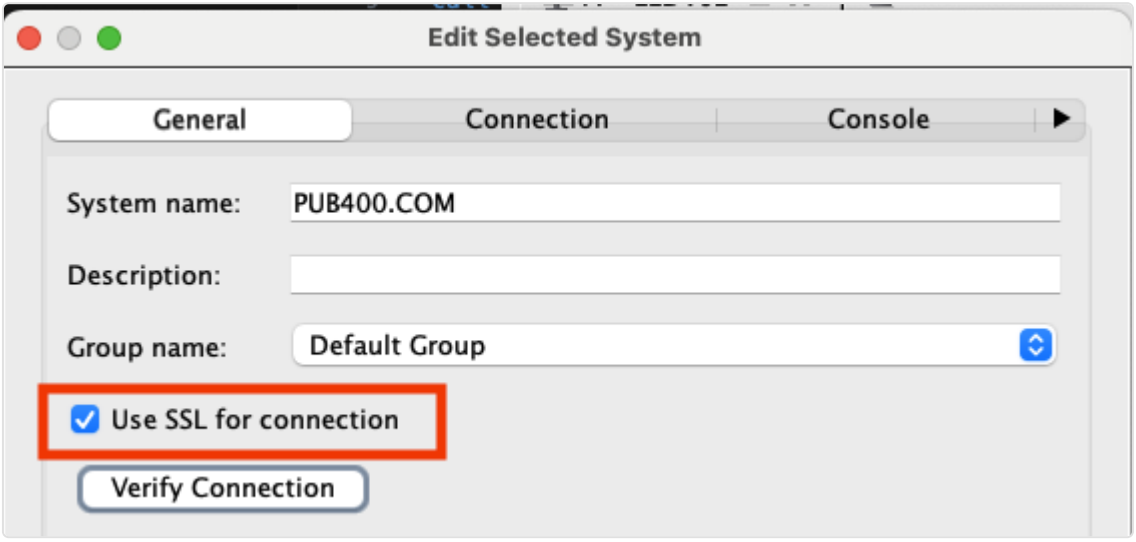


Figure. ACS SSL Connection Configuration

Keep the tools updated

Updating a tool is not only done to benefit from new features and bug fixes but also to make sure they don't carry vulnerabilities.

VS Code will update itself regularly and so will Code for IBM i, every time a new release is published. RDi and ACS may require their updates to be applied manually.

For all the IBM supported products, IBM publishes security bulletins with the corresponding CVE (Common Vulnerability and Exposures) and its severity.

For more information on known vulnerabilities and how to patch them, see IBM Product Security Central.

You can also Subscribe to Security Bulletins (per product familly) to keep yourself updated on vulnerabilities.

Key Takeaways

- Keep the tools updated, at least with security patches
- Ensure the client communications are established over secured channel (i.e. over TLS)
- Secured TCP connections can only be established on a properly configured IBM i
- SSH is secured by default but its configuration can be hardened to tighten security

VS Code for IBM i Development

What are the VS Code extensions useful for IBM i development?

Aside from the IBM i Developer Pack, there are numerous VS Code extensions that can improve the IBM i developer experience when using VS Code.

You'll find below a curated list of such extensions.

IBM i Project Explorer

The IBM i Project Explorer extension provides a dedicated view in VS Code Explorer container view to manage buildable local IBM i Projects, hopefully hosted on Git. The IBM i projects will self-describe how to build themselves as much as possible. The project can be git cloned directly into VS Code for editing and synchronized to the IFS directory it is built from.

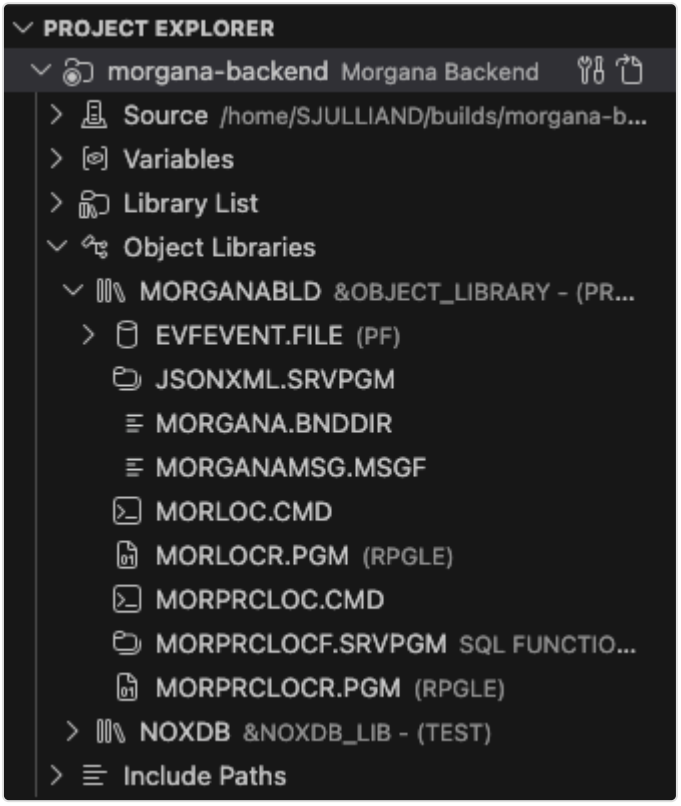


Figure. IBM i Project Explorer

This view lets you display and manage the project's Metadata, library list, variables, object libraries, include paths, and much more. It offers commands to run the project's build. The build logs are automatically loaded in a specific view to easily spot what went wrong.

It is a good extension for whoever is looking at using Git to host and share an IBM i project.

DB2 for IBM i

Db2 for IBM i provides SQL functionality to VS Code.

Key Features

- Statement executor and result set view
- Schemas view
- Query history
- SQL Job Manager, with JDBC options editor and configuration manager

vscode-ibmi-fs

This extension provides interfaces for viewing and managing objects inside of the IBM i QSYS file system. It integrates with Code for IBM i's Object Browser and let you interact with the following object types:

- Save Files (`*SAVF`)
- Data Queues (`*DTAQ`)
- Data Areas (`*DTAARA`)
- Output Queues (`*OUTQ`)
- Job Queues (`*JOBQ`)
- User Spaces (`*USRSPC`)
- User Indexes (`*USRIDX`)

- Message Files (`*MSGF`)
- Message Queues (`*MSGQ`)
- Binding Directories (`*BNDDIR`)
- Programs and Service Programs (`*PGM` / `*SRVPGM`)
- Modules (`*MODULE`)
- Commands (`*CMD`)
- Job Descriptions (`*JOBD`)
- Journal Receivers (`*JRNRCV`)
- Journals (`*JRN`)
- DDM Files (`*DDMF`)
- Subsystem Descriptions (`*SBSD`)
- Classes (`*CLS`)
- Files (`*FILE`)
- Query Definitions (`*QRYDFN`)

Source Orbit

Source Orbit extension is part of the larger Source Orbit project that aims at being a local dependency management tool for IBM i projects.

The extension provides two views to help developer view the impact of their current changes on a project:

- The **Source Impacts** view in the Explorer view container will render the impacted objects for the current active editor.

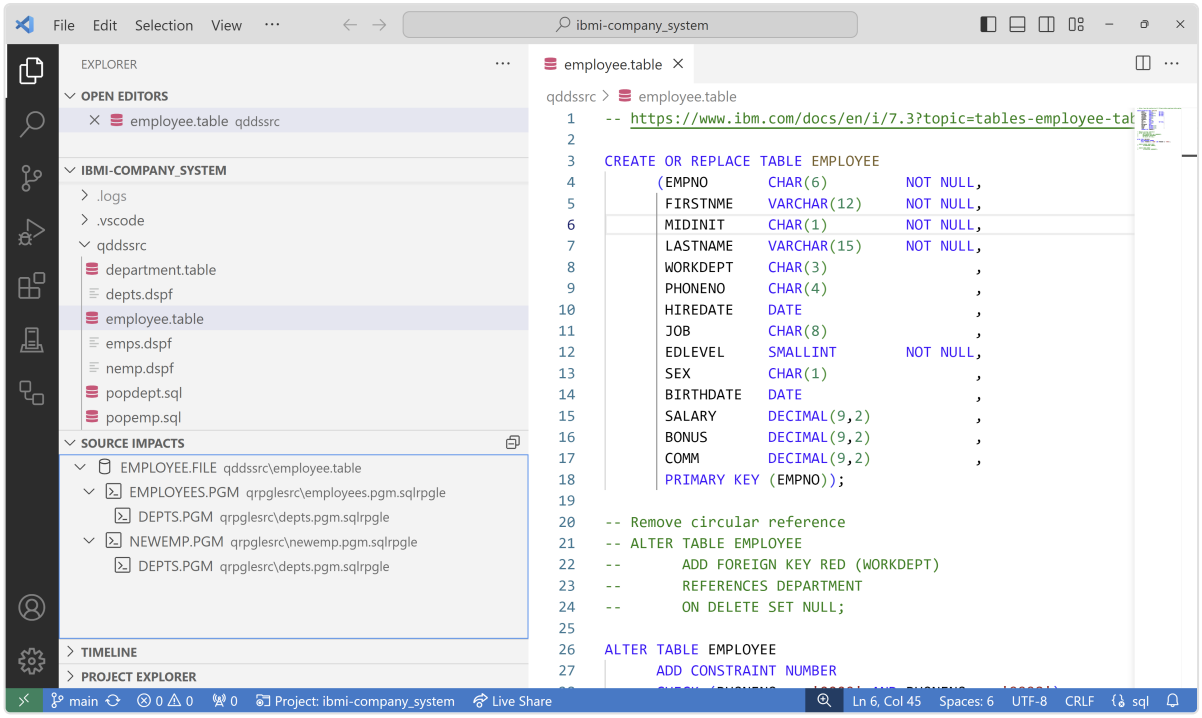


Figure. Source Impacts View

- The **Change Impacts** view in the Source Control view container will render the impacted objects for any files changed detected by Git.

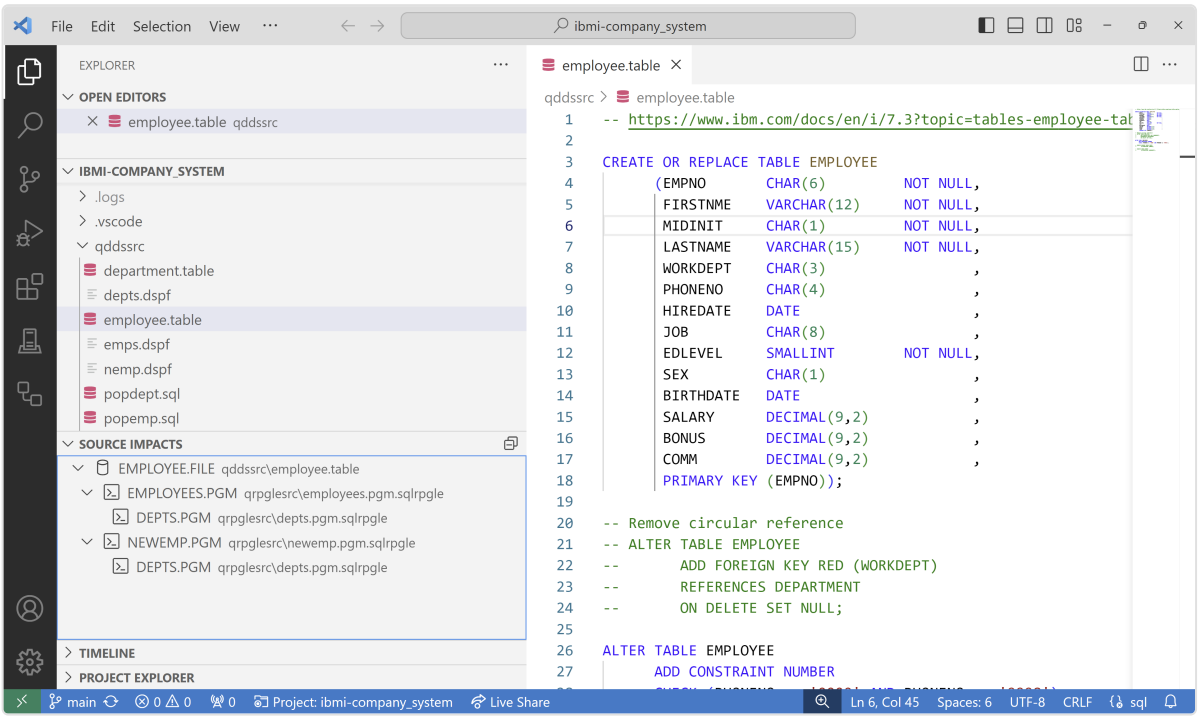


Figure. Change Impacts View

The extension also integrates with the IBM i Project Explorer extension to display an object dependency tree for the IBM i Project being worked on. It can generate build files for `gmake` or `TOBi` and generate dependency reports in JSON and MarkDown formats.

Source Orbit is also a good tool to get you started on migrating an IBM i application to Git as it offers the required tools to migrate from QSYS to streamfiles.

IBM i Testing

The IBM i Testing extension provides the tools to the developer to design and run unit tests and generate code coverage results for RPGLE and COBOL programs on IBM i. It uses the RPGUnit testing framework to achieve this.

It fully integrates with VS Code Testing view container to display the test results:

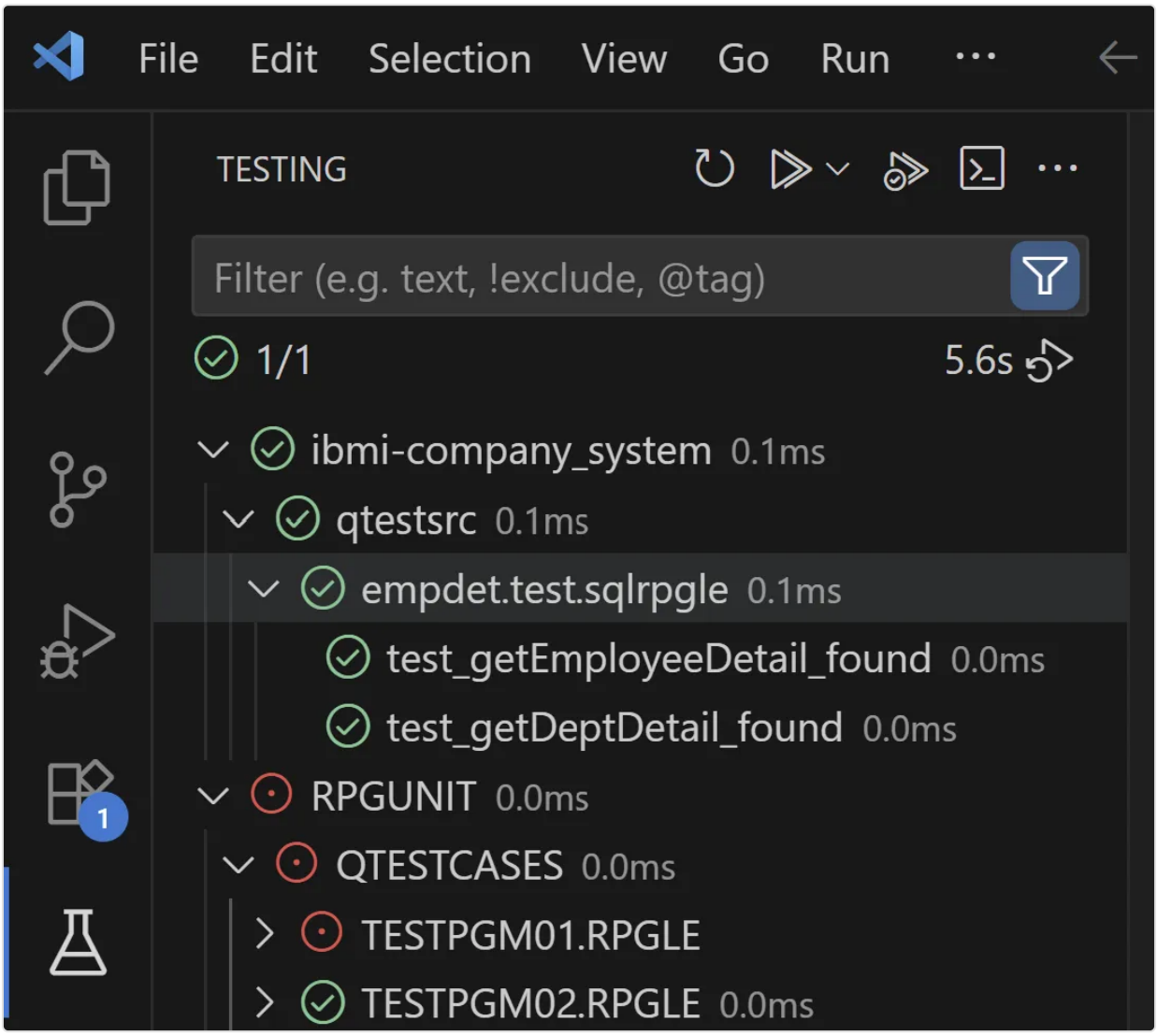


Figure. IBM i Testing View

To simplify the process of writing tests, the extension can generate stubs for individual test cases or an entire test suite, right from the code being edited. It can, for example, write tests for every procedure found in a program. It makes it easy to get started and can greatly accelerate the process of writing unit tests for an entire project.

The extension comes with a companion CLI tool to run the tests outside of VS Code, from a local remote terminal, PASE on IBM i or a CI/CD pipeline.

Community supplied extensions

Apart from the official extensions, there is a growing number of extensions provided by the community.

IBM i Job Log Detective

Analyzes IBM i job logs to quickly identify errors and issues in large job log files.

Key Features

Feature	Description
Custom Tree View	Dedicated "Job Log Detective" panel with filtering and navigation
Smart Analysis	Highlights high-severity messages and analyzes for errors
Timeline View	Chronological view with time buckets showing when issues occurred (icons indicate severity density)
Outline Integration	Messages grouped by type and message ID
Editor Decorations	Gutter flame icons, background highlighting, minimap markers
Multi-Language	English, German, Dutch, French, Italian, Spanish

CL Prompter

CL Prompter is a professional CL (Control Language) prompter for VS Code that replicates the familiar IBM i F4 command prompting experience. Integrates with vscode-for-ibmi extension.

Key Features

Feature	Details
Navigation & Input	Visual focus indicator, Tab navigation, F3=Cancel, Enter=Apply
Smart Formatting	ELEM parameter handling, multi-line comment wrapping, qualified name formatting
Customization	Theme-aware colors, configurable layout, case conversion options
Diagnostics	Save command XML and prompter HTML for troubleshooting

Spooled Files

vscode-ibmi-UserSpooledFiles is an extension that provides views and tools for listing, viewing, and managing IBM i user spooled files directly from VS Code.

Key Features

Feature	Details
Viewing	List user spooled files, add additional users (read-only), open in editor with optional line spacing, open in PDF format
Search & Filter	Search through spooled files, filter by attributes (job number, etc.), sort by name or date
Download	Export to .TXT, .SPLF, or .PDF (*) formats, with optional line spacing
Management	Delete files (by user, name, filter), move to another output queue, multi-file selection
Configuration	Custom spooled file naming patterns, temp file cleanup, line spacing on open, hover tooltips

(*) Viewing as PDF requires installation of a separate PDF viewer.

What is VS Code?

Navigating the ever-evolving landscape of development tools can be challenging. For both new and experienced developers, the choice of a code editor is significant. Among the many options available, one tool has seen a meteoric rise in popularity: Visual Studio Code (VS Code). But what exactly is VS Code, and what makes it such a widely adopted editor?

A Modern Code Editor

At its core, Visual Studio Code (commonly referred to as VS Code) is a free, open-source code editor developed and maintained by Microsoft. A major advantage is its cross-platform availability, running on Windows, macOS, and Linux. This allows development teams to maintain a consistent environment regardless of their individual operating systems.

Upon first launch, users are often greeted by a clean, intuitive interface. Unlike many traditional integrated development environments (IDEs) which can be resource-intensive and slow to start, VS Code provides a lightweight and fast design.

The Power of Extensibility

The foundational philosophy of VS Code is modularity. It provides a robust set of core features, but its true power is realized through its extensive ecosystem of extensions. This “app store” model for a code editor allows for a high degree of customization. For instance, web developers can install extensions for frameworks like React or Vue, along with tools like Prettier for automatic code formatting.

For IBM i developers, this extensibility is particularly valuable. Extensions like Code for IBMi bring modern development tools to traditional languages like RPG and COBOL. These extensions provide features such as syntax highlighting, source code browsing & editing, and integration with the Integrated File System (IFS), allowing developers to work with IBMi resources directly within VS Code.

Key Features of VS Code

Intelligent Code Completion

One of the most praised features of VS Code is IntelliSense. It goes beyond simple autocompletion by providing smart completions based on variable types, function definitions, and imported modules. This feature helps developers write code faster and with fewer errors.

Integrated Debugging

Effective debugging is another area where VS Code excels. It has built-in support for debugging code. For many other languages, including those used on IBMi, debugging capabilities can be added through extensions. The debugging interface is visual and intuitive, allowing users to set breakpoints, step through code, inspect variables, and view the call stack.

Streamlined Version Control

For modern development workflows, version control is essential, and VS Code's built-in support for Git is a significant advantage. It allows developers to view changes, stage files, and commit them directly from the editor. The visual diff tool makes it easy to see exactly what has been changed.

Integrated Terminal

VS Code also includes an integrated terminal, which is a convenient feature that allows users to run shell commands, install packages, or execute applications without leaving the editor. With the Code for IBMi extension, we can open IBMi 5250 terminals as well as a PASE terminal inside VS Code itself. This helps to keep the workspace organized and maintain focus.

Customization and Personalization

VS Code is highly customizable. From themes that alter the color scheme to fine-tuning the editor's behavior through its settings, users can tailor the environment to their personal preferences.

Conclusion

Visual Studio Code is more than just a text editor. It is a versatile and powerful development environment that strikes an excellent balance between simplicity and functionality. Its performance combined with its vast library of extensions and rich feature set makes it an outstanding choice for a wide range of developers, including those working on IBMi.

Setting Up VS Code for IBM i Development

To begin your journey of modern IBM i development, Visual Studio Code (VS Code) paired with the *Code for IBM i* extension offers a powerful and lightweight environment. This section covers the complete procedure for setting up VS Code—including configuration, connection, and basic usage—for IBM i development.

Step-by-Step Setup Procedure

1. Install and Launch VS Code

Download and install Visual Studio (VS Code). Launch the editor and you should see the welcome page.

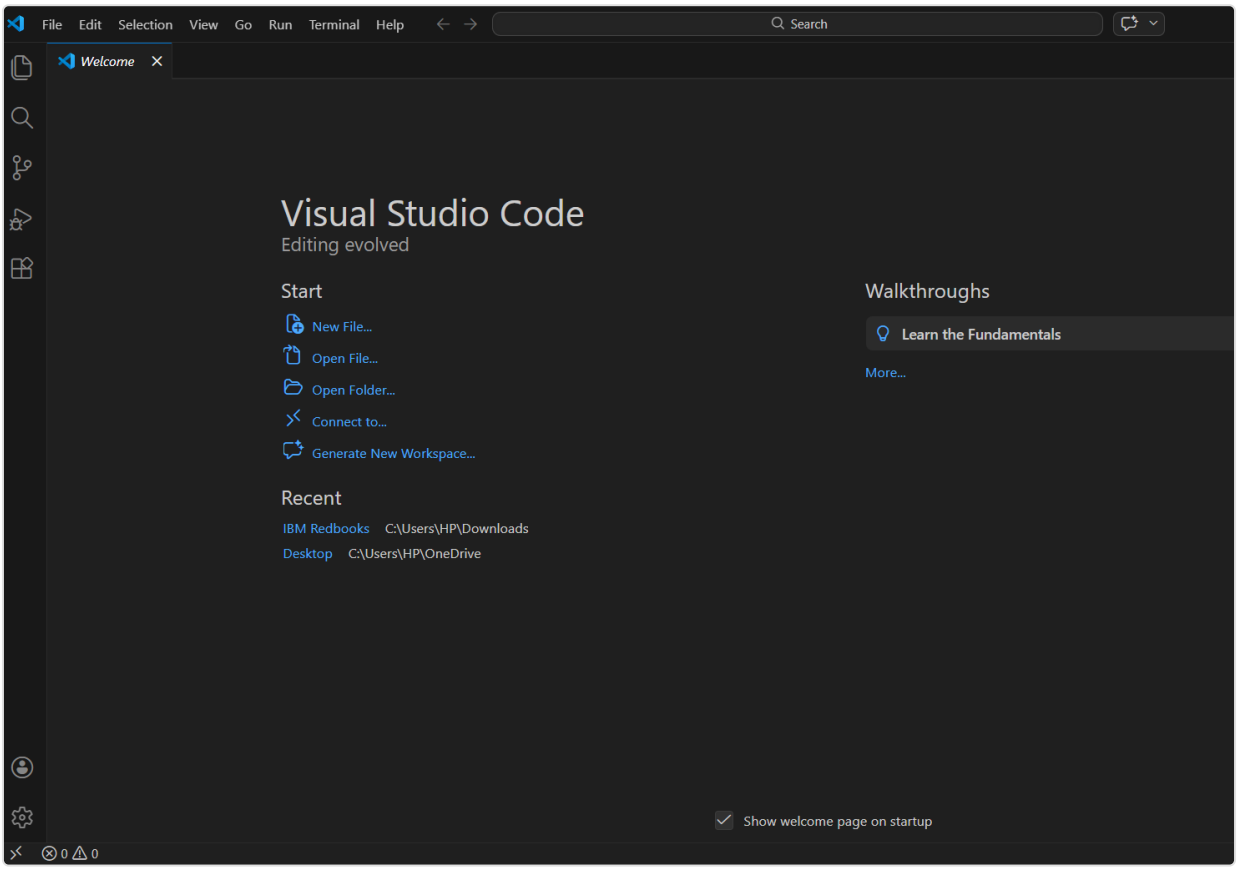


Figure. Visual Studio Code Welcome Page

2. Install the *Code for IBM i* Extension

1. Click on the **Extensions** icon.
2. Search for **IBM i** and choose **Code for IBM i** by Code4i. Alternatively, you can install the whole bundle with an extension named **IBM i Development Pack** which includes a set of extensions for IBM i development in VS code.
3. Click **Install**.

The extension adds a new activity icon (the small IBM i logo) to the side bar after installation.

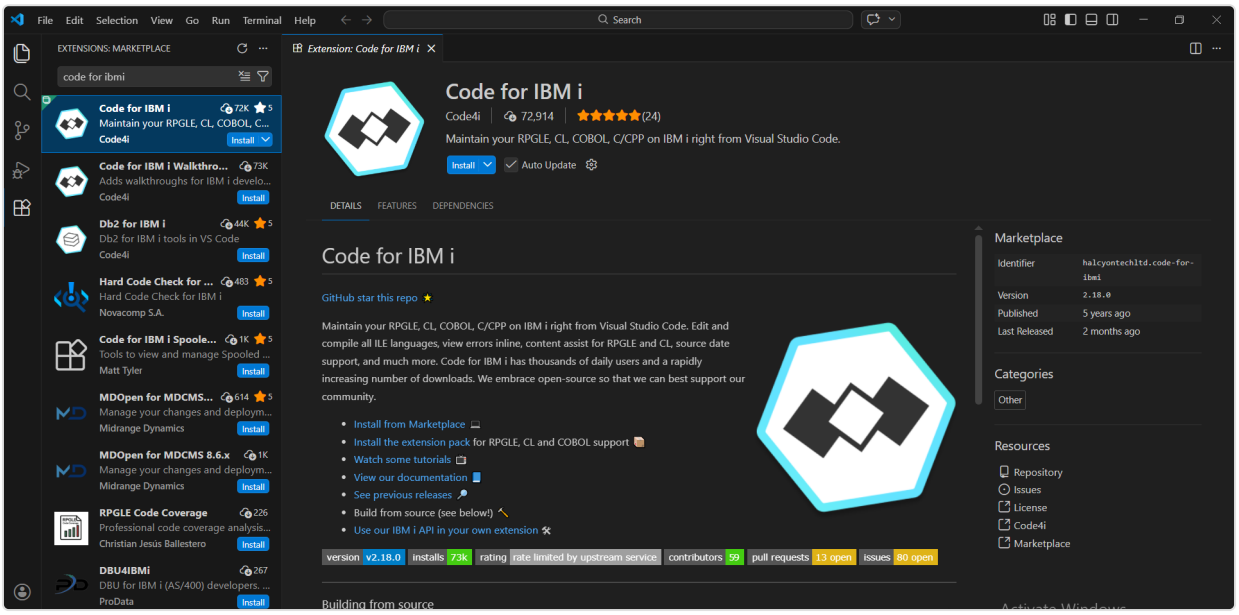


Figure. Code for IBM i Extension Installed

3. Create a Connection to IBM i Server in VS Code

1. Click the **IBM i** icon on the activity bar.
2. Select **Connect to an IBM i**.
3. Fill in the connection details:
   - **Connection Name**: A descriptive name for this connection
   - **Host or IP Address**: Your IBM i system's address
   - **Port (SSH)**: `22` (default). If your IBM i uses different port number for SSH, use that instead of 22.
   - **Username**: Your IBM i user ID
   - **Password or Private Key**: Enter your IBM i system password, or alternatively provide the path to a private SSH key file.
4. Click on Connect or Save.

> **Note:** Sometimes, the server might be configured to allow only public key authentication for SSH connection, in which case the password will not work. If you choose to use a private key, ensure that the corresponding public key is properly configured on the IBM i server for your user account.

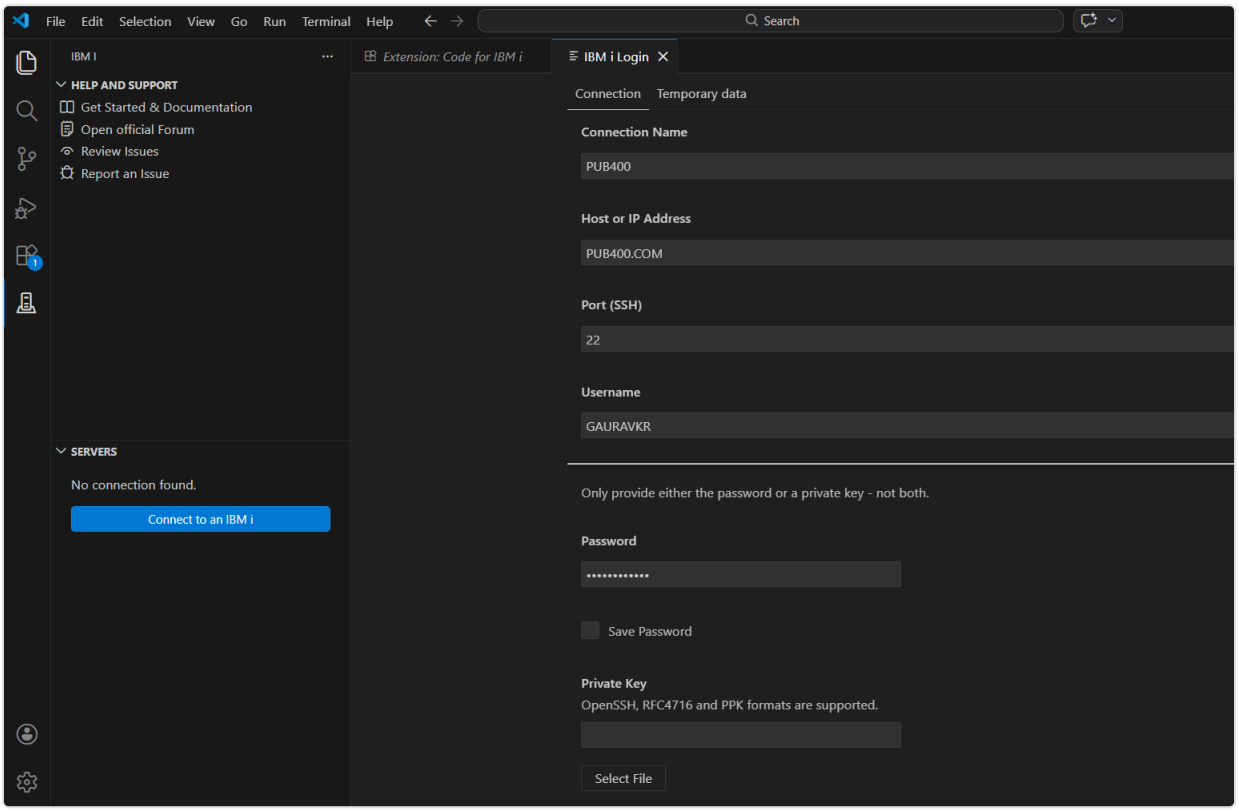


Figure. Connection to IBM i Server

The added connection appears under *Servers*.

4. Connect and Explore

1. Go to IBM i Icon where the saved connection is listed, Click on the **play** icon to connect.
2. It will prompt to enter your password again. After successful authentication, the *IBM i* view should display different sections named:
    - HELP AND SUPPORT
    - ENVIRONMENT
    - USER LIBRARY LIST
    - OBJECT BROWSER
    - IFS BROWSER

5. Browse Objects and Start Coding

User Library List: This section reflects the user library list and allows developers to control libraries that are accessible to a program. You can set a library as current library and add/remove libraries from the library list.

Object Browser: Use the Object browser to navigate through libraries, objects, and source physical files. You will need to create a filter to navigate the objects and source physical files inside libraries which can be managed easily. While creating a filter, you can choose to open the objects in Protected (read-only) mode by selecting the Protected button at the bottom.

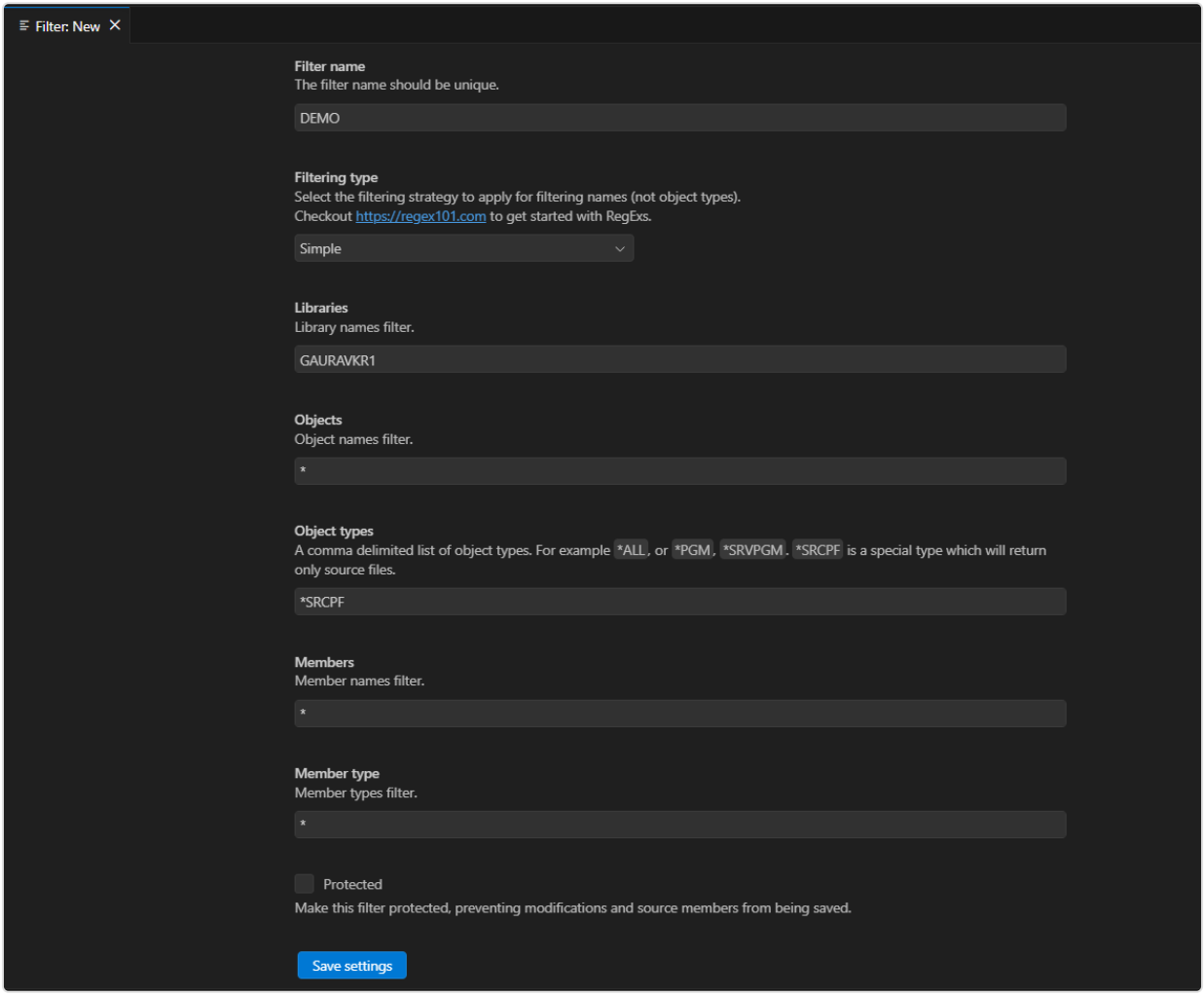


Figure. Creating filter to browse objects

Source Member Editing: Click on a source member to open it in the editor and it is ready for editing.

IFS Browser: The home directory browser is visible by default in this section. We can create new filters to access and edit files in the IFS directories similar to how we do on other operating systems.

Actions: Right-click on source members to see available actions, such as compiling the code.

6. Opening Terminals and Additional Settings

- **Terminal Type**: Hover over the IBM i settings icon displayed on the status bar to select the terminal to open (5250 or PASE).
- **Additional Settings**: Click on the IBM i settings icon to view or modify settings related to features, source code, terminals, debugger etc.

7. Testing Setup

After connecting, create a new source file by right-clicking the library and selecting the **New Source File**. To create a new Source Member, right-click the source file and select **New Member**. Alternatively, for modern development practices and easier integration with version control systems like GIT, consider creating new files with proper file extensions in the IFS Browser or directly in your local VS Code workspace, the extension supports compiling and debugging from these locations as well. Write a simple RPGLE program and compile it using the appropriate option in the Actions button or the `Call` command in the 5250 session.

Summary

This entire setup can be accomplished in under ten minutes. Once configured, VS Code becomes a powerful IDE for remote IBM i development—integrating editing, compiling, debugging, and source control in a familiar, extensible interface.

How do I use VS Code productivity features?

Visual Studio Code (VS Code) has revolutionized the way developers work with IBM i systems, transforming what was once a fragmented experience into a cohesive, modern development environment. Gone are the days of switching between multiple screens for editing, compiling, and debugging. Instead, VS Code integrates everything into a single platform that caters to both seasoned programmers and newcomers to the IBM i ecosystem. In this section, we'll explore the key productivity features that make VS Code an indispensable tool for IBM i development, from code development to database interactions and advanced debugging capabilities.

Code Development and Navigation

Developing code in VS Code feels intuitive and efficient.

Code Authoring Features

Content assist and autocomplete anticipate your needs, suggesting RPG syntax, variable declarations, and even SQL keywords as you type. For instance, start typing `dcl-s` for a standalone variable, and VS Code fills in the complete declaration structure. This feature is particularly valuable for learning modern RPG syntax or when switching between fixed and free formats.

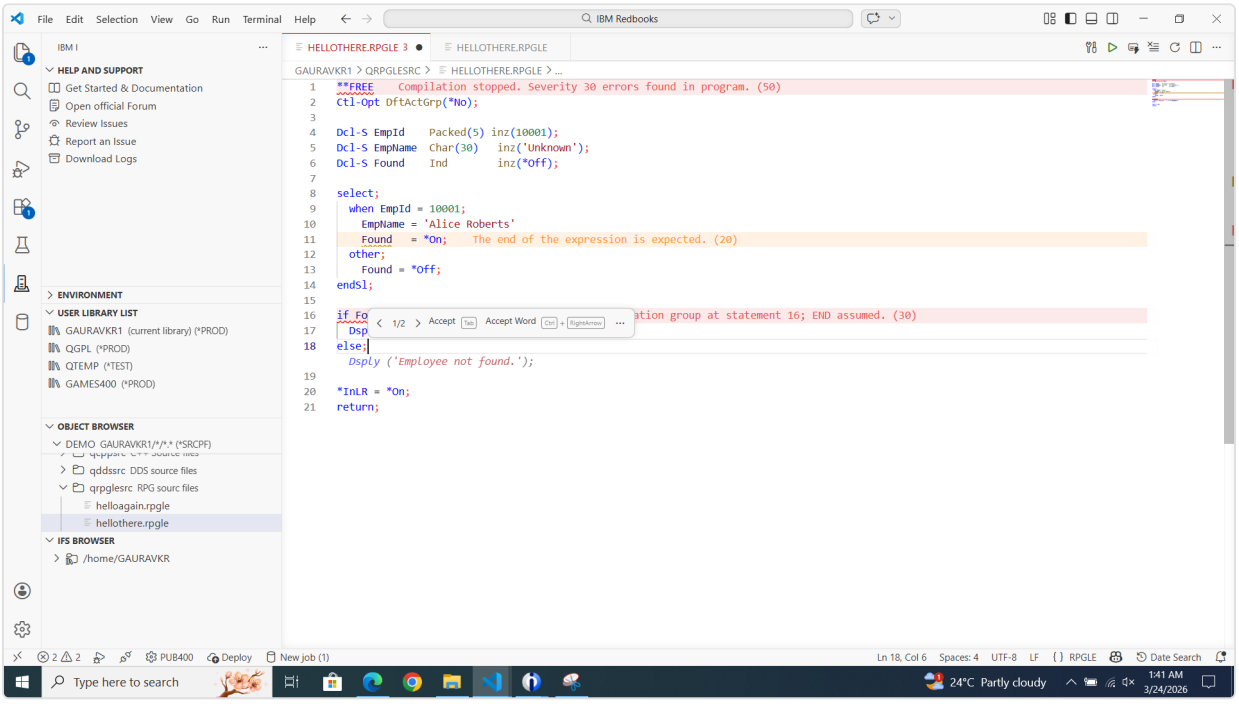


Figure. Content Assist and Syntax Check in Action

Syntax checking and linting features analyze your code in real-time, flagging potential issues like inconsistent indentation or naming conventions. The RPG linter, once enabled, provides immediate feedback as you write, preventing many compilation errors before they occur.

Snippets take this further by allowing you to create reusable code templates. Define a snippet for file headers with author information and copyright notices, or for common error-handling patterns. Once created, insert them with a few keystrokes, saving time on repetitive coding tasks.

Code Navigation and Exploration

Navigation features ensure you never lose your place in complex codebases. You can use 'Peek definition' to hover over a variable or procedure name to see its definition inline, without jumping to another file. You can use 'Find all references' to display every usage of a string across your project, helping you understand dependencies before making changes.

Source comparison tools are invaluable for version control and code reviews. Compare two source members side-by-side, with color-coded differences highlighting additions in green and deletions in red. This works for both source members and IFS files.

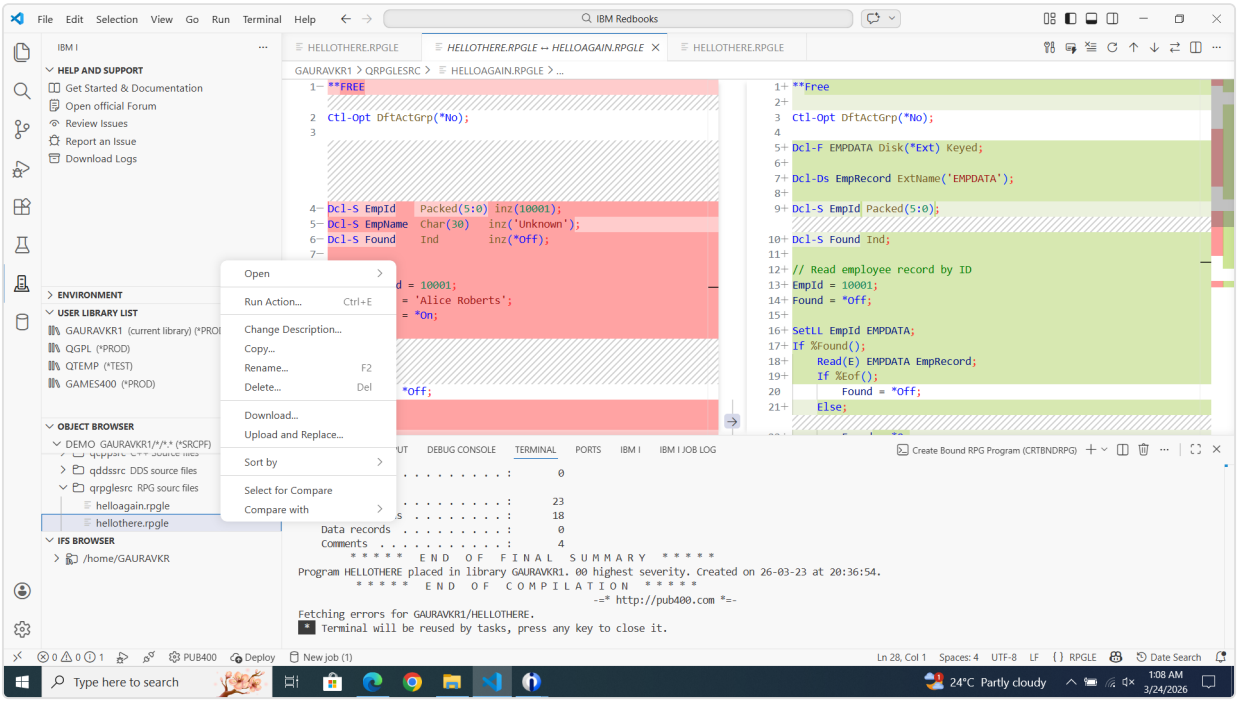


Figure. Source Comparison and Compilation View

Compilation and Build Tasks

Compilation streamline the development workflow by automating the build process directly from the editor. To compile a source member, simply right-click on the member in the Object Browser and select **Run Actions**. A menu will appear with available compilation commands suited to your source type—for example, CRTBNDRPG for RPG ILE programs, CRTCMD for CL commands, or CRTPF for physical files. Select the appropriate action, and VS Code launches the compilation on the IBM i system.

Furthermore, The extension provides a comprehensive default set of actions, but you can also edit and customize existing actions, or create new ones through the Environment view in the IBM i sidebar. Actions are stored in the `code-for-ibmi.actions` section of settings.json, which you can edit directly.

The compilation results are displayed in the integrated terminal output, providing real-time feedback identical to the spooled file format you'd see when compiling directly on IBM i. Error messages, warnings, and diagnostic information are formatted similarly to traditional compilation job logs, making it easy to identify and locate issues. You can click on error lines to jump directly to the problematic code in your editor, enabling rapid error correction and recompilation cycles. Protection modes at the connection or filter level prevent accidental changes to production systems.

Modernization and Code Transformation

For modernization efforts, VS Code excels at transforming legacy code. The extract procedure feature converts old-style subroutines into modern, free-form procedures, automatically identifying variables and creating proper parameter lists. This not only improves readability but also enhances testability and reusability.

Fixed-to-free RPG conversion addresses one of the biggest modernization challenges. There are extensions which can convert entire programs from fixed-format to free-format in seconds, preserving all business logic while updating syntax.

Database and System Integration

At the heart of VS Code's productivity for IBM i lies its seamless integration with system resources. The **DB2 for IBM i extension** stands out as a game-changer, allowing developers to interact with databases without ever leaving their code editor. Imagine browsing through libraries and tables directly in the sidebar, much like navigating folders in a file explorer. You can execute SQL queries with a simple `Ctrl+R`, view results in formatted tables, and even export data as CSV or JSON files—all within the same interface where you're writing your RPG code.

This extension doesn't just display data, it empowers you to manipulate it. To update employee salaries, write the select query with update directive and then you can just click on a cell, make the change, and commit it instantly. For performance tuning, the EXPLAIN feature analyzes query execution plans, showing you whether your SELECT statement is using indexes effectively or resorting to costly table scans. Content assist works here too, suggesting table names, column references, and SQL syntax as you type, reducing errors and speeding up query writing.

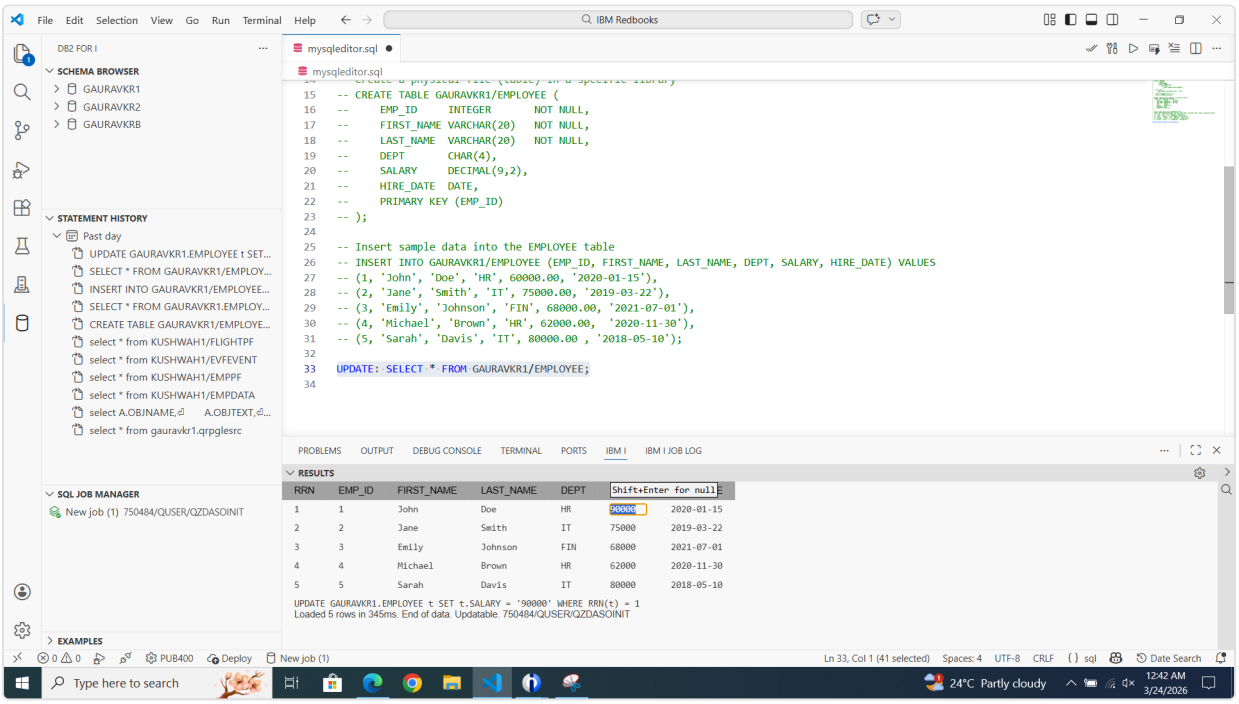


Figure. DB2 for IBM i Interface

Beyond databases, VS Code integrates deeply with the IBM i system itself. The Object Browser provides a hierarchical view of your libraries, source physical files, and members, complete with powerful filtering options. Want to focus only on RPG source files in a specific library? Create a filter, and suddenly your workspace becomes clutter-free. The browser also supports protected mode, perfect for reviewing client code or allowing stakeholders to explore source without risk of accidental changes.

Terminal integration brings the 5250 and PASE environments right into VS Code. Press `Shift+Ctrl+J`, select 5250 as terminal type and you're interacting with your IBM i system through familiar green-screen interfaces or select PASE Unix-like commands, all while maintaining your code context. For file management, the IFS Browser lets you upload, download, and edit files in the integrated file system, bridging the gap between traditional IFS subdirectories and modern file-based development.

Advanced Features: Debugging and Project Management

Debugging in VS Code brings professional-grade tools to IBM i development. Set breakpoints, step through code execution, and inspect variable values—all integrated with your source code. This makes troubleshooting complex RPG programs much more manageable than traditional debugging methods.

Source control integration with Git is seamless, allowing you to commit changes, view diffs, and manage branches directly from the editor. This is crucial for collaborative development and maintaining code history.

Finally, **AI integration** through tools like IBM Bob™ or GitHub Copilot provides intelligent code suggestions, helping developers write code faster and learn new patterns. This AI assistance can suggest complete function implementations or refactor existing code for better performance.

Embracing the Future of IBM i Development

VS Code's productivity features don't just make development faster—they fundamentally change how we approach IBM i projects. By eliminating the need for switching between different screens, reducing manual work, standardizing compilation, and providing powerful modernization tools, VS Code helps teams deliver higher-quality code more efficiently. For comprehensive guidance on VS Code setup, features, tips, and tricks, refer to the Code for IBM i documentation.

How do I migrate from PDM/SEU to VS Code?

What This Covers

This guide walks you through migrating from traditional IBM i development tools (PDM/SEU) to modern Visual Studio Code with IBM i extensions. You'll learn how to set up your development environment, understand the conceptual shifts, and establish workflows that leverage VS Code's powerful features while maintaining your IBM i development productivity.

Before You Start

- ☐ VS Code installed on your workstation (Windows, Mac, or Linux)
- ☐ Network access to your IBM i system (SSH port 22)
- ☐ IBM i user profile
- ☐ IBM i 7.4 or higher
- ☐ SSH daemon (SSHD) running on IBM i (`STRTCPSVR SERVER(*SSHD)`)

Steps

1. **Install Code for IBM i Extension**
   - Open VS Code and navigate to the Extensions view (Ctrl+Shift+X or Cmd+Shift+X)
   - In the extensions, search for IBM i Developer Pack and install it

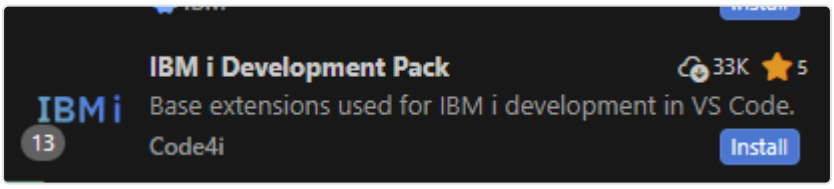


Figure. IBM i Developer Pack Extension

- What to expect: this extension package adds IBM i-specific functionality to VS Code, including connection management, source member editing and syntax highlighting

2. **Create Your First IBM i Connection**
   - Click the `IBM i` icon in the Activity Bar (left sidebar)

     Figure. IBM i Icon in Activity Bar

   - In the `Servers` view, click on the `+` to create a new connection

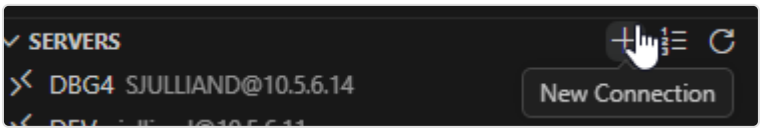
     

     Figure. Create New Connection

   - Enter connection details:
     - Host: Your IBM i system name or IP address
     - Port: 22 (default SSH port)
     - Username: Your IBM i user profile
     - Password or SSH key authentication
   - Save the connection for future use and click on the newly created connection item
   - What to expect: VS Code establishes an SSH connection to your IBM i system

3. **Configure Your Library List**
   - After connecting, open the IBM i view
   - Go to the `User Library List` view
   - Add your development libraries using the `+`
   - Set your current library
     - You can reorder the library list and the set the current library using drag and drop
   - What to expect: This replaces the PDM library list (WRKLIBPDM) and determines which libraries are used during the development and during compilation phases

4. **Set Up Source Physical File Filters**
   - In the `Object Browser`, click on `New Filter...`

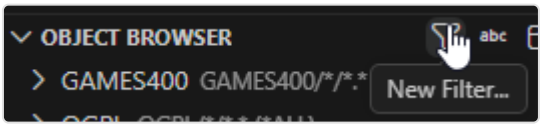
     

     Figure. New Filter in Object Browser

   - Define filters to display your source physical files
     - Leave `Filtering type` on `Simple`
     - Set `Object Types` to `*SRCPF`

- What to expect: Filters replace PDM's WRKMBRPDM command, giving you quick access to source members organized by file and library

5. **Open and Edit Your First Source Member**
   - Expand a filter to see your source physical files
   - Expand a source physical file to list its source members (like PDM's member list)

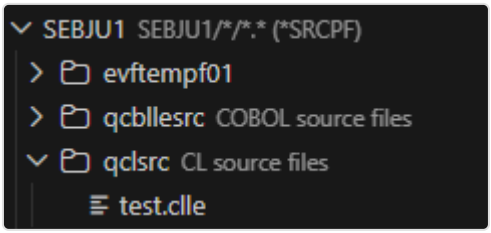


Figure. Source Physical File Members

   - Click on a member to open it in the editor
     - Notice syntax highlighting
   - Make a simple change to test the editor
   - Save with `Ctrl+S` (`Cmd+S` on Mac)
   - What to expect: The member opens in a modern editor with features SEU never had—syntax highlighting, multiple cursors, find/replace with regex, and more. Changes are saved directly to the IBM i system

6. **Learn the New Compile Workflow**
   - With a source member open, click on the `Run Action...` button in the top toolbar

Figure. Run Action Button

   - Select one of the available compile commands
   - View compile results
     - the Problems panel will be populated with compilation errors
     - the compiler's output can be opened by clicking on the `Open output` button in the toaster notification
   - Click on errors to jump directly to the problem line
   - What to expect: This replaces SEU's F3 compile. Errors appear as clickable links, making it much faster to navigate and fix compilation issues than reviewing a spooled file

7. **Configure Custom Actions for Your Build Process**
   - Go to the `Environment` view and expand the `Actions` node

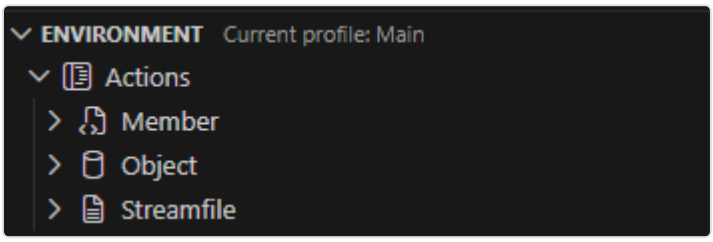


Figure. Environment Actions View

- Next to the `Member` category, click on the `+` to create a new action
- Add a custom compile commands matching your shop standards

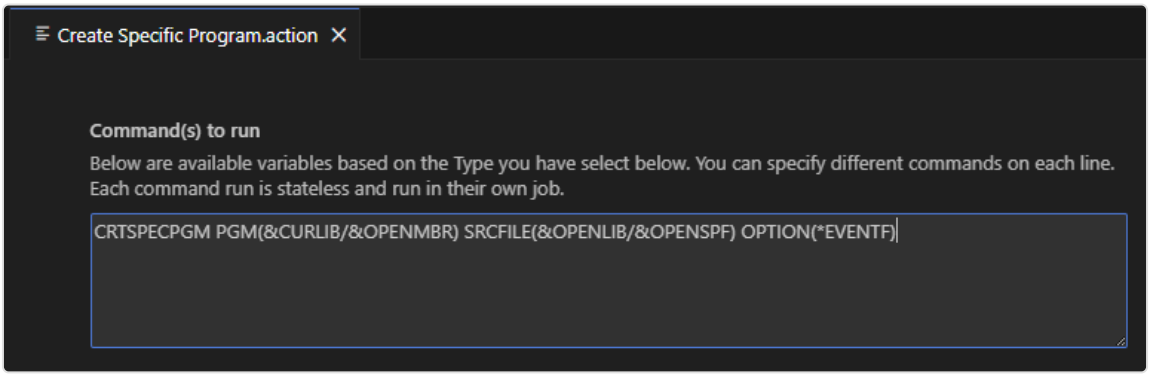


Figure. Custom Action Configuration

- Save with `Ctrl+S` (`Cmd+S` on Mac)
- What to expect: Custom actions replace PDM user-defined options

8. **Explore the IFS Browser**
   - Go to the `IFS Browser` view
   - Navigate the `/home/<your profile>` shortcut
   - Right-click to create, delete, or edit IFS files
   - Click on the `+` button to add new shortcuts

Figure. IFS Browser Add Shortcut

   - Edit stream files directly in VS Code by clicking on them
   - Run actions the same way you did on QSYS members
   - What to expect: This replaces WRKLNK and provides a modern file explorer for the IFS, essential for modern IBM i development with Node.js, Python, or web applications

Verify It Worked

- ☐ You can connect to your IBM i system from VS Code
- ☐ Your library list is configured and displays correctly
- ☐

You can browse and open source members from the `Object Browser`

- ☐
  You can edit a source member and save changes successfully
- ☐
  You can compile a program and see errors in the Problems panel
- ☐
  You can browse and edit IFS files from the `IFS Browser`

Key Takeaways

- **VS Code is a file-based editor**: Unlike SEU which edits members directly on IBM i, VS Code downloads members to your workstation, edits them locally, and uploads on save. This enables features like undo/redo across sessions and offline editing.
- **Filters replace PDM lists**: Instead of WRKMBRPDM, you create reusable filters that appear in your sidebar for quick access to frequently used source files.
- **Actions replace PDM options**: Custom actions (configured in settings) replace PDM's numbered options and user-defined options, providing consistent compile commands.
- **The Problems panel is your new compile listing**: Compilation errors appear as clickable links in the Problems panel, making error resolution much faster than reviewing spooled files.
- **It's a learning curve, but worth it**: Expect 1-2 weeks to become comfortable with VS Code. The productivity gains—syntax highlighting, IntelliSense, multi-cursor editing, and integrated debugging—far outweigh the initial learning investment.
- **You can use both during transition**: Many shops run PDM/SEU and VS Code side-by-side during migration. There's no requirement to switch everything at once.
- **Community support is strong**: The Code for IBM i extension has an active community on GitHub, Zulip, and LinkedIn. Don't hesitate to ask questions or request features.
- **Online documentation is comprehensive**: The Code for IBM i extension has extensive documentation, including tutorials, reference materials, and a knowledge base. For more information, see Code for IBM i.

Alternative IBM i IDEs

MiWorkplace: A Modern, Free IDE for IBM i Development

Introduction

For a long time, development on the IBM i platform has been strongly associated with traditional tools such as SEU (Source Entry Utility) and green-screen interfaces. While robust, these tools no longer meet the expectations of modern developers accustomed to graphical, extensible, and productivity-oriented environments. In response to this shift, new Integrated Development Environments (IDEs) have emerged to modernize the IBM i development experience. One such tool is **MiWorkplace**, a lightweight, free IDE developed and maintained by Remain Software.

MiWorkplace positions itself as a modern alternative to legacy tooling, offering a streamlined, cross-platform development environment tailored specifically for IBM i systems. It combines ease of use with a rich feature set, making it an attractive option for both experienced RPG developers and newcomers to the platform.

MiWorkplace used to be called RPG Next-Gen until original author **Mihael Schmidt** decided to make a commercial version. Later, Remain Software took over maintenance of MiWorkplace and they have since released it as freeware. MiWorkplace can be downloaded from https://miworkplace.com.

A Purpose-Built IDE for IBM i

Unlike generic IDEs adapted for IBM i, MiWorkplace is designed from the ground up for this ecosystem. It is a **fully integrated development environment**.

The IDE is built on Eclipse technology, giving it a familiar structure for developers who have used tools like IBM Rational Developer for i (RDi). However, MiWorkplace differentiates itself through simplicity, performance, and accessibility. It is a **lightweight IDE** that runs on Windows, macOS, and Linux, requiring no server-side installation.

This architecture lowers the barrier to entry significantly. Developers can install and start using the tool quickly without complex setup procedures, making it especially appealing for teams looking to modernize incrementally.

Core Concept: The Remote Explorer

At the heart of MiWorkplace lies the **Remote Explorer**, which acts as the primary interface between the developer and the IBM i system.

Through the Remote Explorer, users can:

- Connect to IBM i servers
- Navigate QSYS libraries and the Integrated File System (IFS)
- Manage source members and stream files
- Perform file operations such as create, rename, copy, and delete

This centralized approach simplifies interaction with IBM i resources. Instead of switching between multiple tools or command-line interfaces, developers can manage everything from a single graphical view.

A particularly powerful feature is the use of **filters**, which allow developers to limit visible objects based on criteria. This is crucial in large enterprise environments where libraries may contain thousands of objects. Filters can be layered and toggled dynamically, improving navigation efficiency.

Advanced Editing Capabilities

Editing source code is the core function of any IDE, and MiWorkplace provides a robust set of features tailored to IBM i languages such as RPG, CL, and DDS.

Key capabilities include:

- Syntax highlighting
- Code completion (including opcodes and built-in functions)
- Outline views for navigating code structure
- Hover-based documentation for variables and procedures
- Support for both fixed and free-format RPG

These features significantly enhance developer productivity compared to traditional editors. The inclusion of templates and snippets further accelerates development by reducing repetitive coding tasks.

Compilation and Debugging Workflow

MiWorkplace integrates closely with the IBM i runtime environment. While it does not compile code locally, it provides seamless mechanisms to **trigger compilation on the server**.

Developers can:

- Start compilation from the UI or context menus
- Configure compile commands and parameters
- Use variables and placeholders in compile options
- View results directly within the IDE

The results of compilation are automatically retrieved and displayed, enabling quick feedback loops.

In addition, the IDE integrates with the **IBM i System Debugger**, allowing developers to debug applications efficiently. Combined with job log analysis tools, this creates a cohesive development and troubleshooting workflow.

Job and System Interaction

A distinguishing feature of MiWorkplace is its deep integration with IBM i system concepts such as jobs and spooled files.

The IDE allows developers to:

- Monitor active jobs
- Filter and inspect job logs
- Analyze messages for debugging
- Access and manage spooled files

These capabilities are essential for real-world IBM i development, where understanding job behavior is often critical to diagnosing issues. By integrating these features directly into the IDE, MiWorkplace eliminates the need for separate tools or terminal sessions.

Extensibility and Automation

MiWorkplace includes several features aimed at improving developer efficiency through customization and automation:

- User-defined actions to automate repetitive tasks
- Internal scripting capabilities
- Configurable templates and snippets

Furthermore, the IDE integrates with documentation tools such as ILEDocs, allowing developers to generate and upload documentation directly from their codebase.

Extensible

Since MiWorkplace is Eclipse based, it enable installation of existing Eclipse plugins.

Github Copilot

One notable addition to MiWorkplace is the integration of **AI-assisted tooling**, including Github Copilot

This reflects a broader trend in software development, where AI is increasingly used to:

- Provide contextual assistance
- Suggest code improvements
- Accelerate learning for new developers

By embedding AI capabilities directly into the IDE, MiWorkplace aligns itself with modern development expectations and helps bridge the knowledge gap often associated with IBM i systems.

DBeaver

DBeaver Community is a free, open-source database management. Manage and explore SQL databases like DB2/400, MySQL, MariaDB, PostgreSQL, SQLite, Apache Family, and more.

- Support for many relational databases
- State of the art Data Editor
- Powerful SQL Editor
- Database schema editor
- DDL view and edit
- Spatial data viewer
- Basic AI assistant for SQL generation
- Basic AI engines (OpenAI, GitHub Copilot)
- Basic ER Diagrams

- Basic charts
- Data export/import in multiple formats
- Task management
- Database maintenance tools
- SSH and proxy
- Basic and SSL authentication

Position in the IBM i Ecosystem

MiWorkplace occupies an interesting position within the IBM i development landscape.

Traditionally, developers have relied on:

- SEU (green-screen editor)
- IBM Rational Developer for i (RDi)
- Visual Studio Code with IBM i extensions

MiWorkplace offers a middle ground:

- More modern and user-friendly than SEU
- Simpler and more lightweight than RDi
- More specialized for IBM i than generic VS Code setups

It is particularly appealing for developers seeking a **free or low-cost alternative** to commercial IDEs like RDi while still benefiting from a rich feature set.

Strengths and Limitations

Strengths

- Lightweight and easy to install
- Cross-platform support
- Deep IBM i integration
- Modern editing features
- Built-in job and system tools
- Free support for core functionality

Limitations

- Less feature-rich than enterprise tools like RDi
- Smaller ecosystem compared to VSCode
- Relies on server-side compilation

Despite these limitations, MiWorkplace strikes a strong balance between functionality and usability.

Conclusion

MiWorkplace represents a significant step forward in modernizing IBM i development. By combining a lightweight architecture with powerful, IBM i-specific features, it provides a compelling alternative to both legacy tools and heavyweight IDEs.

Its emphasis on usability, integration, and modern capabilities—such as AI assistance and cross-platform support—makes it particularly well-suited for teams looking to evolve their development practices without abandoning the strengths of the IBM i platform.

IBM i Development Tools

What is IBM i Access Client Solutions?

IBM i Access Client Solutions (ACS) is a modern client interface and toolkit designed to provide seamless access to IBM i systems. It is developed and maintained by IBM and serves as a comprehensive solution for system administrators, developers, and operators who need to interact with and manage IBM i resources efficiently.

Overview and Purpose

IBM i Access Client Solutions is a replacement for the traditional IBM i Access for Windows client. It represents IBM's commitment to modernizing IBM i connectivity by providing a contemporary platform that works across multiple operating systems. This cross-platform compatibility eliminates the traditional dependency on Windows-only solutions, allowing organizations to standardize their development and administrative tools across diverse environments.

Core Components and Features

ACS comprises several integrated components that work together to provide comprehensive IBM i system access:

5250 Terminal Emulation: ACS includes a sophisticated terminal emulator that supports traditional 5250 green-screen applications. This allows developers and operators to run existing character-based applications while benefiting from modern interface enhancements, keyboard shortcuts, and improved customization options.

IBM i Navigator: The navigator component provides a graphical interface for managing IBM i system resources, including libraries, files, job queues, and system configuration. Users can perform administrative tasks without requiring deep command-line knowledge, making system management more accessible.

Database Access: ACS facilitates access to IBM i databases (DB2 for i), enabling developers to browse database structures, execute SQL queries, and manage database objects directly from the client interface. This is particularly valuable for database administrators and developers working with DB2 for i.

Transfer Services: The tool includes robust file transfer capabilities, allowing users to move data between local machines and IBM i systems with ease. This supports both ASCII and EBCDIC file handling, accommodating the different data formats between systems.

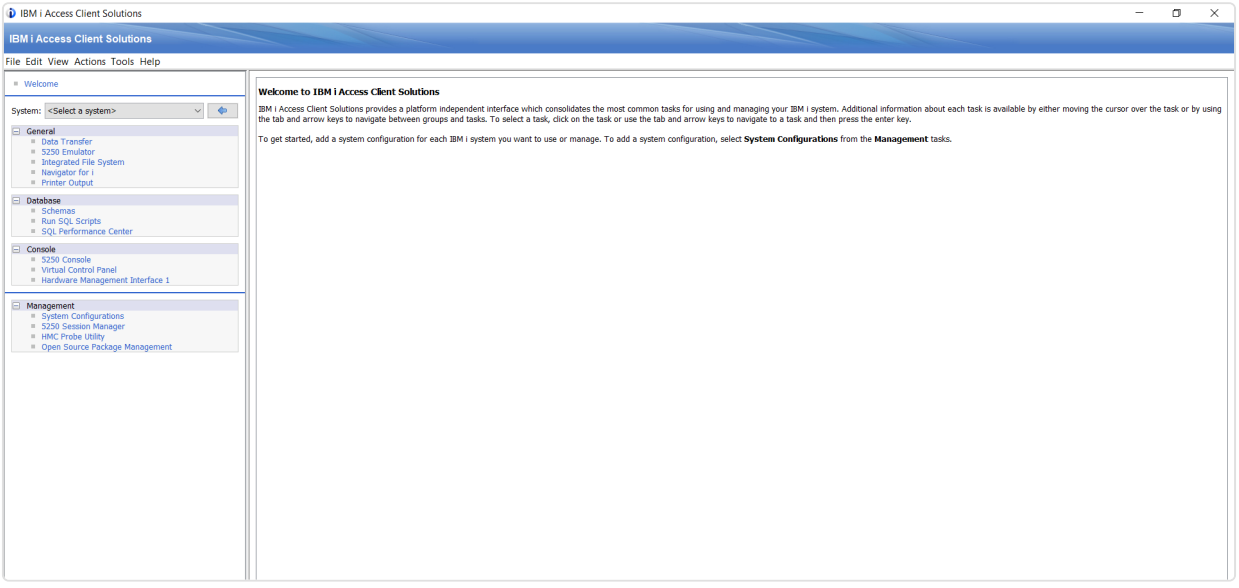


Figure. IBMi Access Client Solutions

Interface Organization

When users launch ACS, they access a web-based interface organized into categories on the left sidebar. Each category groups related tools to help users find what they need. Below is an overview of the main interface sections and their tools:

General Category

This section provides essential tools for system interaction and file management:

- **Data Transfer**: Secure file transfer between your local machine and the IBM i system with support for various formats and protocols.
- **5250 Emulator**: Terminal emulator for traditional 5250 green-screen applications with modern enhancements.
- **Integrated File System**: Browse and manage files in the IBM i Integrated File System (IFS).
- **Navigator for i**: Graphical interface for exploring and managing IBM i system resources.
- **Printer Output**: Manage spool files and printer output from the IBM i system.

Database Category

This section focuses on database administration and SQL operations:

- **Schemas**: Visual browser for exploring database schemas, tables, and views in DB2 for i.
- **Run SQL Scripts**: SQL editor and execution environment for running queries and database maintenance tasks.
- **SQL Performance Center**: Tools for analyzing and optimizing SQL performance.

Console Category

This section provides system console and hardware management:

- **5250 Console**: Direct access to the system console for low-level system operations.
- **Virtual Control Panel**: Web-based interface for managing system partitions and IPL operations.
- **Hardware Management Interface 1**: Manages system hardware components and monitoring.

Management Category

This section contains advanced administrative and configuration tools:

- **System Configurations**: Manage system-wide settings, network configurations, and security policies.
- **5250 Session Manager**: Monitor and manage active 5250 sessions.
- **HMC Probe Utility**: Diagnose and troubleshoot Hardware Management Console connections.
- **Open Source Package Management**: Manage open-source packages on the IBM i system.

Installation and Initial Setup

Installing IBM i Access Client Solutions

To install ACS, visit the IBM website and download the appropriate version for your operating system. ACS is distributed as a Java-based application and requires Java Runtime Environment (JRE) version 8 or higher. Simply download and run the JAR file directly—no traditional installation is needed. Double-click the downloaded JAR file to launch it. This starts the ACS client and opens the ACS interface.

Connecting to an IBM i System

Upon first launch, establish a connection to your IBM i system by navigating to the connection settings under System Configurations and entering system's hostname or IP address and port. Provide valid user credentials. Configure SSL/TLS for secure connections and import certificates if needed for self-signed certificates. Use the "Test Connection" feature to verify connectivity before saving your configuration. ACS remembers your settings, enabling quick access in future sessions, and allows you to configure multiple system connections if needed.

Importance in the IBM i Ecosystem

For IBM i users, ACS is a critical tool that helps preserve investment in existing IBM i applications while modernizing the development and administrative experience. As enterprises increasingly require cross-platform development capabilities and modern development practices, ACS serves as a bridge between legacy IBM i systems and contemporary tooling.

Advanced Development Environments

What is RDi?

What This Covers

This guide explains Rational Developer for i (RDi), IBM's integrated development environment for IBM i systems, including what it is and why it's important for IBM i development.

What It Is

RDi (Rational Developer for i) is an integrated development environment and a modern alternative to PDM/SEU. It's an IDE based on Eclipse, an open source development environment for many languages on many different platforms.

RDi runs on Java and is available for macOS and Windows.

Why It Matters

RDi is a good option for IBM i development because it:

- **Modernizes Development**: Replaces outdated green-screen editors with a modern graphical interface featuring syntax highlighting, code completion, and refactoring tools
- **Increases Productivity**: Provides integrated debugging, source code management, and direct system connectivity, eliminating the need to switch between multiple tools
- **Bridges Legacy and Modern**: Enables developers to maintain existing RPG/COBOL applications while also building new web services and modern integrations
- **Reduces Errors**: Offers real-time syntax checking, code validation, and interactive debugging

How It Works

RDi connects directly to IBM i systems over the network, allowing developers to:

1. **Edit Source Code**: Write and modify programs locally with advanced editor features like IntelliSense, syntax highlighting, and code templates
2. **Compile Remotely**: Send code to the IBM i system for compilation and receive immediate feedback on errors
3. **Debug Interactively**: Set breakpoints, step through code, inspect variables, and control program execution in real-time using the Debug As A Service (DAAS) component
4. **Manage Resources**: Browse libraries, files, and objects on the IBM i system through the Remote Systems Explorer
5. **Integrate with Tools**: Connect to version control systems (Git, SVN), issue trackers, and other development tools within the same environment

The architecture uses a client-server model where RDi (the client) communicates with debug servers and system services running on the IBM i host, enabling seamless remote development and debugging.

Features highlight

Multi-language support

RDi natively supports the languages commonly found and used on IBM i:

- RPG (RPG IV, ILE RPG)

- COBOL (ILE COBOL)
- CL (Control Language)
- DDS (Data Description Specifications)
- SQL
- Java
- C/C++

Supporting these languages means that the editors in RDi provide syntax checking, code coloration and content assist out of the box.

Integrated graphical debugger

The DAAS (Debug As A Service) component enables remote debugging capabilities using Eclipse's PDT (Platform Debug Tools), allowing developers to debug IBM i applications from RDi through the Debug Adapter Protocol (DAP).

The debugger allows to debug batch calls as well as setting Service Entry Points (SEP). The SEP let the user starts a debug session whenever the conditions set in the SEP are met. These conditions include:

- Which program/module/procedure is called
- Which user is calling it
- What entry parameters value is expected to start the debug session

Screen Designer

RDi integrates a WYSIWYG editor to replace the now retired SDA tool.

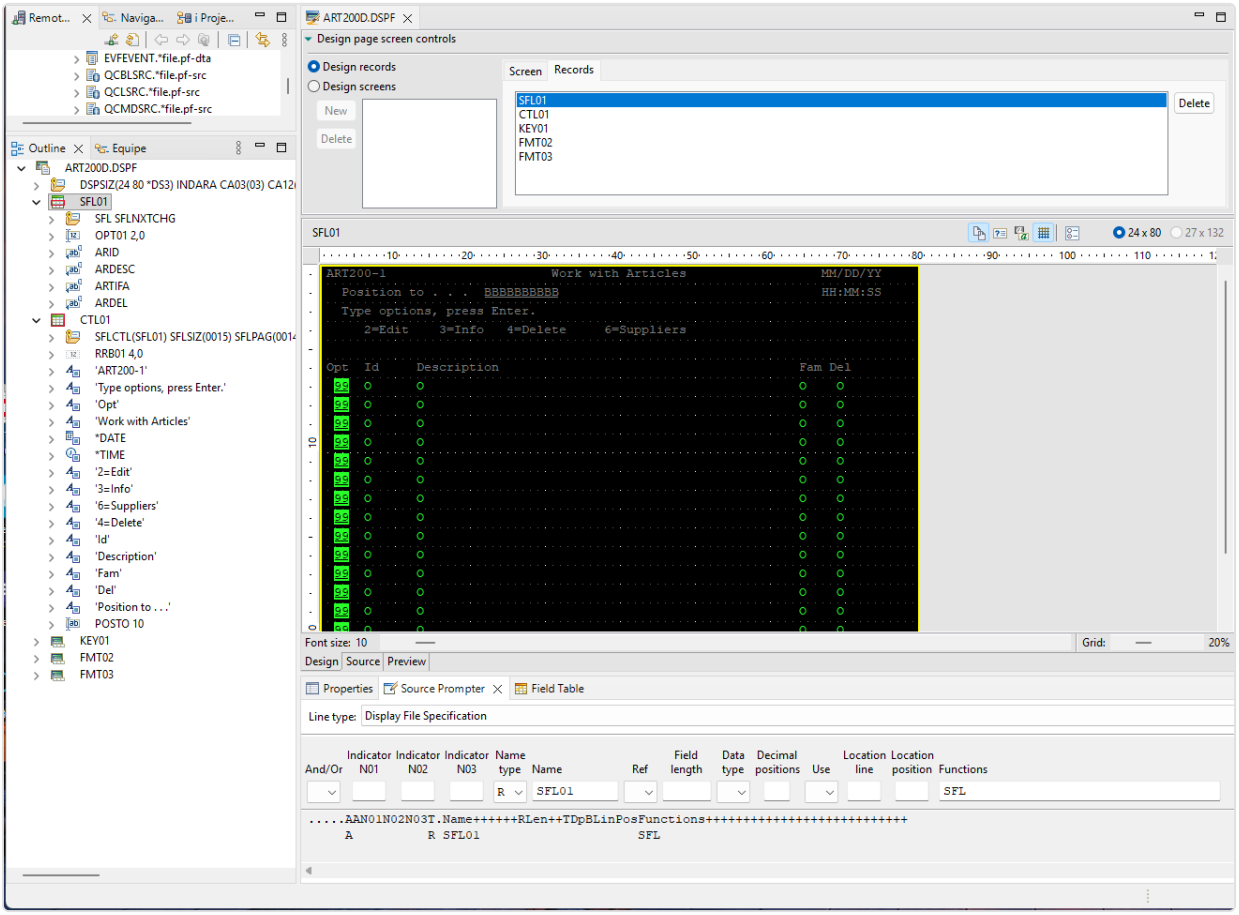


Figure. RDi Screen Designer

It is a visual design tool that allows developers to create and modify 5250 display files (DSPF) using a graphical interface instead of manually coding DDS (Data Description Specifications). Developers can drag and drop fields, format screens visually, and see exactly how the green-screen interface will appear to end users.

Columned RPG Block-nesting/indentation highlight

This feature helps developers understand the structure of columned RPG code by highlighting matching code blocks and their nesting levels.

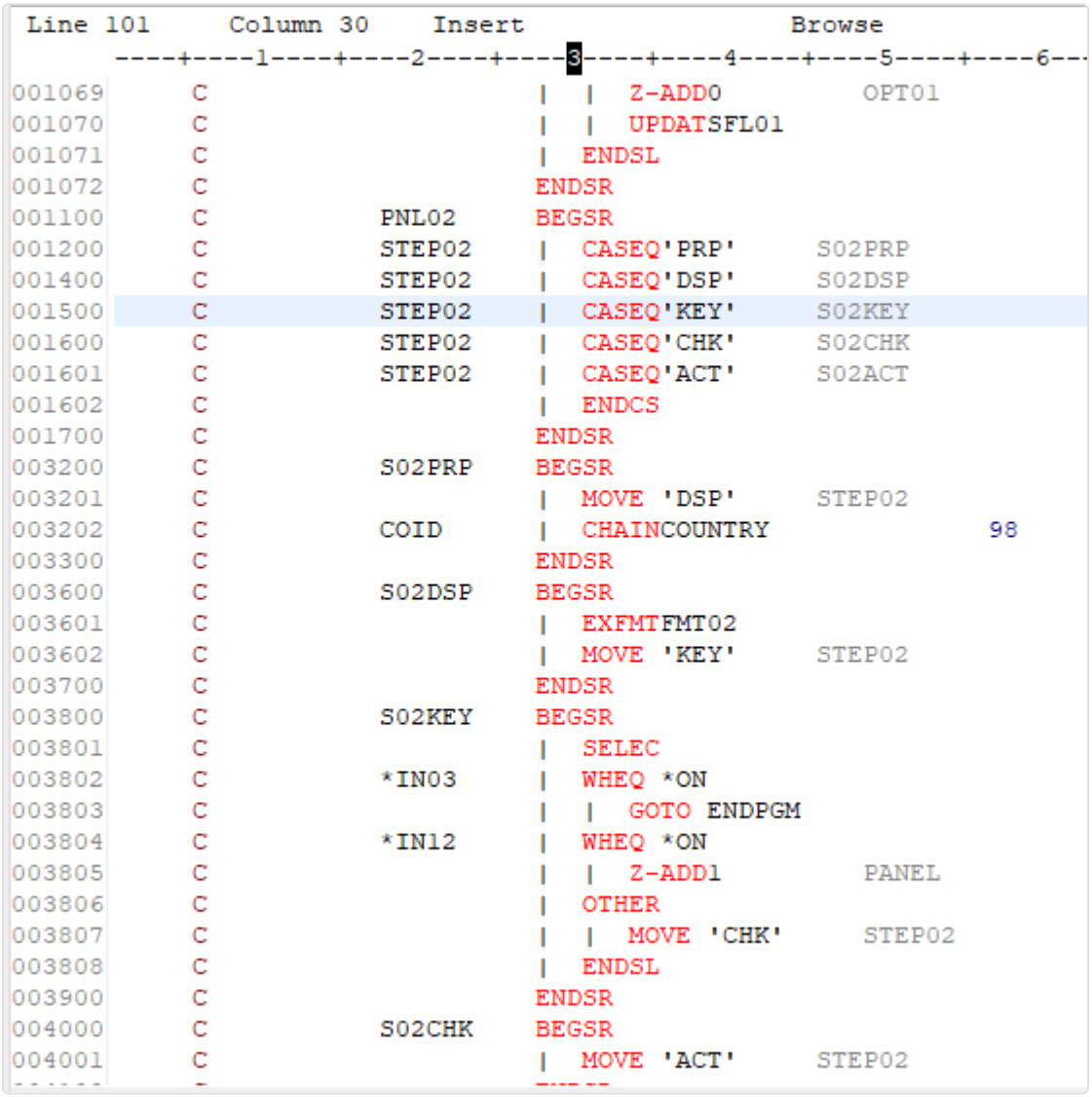


Figure. RPG Block-nesting Highlight

It addresses a unique challenge in RPG programming as it uses a fixed-column format where different parts of the code (specifications, operations, factors) are placed in specific column positions rather than using free-form indentation like modern languages. It makes it easier for developers to follow the code flow thanks to this tool.

RPGLE Outline

The Outline view in RDi supports RPGLE. It provides a structured breakdown of the program.

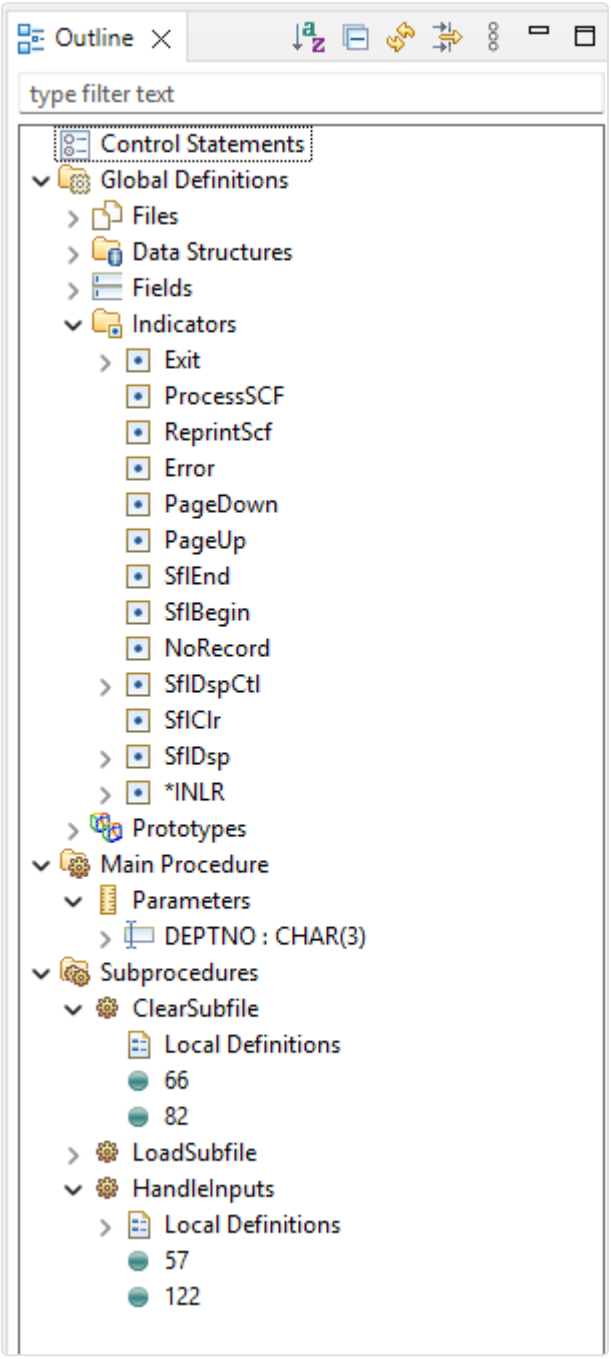


Figure. RPGLE Outline View

Elements are organized hierarchically (e.g. subprocedures appear under their parent procedure, fields appear under their data structure). Clicking any item in the outline instantly jumps to that definition in the source code.

Future considerations

It's been the recommended IBM i developer IDE for the past decade, but it's been losing market shares with VS Code with which it's now on par as of writing. It's still a good choice for coding on IBM i since it's packed with productivity features.

Key Takeaways

- RDi is IBM's official Eclipse-based IDE for IBM i application development and maintenance

- It supports multiple languages including RPG, COBOL, CL, SQL, DDS, Java, and C/C++
- The tool provides modern development features like intelligent code completion, refactoring, and interactive debugging for legacy systems
- RDi is the successor to older tools like WDSC and CODE/400, offering significantly improved productivity and developer experience
- It's a good option for organizations maintaining or modernizing IBM i applications

What is the Visual Studio Code for IBM i - Database Extension?

The **Code for IBM i** offering within the Visual Studio Code marketplace has become the most popular IDE for IBM i application development. As a vibrant open source project, Code for IBM i (aka Code4i) is not just widely used, but also constantly being enhanced to meet the needs and wishes of its user community.

The foundational Code4i offering can be found here: https://marketplace.visualstudio.com/items?itemName=HalcyonTechLtd.code-for-ibmi

Code4i - Database Extension

The **Database Extension** with Code4i provides a feature-rich tool for application developers to build, revise, or test their latest and greated SQL statements. The strategy for this extension is to 'provide an awesome user experience for the IBM i application developer". A passionate group of contributors have taken the Database Extension from its modest initial offering into a best-of-breed tool for the SQL developer.

The Database Extension includes **major features** such as:

- **Schema Browser**
- **Statement History**
- **SQL Job Manager**
- **Examples**
- **Text Files (SQL Scripts, Notebooks, Markdown files, etc)**
- **SQL Job Manager**
- **Syntax Checking Problems**
- **SQL Error Logging Facility (SELF) live report**
- **And much more...**

For more information, see The Database Extension.

Code4i Database Extension

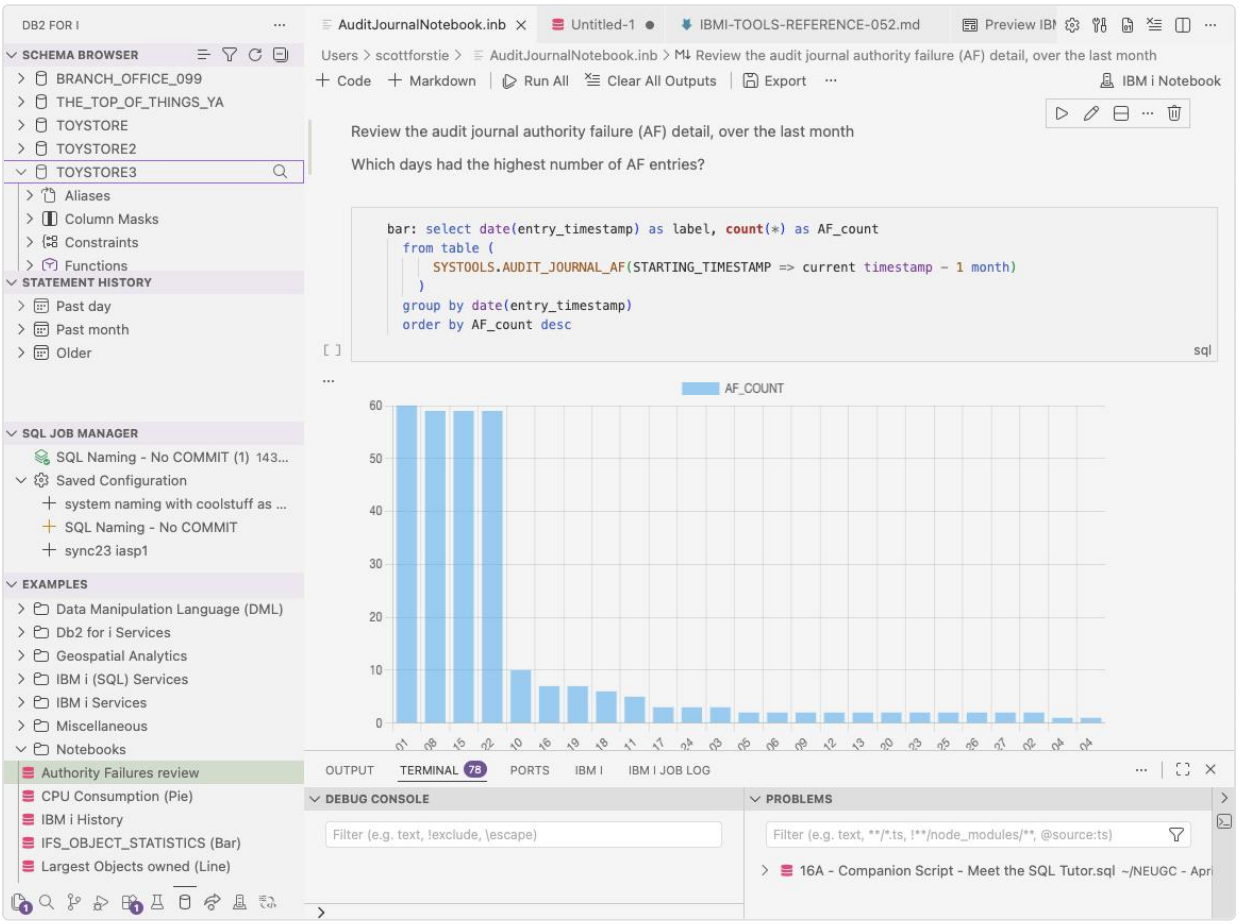


Figure. Code4i Database Extension

Database Extension – Documentation

If the Database Extension is new to you, consider taking a few minutes to review the documentation. The documentation covers many of the great features found within the database extension, and allows you to avoid saying, "I never knew that feature existed!".

For more information, see The Database Extension.

Database Extension – SQL Prefixes

In its simplest form, the Database Extension is a feature-rich environment to construct and test SQL statements. When the SQL statement is a query, the results appear to the user. The extension allows the user to go above and beyond, with the following SQL prefixes.

Database Extension Prefixes:

- **json:** - The query results are rendered using JSON, in a new file.
- **csv:** - The query results are rendered using comma-separated form, in a new file.
- **sql:** - The query results are used to construct an INSERT with VALUES (SQL) statement, in a new file.
- **rpg:** - The query results are used to construct an RPG data structure, in a new file.
- **udtf:** - The query results are used to construct a User Defined Table Function (UDTF), in a new file.
- **cl:** - Enables execution of a CL command.
- **update:** - The query results are rendered in a form that the results are comprised of editable cells.

- **bind:** - Causes the extension to seach backwards in the script for an SQL statement. The SQL statement is executed, with the bind parameter values being substituted for host variables (:H) or parameter markers (?), moving from the start of the statement text and substituting from the left to the right.
- **bar:**, **line:**, **doughnut:**, **pie:**, **polar area:**, **radar:** - Are used within IBM i Notebooks (.inb files) to visually graph the results of an SQL query

The documentation shows full detail and examples. The bind parameter feature is demonstrated in Figure 2. Developers and testers should consider leveraging bind to facilitate creation of unit tests.

> **Note:** that the use of bind provides the user incredible and nimble flexibility when testing their code.

Also shown in the figure is the appearance of SELF, on the lower right bottom. Do yourself a favor and enable SELF for 'New Jobs'. By doing so, Code4i will show you the on-going tally of specific SQL warnings and/or errors. By enabling SELF, you will opt-in to seeing an important quality assurance mechanism.

Example using the bind prefix

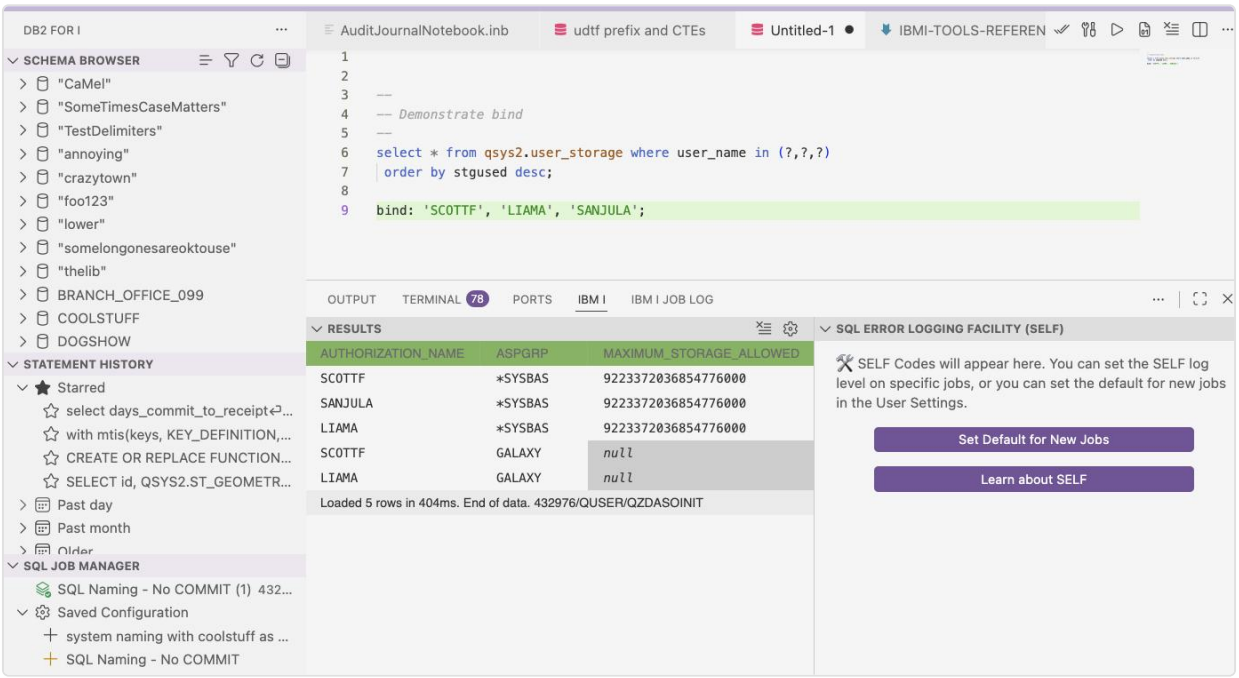


Figure. Example using the bind prefix

Database Extension – Visual Explain

While creating or enhancing an SQL query, take a moment to use the Visual Explain capability, available with the extension.

- **Run and explain statement** - Optimize and execute the query. After the query completes, the query plan is displayed, which includes estimated and actual runtime information.
- **Explain without running** - Optimize but do not execute the query. After the query finishes optimization, the query plan is displayed, which includes estimated runtime information.

Visual Explain Options

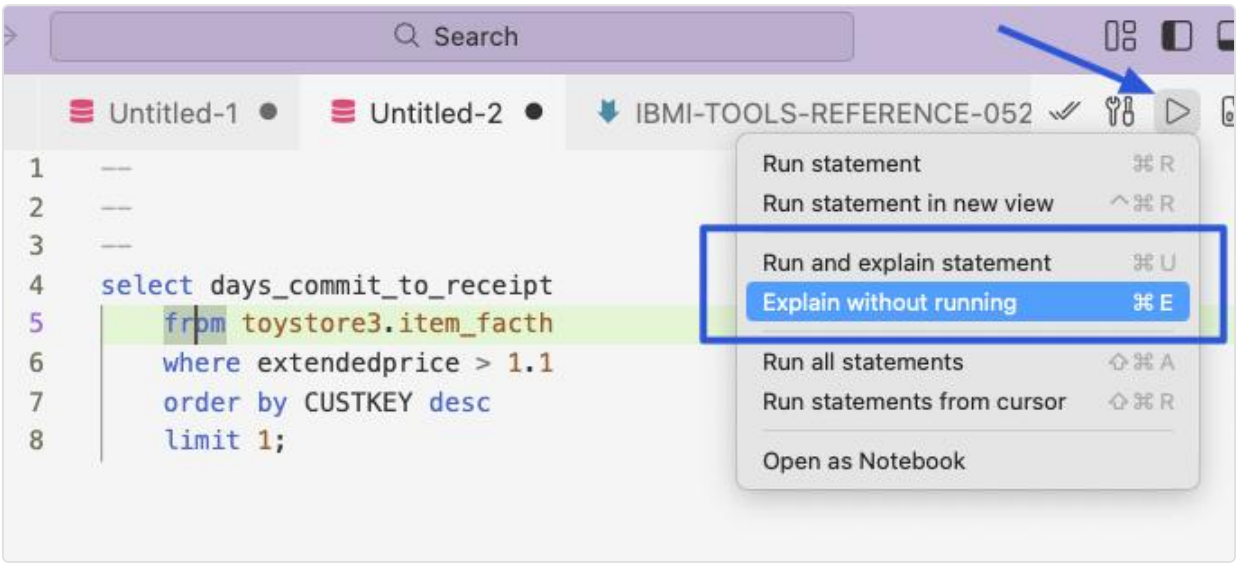


Figure. Visual Explain Options

The Visual Explain details actions that can be taken to examine the Visual Explain detail. For example, by clicking on the lightbulb, any and all Index Advice appears. The 'Advised Indexes and Statistics' twisty includes an action to generate the CREATE INDEX (SQL) statements needed to turn the advice into reality.

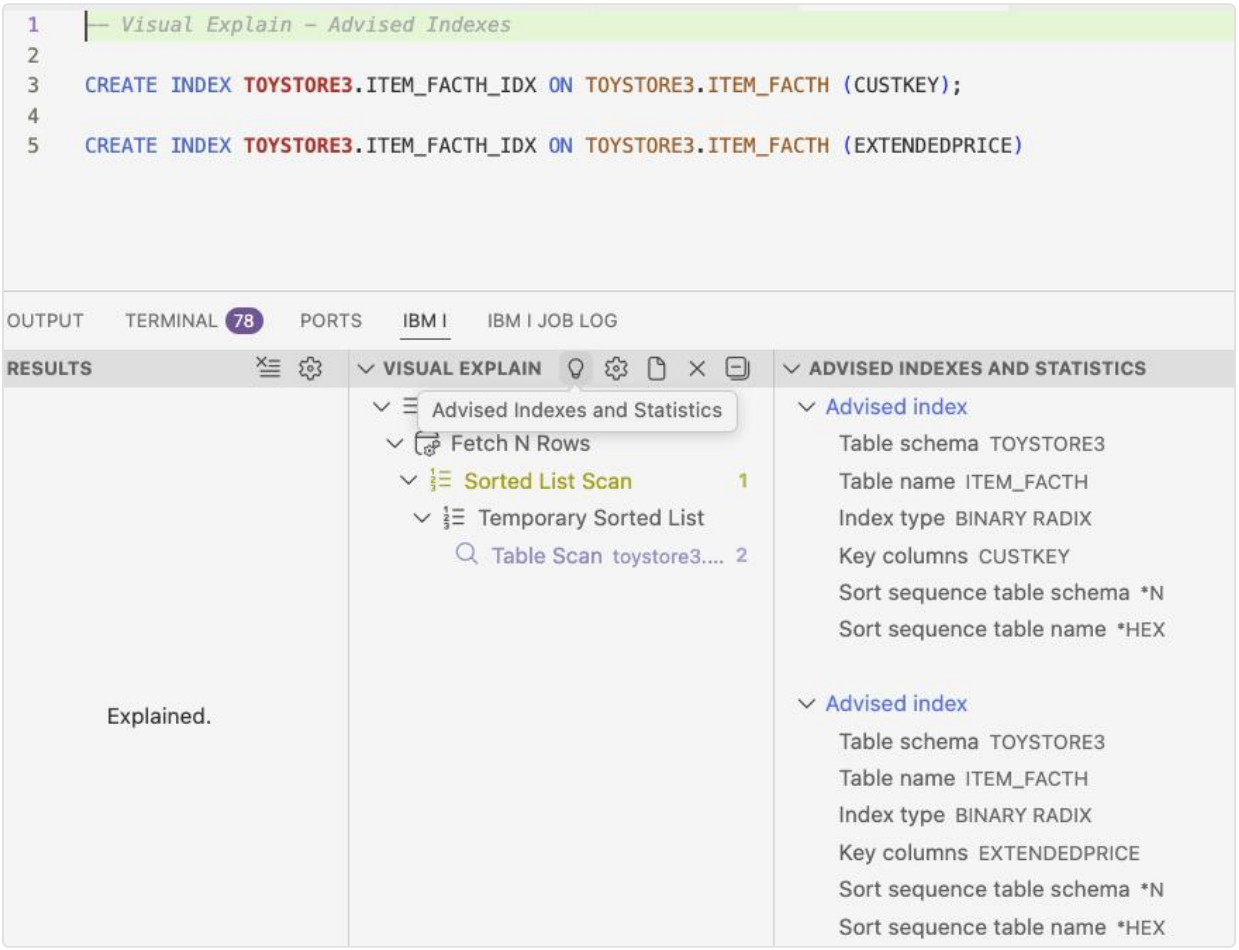


Figure. Visual Explain Detail

Database Extension – Definitely a tool to add to your toolbox

This section has touched on a few of the many Database Extension capabilities. This extension delivers on what you want from a tool: productivity improvements realized. Make the proper choice and add this tool to your toolbox.

IBM i Interfacing and Integration

What is a Stateless Interface?

What is it?

When IBM i developers first encounter the concept of stateless interfaces, the reaction is often confusion or scepticism. Traditional 5250 applications are inherently stateful, they maintain context throughout a user's session, remembering where the user is in a workflow, what data they've entered, and what operations they've performed.

To a tradition developer, this statefulness often feels natural and necessary, a lack of it can leave them wondering how an application possibly function without remembering what the user has been doing up to this point? Yet modern web and mobile applications operate fundamentally differently, embracing statelessness as a core architectural principle.

Understanding what stateless interfaces are, why they matter, and how they differ from traditional IBM i application design is essential for anyone involved in modernization efforts. It represents a fundamental shift in how applications are architected, with profound implications for scalability, reliability, and user experience.

Why It Matters

The distinction between stateful and stateless interfaces directly impacts how you design, deploy, and scale modern applications. Stateful applications, like traditional 5250 programs, maintain session information on the server. When a user navigates through screens, the server remembers their position, their previous inputs, and the context of their work. This approach works well for dedicated terminal connections but creates challenges in modern environments where users might access applications from multiple devices, where connections are unreliable, or where you need to scale to support thousands of concurrent users.

Stateless interfaces solve these challenges by treating each request independently. The application does not maintain session state between requests, instead, each request contains all the information needed to process it.

This architectural shift enables horizontal scaling (adding more servers to handle load), improves reliability (if one server is down for maintenance, users can seamlessly continue on another), and simplifies deployment and maintenance.

Understanding statelessness is crucial for modernization because most modern web frameworks, mobile applications, and API-based architectures assume stateless operation. If you are exposing IBM i business logic through REST APIs, building web interfaces, or enabling mobile access, you are entering the world of stateless design whether you realize it or not.

Key Takeaways

1. How Stateless Differs from Traditional IBM i Applications Traditional IBM i applications maintain state throughout a user's session. When a user signs on, the system allocates resources, a job, memory, perhaps open database cursors, that often persist until the user signs off. As the user navigates through screens, the application remembers where they are and how they got there. System state maintains their values between screen displays.

This stateful approach has advantages. It is intuitive for developers, you can write programs that flow naturally from one step to the next, maintaining context throughout. It's efficient for certain operations, for example you can open database files once and keep them open, avoiding the overhead or repeatedly opening and closing them.

Stateless interfaces work differently. Each request from a browser or mobile app is independent. The server processes the request, returns a response, and forgets about it. If the user submits a form, the server doesn't remember that they previously viewed a list or that they're in the middle of a multi-step process. All necessary context must be included in each request, using methods such as URL referral, inline data, cookies and authentication tokens.

This might seem inefficient or limiting, but it enables capabilities that stateful applications struggle with. A user can open multiple browser tabs, each working independently. They can bookmark a specific page and return to it directly. The application can distribute requests across multiple servers without worrying about which server handled previous requests. If a server crashes, users simply retry their request on another server without losing work.

2. Managing State in Stateless Applications

If stateless interfaces don't maintain state on the server, where does state go? The answer is: it moves to the client and the database. Modern stateless applications use several strategies to manage the state that applications need to function.

Client-side state includes information stored in the browser or mobile app form data the user has entered, the current page or view they're looking at, UI preferences, and temporary working data. JavaScript frameworks like React or Angular excel at managing this client-side state, updating the interface as users interact without requiring server roundtrips for every change.

Stateless applications read from and write to the database as needed, but they don't maintain open cursors or hold locks between requests. Database state represents the persistent information that survives beyond a single user interaction. Session tokens provide a middle ground. While the application itself is stateless, authentication tokens or session identifiers allow the server to retrieve user-specific information when needed. These tokens are typically stored in cookies or local storage and included with each request. The server can use them to look up user preferences, security permissions, or temporary session data stored in a database or cache.

The key insight is that statelessness doesn't mean applications cannot remember anything, it means they do not maintain that memory between requests. This distinction enables the scalability and reliability benefits of stateless architecture while still supporting the functionality users need.

3. Implications for IBM i Modernization Understanding statelessness is crucial when modernizing IBM i applications because it affects how you design APIs, build web interfaces, and architect modern solutions. When you expose IBM i business logic through REST APIs, those APIs should be stateless. Each API call should be independent, containing all necessary information to process the request. This means you cannot rely on program variables persisting between calls or assume that the same job will handle sequential requests from the same user.

This shift requires rethinking traditional IBM i programming patterns: Instead of programs that maintain state across multiple interactions, you design programs that complete discrete operations in single calls. Instead of holding database positions between screen displays, you pass keys or identifiers that allow programs to retrieve the correct records when needed. Instead of relying on job-level variables, you pass context in parameters or retrieve it from the database.

The good news is that IBM i is well suited to stateless operation. RPG programs can be designed to execute quickly and efficiently for single operations. The database is robust and performs well for the read/write patterns that stateless applications require. Modern IBM i web servers and frameworks support stateless operation naturally. The challenge is primarily conceptual, shifting from the stateful mindset of traditional 5250 programming to the stateless patterns of modern web development.

4. Embracing Stateless Design Stateless interfaces represent a fundamental shift from traditional IBM i application design, but they are not inherently better or worse, they are different, with different trade-offs.

For modern web and mobile applications, statelessness enables the scalability, reliability, and flexibility that users expect. For IBM i developers, understanding stateless design is essential for successful modernization, whether you're building APIs, creating web interfaces, or architecting hybrid solutions that bridge traditional and modern approaches.

What communication methods are available?

What This Covers

This article explains the various communication interfaces and protocols available on IBM i for accessing data and executing programs, enabling integration with external applications, development tools, and enterprise systems.

Understanding Communication Interfaces

IBM i provides multiple communication interfaces that allow external applications to:

- Access database tables and views
- Execute stored procedures and SQL statements
- Call RPG, COBOL, and CL programs
- Read and write IFS files
- Interact with data queues and message queues
- Invoke web services

Database Access Interfaces

ODBC (Open Database Connectivity)

Industry-standard interface for database access using SQL.

Characteristics:

- Client-side driver connects to IBM i database
- SQL-based data access with prepared statements and transactions
- Available on Windows, Linux, macOS
- Used by reporting and BI tools

Use Cases: Business intelligence tools, data integration/ETL, desktop applications (Excel, Access), third-party software

Best For: SQL-based data access, read-heavy workloads, reporting and analytics, cross-platform applications

JDBC (Java Database Connectivity)

Java-based interface for database access.

Characteristics:

- Native Java database connectivity
- JT400 (Toolbox for Java) provides IBM i-specific features
- Type 4 (pure Java) driver available
- Connection pooling support

Use Cases: Java applications, application servers (Liberty, Tomcat), web applications, microservices

Best For: Java-based applications, modern web applications, enterprise integration, microservices architectures

OLE DB

Microsoft's database access interface.

Characteristics:

- Windows-specific COM-based architecture
- Used by Microsoft tools and applications
- Supports SQL access to IBM i

Use Cases: Microsoft SSIS, Microsoft Access, Visual Basic applications, .NET applications (via ADO.NET)

Best For: Windows-based integration, Microsoft technology stack, legacy Windows applications

ADO.NET

.NET Framework data access interface.

Characteristics:

- Native .NET data access
- IBM i Access for Windows provides ADO.NET provider
- Supports DataSet and DataReader

Use Cases: .NET applications (C#, VB.NET), ASP.NET web applications, Windows desktop applications, PowerShell scripts

Best For: .NET development, modern Windows applications, Microsoft Azure integration

Mapepire

Modern database access interface for IBM i using server-side components.

Characteristics:

- Server-side SQL execution via database server jobs
- WebSocket-based communication for persistent connections
- JSON-based request/response protocol
- Supports both SQL execution and program calls
- Built on IBM i host servers infrastructure

Use Cases: Modern web applications, real-time data access, cloud-native applications, serverless architectures, Node.js applications

Best For: Modern application development, WebSocket-enabled clients, real-time data streaming, reduced connection overhead, cloud and edge computing

Program Call Interfaces

JT400 ProgramCall

Java API for calling IBM i programs directly.

Characteristics:

- Part of JT400 (Toolbox for Java)
- Call RPG, COBOL, CL programs with parameters
- Handle program messages
- No SQL required

Use Cases: Invoking business logic in ILE programs, integration with existing RPG/COBOL applications, custom Java applications, microservices calling legacy programs

Best For: Direct program invocation, leveraging existing business logic, Java-based integration, avoiding SQL overhead

Stored Procedures

SQL-based interface for executing programs and procedures.

Characteristics:

- Called via SQL CALL statement
- Can wrap RPG, COBOL, CL programs
- Support input/output parameters and result sets
- Accessible via ODBC, JDBC, and other SQL interfaces

Use Cases: Exposing business logic via SQL, database-centric applications, reporting tools integration, cross-platform access

Best For: SQL-based applications, standardized interface, multi-platform access, database-centric architectures

Web Service Interfaces

REST APIs

RESTful web services using HTTP methods.

Characteristics:

- HTTP-based (GET, POST, PUT, DELETE)
- JSON or XML payloads
- Stateless communication
- Can be created via IWS or custom applications

Use Cases: Mobile applications, web applications, microservices integration, cloud service integration, API-first architectures

Best For: Modern web and mobile apps, lightweight communication, public APIs, cloud-native applications

SOAP Web Services

XML-based web services with formal contracts.

Characteristics:

- XML message format with WSDL interface definition
- Protocol-based (WS-* standards)
- Can be created via IWS
- Enterprise integration patterns

Use Cases: Enterprise B2B integration, legacy system integration, formal contract requirements, complex transactions

Best For: Enterprise integration, formal contracts, WS-Security requirements, legacy system compatibility

WebSocket

Full-duplex communication protocol over TCP.

Characteristics:

- Persistent bidirectional connection
- Real-time communication
- Supported by Liberty application server (websocket feature)
- Lower overhead than HTTP polling

Use Cases: Real-time applications, chat applications, live dashboards, notifications, collaborative tools, streaming data

Best For: Real-time bidirectional communication, push notifications, live updates, interactive applications

File Access Interfaces

FTP/SFTP

File Transfer Protocol for moving files.

Characteristics:

- Standard file transfer protocols
- IBM i FTP server built-in
- Access IFS and library files
- Secure variant (SFTP) available

Use Cases: Batch file transfers, data exchange with partners, backup and archive, integration with external systems

Best For: Bulk file transfers, scheduled data exchanges, partner integration, simple file-based integration

Network File System (NFS)

Network file sharing protocol.

Characteristics:

- Mount IBM i IFS as network drive
- Unix/Linux standard
- File-level read/write operations

Use Cases: Unix/Linux integration, shared file access, application data sharing, cross-platform file access

Best For: Unix/Linux environments, shared file systems, application integration, development workflows

Samba/CIFS

Windows file sharing protocol.

Characteristics:

- IBM i can act as file server
- Windows network shares
- Integrated with Windows authentication

Use Cases: Windows integration, shared drives, file access from Windows clients, print services

Best For: Windows environments, user file access, department shares, Windows application integration

Message-Based Interfaces

Data Queues

IBM i native message queuing mechanism.

Characteristics:

- Fast, lightweight queuing
- FIFO or keyed access
- Synchronous or asynchronous
- Accessible from ILE languages and Java

Use Cases: Inter-program communication, event notification, asynchronous processing, high-performance messaging

Best For: IBM i-to-IBM i communication, low-latency messaging, simple queuing needs, native integration

IBM MQ

Enterprise messaging middleware.

Characteristics:

- Guaranteed message delivery
- Transactional messaging
- Cross-platform support
- Point-to-point and pub/sub

Use Cases: Enterprise application integration, cross-platform messaging, reliable message delivery, complex workflows

Best For: Enterprise integration, mission-critical messaging, cross-platform communication, transactional requirements

Apache Kafka

Distributed streaming platform.

Characteristics:

- High-throughput messaging
- Event streaming with persistence and replay
- Distributed architecture

Use Cases: Event-driven architectures, real-time analytics, data streaming, microservices communication

Best For: High-volume messaging, event streaming, real-time processing, modern architectures

Remote Command Execution

SSH (Secure Shell)

Secure remote command execution.

Characteristics:

- Encrypted communication
- Command-line access to PASE shell
- Secure file transfer (SFTP)
- Key-based authentication

Use Cases: Remote administration, automated scripts, DevOps workflows, secure file transfers

Best For: System administration, automation scripts, secure remote access, DevOps practices

IBM i Access Client Solutions

IBM's client access solution.

Characteristics:

- 5250 emulation
- File transfer, printer output
- SQL scripting, system management

Use Cases: Interactive terminal access, system administration, development and testing, user applications

Best For: Interactive access, system management, development, traditional green-screen applications

Specialized Interfaces

DDM (Distributed Data Management)

IBM protocol for remote file access.

Characteristics:

- Access remote IBM i files
- Record-level access
- Native file operations

Use Cases: Distributed applications, multi-system architectures, remote file access, legacy application integration

Best For: IBM i to IBM i communication, record-level file access, distributed systems, legacy applications

DRDA (Distributed Relational Database Architecture)

IBM's distributed database protocol.

Characteristics:

- SQL access to remote databases
- IBM standard protocol
- Cross-platform database access

Use Cases: Distributed database applications, cross-platform SQL access, IBM ecosystem integration, database federation

Best For: IBM database integration, distributed SQL, cross-platform access, enterprise database architectures

Host Servers

IBM i native client/server architecture.

Characteristics:

- Multiple server types (database, file, command, etc.)
- Used by IBM i Access
- Efficient binary protocol
- Integrated with IBM i security

Server Types:

- Central Server (port 8470)
- Database Server (port 8471)
- Network Print Server (port 8472)
- File Server (port 8473)
- Remote Command Server (port 8475)
- Signon Server (port 8476)

Use Cases: IBM i Access Client Solutions, Navigator for i, third-party tools, custom applications using IBM i Access APIs

Best For: IBM i-specific tools, integrated access, multiple service types, efficient communication

Choosing the Right Interface

Database Access

ODBC: BI/reporting tools, cross-platform requirements, standard interface **JDBC**: Java applications, IBM i-specific features (JT400), modern web applications **OLE DB**: Microsoft tools, Windows integration **ADO.NET**: .NET applications, Windows development **Mapepire**: Modern web applications, WebSocket-based communication, real-time data access, cloud-native architectures

Program Execution

ProgramCall: Direct program invocation, Java-based integration, complex parameters **Stored Procedures**: SQL-based applications, cross-platform access, standard interface

Web Services

REST: Modern web/mobile apps, lightweight communication, JSON, public APIs **SOAP**: Enterprise integration, formal contracts, WS-* standards, legacy compatibility **WebSocket**: Real-time bidirectional communication, push notifications, live updates, interactive applications

File Transfer

FTP/SFTP: Batch file transfers, partner integration, scheduled transfers **NFS/CIFS**: Shared file access, network drives, user file access

Messaging

Data Queues: IBM i-to-IBM i, low latency, simple queuing **IBM MQ**: Enterprise integration, guaranteed delivery, cross-platform **Kafka**: High-volume streaming, event-driven architecture, real-time processing

Security Considerations

Authentication Methods:

- IBM i user profiles
- Kerberos
- SSL/TLS certificates
- Token-based (OAuth, JWT)

Transport Security:

- SSL/TLS for encrypted communication
- SSH for secure shell access
- VPN for network-level encryption

Best Practices:

- Use encrypted connections for sensitive data
- Implement proper authentication
- Follow principle of least privilege
- Monitor and audit access

Performance Optimization

Connection Pooling:

- Reduces connection overhead
- Applicable to JDBC, ODBC, HTTP connections
- Improves response times

Batch Operations:

- Process multiple records together
- Reduce round trips
- Applicable to SQL, file transfers, messaging

Caching:

- Cache frequently accessed data
- Application-level and database caching
- Improves performance for reference data

Key Takeaways

- IBM i provides numerous communication interfaces for data access and program execution
- Database interfaces: ODBC, JDBC, OLE DB, ADO.NET, Mapepire for SQL-based access
- Mapepire offers modern WebSocket-based communication for real-time data access and cloud-native applications
- Program call interfaces: JT400 ProgramCall and stored procedures for executing business logic
- Web services: REST, SOAP, and WebSocket protocols for modern and enterprise integration
- File access: FTP/SFTP, NFS, CIFS/Samba for file transfers and sharing
- Message-based: Data Queues, IBM MQ, Kafka for asynchronous communication
- Specialized: DDM, DRDA, Host Servers for IBM i-specific capabilities
- Choice depends on use case, platform, performance requirements, and infrastructure
- Security through authentication, encryption, and access control across all interfaces
- Performance optimization via connection pooling, batch operations, and caching

When do I use message queues?

What This Covers

This article explains when and why to use message queuing technologies on IBM i, focusing on IBM MQ and Apache Kafka as enterprise messaging solutions for decoupling applications, ensuring reliable message delivery, and building scalable distributed systems.

Understanding Message Queues

What are Message Queues?

Message queues are middleware systems that enable asynchronous communication between applications by temporarily storing messages until the receiving application is ready to process them.

Core Concepts:

- **Producer**: Application that sends messages
- **Consumer**: Application that receives and processes messages
- **Queue/Topic**: Storage location for messages
- **Broker**: Server that manages queues and routes messages

Key Benefits:

- Decoupling: Producers and consumers operate independently
- Asynchronous Processing: Producers don't wait for consumers
- Reliability: Messages persist until successfully processed
- Scalability: Add consumers to handle increased load
- Fault Tolerance: System continues operating if components fail

Message Queuing on IBM i

IBM MQ (formerly WebSphere MQ)

IBM's enterprise messaging middleware providing reliable, secure message delivery across platforms.

Key Characteristics:

- Guaranteed message delivery (exactly-once semantics)
- Transactional messaging with commit/rollback
- Point-to-point and publish/subscribe patterns
- Native IBM i integration
- Cross-platform messaging (IBM i, Windows, Linux, z/OS)
- Callable from RPG, COBOL, CL, Java

Apache Kafka

Distributed streaming platform designed for high-throughput, fault-tolerant event streaming.

Key Characteristics:

- High throughput (millions of messages per second)
- Distributed and scalable architecture
- Message retention and replay capabilities
- Publish/subscribe with consumer groups
- Stream processing capabilities
- Runs on IBM i using Java; accessible from all IBM i languages

When to Use Message Queues

1. Decoupling Applications

Problem: Tightly coupled systems fail when dependencies are unavailable.

Solution: Message queues allow systems to communicate without direct dependencies.

```
Without Queues: Order System → (Direct Call) → Inventory/Shipping/Billing
                (Any failure stops order processing)

With Queues:    Order System → Message Queue → Inventory
                                             → Shipping
                                             → Billing
                (Order system continues; consumers process independently)
```

IBM i Example: RPG order entry sends messages to IBM MQ. Separate programs consume messages to update inventory, create shipments, and generate invoices—all independently.

2. Asynchronous Processing

Problem: Long-running operations slow down user interactions.

Solution: Queue tasks for background processing, return immediate response.

Use Cases: Email notifications, report generation, data exports, batch operations

IBM i Example: Web service queues large data import to IBM MQ instead of processing synchronously. Background job processes import and sends completion notification.

3. Load Leveling

Problem: Traffic spikes overwhelm systems, causing failures.

Solution: Messages queue during spikes; consumers process at sustainable rate.

IBM i Example: Month-end transactions flood the system. IBM MQ queues transactions; multiple batch jobs process at controlled rate, preventing overload.

4. Guaranteed Delivery

Problem: Cannot afford to lose critical messages during failures.

Solution: IBM MQ provides transactional delivery with persistence.

Use Cases: Financial transactions, order processing, regulatory compliance, audit trails

IBM i Example: Payment processing uses IBM MQ to ensure every payment instruction is delivered exactly once with full transactional integrity.

5. System Integration

Problem: Integrating IBM i with external systems, cloud services, or partners.

Solution: Message queues provide integration backbone without tight coupling.

Use Cases: Cloud integration, partner EDI, multi-system workflows, legacy modernization

IBM i Example: IBM i sends order data to Kafka topics. Cloud analytics consume real-time stream; data warehouse subscribes for historical analysis—without direct coupling.

6. Event-Driven Architectures

Problem: Need real-time reactions to business events across multiple systems.

Solution: Publish events to Kafka; multiple consumers react independently.

Use Cases: Real-time inventory updates, customer activity tracking, IoT data, fraud detection

IBM i Example: Every inventory change publishes to Kafka. Multiple consumers react: data warehouse updates, reorder alerts, real-time dashboard—all from single event stream.

7. Microservices Communication

Problem: Services need to communicate without tight coupling.

Solution: Message queues enable independent service deployment and scaling.

IBM i Example: Breaking down monolithic RPG application—order processing, inventory, and shipping become separate services communicating via IBM MQ.

8. Workflow Orchestration

Problem: Complex business processes involving multiple steps and systems.

Solution: Route messages through workflow stages with full audit trail.

IBM i Example: Purchase order approval workflow routes to different queues based on amount. Each approval triggers next step with complete audit trail.

IBM MQ vs. Kafka: Choosing the Right Tool

Use IBM MQ When:

✓ Transactional integrity is critical (financial transactions, order processing) ✓ Guaranteed exactly-once delivery required ✓ Point-to-point messaging is primary pattern ✓ Strong security and governance needed ✓ Mature, proven technology preferred

Use Kafka When:

✓ High throughput essential (millions of messages/second) ✓ Event replay capability needed ✓ Publish/subscribe is primary pattern ✓ Stream processing required ✓ Integration with modern data platforms

Hybrid Approach

Many organizations use both:

- **IBM MQ** for critical transactional messaging
- **Kafka** for high-volume event streaming
- Bridge between them for specific use cases

Implementation Considerations

Performance

IBM MQ: Optimized for reliability; excellent for moderate volumes (thousands/second); efficient for request/reply patterns

Kafka: Optimized for throughput (millions/second); lower latency for streaming; scales horizontally

Resource Requirements

IBM MQ: Moderate CPU/memory; well-suited for IBM i resource model

Kafka: Higher memory for caching; Java-based (JVM overhead); benefits from multiple cores

Security

IBM MQ: Native IBM i security integration; SSL/TLS; extensive audit capabilities

Kafka: SSL/TLS encryption; SASL authentication; ACL-based authorization

Best Practices

Message Design

- Keep messages small and self-contained
- Use standard formats (JSON, XML, Avro)
- Include all context needed for processing

Error Handling

- Implement retry logic with exponential backoff
- Use dead-letter queues for persistent failures
- Monitor and alert on error conditions
- Handle poison messages appropriately

Scalability

- Design for horizontal scaling with multiple consumers
- Implement backpressure to prevent resource exhaustion
- Monitor queue depths and processing rates

Monitoring

- Track queue depth trends
- Monitor message processing rates and error rates
- Implement real-time dashboards and alerting

- Maintain historical data for capacity planning

Migration Strategy

Phase 1: Pilot Project

Select non-critical use case, implement with small scope, validate approach

Phase 2: Expand Usage

Apply to additional use cases, standardize patterns, develop reusable components

Phase 3: Enterprise Adoption

Establish messaging standards, integrate with enterprise architecture, implement governance

Modernizing Existing Systems

- Insert message queues between tightly coupled components
- Migrate one integration at a time
- Maintain backward compatibility during transition

Common Pitfalls to Avoid

1. **Using queues for everything**: Not all communication needs queuing; consider latency requirements
2. **Ignoring message ordering**: Understand ordering guarantees; design for out-of-order delivery when possible
3. **Inadequate error handling**: Implement comprehensive retry logic and monitoring
4. **Poor message design**: Avoid overly large messages; version message formats
5. **Insufficient monitoring**: Implement proactive monitoring with meaningful alerts

Key Takeaways

- Message queues enable asynchronous, decoupled communication between applications, improving reliability and scalability
- IBM i supports both IBM MQ (transactional, guaranteed delivery) and Apache Kafka (high-throughput event streaming)
- Use message queues for decoupling applications, asynchronous processing, load leveling, guaranteed delivery, system integration, event-driven architectures, microservices, and workflow orchestration
- IBM MQ excels at transactional integrity and exactly-once delivery for critical business processes
- Kafka excels at high-throughput event streaming, message replay, and real-time data pipelines
- Both integrate with IBM i and are accessible from RPG, COBOL, CL, Java, and other languages
- Proper message design, error handling, monitoring, and testing are essential for success
- Start with pilot projects and gradually expand as expertise grows
- Evaluate based on specific requirements—message queues are powerful but not appropriate for every scenario

When do I use Data Queue?

What This Covers

This procedure covers on how to use Data Queues on the IBM i, including their types, capabilities, and use cases. It also explores traditional and modern approaches to working with Data Queues, along with sample implementations.

What It Is

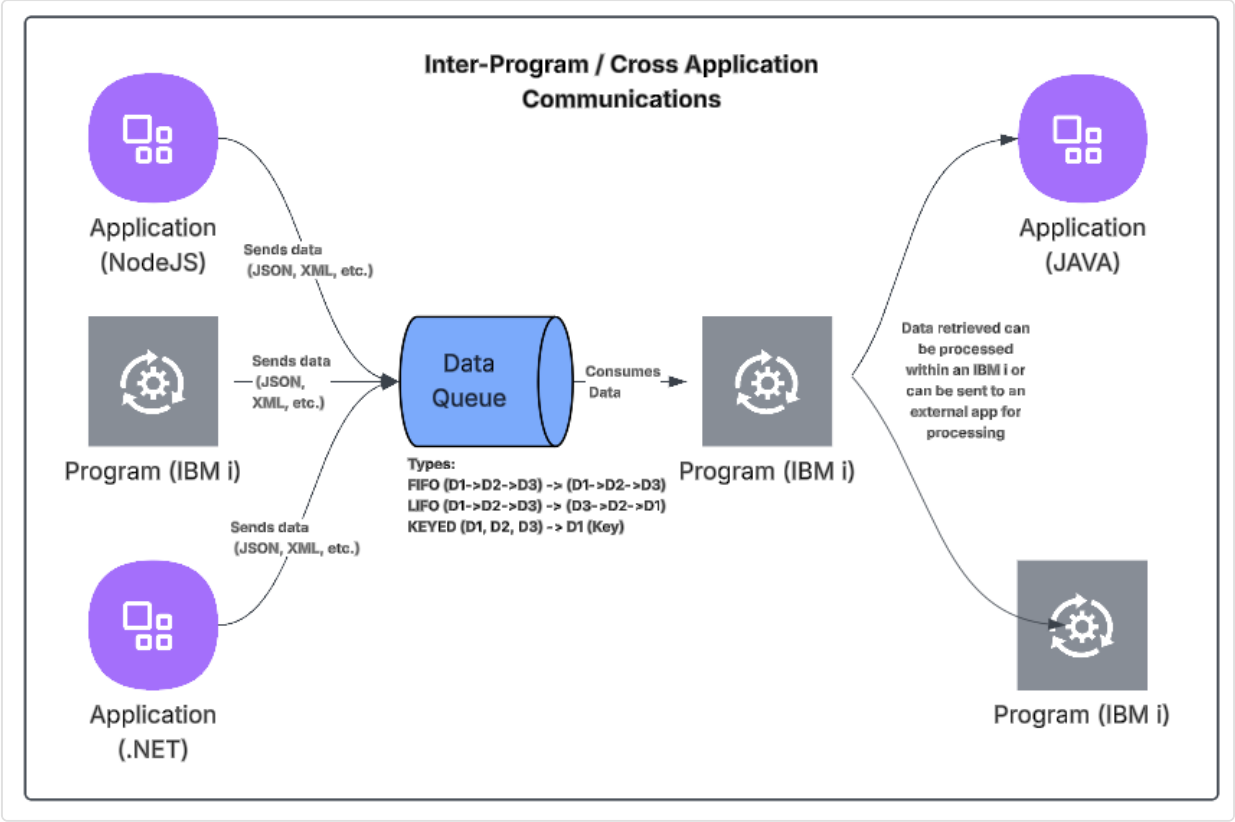


Figure. Data Queue Diagram

- Data Queues are of *DTAQ type objects which are used for inter-program or cross-platform communication on the IBM i. They provide a mechanism for sending and receiving messages between different jobs or processes, either on the same system or across different systems.
- Data Queues are of the following types:
  - Keyed Data Queues
  - Non-Keyed Data Queues
- Messages in the Non-Keyed data queues can be processed in First-In-First-Out (FIFO) or Last-In-First-Out (LIFO) order, while messages in the Keyed data queues can be processed based on their keys
- Once a message is read from the data queue, it is removed from the queue. This means that messages can only be read once, and if multiple jobs are reading from the same data queue, they will each receive different messages
- The unread messages persist until the queue is cleared or deleted

Keyed vs. Non-Keyed Data Queues

Both types share the same *DTAQ object type, but they behave differently when it comes to how messages are stored and retrieved.

	Non-Keyed Data Queue	Keyed Data Queue
Message Order	FIFO or LIFO — messages come off in sequence	By key — you pick which message to pull
Key Field	None	1–256 byte character key, set at queue creation
Retrieval	Next in line, no filtering	Selective: match on exact key, or use comparison operators (EQ, GE, GT, LE, LT)
Creation	CRTDTAQ DTAQ(LIB/MYQUEUE) MAXLEN(512)	CRTDTAQ DTAQ(LIB/MYQUEUE) MAXLEN(512) SEQ(*KEYED) KEYLEN(10)
Best For	Simple pipelines, event streams, logging	Priority routing, multi-category processing, request-reply correlation
When to Pick	All messages are equal — process them in order	You need to pull specific messages based on a category, priority, or correlation ID

The key difference: non-keyed queues are a conveyor belt. Keyed queues are a filing cabinet — you reach in and grab exactly the folder you need.

Why It Matters

Data Queues are powerful and can be used in quite a few scenarios and here are some example use cases:

Inter-Program Communication

- **Scenario**: All the changes that are happening to the Customer Master data on the IBM i needs to be recorded in a log file for auditing purposes. A data queue can be used to send messages from the program that is responsible for creating/updating the Customer Master data to another program that is responsible for writing the log file. The two processes stay decoupled — the logging job runs independently, so it doesn't slow down the main program.

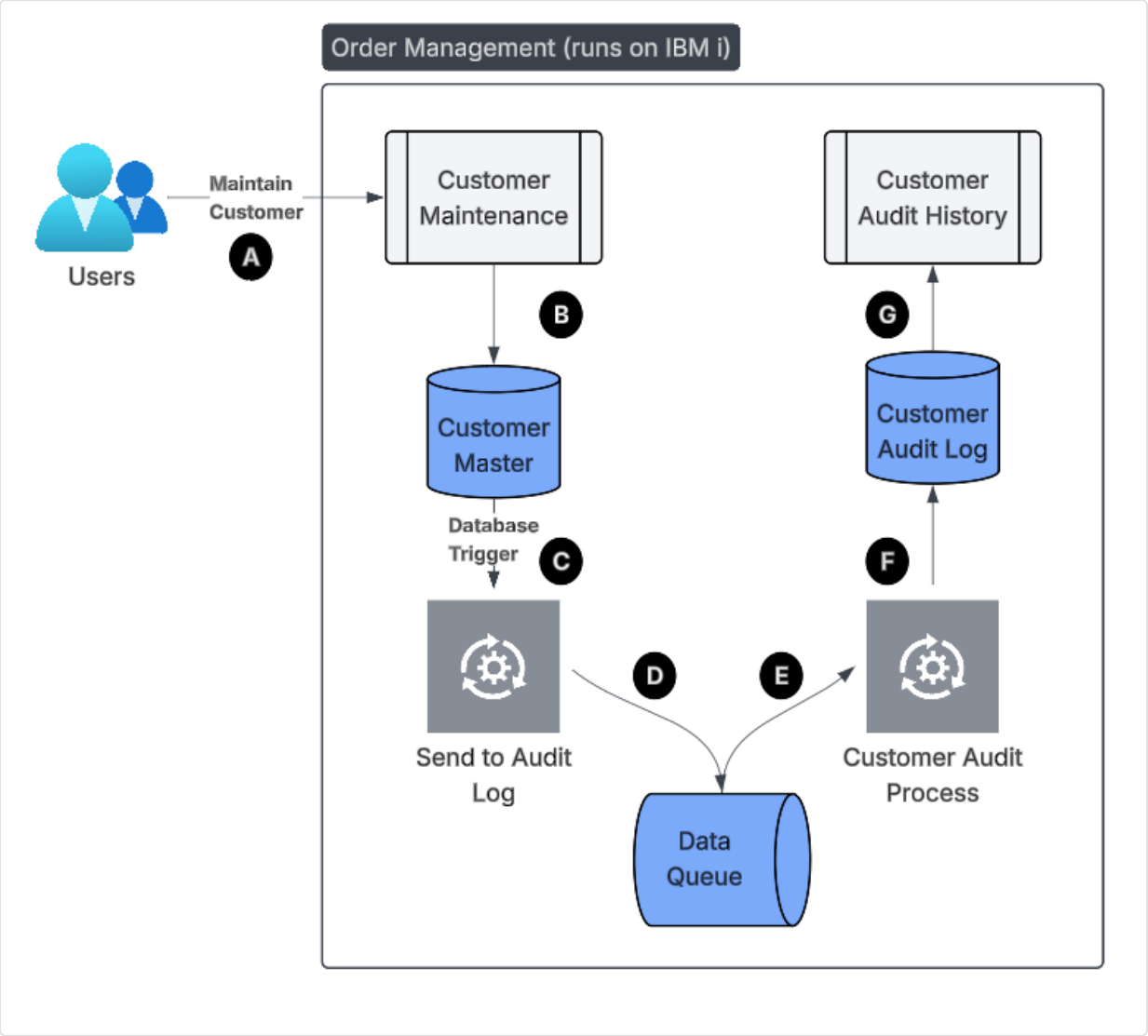


Figure. Inter Program Communication

Cross Platform Data Synchronization

- **Scenario**: Every time a new Customer record is created on the IBM i, a message needs to be sent to an external system so that the Customer data can be synchronized to a datawarehouse. A data queue can be used in this scenario to record the new Customer record creation event and there are certain middleware tools (i.e. TIBCO, Mulesoft etc.) that have native capabilities to listen to the data queue and forward the message to the external system in real-time. You get near real-time data synchronization between the IBM i and the external system — no complex integration layer or batch processing required.

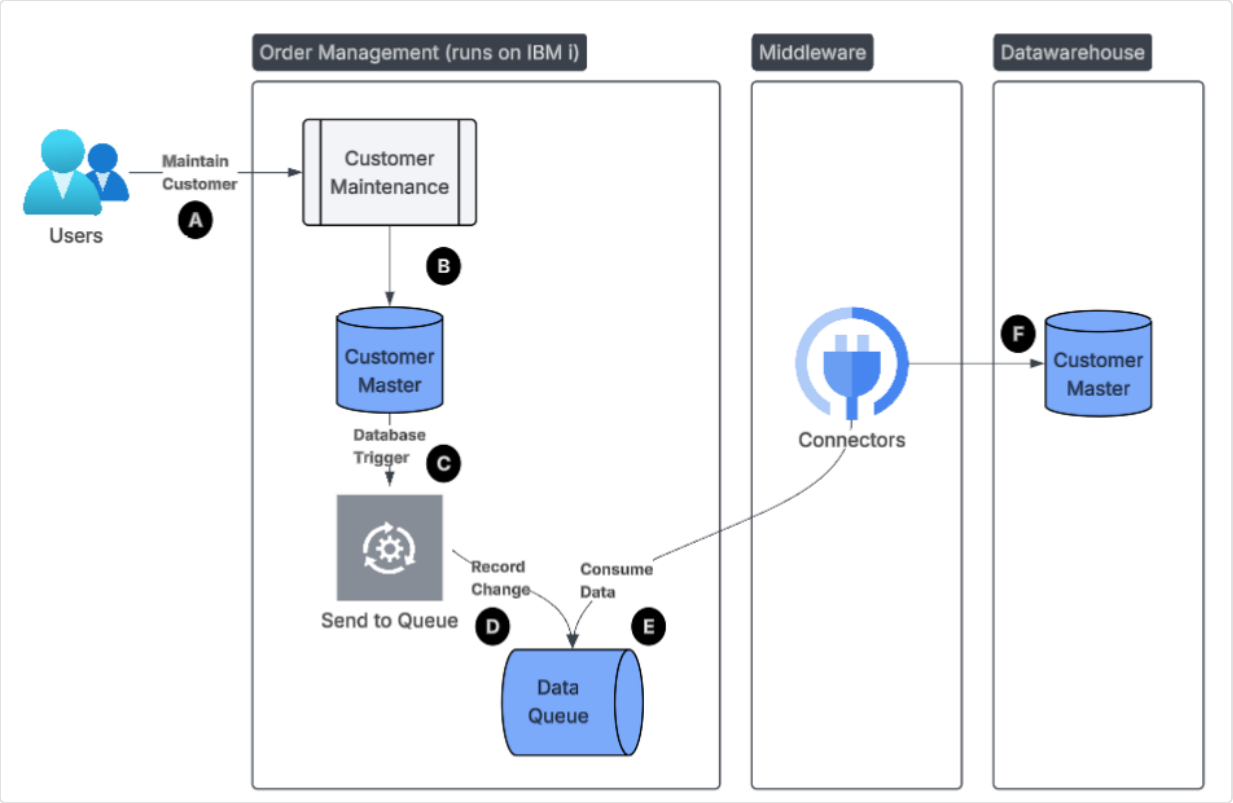


Figure. Data Sync

Event Driven

- **Scenario**: The Billing System which runs on the IBM i is notified in real-time once a Customer has been serviced. This can be achieved by sending a message to a data queue once the service event occurs, and the Billing System can listen to the data queue and process the message in real-time to generate the bill for the customer. It's event-driven — the billing job reacts to what happened instead of polling on a schedule.

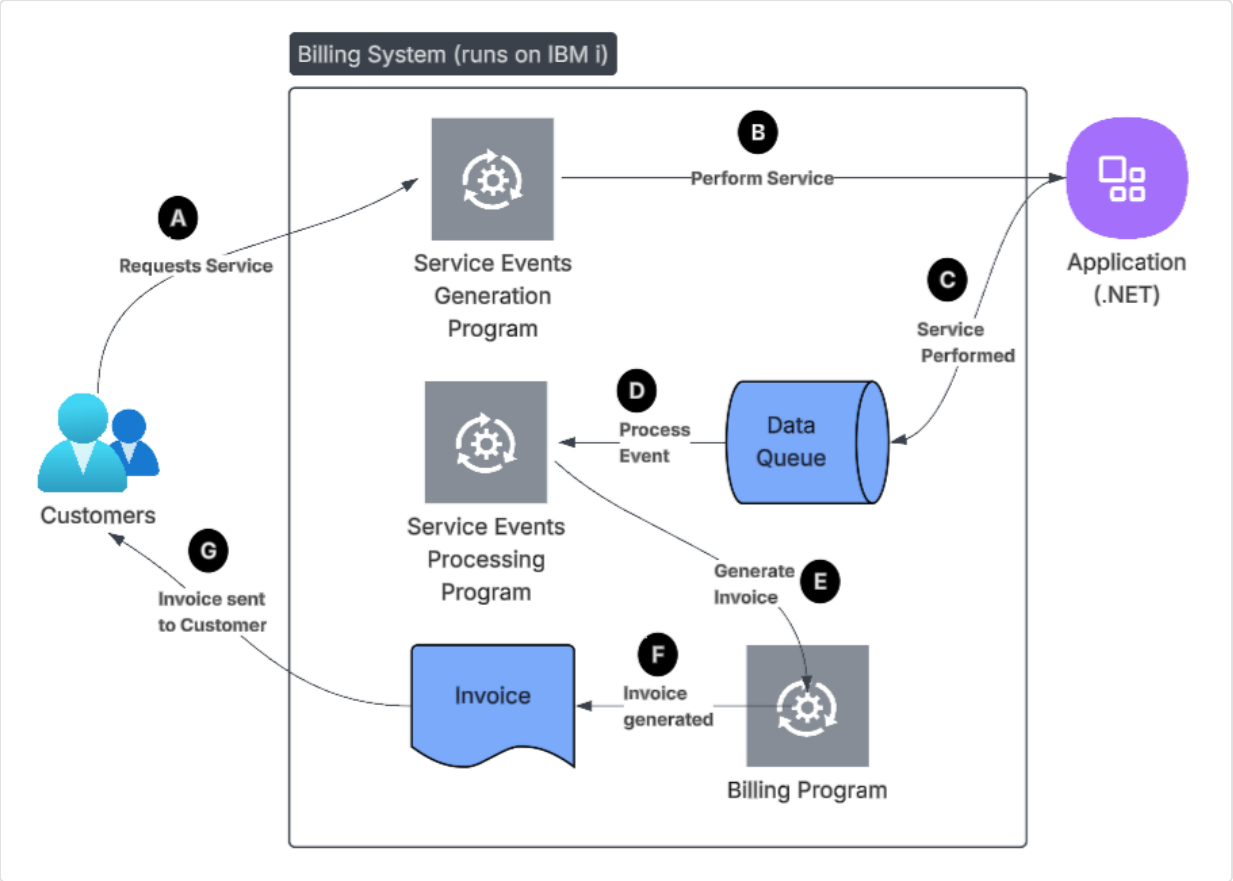


Figure. Event Driven

Priority-Based Order Processing

- **Scenario**: The Order Management System receives orders throughout the day with different priority levels — RUSH, STANDARD, and ECONOMY. All orders land on the same keyed data queue, with the priority level as the key. The processing job pulls RUSH orders first by requesting `key_data => 'RUSH'`. Once those are cleared, it moves to STANDARD, then ECONOMY. The lower-priority orders don't get lost — they just wait their turn. This is something you can't do with a non-keyed queue. With FIFO, a RUSH order that arrives after 50 ECONOMY orders sits at the back of the line.

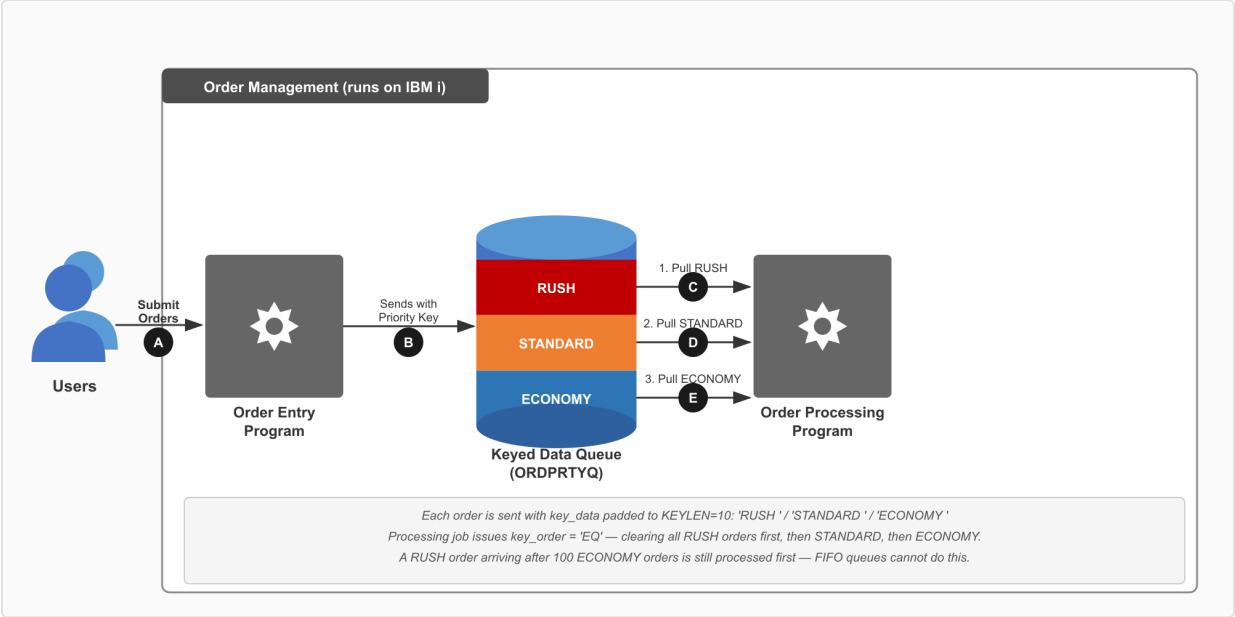


Figure. Priority Based Order Processing

How It Works

Traditional Approach	Modern Approach
SNDDTAQ	IBM i Services, Language-specific libraries (NodeJS, Java, Python etc.)
RCVDTAQ	IBM i Services, Event Driven Patterns

Non-Keyed Data Queue

```
  -- Send a message to the data queue
  call qsys2.send_data_queue(
  message_data       => '{"customer":"C2001","action":"UPDATE","details":{"field":"ADDRESS","old":"123 OLD
  data_queue         => 'CUSTAUDQ',
  data_queue_library => 'MYLIB');
```

```
-- Retrieve the message from the data queue
select *
  from table (
      qsys2.receive_data_queue(
        data_queue => 'CUSTAUDQ',
        data_queue_library => 'MYLIB')
    );
```

Keyed Data Queue

Keyed queues use the same QSYS2 services — you just add the `key_data` parameter on send, and `key_data` + `key_order` on receive. The queue itself has to be created with `SEQ(*KEYED)` though, and there's no SQL equivalent for that — it's a CL command.

A. Create a keyed data queue

```
-- CL command — run this from CL or via QCMDEXC
CRTDTAQ DTAQ(MYLIB/ORDPRTYQ) MAXLEN(512) SEQ(*KEYED) KEYLEN(10)
        TEXT('Order priority queue - keyed by priority level')
```

This creates a data queue called `ORDPRTYQ` in `MYLIB` with a 10-character key. The key length is fixed at creation — every message sent to this queue must include a key of exactly that length. Pick your key length based on the longest key value you'll use (RUSH is 4 chars, STANDARD is 8, ECONOMY is 7 — so 10 gives you room).

B. Send messages with different priority keys

```
-- Send a RUSH order
call qsys2.send_data_queue(
  message_data       => '{"order":"W9012445511","customer":"C2001","items":3,"total":847.50}',
  data_queue         => 'ORDPRTYQ',
  data_queue_library => 'MYLIB',
  key_data           => 'RUSH      ');
  -- Note: key is padded to 10 chars to match KEYLEN

-- Send a STANDARD order
call qsys2.send_data_queue(
  message_data       => '{"order":"W7484738837","customer":"C1001","items":1,"total":125.00}',
  data_queue         => 'ORDPRTYQ',
  data_queue_library => 'MYLIB',
  key_data           => 'STANDARD  ');

-- Send an ECONOMY order
call qsys2.send_data_queue(
  message_data       => '{"order":"W6501223344","customer":"C3045","items":5,"total":2340.00}',
  data_queue         => 'ORDPRTYQ',
  data_queue_library => 'MYLIB',
  key_data           => 'ECONOMY   ');
```

All three messages sit on the same queue. Without keys, you'd get them back in FIFO order. With keys, you choose.

C. Receive by key — selective retrieval

```
-- Pull only RUSH orders (exact match), wait up to 10 seconds
select *
  from table (
      qsys2.receive_data_queue(
        data_queue         => 'ORDPRTYQ',
        data_queue_library => 'MYLIB',
        key_data           => 'RUSH      ',
        key_order          => 'EQ',
        wait_time          => 10)
    );
-- Returns: {"order":"W9012445511","customer":"C2001","items":3,"total":847.50}
-- If no RUSH messages arrive within 10 seconds, returns empty result set
```

The `wait_time` parameter controls how long the receive call blocks when no matching message is on the queue:

wait_time	Behavior
0	Return immediately — empty result if nothing matches
n	Wait up to n seconds for a matching message
-1	Wait forever until a message arrives

In a real processing loop, you'd almost always use `wait_time => -1` so the job sits and listens. Use `0` for a quick peek, and a positive value when you need a timeout.

```
-- Pull any message with a key >= 'RUSH' (catches RUSH and STANDARD)
select *
  from table (
      qsys2.receive_data_queue(
        data_queue         => 'ORDPRTYQ',
        data_queue_library => 'MYLIB',
        key_data           => 'RUSH      ',
        key_order          => 'GE')
    );
-- Watch out: comparison is alphabetic, not by business priority.
-- 'RUSH' < 'STANDARD' alphabetically, so GE on 'RUSH' returns RUSH or STANDARD — not ECONOMY.
-- If you need true priority ordering, use numeric keys like '01', '02', '03'.
```

The `key_order` parameter accepts five comparison operators:

key_order	Meaning	Returns message where key is...
EQ	Equal	Exactly matches the key_data
GE	Greater than or equal	>= key_data (alphabetically)
GT	Greater than	> key_data (alphabetically)
LE	Less than or equal	<= key_data (alphabetically)
LT	Less than	< key_data (alphabetically)

One thing to keep in mind — the comparison is character-based, not numeric. If you're using keys for true ranked priority, use zero-padded numbers (`'01'`, `'02'`, `'03'`) instead of words. That way `GE '01'` gives you priority 1 first, then 2, then 3 — in the right order.

Key Takeaways

- Use Data Queues when you need:
  - fast, low-overhead messaging between jobs on IBM i
  - Cross-platform messaging between IBM i and other systems
  - Event-driven processing on IBM i
  - Selective message retrieval based on a key — that's what keyed data queues are for
  - Priority-based processing where certain messages need to be handled before others

- When not to use Data Queues:
  - For large message payloads
  - When you need durable, queryable history of messages
  - For complex message routing or transformation needs
  - When you need to fan-out i.e. one message and multiple consumers
  - If all messages are processed in arrival order, a non-keyed queue is simpler — don't add a key just because you can

Best practice for TCPIP communication

What This Covers

Best practice for TCP/IP communication in IBM i applications centers on resilience, security, modular design, performance optimization, and operational transparency. By treating network communication as a critical architectural component, organizations can ensure they are stable, resilient and scalable.

What It Is

We will describe each of the best practices in turn, explain the benefit it offers then provide a link to explain how to implement it.

Why It Matters

When designing or modifying an IBM i application that communicates over TCP/IP, following best practices is critical to ensure reliability, performance, scalability, and security. Because IBM i systems are often central to core business operations, poorly implemented network communication can quickly become a single point of failure.

Aggregated Ethernet This is the preferred Native IBM i networking standard used to remove the physical Network Interface Connections (NIC) being a single point of failure. Using Aggregated Ethernet allows you to logically group 2 to 8 physical NICs from multiple network cards whilst all using a single IP address linked to a single line description.

It is possible to spread these aggregated NICs across network switches, which then allows you to remove the network switches from being a single point of failure. As well as introducing automatic failover and load balancing across these aggregated ports another benefit is that the overall speed of the of the line increases. So, for example, if you have 4 x 1Gbs NICs aggregated on a single ethernet line, you will effectively have a 4Gbs interface available to use. To find out more about Aggregated Ethernet, see Configuring Ethernet Link Aggregation.

> **Note:** If your IBM i LPAR is networking is virtualized using VIOS, then you should use VIOS Shared Ethernet Adaptors (SEA) instead of Aggregated Ethernet. TLS Encryption

The foundation of a secure application ais that all the network traffic that connects to it should be encrypted using Transport Layer Security (TLS). Any new application being written today should be using TLS 1.3. Any that are using TLS 1.2 should be migrated to TLS 1.3 as soon is is viable. For more information on IBM recommend protocols and ciphers, see Configuring Your IBM i System Secure Sockets Layer (SSL)/Transport Layer Security (TLS) Protocols and Cipher Suites.

Historically, an argument was made for not encrypting traffic that runs solely on a locally on the TCP stack or over internal virtual IP, this argument is now obsolete, and we need to assume that all traffic could be intercepted and so must be protected by encryption. If your application has certificates which are not managed by DCM, it is vital that you build robust and secure processes to manage them. For more information on configuring and managing TLS encryption for the Digital Certificate Manager (DCM), see Digital Certificate Manager.

If the overhead of encryption would place a performance burden on your application, then you should consider implementing a cryptographic coprocessor. For more information, see IBM PCIe Cryptographic Coprocessor.

Network Segmentation Where appropriate, consider binding different applications to different IP addresses and potentially configuring the different IP addresses to use different NICs. Don't assume that just because applications services exist on the same LPAR that it is appropriate for them to be able to be accessed by one another.

This combination of network and application segmentation will make it easier for your network team to build firewall rules that are better suited to each application. If an application is Soley used inside an LPAR and is only accessed by other applications resident on that LPAR, then consider binding this application to the Local Host 127.0.01 interface so that it is not directly addressable from outside of the LPAR.

Disable Network Services that are not used By default, IBM i runs a number of common network services that may not be required on all servers, these include listener jobs to SMNP, LPR, FTP, SMTP, POP, REXEC & SMB (Netserver) if these are not required, then they should be disabled. Disabling a listening service like SMTP or FTP does not stop you from sending outbound email or initialing secure file transfers FTPS or sFTP from your application. ProTip: When designing a new application, work out the minimum viable network services required for it to function. When testing modifications, repeat this process. Lock down access to management interface, some interfaces are designed to have additional management functions, these include but are not limited to TELNET, SSH and

Navigator, if you do need to have these applications running, limit access to them to just the users that need this and ensure that they will not respond to none encrypted traffic. From IBM i 7.6 onward, you can more easily achieve this for all IBM i native host services using the new CFGHOSTSVR command. For more information, see How to Disable TCP/IP Ports for IBM i Host Servers.

If you need to have the SSH listening job running on your server, limit which users are able to connect to SSH, and ensure those users that must connect authenticate using keys rather than passwords. For more information on limiting access to SSH, see Allowing or Denying Access to the IBM i Secure Shell Daemon (SSHD) Using Group Profiles.

Key Takeaways

Design your application communication based on the premise that any network traffic could become visible to third parties outside of the application, trust nothing by default and encrypt everything.

Patching your IBM i is more important than ever, this is now more than just applying the latest PTFs, it is necessary to update your Open-Source Components, Digital Certifications and API access tokens.

Avoid using older encryption protocols and ciphers, regularly check IBM support pages for current recommendations and ensure your application meets these standards.

Consider separating different groups of applications onto different NICs so as to segregate network traffic and make it easier for you to build more robust firewall rules on your network.

Disable network services that are not required and limited to access to those that are to solely to the users that require access to them.

What is ODBC/JDBC and what is Mapepire?

Introduction

IBM i sits on top of one of the most capable relational databases in the enterprise world, Db2 for i.

For applications running natively on the platform, accessing that data is straightforward, but for web applications, microservices, reporting tools and modern open-source frameworks running on other platforms, a connectivity layer is needed.

That is where ODBC and JDBC come in and where Mapepire is beginning to offer a compelling alternative.

What is ODBC?

ODBC, Open Database Connectivity, is a standard API developed by Microsoft in the early 1990s that allows applications to communicate with any relational database through a common interface.

The application talks to an ODBC driver and the driver handles the translation to the specific database underneath.

The application does not need to know whether it is talking to SQL Server, PostgreSQL, Oracle, or Db2 for i, the API is the same. IBM provides an ODBC driver for IBM i as part of IBM i Access Client Solutions (ACS). Once installed and configured on a Windows, Linux, or Mac machine, applications on that machine can connect to a remote IBM i server and query Db2 for i using standard SQL.

Tools like Microsoft Excel, Power BI, Python and many other platforms all support ODBC, making it a practical bridge between IBM i data and the wider analytics and integration ecosystem.

What is JDBC?

JDBC, Java Database Connectivity, is the Java equivalent of ODBC. It defines a standard set of interfaces in Java for connecting to relational databases, executing SQL and processing result sets.

Any Java application that uses JDBC can theoretically connect to any database that provides a JDBC driver, without the application code itself needing to change.

IBM provides the JT400 JDBC driver (also known as the Toolbox for Java driver) for Db2 for i. It is open source, actively maintained and available on Maven Central, making it straightforward to include in any Java or JVM-based project.

There is also a native JDBC driver available when Java runs directly on the IBM i partition itself.

JDBC is widely used in Java-based middleware, application servers (such as JBoss or WebSphere) and frameworks like Spring Boot. If your organisation has Java applications that need to read or write IBM i data, JDBC via JT400 is the standard approach.

What is Mapepire?

Mapepire is an open-source database connectivity project specifically designed for IBM i, developed under the IBM i Open Source community.

Its goal is to provide a modern, language-agnostic way to connect to Db2 for i from any environment, without requiring ODBC drivers or a JVM. At its core, Mapepire runs a lightweight server process on the IBM i server itself.

Client applications connect to that server over a WebSocket connection using a simple JSON-based protocol.

Because WebSockets are natively supported in virtually every modern programming language and runtime, Python, Node.js, Go, Rust, .NET and more — Mapepire makes it possible to query Db2 for i from practically anywhere, without installing any IBM-specific drivers on the client side.

ODBC/JDBC vs Mapepire: Which Should You Use?

These technologies are complementary rather than competing.

ODBC remains the right choice for desktop tools, BI platforms and other systems that expect an ODBC data source. Power BI, Excel and most commercial reporting tools will use ODBC.

JDBC is the right choice for Java applications and JVM-based frameworks where the JT400 driver integrates naturally into the dependency model.

Mapepire is the right choice for modern, lightweight, or cloud-native applications, especially those written in Python, Node.js or other languages where ODBC driver management is awkward, where async programming patterns are important or where you want to avoid any client-side driver dependency at all.

It is also an excellent fit for internal APIs and microservices that need to surface IBM i data to web front-ends.

The Bigger Picture

ODBC, JDBC and Mapepire all exist to solve the same fundamental problem, getting data out of and into Db2 for i from the outside world.

The right choice depends on your application architecture, your team's language preferences, and where your code runs. What matters most is that IBM i data no longer has to be isolated behind a green screen. With these connectivity options, Db2 for i is as accessible as any other enterprise database and with Mapepire in particular, that access is becoming progressively easier to achieve from the modern, open-source tooling that developers reach for by default.

Andy Youens March 2026

How to Configure TLS

Perhaps the easiest way to learn how to configure an IBM i TLS connection for a web server is to enable it for the IBM i Navigator interface itself. Even though this process does not configure TLS for your web server, you will be able to easily follow this process to enable TLS for your own webserver.

> **Note:** This is in effect a pre-requisite for your own web server, as it would be poor security to leave the web server admin interface unencrypted on an IBM i LPAR that needs to run encrypted web servers for production web servers.

You can get more information on this process and more at the following IBM website: Enabling TLS for IBM Navigator for i

Step 1: TLS Wizard

Navigator for i can be configured to use TLS using the **Network** → **Web Administration** → **Application Servers** → **ADMIN1** → **Configure TLS wizard** in Navigator for i.

The Navigator for i application server Configure TLS wizard is now available with the IBM i HTTP group update approved in 2024. IBM recommends utilizing the Navigator for i wizard to Configure or Re-configure your ADMINx application servers for TLS.

If Navigator for i is not available, another option is to execute the "Disable TLS" and "Configure TLS" wizards under **Manage** → **Application Servers** → **ADMIN1**

Navigator for i:

- Runs on the Admin1 HTTP server job using ports 2002 (Non-secure) and 2003 (with TLS configured)
- Non-TLS URL used to connect is `http://hostName:2002/Navigator`
- TLS URL is `https://hostName:2003/Navigator`

You can enable HTTPS by either using an existing certificate store or by using the Digital Certificate Manager *SYSTEM store.

Launch TLS Wizard

1. Click **Network** → **Web Administration** → **Application Servers**
2. Select **Admin1** on Application Servers list, right-click and select **Configure TLS Wizard**
3. Set TLS protocol. Also select if the Non-TLS port should be disabled after the wizard. Click **Next**
4. Select your intended certificate store using one of the following three options:

A. Using the DCM *SYSTEM Store:

Select this option to use the system certificate store.

B. Specify a path to the certificate store:

Provide the full path to your certificate store file.

C. Select using the certificate store table:

> **Note:** Certificate stores can be added to the table in **Network** → **Web Administration** → **Certificate Stores**. This process is meant to simplify the use of certificate stores other than the *SYSTEM store.

5. This will prompt the user to enter the certificate store password
6. Select an existing certificate from the searchable drop-down list populated from the certificate store location provided. On this step Navigator will check that the certificate is valid for hostname verification.

> **Note:** This step also informs the user of the certificate's expiry date. There are also warnings if the certificate is expired, or will expire in the next 2 weeks.

7. Select **Default Ciphers** and click **Next**
8. Confirm the information and click **Finish**

Step 2: Enabling Secure Connections

Turn on TLS Connections

1. Go to **Serviceability** → **Connection Properties** and select the **TLS Connection** tab
2. Test or set TLS Enablement
3. Before non-secure ports are disabled, an administrator should turn on Global TLS by setting **"Use TLS for All Users"**

Do I need a dedicated HTTP server over my Application Server?

What This Covers

This article explains the architectural considerations for using a dedicated Apache HTTP Server in front of WebSphere Liberty on IBM i versus accessing Liberty directly, helping you make an informed decision based on your specific requirements.

Understanding the Architecture Options

On IBM i, you have two primary approaches for serving web applications through WebSphere Liberty:

1. **Direct Access**: Clients connect directly to Liberty's HTTP endpoint
2. **Fronted Architecture**: Apache HTTP Server sits in front of Liberty, proxying requests

Both are valid architectures, but each serves different needs and use cases.

When You DON'T Need a Dedicated HTTP Server

Simple Application Scenarios

If your environment matches these characteristics, direct Liberty access may be sufficient:

- **Single application deployment**: You're running one web application or a small number of related applications

- **Internal-only access**: The application is accessed only within your corporate network
- **No SSL offloading requirements**: Liberty can handle your SSL/TLS termination needs
- **Minimal static content**: Your application serves primarily dynamic content
- **No complex routing**: You don't need sophisticated URL rewriting or request routing
- **Limited scalability requirements**: Your traffic patterns don't require load balancing across multiple Liberty instances

Liberty's Built-in Capabilities

Liberty includes robust HTTP server functionality:

- Native HTTP/HTTPS support with modern TLS protocols
- Efficient servlet container with low overhead
- Integrated security features
- Good performance for dynamic content
- Simplified configuration and management

When You SHOULD Use a Dedicated HTTP Server

Enterprise-Grade Requirements

A dedicated Apache HTTP Server becomes valuable when you need:

1. Load Balancing and High Availability

- Distribute traffic across multiple Liberty instances
- Implement failover strategies for zero-downtime deployments
- Session affinity (sticky sessions) for stateful applications
- Health checking and automatic instance removal

```
[Client] → [Apache] → [Liberty Instance 1]
                    → [Liberty Instance 2]
                    → [Liberty Instance 3]
```

> **Note:** This becomes less important/interesting if Load Balancing is performed upstream at a network level

2. SSL/TLS Offloading

- Centralize certificate management in one location (especially for large deployments)
- Reduce CPU overhead on Liberty instances by handling encryption/decryption at the front end
- Simplify certificate renewal and management processes
- Enable HTTP/2 and modern protocol support uniformly

3. Static Content Optimization

- Apache excels at serving static files (images, CSS, JavaScript, PDFs)
- Implement aggressive caching strategies
- Reduce load on Liberty for non-dynamic content
- Better performance for mixed content applications

4. Advanced Request Routing

- URL rewriting and redirection rules
- Virtual host management for multiple domains
- Path-based routing to different backend services
- API gateway functionality

5. Security Hardening

- Additional security layer (defense in depth)
- Web Application Firewall (WAF) integration
- Rate limiting and DDoS protection
- IP-based access control
- Request filtering and validation

6. Legacy Integration

- Serve CGI applications alongside modern Liberty apps
- Support for older protocols or authentication methods
- Bridge between different application generations
- Gradual migration path from legacy to modern architectures

7. Centralized Logging and Monitoring

- Unified access logs across all applications
- Centralized monitoring and analytics
- Easier troubleshooting with consolidated logs
- Integration with enterprise monitoring tools

8. Compliance and Audit Requirements

- Centralized audit trail for all HTTP traffic
- Compliance with security standards requiring layered architecture
- Separation of concerns for security reviews
- Easier demonstration of security controls

Performance Considerations

The Overhead Question

Adding Apache introduces a proxy hop, which adds minimal latency (typically 1-5ms) but provides:

- **Better throughput** for static content
- **Connection pooling** to backend Liberty instances
- **Request queuing** during traffic spikes
- **Compression** at the edge

When Direct Access Performs Better

For pure dynamic content with low traffic volumes, direct Liberty access may offer slightly better response times by eliminating the proxy hop.

IBM i Specific Considerations

Integrated Web Application Server

IBM i's integrated application server for Liberty provides:

- Tight integration with IBM i security
- Efficient use of system resources
- Simplified administration through IBM i interfaces
- Native support for IBM i objects and services

Apache HTTP Server on IBM i

The IBM HTTP Server for i (powered by Apache) offers:

- Native IBM i implementation
- Integration with IBM i security and user profiles
- Support for IBM i-specific modules
- Proven stability and performance on the platform

Decision Framework

Choose Direct Liberty Access When:

√ You have a single application or small application portfolio √ All access is internal to your organization √ You don't need load balancing or high availability √ Your application is primarily dynamic content √ You want to minimize architectural complexity √ You have limited resources for infrastructure management

Choose Apache HTTP Server Front-End When:

√ You need to load balance across multiple Liberty instances √ You serve significant static content √ You require advanced security features or WAF integration √ You need sophisticated URL routing or rewriting √ You're running multiple applications that need unified access √ You have compliance requirements for layered security √ You need SSL offloading for performance √ You're integrating legacy and modern applications

Migration Path

You can start with direct Liberty access and add Apache later if needs evolve:

1. **Phase 1**: Deploy Liberty with direct access for initial rollout
2. **Phase 2**: Add Apache as traffic grows or requirements change
3. **Phase 3**: Implement load balancing as you scale horizontally

This approach minimizes initial complexity while preserving future flexibility.

Best Practice Recommendation

For **production enterprise applications**, especially those serving external users or requiring high availability, the Apache + Liberty architecture is recommended despite the additional complexity. The benefits in security, scalability, and operational flexibility typically outweigh the overhead.

For **internal tools, development environments, or simple applications**, direct Liberty access provides a simpler, more maintainable solution.

Key Takeaways

- Direct Liberty access is simpler and sufficient for many internal applications and development scenarios
- A dedicated Apache HTTP Server provides enterprise-grade features like load balancing, SSL offloading, and advanced security

- The decision should be based on your specific requirements for scalability, security, performance, and operational complexity
- You can start simple with direct access and add Apache as your needs evolve
- On IBM i, both options are well-supported and integrate seamlessly with platform security and management tools
- Consider your long-term scalability and high availability requirements when making the architectural decision

What are SOAP vs REST web services?

What This Covers

This article explains the fundamental differences between SOAP and REST web services, their use cases, and how IBM i's Integrated Web Services (IWS) supports both approaches for exposing RPG programs and service programs as web services.

Understanding Web Services

Web services enable applications to communicate over networks using standard protocols. They allow your IBM i business logic to be consumed by external applications, mobile apps, web applications, or other systems regardless of platform or programming language.

IBM i's **Integrated Web Services (IWS)** provides native tooling to expose your existing RPG, COBOL, and CL programs as web services without requiring code changes to your core business logic.

SOAP Web Services

What is SOAP?

SOAP (Simple Object Access Protocol) is a protocol-based approach to web services that uses XML for message formatting and typically runs over HTTP/HTTPS.

Key Characteristics:

- **Protocol-driven**: Strict standards and specifications (WS-* standards)
- **XML-only**: All messages are XML-formatted
- **Contract-first**: Requires a WSDL (Web Services Description Language) document
- **Stateful or stateless**: Can maintain state between requests
- **Built-in error handling**: Standardized fault messages
- **Transport agnostic**: Can use HTTP, SMTP, TCP, or other protocols

SOAP Message Structure:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Header>
    <!-- Optional header information -->
  </soap:Header>
  <soap:Body>
    <GetCustomer xmlns="http://example.com/services">
      <customerId>12345</customerId>
    </GetCustomer>
  </soap:Body>
</soap:Envelope>
```

SOAP in IBM i IWS

When you deploy a SOAP web service using IWS:

1. **WSDL Generation**: IWS automatically generates a WSDL from your RPG program parameters

2. **XML Mapping**: Input/output parameters are automatically mapped to/from XML
3. **Type Safety**: Strong typing ensures data integrity
4. **Standards Compliance**: Full support for WS-Security, WS-Addressing, and other standards

IWS SOAP Advantages:

- **Automatic conversion**: Data structures, arrays, and complex types are handled automatically
- **Enterprise integration**: Works seamlessly with enterprise service buses (ESBs)
- **Security standards**: Built-in support for WS-Security for authentication and encryption
- **Transaction support**: Can participate in distributed transactions
- **Formal contracts**: WSDL provides clear interface documentation

REST Web Services

What is REST?

REST (Representational State Transfer) is an architectural style (not a protocol) that uses standard HTTP methods and can support multiple data formats.

Key Characteristics:

- **Architecture-driven**: Follows REST principles (stateless, cacheable, uniform interface)
- **Multiple formats**: Supports JSON, XML, plain text, or other formats
- **Resource-oriented**: URLs represent resources, HTTP methods represent actions
- **Stateless**: Each request contains all necessary information
- **Lightweight**: Simpler message structure, less overhead
- **HTTP-native**: Uses standard HTTP methods (GET, POST, PUT, DELETE, PATCH)

REST Request Examples:

```
GET /api/customers/12345 HTTP/1.1
Host: example.com
Accept: application/json

POST /api/customers HTTP/1.1
Host: example.com
Content-Type: application/json

{
  "name": "John Doe",
  "email": "john@example.com"
}
```

REST in IBM i IWS

IWS provides robust REST support with both JSON and XML capabilities:

1. **Flexible deployment**: Deploy the same RPG program as REST with JSON or XML
2. **HTTP method mapping**: Map RPG procedures to GET, POST, PUT, DELETE operations
3. **URI templates**: Define resource-oriented URLs
4. **Content negotiation**: Support multiple response formats from the same service

IWS REST Advantages:

- **JSON support**: Native JSON parsing and generation for modern applications
- **XML support**: Traditional XML for legacy system integration

- **Simpler consumption**: Easier for web and mobile developers to consume
- **Better performance**: Less overhead than SOAP for simple operations
- **Caching**: HTTP caching mechanisms improve performance
- **Modern integration**: Ideal for mobile apps, SPAs, and microservices

IBM i IWS: Supporting Both Approaches

Unified Deployment Model

IWS allows you to deploy the **same RPG program** as both SOAP and REST services:

```
RPG Program/Service Program
         ↓
    IWS Server
    ↙        ↘
SOAP Service  REST Service
(XML)         (JSON/XML)
```

Data Format Support

JSON (JavaScript Object Notation)

- **Modern standard**: Preferred format for web and mobile applications
- **Lightweight**: Smaller payload size than XML
- **Native JavaScript**: Direct consumption in web browsers
- **Human-readable**: Easy to debug and understand

Example JSON response:

```
{
  "customerId": "12345",
  "name": "John Doe",
  "email": "john@example.com",
  "orders": [
    {"orderId": "ORD-001", "amount": 150.00},
    {"orderId": "ORD-002", "amount": 275.50}
  ]
}
```

XML (eXtensible Markup Language)

- **Enterprise standard**: Widely used in B2B integration
- **Schema validation**: Strong typing with XSD schemas
- **Transformation**: XSLT for data transformation
- **Namespace support**: Prevents naming conflicts

Example XML response:

```
<customer>
  <customerId>12345</customerId>
  <name>John Doe</name>
  <email>john@example.com</email>
  <orders>
    <order>
      <orderId>ORD-001</orderId>
      <amount>150.00</amount>
    </order>
    <order>
      <orderId>ORD-002</orderId>
      <amount>275.50</amount>
    </order>
  </orders>
</customer>
```

Comparison Matrix

Aspect	SOAP	REST
Protocol	Strict protocol with standards	Architectural style using HTTP
Data Format	XML only	JSON, XML, or others
Message Size	Larger (XML envelope overhead)	Smaller (especially with JSON)
Learning Curve	Steeper (complex standards)	Gentler (uses familiar HTTP)
Tooling	Extensive enterprise tools	Simpler, widely available tools
Security	WS-Security standards	HTTPS, OAuth, JWT
State	Can be stateful	Stateless by design
Caching	Difficult	HTTP caching built-in
Error Handling	Standardized SOAP faults	HTTP status codes
Best For	Enterprise integration, formal contracts	Mobile apps, web apps, microservices

When to Use SOAP

Ideal SOAP Scenarios:

√ **Enterprise B2B integration**: Trading partners require SOAP √ **Formal contracts required**: WSDL provides explicit interface definition √ *WS- standards needed**: Security, transactions, reliable messaging √ **Complex operations**: Multi-step transactions or workflows √ **Legacy system integration**: Existing SOAP infrastructure √ **ACID transactions**: Need distributed transaction support √ **Asynchronous processing**: Built-in support for async operations

IBM i IWS SOAP Use Cases:

- Integration with SAP, Oracle, or other enterprise systems

- Financial services requiring WS-Security
- Healthcare systems using HL7 standards
- Government systems with strict compliance requirements
- Legacy modernization maintaining existing SOAP contracts

When to Use REST

Ideal REST Scenarios:

✓ **Mobile applications**: Lightweight, JSON-based communication ✓ **Web applications**: Modern SPAs and progressive web apps ✓ **Public APIs**: Easy consumption by third-party developers ✓ **Microservices**: Service-to-service communication ✓ **CRUD operations**: Simple create, read, update, delete operations ✓ **Performance critical**: Lower overhead and better caching ✓ **Cloud-native**: Modern cloud architectures

IBM i IWS REST Use Cases:

- Mobile apps accessing IBM i data
- Web portals consuming IBM i business logic
- IoT devices sending data to IBM i
- Modern e-commerce integrations
- Real-time dashboards and analytics
- API-first application development

IBM i IWS Implementation Considerations

Deployment Process

Both SOAP and REST services in IWS follow a similar deployment pattern:

1. **Identify the program**: Select RPG/COBOL program or service program
2. **Configure parameters**: Map input/output parameters
3. **Choose service type**: SOAP or REST (or both)
4. **Select data format**: XML for SOAP; JSON or XML for REST
5. **Deploy**: IWS generates the service endpoint
6. **Test**: Use built-in testing tools or external clients

Security Integration

IWS integrates with IBM i security:

- **User profiles**: Authenticate using IBM i user profiles
- **Authorization lists**: Control access to services
- **SSL/TLS**: Secure communication channels
- **Exit programs**: Custom authentication and authorization logic

Performance Optimization

- **Connection pooling**: Reuse program activations
- **Caching**: HTTP caching for REST services
- **Compression**: Reduce payload size
- **Asynchronous processing**: For long-running operations

Hybrid Approach: Best of Both Worlds

Many organizations use **both** SOAP and REST:

- **SOAP for B2B**: Enterprise partner integration
- **REST for B2C**: Customer-facing mobile and web apps
- **Same business logic**: IWS exposes the same RPG programs both ways

This approach maximizes flexibility while leveraging existing investments in IBM i business logic.

Migration Strategy

From SOAP to REST

If you have existing SOAP services and want to modernize:

1. **Parallel deployment**: Deploy REST alongside existing SOAP
2. **Gradual migration**: Move consumers to REST incrementally
3. **Maintain SOAP**: Keep for partners who require it
4. **Leverage IWS**: Same RPG program serves both

From Custom APIs to IWS

If you have custom-built web services:

1. **Assess current services**: Document existing functionality
2. **Deploy via IWS**: Expose same programs through IWS
3. **Test thoroughly**: Ensure compatibility
4. **Migrate consumers**: Update client applications
5. **Retire custom code**: Reduce maintenance burden

Key Takeaways

- SOAP is a protocol-based approach using XML, ideal for enterprise integration with formal contracts and advanced security requirements
- REST is an architectural style using HTTP methods and supporting multiple formats (JSON/XML), ideal for modern web and mobile applications
- IBM i's Integrated Web Services (IWS) natively supports both SOAP and REST, allowing you to expose the same RPG programs through either approach
- JSON is the preferred format for REST services in modern applications due to its lightweight nature and native JavaScript support
- XML remains important for enterprise B2B integration and systems requiring schema validation
- You can deploy the same business logic as both SOAP and REST services, choosing the best approach for each consumer
- The choice between SOAP and REST should be driven by your integration requirements, consumer preferences, and architectural goals
- IWS eliminates the need to modify your core RPG business logic, providing automatic conversion between program parameters and web service formats

How do I create REST services?

Introduction

REST (Representational State Transfer) services are nowadays widely used by web applications to enable communication via HTTP. It follows REST architecture style, allowing stateless transfer, utilization of HTTP methods (GET, POST, PUT, DELETE), and client-server model.

This section of the chapter will highlight numerous ways to create a REST service on IBM i.

Example Users API

Throughout this section, we will implement one example REST web service to fetch user details (GET HTTP method) using different approaches.

The request endpoint includes user id for which the record will be fetched from the database. Example – 'users/U1001'

The JSON response will be:

```
{
  "userId": "U1001",
  "name": "Test User 1",
  "email": "test.user1@example.com",
  "status": "active"
}
```

1. Leveraging HTTP server (powered by Apache)

An HTTP server is responsible for handling & routing the client requests from a browser and send back the responses via HTTP/HTTPS protocol.

On IBM i, you can create an HTTP server by using IBM Navigator under the Network - Web Admin section. Additionally they can be created within the IBM Web Administration for i until its depricate.

Step by step process

1. Open IBM Web Administration for i: `http://<ibmi-host>:2001/HTTPAdmin`
2. Create a new server HTTP Server.

or

1. Open IBM Navigator `http://<ibmi-host>:2002/Navigator`
2. Create a new server HTTP Server. Under Network - Web Admin - HTTP Server
3. Set the configuration in the configuration file of the server (httpd.conf) to allow the HTTP server to access the required objects. For example, to create the example 'users' REST service, below configuration will work. (Leave the default configuration as it is)

```
ScriptAliasMatch /users/([a-z0-9]+) /Qsys.Lib/usersdtl.Lib/$1.pgm
<Directory /Qsys.Lib/usersdtl.Lib>
 Require all granted
</Directory>
```

This will allow the server to access all the objects in USERDTL library. 4. Start the HTTP server. 5. Create an RPG ILE CGI program: This program should be created in the configured library (USERDTL).

The program will include the logic to get the client request (URL, request body, request header, etc.), fetch the user details from the database in JSON format, and send back the response to the HTTP server. For more information, see CGI programming.

6. Access the created REST web service using configured URL. `http://<ibmi-host>:<port>/users/GETUSERDTL.pgm/U1001`

When to use

1. While building interactive web application using HTML, CSS, and JavaScript.
2. To incorporate all the latest features of Apache.
3. No external dependencies.
4. To implement the backend logic using CGI Programming.

2. Leveraging IWS Server

Integrated Web Services (IWS) server on IBM i provides the capabilities to externalize existing programs running on IBM i, such as RPG or COBOL as web services. IWS server allows you quickly build and deploy an existing ILE object as a web service with no up-front cost and without the need to rewrite the core infrastructure.

Step by step process

1. Open IBM Web Administration for i: `http://<ibmi-host>:2001/HTTPAdmin` or
2. Open IBM Navigator for i: `http://<ibmi-host>:2002/Navigator` Under Network > Web Administration
3. Create a new Web Services Server.
4. Start the IWS server.
5. Click on 'Deploy a new service' to deploy an ILE object as a web service. This step will allow you to deploy an existing program or service program (*SRVPGM) running on IBM i as a web service & set the required configuration like the request URL.

4. Access the REST web service using configured URL, like: `http://ibmi-host:port/web/services/GETUSERDTL/users/U1001`

When to use

1. To externalize existing RPG ILE program as a web service without rewriting core infrastructure.
2. To quickly build and deploy a service with no up-front cost.
3. To utilize its built-in security control features.

3. Node.js

Using Node.js is a robust approach to create a REST web service on IBM i. It provides the required environment to build scalable web application and manage concurrent request efficiently.

Express is a widely used framework in Node.js for building web services. It allows easy routing & middleware management. It also has the capabilities to create REST service and effortlessly manage HTTP requests.

Prerequisites Node & npm (Node package manager) should be installed on the server. If not yet installed, install it via Open Source Package Management option available on IBM i Access Client Solutions (ACS) or install it via yum (a command line tool to install packages).

Step by step process

1. Open PASE environment on the IBM i using VS code or by calling the command 'Call QP2TERM'.
2. Create a project directory on the IFS & make it as your current directory.
3. Initialize the project: npm init -y

4. Install all the necessary dependencies in your project directory: a. Install express `npm install express`

   b. Install any database connector like: `npm install idb-connector`
5. Create a JavaScript server program. This program will include the logic to create the express application, establish connection with db2 for i, process the API request, and generate API response.
6. Run the server program: `node myserver.js`
7. Lastly, invoke the created REST web service by requesting the URL. For example: `http://<ibmi-host>:<port>/users/U1001`

When to use

1. To utilize numerous packages available through npm (Node package manager).
2. To seamlessly integrate with existing IBM i applications.
3. To build modern and scalable web application.
4. Require event-driven architecture.

4. Python

Python is nowadays one of the widely used programming language for application development. On IBM i, we can also use Python for creating a REST web service like the users API to fetch user information.

Flask is a web server gateway interface (WSGI) framework in Python. It is highly suitable for building a REST web service, as it is lightweight and provides capabilities to scale up complex applications easily.

Now we will focus on the fundamentals to make use of Python on IBM i to deliver incredibly scalable web services.

Prerequisites

1. The desired python version should be installed.
2. pip (python package manager) should be installed.
3. IBM i 7.2 or higher.
4. Choose and install a framework, like Flask.

Step by step process

1. Open PASE environment on the IBM i using VS code or by calling the command 'Call QP2TERM'.
2. Create a project directory on the IFS & make it as your current directory.
3. Install the required dependencies, like `pyodbc` to enable database connectivity.
4. Create the python script in the project directory to establish connection, process the API request, and generate API response.
5. Run the python application: python app.py
6. Invoke the created REST web service by requesting the URL, like: `http://<ibmi-host>:<port>/users/U1001`

When to use

1. To make use of vast ecosystem of frameworks and libraries for web development.
2. To additionally introduce automation in the application.
3. For rapid development.

Conclusion

IBM i provides numerous ways to create a REST services. The most suitable approach differs based on your use case. By leveraging the right approach, you can create robust and secure application.

How do I use Integrated Web Services Server?

Introduction

Integrated Web Services (IWS) server on IBM i provides the capabilities to externalize existing programs running on IBM i, such as RPG or COBOL as web services. With IWS, you can quickly build and deploy an existing ILE object as a web service with no up-front cost. This allows you to focus on enhancing the functionalities & user experience, without the need to rewrite the core infrastructure.

This section of the chapter will walk you through the entire process to use an Integrated Web Services Server on IBM i, note: In the future IBM Navigator for i will be the preferred way to create an IWS server.

#1. Create an IWS server The initial step is to create an IWS server using IBM Web Administration for i.

1. Open IBM Web Administration for i: `http://<ibmi-host>:2001/HTTPAdmin`
2. Click on 'Create a new Web Services Server'.

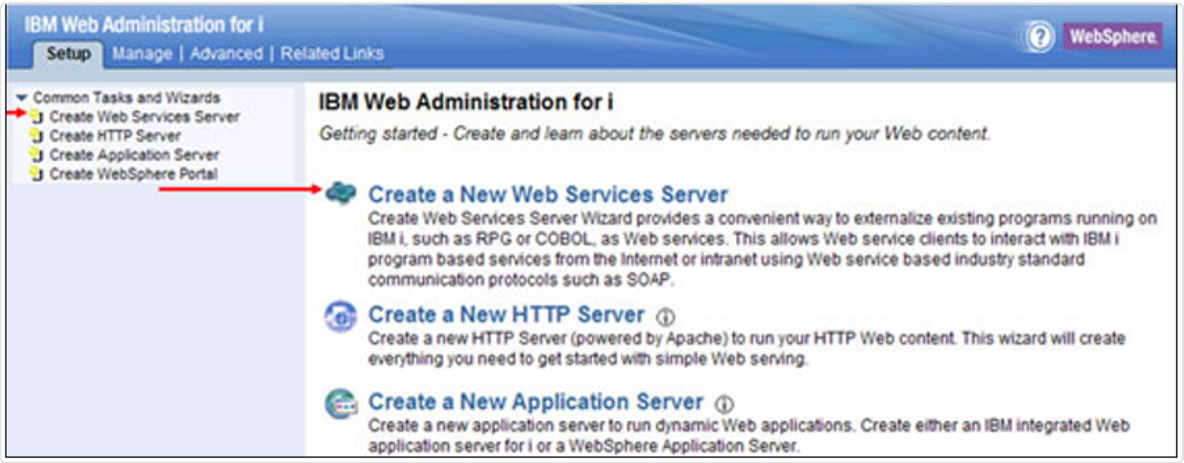


Figure. Create IWS server wizard

3. Creating the server is a 5-step process.

- Step 1: Specify the name of the server.
- Step 2: Specify the network attributes of the server. This includes the IP address & the PORT number which the server will listen.
- Step 3: The third step displays the default configuration for the subsystem under which the server jobs will run, for example QHTTPSVR. You can change this configuration if you want to run the jobs in some other subsystem.
- Step 4: Specify the runtime user profile for the server. You can also use the default user profile, QWSERVICE.
- Step 5: In the last step, server information will be displayed, click on Finish to create the server.

4. Start the created server. You will see the status as `RUNNING` once the IWS server is started.

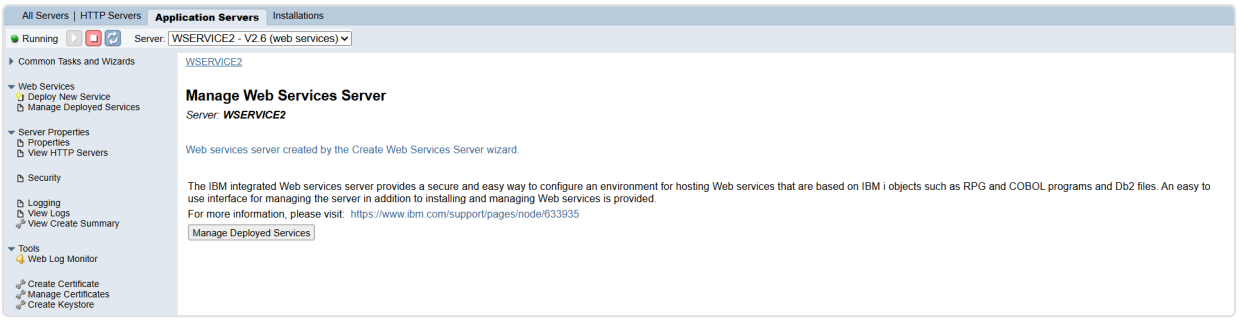


Figure. IWS Server status Output

5. Select any existing ILE object which you want to externalize as a web service. For example, an RPG ILE program or a service program (*SRVPGM) which accepts customer id as input parameter and returns corresponding customer information from the database. To make this ILE object compatible to be utilized by the IWS server as a web service, Program Call Markup Language (PCML) comes in use. It is a tag language that describes all input and output parameters of programs. It is used to generate Java proxy code that performs the call to the web service implementation code that is written in RPG or COBOL.
6. Deploy the ILE object as a service on IWS server. It includes the following steps:

a. The process begins by specifying the type of web service you want to create (REST or SOAP). It also involves selecting the path to the ILE program.

Figure. Deploy Service step 1

b. Specify the name of the resource to be created (example: GETCUSTDTL) & the URL path template to call the web service. Example: `/customer/{cust:\w+}`. It signifies, the API request endpoint must contain `/customer` & the customer id (\w+ is a regex pattern for character value, like 'C1001').

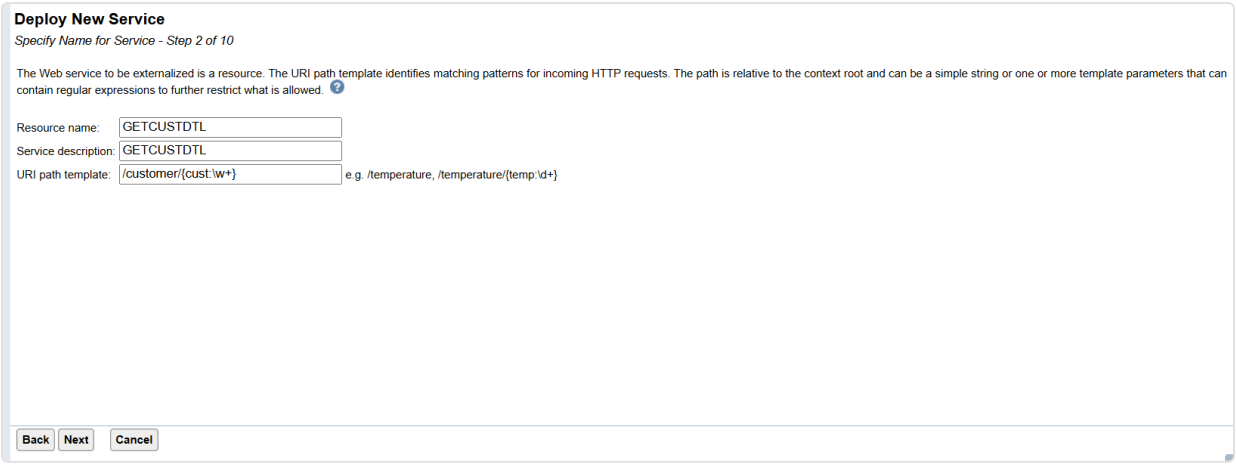


Figure. Deploy Service step 2

c. Specify the security constraints like the API authentication method.

d. In the fourth step, the procedure/parameter names from the ILE object are listed, along with their usage (input/output). It allows you to determine the API request & response data. For example, `customerId` will be input parameter (part of API request) & other customer details like name, email, etc. will be output parameters (part of API Json response).

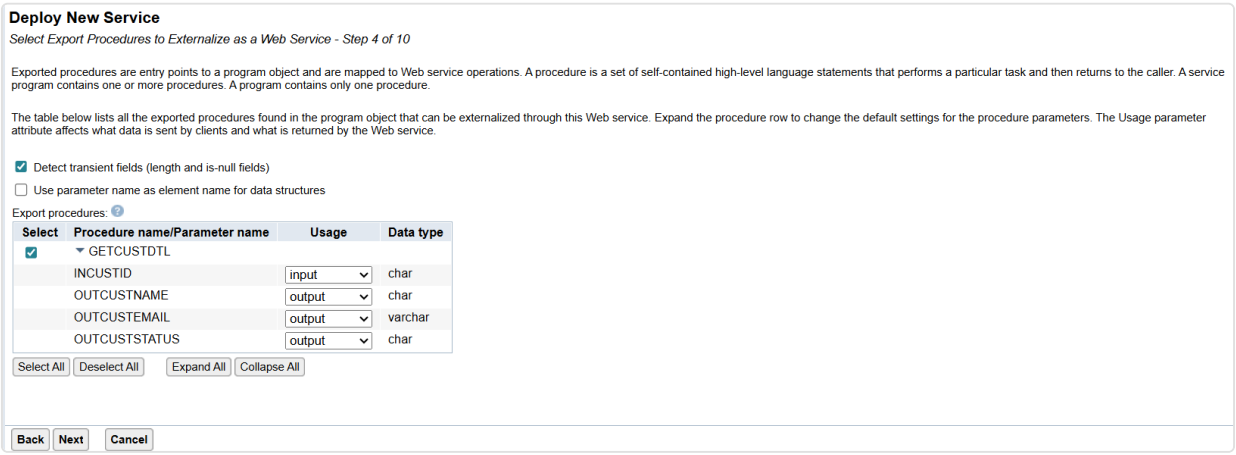


Figure. Deploy Service step 4

e. In the fifth step, you can control the API call details like the HTTP status code for success & failure.

f. This is an important step where you decide information like the HTTP request method (GET/PUT/POST/DELETE) based on your API, HTTP header, response & request media type like *JSON. Also, you need to select the input source for the ILE program input parameter (custid). For example, *PATH_PARAM requires the customer id to be specified in the URL path.

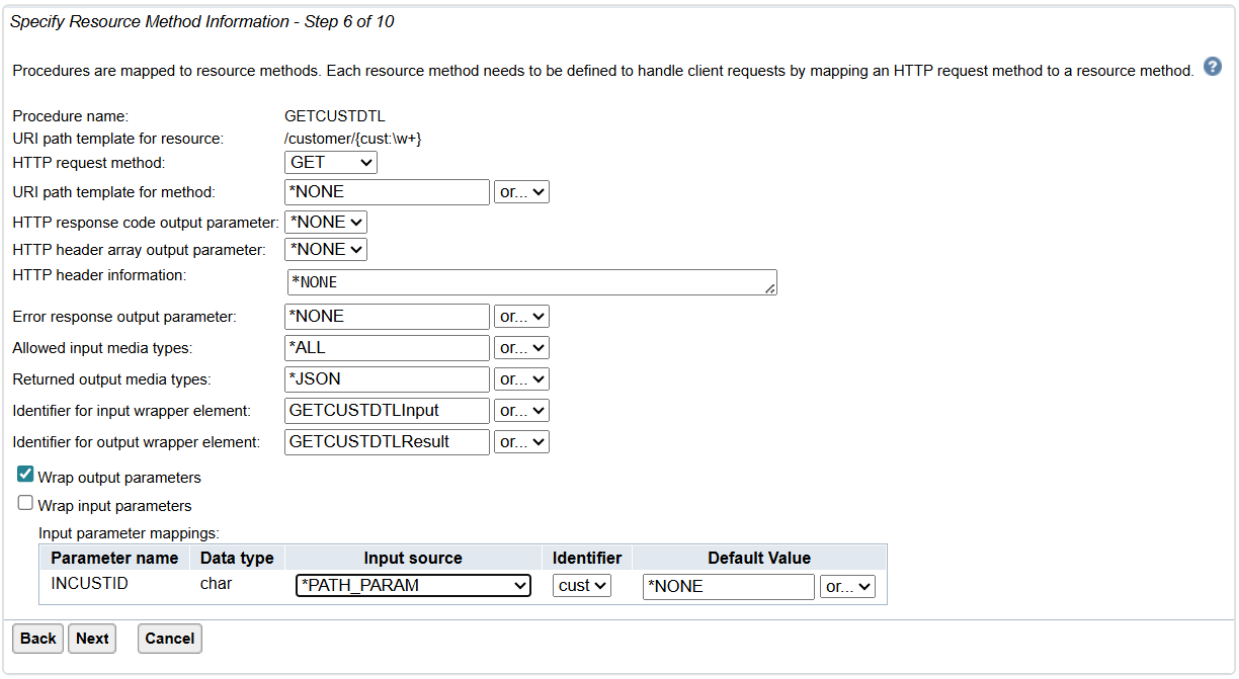


Figure. Deploy Service step 6

g. Specify the IBM i user id under which the web service will run.

h. In the eighth step, you can configure the library list required to run the ILE program in the backend.

i. In this step, you can access any of the listed environment variables while calling the API. For the `/customer` API, none of the entries are required.

j. At the last step, all the web service information will be displayed, click on Finish to deploy the service.

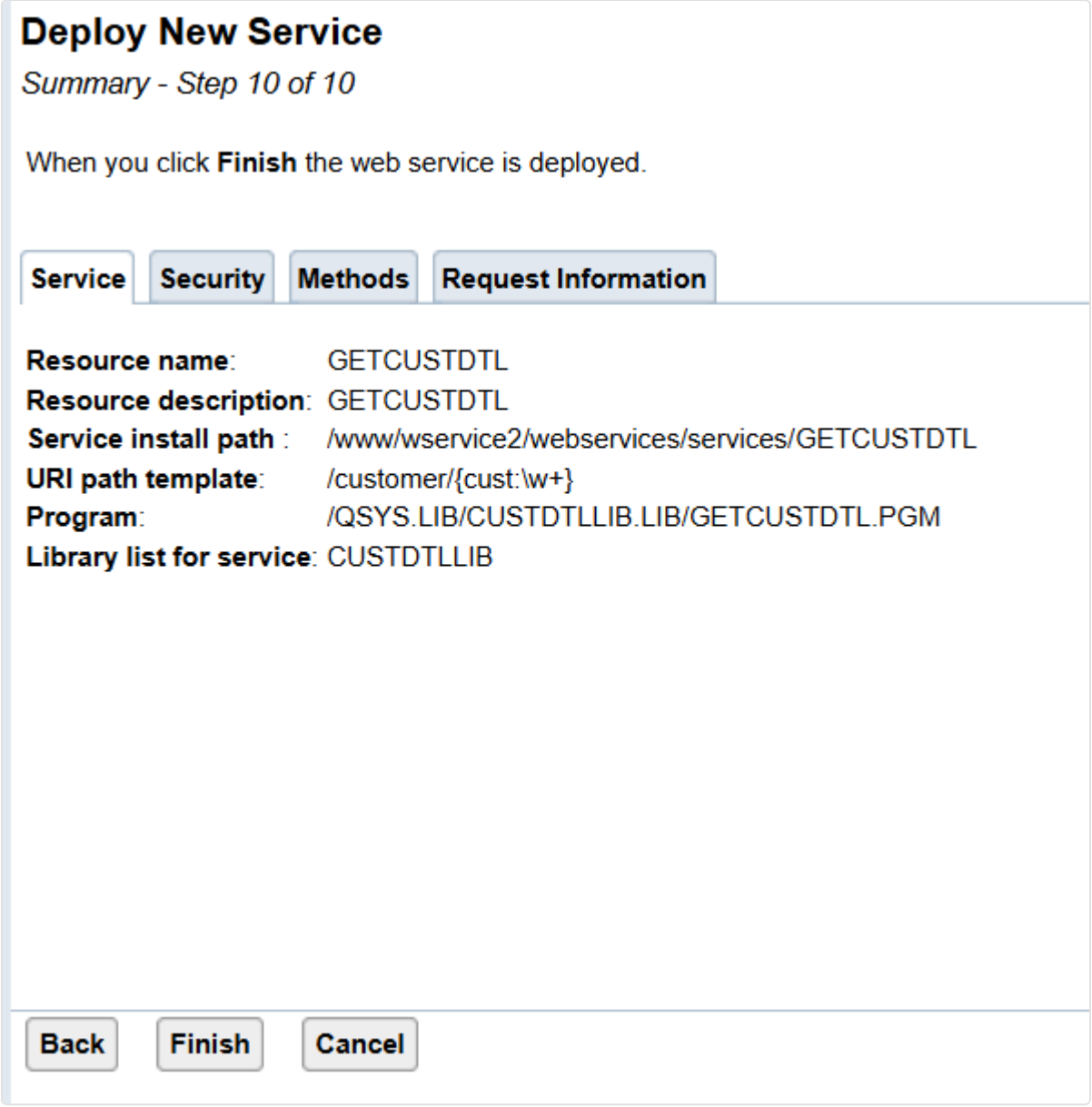


Figure. Deploy Service step 10

7. Test the deployed service Before testing the deployed service, make sure it is in `Running` status on the 'Manage Deployed Service' wizard. On the same wizard, you can view the Swagger definition to know what the web service does before testing it.

You can access the created web service via any API testing tools. For example, the customer API request will be: URL: `http://ibmi-host:port/web/services/GETCUSTDTL/customer/C1001` HTTP Request Method: `GET`

To check the status of the jobs running under the created IWS server, run the command `WRKACTJOB job(IWS_SERVER)`.

Conclusion

The IWS server on IBM i provides a robust and flexible way to expose your business logic and data as a modern web service. You can create multiple services on one IWS server. Using IWS server evidently helps to enhance the value of an application on IBM i by making it a web service without major modifications in the architecture.

How do I deploy on Integrated Application Server?

What This Covers

This article explains how to deploy custom applications on IBM i's Integrated Application Server, which runs on WebSphere Liberty, providing a modern, lightweight Jakarta EE application server tightly integrated with IBM i.

Understanding the Integrated Application Server

What is the Integrated Application Server?

The **Integrated Application Server** is IBM i's native application server based on WebSphere Liberty. It provides a full-featured EE runtime environment that is:

- **Integrated**: Deeply integrated with IBM i security, database, and system services
- **Lightweight**: Fast startup times and efficient resource usage
- **Modern**: Supports Jakarta EE standards
- **Flexible**: Supports servlets, JSPs, REST services, JPA, and more
- **Scalable**: Multiple server instances can run simultaneously

Key Benefits

Native IBM i Integration:

- Direct access to Db2 for i without external drivers
- Integration with IBM i user profiles and authorities
- Native file system access
- Call RPG, COBOL, and CL programs directly
- Participate in IBM i commitment control

Simplified Management:

- Managed through IBM i interfaces (Navigator, commands)
- Integrated with IBM i security model
- Automatic startup and monitoring
- Centralized logging and diagnostics

Modern Development:

- Support for modern Java frameworks
- RESTful web services
- Microservices architectures
- Cloud-ready applications

Deployment Methods

1. Administrative Console Deployment

The IBM i Navigator for i provides a web-based interface for deploying applications.

Steps:

1. Open IBM Navigator for i in web browser
2. Navigate to **IBM i Management** → **Application Administration** → **Application Servers**
3. Select your application server instance
4. Click **Deploy Application**
5. Browse to your WAR/EAR file

6. Configure application settings (context root, name)
7. Click **Deploy**

Best For:

- Interactive deployments
- Testing and development
- Small-scale deployments
- Learning and exploration

2. Command Line Deployment

Use IBM i CL commands for scripted and automated deployments.

Approach:

- Use STRQSH to execute Liberty commands
- Modify server.xml configuration file directly
- Copy application files to appropriate directories
- Restart server to apply changes

Best For:

- Automated deployments
- CI/CD pipelines
- Production deployments
- Repeatable processes

3. Dropins Directory

Place applications in the dropins directory for automatic deployment.

Process:

1. Copy WAR file to the server's dropins directory
2. Server automatically detects and deploys the application
3. No server restart required

Dropins Directory Location:

```
/www/[server-name]/[instance-name]/dropins/
```

Best For:

- Quick deployments
- Development environments
- Simple applications
- Rapid testing

4. Server Configuration File

Add application definitions directly to server.xml for controlled deployment.

Approach:

- Edit server.xml in the server instance directory
- Add webApplication or enterpriseApplication elements

- Specify location, context root, and other settings
- Server automatically loads configured applications

Best For:

- Production environments
- Version-controlled deployments
- Complex configurations
- Multiple applications

Application Types

Web Applications (WAR)

Characteristics:

- Single WAR file containing web application
- Includes servlets, JSPs, HTML, CSS, JavaScript
- WEB-INF directory contains classes and libraries
- Deployed with specific context root

Typical Structure:

- WEB-INF/web.xml (deployment descriptor, optional)
- WEB-INF/classes/ (compiled Java classes)
- WEB-INF/lib/ (JAR dependencies)
- Static resources (HTML, CSS, JS, images)

URL Access:

```
https://[hostname]:[port]/[context-root]/
```

Enterprise Applications (EAR)

Characteristics:

- Contains multiple modules (WAR, EJB JAR, etc.)
- META-INF/application.xml defines structure
- Shared libraries across modules
- More complex than WAR deployments

Best For:

- Complex enterprise applications
- Multiple integrated modules
- Shared resources and libraries

Loose Applications

Characteristics:

- Deploy without packaging into WAR/EAR
- Point to exploded directory structure
- Faster development cycle
- Changes detected automatically

Best For:

- Development environments
- Rapid iteration and testing
- Debugging
- Hot deployment scenarios

Server Configuration

Creating a Server Instance

Using Navigator:

1. Navigate to **Application Servers** in Navigator
2. Click **New Server**
3. Specify server name and HTTP port
4. Configure initial features
5. Start the server

Key Configuration Elements:

- Server name and description
- HTTP and HTTPS ports
- Enabled Liberty features
- Security settings
- Data source definitions

Configuring Features

Liberty uses a feature-based architecture. Enable only the features your application needs.

Common Feature Categories:

- **Web**: servlet, jsp, websocket
- **REST**: restfulWS, jsonp, jsonb
- **Database**: jdbc, jpa
- **Security**: appSecurity, ssl
- **Management**: localConnector, restConnector

Feature Configuration: Features are specified in the featureManager section of server.xml. Only enabled features are loaded, keeping the server lightweight.

Configuring Data Sources

Db2 for i Data Source:

- Use JT400 (jt400.jar) for native IBM i database access
- Configure JNDI name for application lookup
- Set connection pool parameters
- Specify database name (*LOCAL for local database)

Connection Pool Settings:

- Maximum pool size
- Minimum pool size
- Connection timeout
- Maximum idle time

- Reap time for idle connections

Security Configuration

Authentication Options:

- Basic registry (username/password in server.xml)
- IBM i user profiles (native integration)
- LDAP registry
- Custom user registry

IBM i User Profile Integration: Applications can authenticate using IBM i user profiles, providing seamless integration with existing security infrastructure.

SSL/TLS Configuration:

- Configure keystore for SSL certificates
- Enable HTTPS endpoint
- Specify SSL protocols and cipher suites
- Configure client certificate authentication if needed

Deployment Best Practices

1. Environment-Specific Configuration

Use Variable Substitution:

- Define variables in bootstrap.properties
- Reference variables in server.xml using ${variable.name}
- Maintain separate properties files per environment
- Keep sensitive data out of version control

Benefits:

- Same server.xml across environments
- Easy environment promotion
- Reduced configuration errors

2. Application Versioning

Strategies:

- Include version in application name
- Use separate context roots for versions
- Enable side-by-side deployments
- Facilitate rollback if needed

3. Resource Management

Configure Appropriate Limits:

- Connection pool sizes based on load
- Thread pool sizes for concurrent requests
- JVM heap size for application memory needs
- Timeout values for operations

Monitor Resource Usage:

- Connection pool utilization
- Thread pool statistics
- Memory consumption
- Response times and throughput

4. Logging Configuration

Configure Logging Levels:

- Set appropriate trace specifications
- Configure log file rotation
- Specify maximum file sizes
- Define retention policies

Log Locations:

- Messages log: General server messages
- Trace log: Detailed diagnostic information
- FFDC: First Failure Data Capture for errors

5. Health Monitoring

Enable Health Checks:

- Configure readiness checks
- Configure liveness checks
- Expose health endpoints
- Integrate with monitoring systems

Monitor Application Health:

- Response times
- Error rates
- Resource utilization
- Availability

Managing Deployed Applications

Starting and Stopping

Using Navigator:

- Navigate to application in Application Servers
- Use Start/Stop/Restart actions
- View application status

Using Commands:

- STRTCPSVR for starting HTTP server
- ENDTCPSVR for stopping HTTP server
- Commands work with server instances

Updating Applications

Development Approach:

- Use loose applications for hot deployment
- Changes detected automatically
- Fast iteration cycle

Production Approach:

- Deploy new version alongside old
- Update routing to new version
- Drain connections from old version
- Remove old version after validation

In-Place Update:

- Stop application
- Replace application file
- Start application
- Verify functionality

Monitoring

Check Server Status:

- Use Navigator to view server state
- Check HTTP endpoint accessibility
- Review active applications

View Logs:

- Access logs through Navigator
- Use DSPF command for IFS log files
- Monitor for errors and warnings

Performance Monitoring:

- Use Navigator performance monitoring
- Review thread dumps for issues
- Analyze connection pool statistics
- Monitor JVM memory usage

Troubleshooting

Application Won't Start

Check These Areas:

1. Review messages.log for errors
2. Verify all required features are enabled
3. Check for port conflicts
4. Validate server.xml syntax
5. Ensure all dependencies are available

Application Not Accessible

Verify:

1. Server is running
2. HTTP endpoint is configured correctly
3. Firewall allows traffic on configured ports
4. Context root matches URL
5. Application started successfully

Performance Issues

Investigate:

1. Enable detailed tracing
2. Check connection pool exhaustion
3. Review thread pool utilization
4. Analyze database query performance
5. Check for memory pressure

Common Causes:

- Insufficient connection pool size
- Thread pool saturation
- Memory leaks
- Inefficient database queries
- Resource contention

Memory Issues

Address By:

1. Increasing JVM heap size
2. Analyzing heap dumps
3. Enabling garbage collection logging
4. Identifying memory leaks
5. Optimizing application code

Integration with IBM i

Calling RPG Programs

Applications can call RPG, COBOL, and CL programs using:

- JT400 toolkit (ProgramCall API)
- Stored procedures
- Data queues
- User-defined functions

Accessing IBM i Files

Access Methods:

- Direct IFS file access using Java File APIs
- Database access through JDBC
- JPA for object-relational mapping

- Native file system integration

Using IBM i Security

Integration Points:

- Applications run under IBM i user profiles
- File and database access uses profile authorities
- Seamless integration with IBM i security model
- No separate security infrastructure needed

Key Takeaways

- IBM i's Integrated Application Server runs on WebSphere Liberty, providing modern Jakarta EE runtime
- Deploy applications using Navigator, command line, dropins directory, or server configuration
- Support for WAR, EAR, and loose application deployments
- Configure features, data sources, and security through server.xml
- Tight integration with IBM i security, Db2 for i, and ILE programs
- Multiple deployment methods support different workflows
- Use environment-specific configuration for portability
- Comprehensive monitoring and logging capabilities
- Hot deployment for development; controlled updates for production
- Direct access to IBM i resources from Java applications

How do I use Integrated Web Services for ILE?

What This Covers

This procedure covers IBM i's Integrated Web Services (IWS) — the **server** side (exposing ILE programs as REST or SOAP web services) and the **client** side (consuming external SOAP services from ILE using generated stubs).

What It Is

IWS is two things. Most people know one side and miss the other.

IWS Server takes an existing `*PGM` or `*SRVPGM` and exposes it as a REST or SOAP web service — no RPG code changes. Deploy through the IBM Navigator for i interface (http://your-ibmi:2002/Navigator) or the Web Admin GUI interface (`http://your-ibmi:2001/HTTPAdmin`- note will be depricated in the future) or using scriptin oiptions like `installWebService.sh`. The server handles HTTP routing, JSON/XML serialization, and PCML (Program Call Markup Language) to parameter mapping.

IWS Client takes a partner's WSDL (Web Services Description Language) and generates ILE stubs your RPG calls to consume a SOAP service. Run `wsdl2ws.sh` once, get a `*SRVPGM`, call each SOAP operation as a procedure. The stubs own XML envelope construction, HTTP transport, and response parsing.

Component	Direction	Protocols	Key Tool	Use Case
IWS Server	Expose application data outward	REST (JSON/XML) and SOAP 1.1/1.2	/QIBM/ProdData/OS/WebServices/bin/installWebService.sh or admin console	Mobile apps, web frontends, cross-platform partners calling your RPG
IWS Client	Consume external services	SOAP (document/literal only)	/QIBM/ProdData/OS/WebServices/V1/client/bin/wsdl2ws.sh	Your RPG calling a partner's SOAP gateway (payments, EDI, ratings)

Both ship under `/QIBM/ProdData/OS/WebServices/` and are standard on IBM i 7.5 and 7.6.

Which side do you need? Partner says "here's our WSDL" → IWS Client. Partner says "here's our REST API" → QSYS2 HTTP functions, not IWS. You need to expose your application logic as an API → IWS Server (REST or SOAP).

Why It Matters

Exposing. Your `CalcFreight` service program takes a ZIP code, weight, and service level — returns a rate. Today it's called from 5250. Tomorrow the web team needs it for the portal and mobile needs it for the driver app. Without IWS, you'd write a Java wrapper and maintain it forever. With IWS, deploy `CalcFreight` as REST in five minutes. JSON in, JSON out, same ILE code, no Java.

Consuming. The bank published a WSDL — `SubmitPayment`, `GetPaymentStatus`, `VoidTransaction`. Without the IWS Client, you'd build SOAP envelopes by hand (XML escaping, namespace prefixes — breaks when the vendor adds a field) or route through Java. With `wsdl2ws.sh`, run it once, get a `*SRVPGM`, call `SubmitPayment()` like any other procedure.

How It Works

A. IWS Server — Expose an ILE program as a REST web service

IWS Server wraps your compiled `*SRVPGM` or `*PGM` with an HTTP endpoint. Your program must have PCML embedded (compile with `PGMINFO(*PCML *MODULE)`) or a separate PCML file.

Deploy via QShell or SSH:

```
cd /QIBM/ProdData/OS/WebServices/V1/server/bin

./installWebService.sh \
  -server MYAPISVR \
  -programObject /qsys.lib/apilib.lib/freightsvc.srvpgm \
  -service CalcFreight \
  -serviceType *REST \
  -detectFieldLengths \
  -propertiesFile /home/DEVUSER/ws/calcfreight.props
```

The `-serviceType *REST` flag is what makes this a REST service instead of SOAP. Without it, the default is `*SOAP11`.

The properties file controls URI templates, HTTP methods, and parameter injection:

```
### /home/DEVUSER/ws/calcfreight.props

uri.path.template=/freight/{zipCode}

CALCFREIGHT.uri.path.template=
CALCFREIGHT.wrap.input.parameters=false
CALCFREIGHT.wrap.output.parameter=true
CALCFREIGHT.ZIPCODE.pathparam=zipCode
CALCFREIGHT.WEIGHT.queryparam=weight
CALCFREIGHT.SVCLEVEL.queryparam=level
```

With this deployed, external clients call your RPG like any REST API:

```
GET http://your-ibmi:10010/web/services/CalcFreight/85281?weight=12.5&level=GROUND
```

Response (JSON, automatically serialized from your RPG output parameters):

```
{
  "rate": 14.75,
  "carrier": "FEDEX",
  "transitDays": 3,
  "effectiveDate": "2026-03-25"
}
```

Your RPG didn't change. The IWS server handles HTTP routing, JSON serialization, and PCML-to-parameter mapping. You can also deploy through the **admin console wizard** — same result, point-and-click.

Change `-serviceType *SOAP12` to deploy as SOAP instead. The server auto-generates the WSDL. You can deploy the same program as both REST and SOAP with two `installWebService.sh` runs using different service names.

B. IWS Client — Generate stubs from a partner's WSDL

Start in QShell (`STRQSH`) or SSH. The `wsdl2ws.sh` script takes a WSDL and generates everything:

```
cd /QIBM/ProdData/OS/WebServices/V1/client/bin

./wsdl2ws.sh -lrpg \
  -o /home/DEVUSER/wsstubs/payment \
  -s /qsys.lib/pmtlib.lib/pmtws.srvpgm \
  -ms 4096 -ma 100 \
  https://payments.examplebank.com/gateway?wsdl
```

Key flags: `-lrpg` generates RPG stubs, `-o` sets the output directory, `-s` compiles into a `*SRVPGM`, `-ms` / `-ma` control max string and array sizes (bump these if you see truncation or compile failures from the 16 MB DS limit).

Common failures: HTTPS WSDL without the CA cert imported in DCM (import into `*SYSTEM` first). RPC/encoded WSDLs (only document/literal is supported). The generated RPG stubs are built on top of C stubs — you'll see `.c` files in the output directory. Leave them alone.

C. Understand the generated files

The output directory contains RPG source, C source, and include files. File names derive from the WSDL's port type — `PaymentPortType` in this example. The file you care about is `PaymentPortType.rpgleinc` — it defines the data structures, lifecycle procedures, and operation procedures:

```
**FREE

// --- Data structures for each operation ---
DCL-DS SubmitPmt_In QUALIFIED;
    accountNum  LIKE(xsd_string);
    routingNum  LIKE(xsd_string);
    amount      LIKE(xsd_decimal);
    referenceId LIKE(xsd_string);
END-DS;

DCL-DS SubmitPmt_Out QUALIFIED;
    confirmId  LIKE(xsd_string);
    status     LIKE(xsd_string);
    timestamp  LIKE(xsd_string);
END-DS;

// --- Lifecycle: create/destroy the WS handle ---
DCL-PR stub_create_PaymentPortType POINTER
       EXTPROC('stub_create_PaymentPortType');
END-PR;

DCL-PR stub_destroy_PaymentPortType
       EXTPROC('stub_destroy_PaymentPortType');
    handle POINTER VALUE;
END-PR;

// --- Operation procedures — one per WSDL operation ---
DCL-PR stub_op_SubmitPayment
       EXTPROC('stub_op_SubmitPayment');
    handle POINTER VALUE;
    input  LIKEDS(SubmitPmt_In);
    output LIKEDS(SubmitPmt_Out);
END-PR;

DCL-PR stub_op_GetPaymentStatus
       EXTPROC('stub_op_GetPaymentStatus');
    handle POINTER VALUE;
    input  LIKEDS(GetStatus_In);
    output LIKEDS(GetStatus_Out);
END-PR;
```

> **Note:** The actual tool generates fixed-format D-specs (it predates fully free RPG). The above shows the equivalent declarations in fully free format. Your calling program can be `**FREE` regardless — the precompiler handles mixed includes.

`PaymentPortType_xsdtypes.rpgleinc` maps XSD primitives to RPG types. `xsd_string` is a VARCHAR sized by your `-ms` setting. Check it if you're seeing truncation.

D. Call the stubs from your RPG program

Here's a complete RPG program that submits a payment through the generated service program:

```
**FREE
CTL-OPT DFTACTGRP(*NO) ACTGRP(*CALLER)
        BNDDIR('PMTLIB/PMTBND');

// Pull in the generated data structures and prototypes
/INCLUDE /home/DEVUSER/wsstubs/payment/PaymentPortType.rpgleinc
/INCLUDE /home/DEVUSER/wsstubs/payment/PaymentPortType_xsdtypes.rpgleinc

DCL-PI SubmitPmt;
    pAcctNum    CHAR(17)     CONST;
    pRoutNum    CHAR(9)      CONST;
    pAmount     PACKED(11:2) CONST;
    pRefId      CHAR(20)     CONST;
    pConfirmId  CHAR(30);
    pStatus     CHAR(10);
END-PI;

DCL-S wsHandle POINTER;
DCL-DS reqData LIKEDS(SubmitPmt_In) INZ;
DCL-DS rspData LIKEDS(SubmitPmt_Out) INZ;

// --- Get a handle to the web service ---
wsHandle = stub_create_PaymentPortType();

IF wsHandle = *NULL;
    pConfirmId = *BLANKS;
    pStatus = '*WS-FAIL';
    RETURN;
ENDIF;

// --- Populate the request data structure ---
reqData.accountNum = %TRIM(pAcctNum);
reqData.routingNum = %TRIM(pRoutNum);
reqData.amount = pAmount;
reqData.referenceId = %TRIM(pRefId);

// --- Call the SOAP operation ---
MONITOR;
    stub_op_SubmitPayment(wsHandle : reqData : rspData);
ON-ERROR;
    // Network timeout, SOAP fault, SSL failure — all land here
    pConfirmId = *BLANKS;
    pStatus = '*CALL-ERR';
    stub_destroy_PaymentPortType(wsHandle);
    RETURN;
ENDMON;

// --- Map the response back to the caller ---
pConfirmId = rspData.confirmId;
pStatus = rspData.status;

// --- Always clean up the handle ---
stub_destroy_PaymentPortType(wsHandle);

RETURN;
```

The handle is a pointer — `stub_create_*` allocates memory. Always pair it with `stub_destroy_*`. In a batch loop, create once before and destroy once after. MONITOR the stub call — timeouts, SOAP faults, and SSL failures throw RPG exceptions that kill your job without it. You never touch XML; the stubs handle serialization both ways.

E. Build the caller program

Compile your calling program and bind to the generated `*SRVPGM`:

```
CRTBNDDIRE BNDDIR(PMTLIB/PMTBND);

ADDBNDDIRE BNDDIR(PMTLIB/PMTBND)
            OBJ((PMTLIB/PMTWS *SRVPGM));

CRTBNDRPG PGM(PMTLIB/SUBMITPMT)
          SRCSTMF('/home/DEVUSER/src/submitpmt.rpgle')
          DBGVIEW(*SOURCE)
          DFTACTGRP(*NO);
```

The binding directory approach lets every program that calls the payment gateway just specify `BNDDIR('PMTLIB/PMTBND')` in its `CTL-OPT`.

F. The end-to-end flow

```
sequenceDiagram
    participant Dev as Developer (QShell)
    participant WSDL as Partner WSDL
    participant Tool as wsdl2ws.sh
    participant SrvPgm as Generated *SRVPGM
    participant RPG as Your RPG Program
    participant SOAP as Partner SOAP Endpoint

    Dev->>WSDL: Retrieve WSDL (one-time setup)
    Dev->>Tool: Run wsdl2ws.sh -lrpg
    Tool->>Tool: Generate C + RPG stubs
    Tool->>SrvPgm: Compile into *SRVPGM

    RPG->>SrvPgm: stub_create_PaymentPortType()
    SrvPgm-->>RPG: handle (pointer)
    RPG->>SrvPgm: stub_op_SubmitPayment(handle, input, output)
    SrvPgm->>SOAP: SOAP envelope over HTTPS
    SOAP-->>SrvPgm: SOAP response XML
    SrvPgm-->>RPG: output data structure
    RPG->>SrvPgm: stub_destroy_PaymentPortType()
```

The top half happens once per partner WSDL. The bottom half runs every time your RPG calls the service.

G. SSL and WSDL maintenance

Import the partner's CA certificate into DCM (`*SYSTEM` store) before testing — both the Server (incoming HTTPS) and Client (outbound calls) need this. The stubs use `*SYSTEM` by default.

When a partner updates their WSDL, re-run `wsdl2ws.sh` with the same flags. The tool regenerates stubs and recompiles the `*SRVPGM`. If field names changed, fix your RPG references and rebuild.

Key Takeaways

- **IWS is two tools.** Server exposes your application logic as REST or SOAP. Client consumes external SOAP via generated stubs.
- **Server supports REST and SOAP.** `-serviceType *REST` for JSON/REST, `-serviceType *SOAP12` for SOAP. Same program can be both.
- **Client handles SOAP only.** For consuming REST from RPG, use QSYS2 HTTP functions.
- **No RPG changes to expose.** IWS Server reads PCML and maps parameters. Compile with `PGMINFO(*PCML *MODULE)`.

- **DCM first, code second.** Import certificates before testing — both directions need DCM setup.
- **Always MONITOR stub calls and destroy handles.** Especially in batch loops — leaked handles eat memory, unhandled faults kill jobs.

How do I use SQL for consuming Web Services?

What This Covers

This procedure covers how to use SQL for consuming REST APIs or Web Services from an IBM i server.

What It Is

If your **IBM i is at 7.5 and 7.6**, you have access to **HTTP functions in the QSYS2** schema or library for consuming REST APIs and Web Services.

If you're using an older release of IBM i or for some other reason, you can still use the HTTP functions in the SYSTOOLS schema or library for consuming REST APIs and Web Services. However, using the **HTTP functions in SYSTOOLS is not recommended**.

Comparison between QSYS2 and SYSTOOLS HTTP functions:

Feature	QSYS2 (Modern)	SYSTOOLS (Legacy)
Available since	Established in 2022	Established in 2012
Engine	Based upon AXISC API set (no JAVA)	Java-based (spawns JVM jobs)
Support	Many enhancements since inception	No recent or expected enhancements
Performance	Faster performance	Slower performance
Scalability	Many Header controls & options	Few Header controls & options
Security	User must point to a certificate store	Java's certificate store is used
Ease of Use	Straightforward — mirrors standard HTTP semantics	Familiar to long-time users, but JVM overhead and limited options

This document will only focus on the HTTP functions in QSYS2 schema or library as they are the modern and recommended way to consume REST APIs and Web Services from IBM i.

For more information about the HTTP functions in QSYS2 schema or library, see HTTP functions overview.

Why It Matters

Need to check a customer's credit score before approving an order? Or pull shipping rates from a carrier API during order entry? Without SQL HTTP functions, that means writing a service program with ILE C HTTP bindings, or standing up a middleware layer just to make an outbound call.

With QSYS2.HTTP_GET, it's one SQL statement — callable from RPG, from a stored procedure, or straight from ACS Run SQL Scripts. No new components to deploy, no new language to learn, and no JVM spinning up in the background.

How It Works

For illustration purposes, we'll use the following public API URL: `https://restful-api.dev/`

The examples below start with a public API (no auth), then add API key authentication.

Example usage of HTTP_GET

1. Returns a JSON response

```
VALUES
    QSYS2.HTTP_GET('https://api.restful-api.dev/objects', '');
```

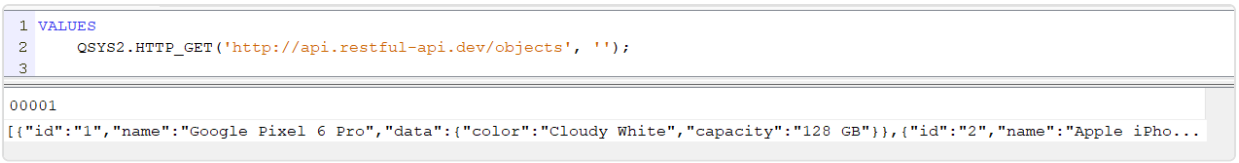


Figure. HTTP_GET with JSON response

2. Parses the JSON response using JSON_TABLE to extract the specific fields from the response

```
SELECT Id,
       COALESCE(Name, '') AS Name,
       COALESCE(Color, '') AS Color,
       COALESCE(Capacity, '') AS Capacity,
       COALESCE(Generation, '') AS Generation,
       COALESCE(Model, '') AS Model,
       COALESCE(Price, 0) AS Price
    FROM JSON_TABLE(
            QSYS2.HTTP_GET('https://api.restful-api.dev/objects', ''),
            '$'
            COLUMNS(
                Id CHAR(20) PATH '$.id',
                Name CHAR(50) PATH '$.name',
                NESTED '$.data' COLUMNS(
                    Color VARCHAR(50) PATH '$.color',
                    Capacity VARCHAR(50) PATH '$."capacity GB"',
                    Price DECIMAL(9, 2) PATH '$.price',
                    Generation VARCHAR(50) PATH '$.generation',
                    Model VARCHAR(50) PATH '$."CPU model"'
                )
            )
        ) AS X;
```

```
SELECT ID,
       COALESCE(NAME,'')        as NAME,
       COALESCE(COLOR,'')       as COLOR,
       COALESCE(CAPACITY,'')    as CAPACITY,
       COALESCE(GENERATION,'')  as GENERATION,
       COALESCE(MODEL,'')       as MODEL,
       COALESCE(PRICE,0)        as PRICE
    FROM JSON_TABLE(
            QSYS2.HTTP_GET('http://api.restful-api.dev/objects', ''),
            '$'
            COLUMNS(
                ID CHAR(20) PATH '$.id', NAME CHAR(50) PATH '$.name',
                Nested '$.data'
                COLUMNS(
                   color        varchar(50)     Path '$.color',
                   capacity     varchar(50)     Path '$."capacity GB"',
                   price        decimal(9,2)    Path '$.price',
                   generation   varchar(50)     Path '$.generation',
                   model        varchar(50)     Path '$."CPU model"'
                )
            )
        ) AS X;
```

ID	NAME	COLOR	CAPACITY	GENERATION	MODEL	PRICE
1	Google Pixel 6 Pro	Cloudy White				0.00
2	Apple iPhone 12 Mini, 256GB, Blue					0.00
3	Apple iPhone 12 Pro Max	Cloudy White	512			0.00
4	Apple iPhone 11, 64GB	Purple				389.99
5	Samsung Galaxy Z Fold2	Brown				689.99
6	Apple AirPods			3rd		120.00
7	Apple MacBook Pro 16				Intel Core i9	1849.99
8	Apple Watch Series 8					0.00
9	Beats Studio3 Wireless					0.00
10	Apple iPad Mini 5th Gen					0.00
11	Apple iPad Mini 5th Gen					0.00
12	Apple iPad Air					0.00
13	Apple iPad Air					0.00

Done: 13 rows retrieved.

Figure. HTTP_GET with JSON response parsed using JSON_TABLE function

3. Uses the VERBOSE version of the function to get the additional information about the HTTP Header response

```
SELECT *
    FROM TABLE (
            QSYS2.HTTP_GET_VERBOSE('https://api.restful-api.dev/objects', '')
        ) AS X;
```

```
SELECT *
    FROM TABLE (
            QSYS2.HTTP_GET_VERBOSE('http://api.restful-api.dev/objects', '')
        ) AS X;
```

RESPONSE_MESSAGE	RESPONSE_HTTP_HEADER
[{"id":"1","name":"Google Pixel 6 Pro","data":{"col...	{"HTTP_STATUS_CODE":200,"Date":"Wed, 04 Mar 2026 06:05:05 GMT","Content-T...

Figure. HTTP_GET_VERBOSE with additional information about the HTTP response

> **Note:** `<your_api_key_here>` is a placeholder — don't hardcode a real key in SQL. It'll surface in the SQL plan cache, job logs, and source history.

Keep keys in a secured credentials table and retrieve them at runtime inside a stored procedure:

```
-- One-time setup
CREATE TABLE CUSTDB.API_CREDENTIALS (
    SERVICE_NAME  VARCHAR(50)  NOT NULL PRIMARY KEY,
    API_KEY       VARCHAR(500) NOT NULL
);
REVOKE ALL ON TABLE CUSTDB.API_CREDENTIALS FROM PUBLIC;
GRANT SELECT ON TABLE CUSTDB.API_CREDENTIALS TO SVCPRF001;  -- your service profile

INSERT INTO CUSTDB.API_CREDENTIALS VALUES ('RESTFUL_API_DEV', 'your_actual_key_here');
```

Then build the options string dynamically inside your calling procedure:

```
DECLARE v_key  VARCHAR(500);
DECLARE v_opts VARCHAR(1000);

SELECT API_KEY INTO v_key
    FROM CUSTDB.API_CREDENTIALS
    WHERE SERVICE_NAME = 'RESTFUL_API_DEV';

SET v_opts = '{"headers":{"Content-Type":"application/json","x-api-key":"' CONCAT v_key CONCAT '"}}';

-- Substitute v_opts wherever the examples below show a hardcoded options JSON
SET v_response = QSYS2.HTTP_GET(url, v_opts);
```

Example usage of HTTP_GET with Auth Key

1. This example pulls a single product by ID. The API requires an x-api-key header — pass it as JSON in the second parameter.:

```
SELECT Id,
       COALESCE(Name, '') AS Name,
       COALESCE(Color, '') AS Color,
       COALESCE(Capacity, '') AS Capacity,
       COALESCE(Generation, '') AS Generation,
       COALESCE(Model, '') AS Model,
       COALESCE(Price, 0) AS Price
    FROM JSON_TABLE(
            QSYS2.HTTP_GET('https://api.restful-api.dev/collections/products/objects/ff8081819c5368bb019ca
                {
                    "headers": {
                    "Accept": "application/json",
                    "x-api-key": "<your_api_key_here>"
                    }
                }
            '),
            '$'
            COLUMNS(
                Id CHAR(20) PATH '$.id', Name CHAR(50) PATH '$.name', NESTED '$.data' COLUMNS(
                    Color VARCHAR(50) PATH '$.color',
                    Capacity VARCHAR(50) PATH '$."capacity GB"',
                    Price DECIMAL(9, 2) PATH '$.price',
                    Generation VARCHAR(50) PATH '$.generation',
                    Model VARCHAR(50) PATH '$."CPU model"'
                )
            )
        ) AS X;
```

```
SELECT ID,
       COALESCE(NAME, '') AS NAME,
       COALESCE(COLOR, '') AS COLOR,
       COALESCE(CAPACITY, '') AS CAPACITY,
       COALESCE(GENERATION, '') AS GENERATION,
       COALESCE(MODEL, '') AS MODEL,
       COALESCE(PRICE, 0) AS PRICE
    FROM JSON_TABLE(
           QSYS2.HTTP_GET('http://api.restful-api.dev/collections/products/objects/ff8081819c5368bb019ca68653db0830', '
               {
                   "headers": {
                   "Accept": "application/json",
                   "x-api-key": "[redacted]"
                   }
               }
           '),
           '$'
           COLUMNS(
               ID CHAR(20) PATH '$.id', NAME CHAR(50) PATH '$.name', NESTED '$.data' COLUMNS(
                   color VARCHAR(50) PATH '$.color',
                   capacity VARCHAR(50) PATH '$."capacity GB"', price DECIMAL(9, 2) PATH '$.price',
                   generation VARCHAR(50) PATH '$.generation', model VARCHAR(50) PATH '$."CPU model"'
               )
           )
       ) AS X;
```

ID	NAME	COLOR	CAPACITY	GENERATION	MODEL	PRICE
ff8081819c5368bb019c	Apple MacBook Pro 16				Intel Core i9	1849.99

Figure. HTTP_GET with Auth Key and JSON response parsed using JSON_TABLE function

Example usage of HTTP_POST with Auth Key

1. Using the HTTP_POST for adding a new Product

```
WITH request (requestBody) AS (
        VALUES
            ('
{
  "name": "Apple MacBook Pro 17",
  "data": {
    "year": 2026,
    "price": 1649.99,
    "CPU model": "Intel Core i109",
    "Hard disk size": "2 TB"
  }
}
')
    )
    SELECT QSYS2.HTTP_POST('https://api.restful-api.dev/collections/products/objects', requestBody, '
{
   "headers": {
   "Content-Type": "application/json",
   "x-api-key": "<your_api_key_here>"
   }
}
') AS responseOutput
        FROM request;
```

```
WITH request (requestBody) AS (
        VALUES
            ('
{
  "name": "Apple MacBook Pro 17",
  "data": {
    "year": 2026,
    "price": 1649.99,
    "CPU model": "Intel Core i109",
    "Hard disk size": "2 TB"
  }
}
')
    )
    SELECT QSYS2.HTTP_POST('http://api.restful-api.dev/collections/products/objects', requestBody, '
{
   "headers": {
   "Content-Type": "application/json",
   "x-api-key": "[redacted]"
   }
}
') AS responseOutput
        FROM request;
```

```
RESPONSEOUTPUT
{"id":"ff8081819c5368bb019cbc5c3c742872","name":"Apple MacBook Pro 17","data":{"year":2026,"price":1649.99,
```

Figure. HTTP_POST with Auth Key and JSON request body for adding a new Product

2. Using the HTTP_POST_VERBOSE for adding a new Product

```
WITH request (requestBody) AS (
        VALUES
            ('
{
  "name": "Apple MacBook Pro 20",
  "data": {
    "year": 2026,
    "price": 1600.45,
    "CPU model": "Intel Core i20",
    "Hard disk size": "5 TB"
  }
}
')
    )
    SELECT responseOutput.response_message,
           responseOutput.response_http_header
        FROM TABLE (
                QSYS2.HTTP_POST_VERBOSE('https://api.restful-api.dev/collections/products/objects',
                    (SELECT requestBody
                            FROM request), '
    {
    "headers": {
    "Content-Type": "application/json",
    "x-api-key": "<your_api_key_here>"
    }
   }')
            ) AS responseOutput
;
```

```
WITH request (requestBody) AS (
        VALUES
            ('
{
  "name": "Apple MacBook Pro 20",
  "data": {
    "year": 2026,
    "price": 1600.45,
    "CPU model": "Intel Core i20",
    "Hard disk size": "5 TB"
  }
}
')
    )
    SELECT responseOutput.response_message,
           responseOutput.response_http_header
        FROM TABLE (
                QSYS2.HTTP_POST_VERBOSE('http://api.restful-api.dev/collections/products/objects',
                    (SELECT requestBody
                            FROM request), '
    {
    "headers": {
    "Content-Type": "application/json",
    "x-api-key": "[redacted]"
    }
   }')
            ) AS responseOutput
;
```

RESPONSE_MESSAGE	RESPONSE_HTTP_HEADER
{"id":"ff8081819c5368bb019cbc6357d0287c","name":"Apple MacBook Pro 20",...	{"HTTP_STATUS_CODE":200,"Date":"Thu, 05 Mar 2026 05:05:53

Figure. HTTP_POST_VERBOSE with Auth Key and JSON request body for adding a new Product

Example usage of HTTP_PUT with Auth Key

1. HTTP_PUT replaces a resource in full — same structure as HTTP_POST, but pointed at a specific resource URL rather than the collection

```
WITH request (requestBody) AS (
        VALUES
            ('
{
  "name": "Apple MacBook Pro 17",
  "data": {
    "year": 2026,
    "price": 1799.99,
    "CPU model": "Intel Core i109",
    "Hard disk size": "2 TB"
  }
}
')
    )
    SELECT QSYS2.HTTP_PUT(
               'https://api.restful-api.dev/collections/products/objects/ff8081819c5368bb019ca68653db0830'
               requestBody,
               '{
                   "headers": {
                       "Content-Type": "application/json",
                       "x-api-key": "<your_api_key_here>"
                   }
               }'
           ) AS responseOutput
        FROM request;
```

```
54 WITH request (requestBody) AS (
55         VALUES
56             ('
57 {
58   "name": "Apple MacBook Pro 17",
59   "data": {
60     "year": 2026,
61     "price": 1799.99,
62     "CPU model": "Intel Core i109",
63     "Hard disk size": "2 TB"
64   }
65 }
66 ')
67     )
68     SELECT QSYS2.HTTP_PUT(
69                 'http://api.restful-api.dev/collections/products/objects/ff8081819c5368bb019ca68653db0830',
70                 requestBody,
71                 '{
72                     "headers": {
73                         "Content-Type": "application/json",
74                         "x-api-key": "[redacted]"
75                     }
76                 }'
77             ) AS responseOutput
78         FROM request;
79
```

```
RESPONSEOUTPUT
{"id":"ff8081819c5368bb019ca68653db0830","name":"Apple MacBook Pro 17","data":{"year":2026,"price":1799...
```

Figure. HTTP_PUT with Auth Key for updating an existing Product

Example usage of HTTP_DELETE with Auth Key

1. HTTP_DELETE takes no request body — just the resource URL and headers

```
VALUES QSYS2.HTTP_DELETE(
           'https://api.restful-api.dev/collections/products/objects/ff8081819c5368bb019ca68653db0830',
           '{
               "headers": {
                   "x-api-key": "<your_api_key_here>"
               }
           }'
       ) ;
```

```
84 VALUES QSYS2.HTTP_DELETE(
85           'http://api.restful-api.dev/collections/products/objects/ff8081819c5368bb019ca68653db0830',
86           '{
87               "headers": {
88                   "Content-Type": "application/json",
89                   "x-api-key": "[redacted]"
90               }
91           }'
92       ) ;
93
94
```

```
00001
{"message":"Object with id = ff8081819c5368bb019ca68653db0830 has been deleted."}
```

Figure. HTTP_DELETE with Auth Key for removing a Product

Handling HTTP errors

The non-VERBOSE functions don't tell you when a call fails — a 401 or 503 comes back as a response body that looks like an error message, not an SQL exception. You'll mistake it for a data problem.

Use the VERBOSE variants to get the HTTP status alongside the response body. A quick test with a bad API key shows what failure actually looks like:

```
SELECT responseOutput.response_message,
       JSON_VALUE(responseOutput.response_http_header, '$.Status') AS http_status
    FROM TABLE (
            QSYS2.HTTP_GET_VERBOSE(
                'https://api.restful-api.dev/objects',
                '{"headers":{"x-api-key":"wrong_key_value"}}'
            )
        ) AS responseOutput;
```

The `response_http_header` column is a JSON document; `JSON_VALUE` pulls the status code out of it. In a production stored procedure, check it and SIGNAL before trying to parse the body:

```
CREATE OR REPLACE PROCEDURE CUSTDB.GET_PRODUCTS (
    OUT p_response VARCHAR(16000000)
)
LANGUAGE SQL
BEGIN
    DECLARE v_status  VARCHAR(10);
    DECLARE v_body    VARCHAR(16000000);
    DECLARE v_headers CLOB(16M);

    SELECT response_message, response_http_header
        INTO v_body, v_headers
        FROM TABLE (
                QSYS2.HTTP_GET_VERBOSE(
                    'https://api.restful-api.dev/objects',
                    '{"headers":{"Accept":"application/json"}}'
                )
            ) AS x;

    SET v_status = JSON_VALUE(v_headers, '$.Status');

    IF v_status <> '200' THEN
        SIGNAL SQLSTATE '38501'
            SET MESSAGE_TEXT = 'HTTP call failed. Status: ' CONCAT v_status CONCAT
                               '. Body: ' CONCAT LEFT(v_body, 200);
    END IF;

    SET p_response = v_body;
END;
```

The SIGNAL raises a catchable SQL exception — the status code and the first 200 characters of the error body are embedded in the message text, which is usually enough to diagnose without filling your logs.

Before any of these examples work against an HTTPS endpoint in production, the target server's CA certificate must be trusted in DCM — add it via Digital Certificate Manager in IBM Navigator for i.

Key Takeaways

- **QSYS2 HTTP functions** are the recommended path — native ILE, no JVM
- Start with the **VERBOSE** variants during development — they return HTTP status codes and headers, which you'll need to diagnose auth failures and timeouts
- Pass **Auth headers** as JSON in the second parameter — works for API keys, Bearer tokens, and basic auth
- **Never hardcode API keys in SQL** — store them in a secured credentials table, grant SELECT only to your service profile, and retrieve at runtime inside a stored procedure

- **Check HTTP status before parsing** — use the VERBOSE variants, extract the status with `JSON_VALUE(response_http_header, '$.Status')`, and SIGNAL on non-2xx; a 401 or 503 returns a response body too, it won't automatically throw an SQL exception
- **HTTPS** requires DCM setup — configure Digital Certificate Manager to trust the remote server's CA certificate before calling HTTPS endpoints

What data interchange formats should I use?

What This Covers

This article explains the data interchange formats available on IBM i—JSON, XML, and fixed format—with focus on using the DATA-GEN operation code for producing output and embedded SQL for parsing using Db2 for i built-in functionality.

Data Interchange Formats on IBM i

IBM i provides native support for three primary data interchange formats, each suited to different use cases.

JSON (JavaScript Object Notation)

What it is: Lightweight, human-readable format that has become the standard for modern web APIs.

Example:

```
{
  "customerId": "12345",
  "name": "John Doe",
  "orders": [
    {"orderId": "ORD-001", "amount": 150.00}
  ]
}
```

Producing JSON from RPG:

Use the DATA-GEN operation code to generate JSON from RPG data structures:

- Define data structure (can be externally described)
- Populate the data structure
- Use DATA-GEN to generate JSON
- Output to variable or stream file

Benefits: Automatic field name mapping, support for nested structures and arrays, no manual JSON construction.

Parsing JSON in RPG:

Use embedded SQL with Db2 for i JSON functions:

- **JSON_TABLE**: Converts JSON to relational table format
- **JSON_VALUE**: Extracts scalar value from JSON
- **JSON_QUERY**: Extracts JSON object or array
- **JSON_OBJECT**: Creates JSON object from SQL data
- **JSON_ARRAY**: Creates JSON array from SQL data

When to Use JSON:

- Modern web APIs and microservices
- Mobile application backends
- Cloud service integration

- RESTful web services
- JavaScript-based applications

XML (eXtensible Markup Language)

What it is: Structured, hierarchical format widely used in enterprise integration.

Example:

```
<customer>
  <customerId>12345</customerId>
  <name>John Doe</name>
  <orders>
    <order>
      <orderId>ORD-001</orderId>
      <amount>150.00</amount>
    </order>
  </orders>
</customer>
```

Producing XML from RPG:

Use DATA-GEN operation code with same approach as JSON:

- Automatic XML element generation from data structures
- Support for attributes and nested elements
- Schema validation options
- Namespace support

Parsing XML in RPG:

Use embedded SQL with Db2 for i XML functions:

- **XMLTABLE**: Converts XML to relational table format
- **XMLQUERY**: Extracts XML fragments using XQuery
- **XMLEXISTS**: Tests for XML content existence
- **XMLCAST**: Converts XML values to SQL types
- **XMLELEMENT**: Creates XML elements from SQL data

When to Use XML:

- Enterprise B2B integration
- SOAP web services
- Schema validation requirements
- Complex hierarchical data
- Legacy system integration
- EDI transformations

Fixed Format Data

What it is: Data using predetermined field positions and lengths, common in traditional IBM i applications.

Example:

```
12345John Doe         ORD-001  00015000
```

Using Externally Described Files:

Externally described files (physical files, logical files, or DDL tables) serve as metadata:

- Single source of truth for data structure
- Automatic field definitions
- Consistent across applications
- Type safety
- Both producer and consumer observe common external definition

When to Use Fixed Format:

- Legacy system integration
- High-volume batch processing
- Mainframe integration
- Performance-critical applications
- Internal IBM i communication

Format Comparison

Aspect	JSON	XML	Fixed Format
Readability	High	Medium	Low
Size	Compact	Verbose	Most Compact
Parsing Speed	Fast	Moderate	Fastest
Schema Validation	JSON Schema	XSD	External Description
Web Support	Excellent	Good	None
Self-Describing	Yes	Yes	No

DATA-GEN Operation Code

The DATA-GEN operation code generates JSON or XML from RPG data structures.

Key Capabilities:

- Generate from data structures (can be externally described)
- Support for nested structures and arrays
- Naming convention options
- Formatting control

Best Practices:

- Use externally described files for consistency
- Maintain single source of truth
- Handle special characters appropriately
- Test with edge cases

Embedded SQL for Parsing

JSON Functions

JSON_TABLE converts JSON to relational format for SQL queries:

- Supports nested structures
- Returns result set
- Enables SQL operations on JSON data

JSON_VALUE extracts single scalar values:

- Supports path expressions
- Type conversion
- Null handling

XML Functions

XMLTABLE converts XML to relational format:

- XPath/XQuery support
- Namespace handling
- Returns result set

XMLQUERY extracts XML fragments:

- XQuery expressions
- Returns XML

Benefits of SQL Approach

- Native database integration
- No external libraries needed
- Efficient processing
- Standard SQL syntax
- Join with database tables

Choosing the Right Format

Use JSON When:

✓ Building modern web APIs or microservices ✓ Integrating with JavaScript applications ✓ Need lightweight, human-readable format ✓ Working with mobile applications ✓ Implementing RESTful services

Use XML When:

✓ Enterprise B2B integration required ✓ SOAP web services needed ✓ Schema validation is critical ✓ Complex hierarchical data structures ✓ Industry standards mandate XML

Use Fixed Format When:

✓ Integrating with legacy systems ✓ High-volume batch processing ✓ Performance is critical ✓ Simple, predictable data structures ✓ Internal IBM i communication

Implementation Approach

Step 1: Define Data Structure

- Create external file description or data structure
- Define fields with appropriate types

- Consider future extensibility

Step 2: Choose Format

- Consider target system requirements
- Evaluate performance needs
- Assess data complexity

Step 3: Implement Generation

- Use DATA-GEN for JSON/XML
- Configure formatting options
- Test with sample data

Step 4: Implement Parsing

- Use embedded SQL functions
- Define path expressions
- Extract to RPG variables
- Handle errors and nulls

Step 5: Test and Validate

- Test valid and invalid data
- Verify edge cases
- Check performance
- Validate error handling

Best Practices

External File Descriptions:

- Maintain as single source of truth
- Use meaningful field names
- Document field purposes
- Plan for versioning

DATA-GEN Usage:

- Generate once, use multiple times
- Consider caching for repeated use
- Handle null values appropriately
- Monitor resource usage

SQL Parsing:

- Use appropriate functions for task
- Validate input format
- Provide meaningful error messages
- Test performance with realistic volumes

Key Takeaways

- IBM i supports JSON, XML, and fixed format data interchange through native capabilities
- DATA-GEN operation code generates JSON and XML from RPG data structures
- Externally described files provide single source of truth for data structure definitions

- Embedded SQL provides JSON and XML parsing through Db2 for i built-in functions
- JSON is best for modern web APIs, mobile apps, and microservices
- XML excels in enterprise B2B integration and schema validation scenarios
- Fixed format remains valuable for legacy integration and high-performance processing
- SQL JSON_TABLE and XMLTABLE functions enable efficient parsing and querying
- Choice of format depends on target systems, performance needs, and data complexity
- Native IBM i support eliminates need for external libraries or tools

How do I use MCP for AI Integration?

What This Covers

This article explains the Model Context Protocol (MCP) and how to use MCP servers on IBM i to integrate AI capabilities, enabling AI assistants to access IBM i data, execute programs, and interact with business logic.

Understanding MCP (Model Context Protocol)

What is MCP?

Model Context Protocol (MCP) is an open protocol that standardizes how AI applications communicate with data sources and tools. It provides a consistent interface for AI assistants to access external resources.

Key Components:

MCP Server:

- Runs on or connects to IBM i
- Exposes IBM i resources to AI applications
- Handles authentication and authorization
- Translates AI requests to IBM i operations

MCP Client:

- AI application or assistant
- Sends requests to MCP server
- Receives structured responses
- Processes data for AI model

Why MCP Matters for IBM i:

- Standardized AI integration
- No custom API per AI tool
- Secure access to IBM i resources
- Consistent interface across AI platforms
- Reusable integration components

MCP Server Capabilities on IBM i

1. Resource Access

Database Access:

- Query Db2 for i tables and views
- Execute SQL statements
- Retrieve result sets
- Support complex queries

File System Access:

- Read IFS files
- List directories
- Access stream files
- Retrieve file metadata

Program Execution:

- Call RPG, COBOL, CL programs
- Pass parameters
- Retrieve output
- Handle program messages

2. Tool Invocation

Predefined Tools:

- Database query tools
- File access tools
- Program call tools
- System information tools

Custom Tools:

- Business-specific operations
- Complex workflows
- Data transformations
- Integration with external systems

3. Security and Authorization

Authentication:

- IBM i user profile integration
- Token-based authentication
- Session management
- Credential validation

Authorization:

- Role-based access control
- Resource-level permissions
- Audit logging
- Compliance support

MCP Server Architecture

Deployment Options

Option 1: Native IBM i MCP Server

- Runs directly on IBM i
- Java or Node.js based
- Direct access to IBM i resources
- Minimal latency

Option 2: External MCP Server

- Runs on separate system
- Connects via JDBC, ODBC, etc.
- Can aggregate multiple IBM i systems
- Additional deployment flexibility

Communication Flow

```
AI Application → MCP Protocol → MCP Server → IBM i Resources
                                              (Db2, IFS, Programs)
```

Key Capabilities Enabled by MCP

Natural Language Database Queries

AI converts natural language to SQL queries:

- User: "Show me top 10 customers by revenue this year"
- AI generates and executes SQL via MCP
- Results formatted for user

Benefits: No SQL knowledge required, complex queries from simple questions, ad-hoc reporting

Intelligent Data Analysis

AI analyzes IBM i data and provides insights:

- Queries sales data
- Identifies trends and patterns
- Generates recommendations
- Explains findings

Benefits: Automated analysis, pattern recognition, predictive insights

Automated Business Operations

AI executes business processes:

- User: "Create order for customer 12345"
- AI calls RPG order entry program via MCP
- Parameters passed automatically
- Results confirmed

Benefits: Natural language interfaces, automated workflows, improved user experience

Documentation and Knowledge Access

AI accesses and explains IBM i resources:

- Retrieves program source or documentation
- Explains logic and parameters
- Provides usage examples

Benefits: Self-service documentation, knowledge preservation, developer assistance

System Monitoring and Diagnostics

AI monitors system health:

- Queries system status
- Analyzes performance metrics
- Identifies issues
- Recommends actions

Benefits: Proactive monitoring, intelligent diagnostics, reduced downtime

Implementation Steps

Step 1: Choose MCP Server

Options:

- Open-source frameworks
- Custom implementation
- Vendor solutions
- Cloud-based services

Step 2: Configure IBM i Access

Database: Configure JDBC/ODBC, connection pooling, permissions **Programs:** Identify programs to expose, define parameters, configure security **Files:** Define accessible directories, set permissions

Step 3: Define Tools

Tool Definition:

- Specify name and description
- Define input parameters
- Document expected outputs
- Provide usage examples

Step 4: Implement Security

Authentication: IBM i user authentication, token management, session policies **Authorization:** Role-based access, resource permissions, audit logging

Step 5: Test and Deploy

Testing: Test tools individually, verify security, validate performance **Deployment:** Deploy server, configure monitoring, set up logging

Use Cases

Business Intelligence

Enable natural language queries for business users without SQL knowledge.

Application Modernization

Provide modern conversational interfaces to legacy RPG/COBOL applications.

Developer Assistance

Help developers understand IBM i code, generate documentation, suggest improvements.

Automated Operations

Automate routine system administration tasks through natural language commands.

Customer Service

Enable service reps to access information quickly using natural language queries.

Security Considerations

Authentication and Authorization

- Use IBM i user profiles
- Implement least privilege access
- Require strong authentication
- Monitor access patterns

Data Protection

- Encrypt data in transit (TLS/SSL)
- Sanitize sensitive data
- Implement data masking
- Comply with privacy regulations

Audit and Compliance

- Log all MCP server access
- Track tool invocations
- Monitor for anomalies
- Maintain audit trails

Rate Limiting

- Limit requests per user/session
- Implement query complexity limits
- Monitor resource usage
- Detect and block abuse

Performance Optimization

Connection Management: Use connection pooling, maintain persistent connections, optimize queries **Query Optimization:** Limit result sets, use indexes, implement pagination **Resource Caching:** Cache tool definitions, frequently accessed data, implement TTL policies

Best Practices

Start Small

- Begin with read-only operations
- Limit scope initially
- Validate thoroughly
- Expand gradually

Monitor and Validate

- Log all operations
- Validate AI-generated queries

- Review results
- Implement safeguards

User Training

- Educate users on capabilities
- Set appropriate expectations
- Provide usage guidelines
- Gather feedback

Limitations

Technical:

- MCP protocol still evolving
- Limited tooling and frameworks
- Integration complexity
- Performance overhead

Operational:

- Requires AI model access
- Potential for incorrect interpretations
- Need for validation and testing
- Training and adoption challenges

Key Takeaways

- MCP (Model Context Protocol) standardizes how AI applications access IBM i resources
- MCP servers expose IBM i databases, programs, and files to AI assistants
- Enables natural language interfaces to IBM i data and business logic
- Key capabilities include database queries, program execution, file access, and system monitoring
- Can run natively on IBM i or externally with IBM i connections
- Use cases include business intelligence, application modernization, developer assistance, and automated operations
- Security through IBM i user authentication and role-based access control
- Requires careful implementation of authentication, authorization, and audit controls
- Start with read-only operations and expand gradually
- MCP enables AI-powered interfaces without custom API development for each AI tool
- Protocol is evolving with growing ecosystem support

What are the considerations that need to be made to ensure data interfaces are secure?

What This Covers

This article explains the critical security considerations for IBM i data interfaces, focusing on preventing unauthorized access, protecting credentials, encrypting data in transit, and implementing defense-in-depth strategies.

Critical Security Principles

1. Never Store Cleartext Passwords

The Problem: Cleartext passwords in configuration files, source code, or databases can be easily compromised.

Solutions:

Use IBM i Password Encryption:

- Store passwords using IBM i's password encryption
- Use password phrases instead of passwords
- Leverage IBM i validation list entries

Use Token-Based Authentication:

- OAuth 2.0 tokens with expiration
- JWT (JSON Web Tokens)
- API keys with rotation policies
- Short-lived tokens with refresh mechanisms

Use Certificate-Based Authentication:

- Client certificates for mutual TLS
- No passwords transmitted or stored
- Certificate lifecycle management

Never:

- Store passwords in source code
- Include passwords in configuration files
- Log passwords in application logs
- Transmit passwords in URLs
- Store passwords in databases without encryption

2. Encrypt Data in Transit

The Problem: Unencrypted network traffic can be intercepted, exposing sensitive data and credentials.

Solutions:

Use TLS/SSL for All Connections:

- HTTPS for web services and APIs
- FTPS or SFTP for file transfers
- SSL/TLS for database connections
- Encrypted messaging protocols

TLS Configuration:

- Use TLS 1.2 or higher
- Disable older protocols (SSL 2.0, SSL 3.0, TLS 1.0, TLS 1.1)
- Configure strong cipher suites
- Regularly update TLS configurations

Certificate Management:

- Use certificates from trusted CAs
- Monitor certificate expiration
- Automate certificate renewal
- Maintain certificate revocation lists

3. Implement Strong Authentication

Multi-Factor Authentication (MFA):

- Require MFA for external access
- Use time-based one-time passwords (TOTP)
- Support hardware tokens

IBM i User Profile Integration:

- Leverage IBM i's security model
- Implement password policies (length, complexity, expiration)
- Enable account lockout after failed attempts

Strong Password Policies:

- Minimum 8-12 characters
- Require complexity (uppercase, lowercase, numbers, special characters)
- Enforce password expiration
- Prevent password reuse

Service Accounts:

- Use dedicated service accounts for applications
- Limit service account privileges
- Rotate credentials regularly
- Monitor service account usage

4. Apply Principle of Least Privilege

Database Access:

- Grant only necessary table/view access
- Use views to restrict column access
- Implement row-level security
- Separate read and write permissions

Program Access:

- Limit which programs can be called
- Use adopted authority carefully
- Restrict access to sensitive programs

File System Access:

- Restrict IFS directory access
- Use object authorities appropriately
- Monitor file access patterns

API Access:

- Implement role-based access control (RBAC)
- Define granular permissions per endpoint
- Use scopes for OAuth tokens
- Regularly review and audit permissions

5. Secure Network Configuration

Firewall Configuration:

- Close unnecessary ports
- Restrict access by IP address/range
- Implement network segmentation
- Use DMZ for external-facing services

Port Management:

- Change default ports where possible
- Document all open ports
- Regularly audit port usage

IP Whitelisting:

- Restrict access to known IP addresses
- Implement geo-blocking if appropriate
- Use VPN for remote access

Interface-Specific Security

Database Interfaces (ODBC, JDBC)

Connection Security:

- Always use SSL/TLS for database connections
- Never embed credentials in connection strings
- Use connection pooling with secure credential management
- Implement connection timeout policies

SQL Injection Prevention:

- Use parameterized queries exclusively
- Never concatenate user input into SQL
- Validate and sanitize all input
- Implement input length limits

Access Control:

- Use database roles and privileges
- Implement views for data access control
- Enable SQL auditing
- Monitor for suspicious queries

Web Services (REST, SOAP)

Authentication:

- Implement OAuth 2.0 or JWT
- Use API keys with rotation
- Require authentication for all endpoints
- Implement rate limiting

Authorization:

- Validate permissions for each request
- Implement scope-based access control
- Use HTTPS exclusively
- Validate all input data

API Security:

- Implement CORS policies appropriately
- Use API gateways for centralized security
- Remove server identification headers
- Implement security headers (HSTS, CSP, X-Frame-Options)

File Transfer (FTP, SFTP)

Protocol Selection:

- Use SFTP or FTPS exclusively
- Disable plain FTP
- Implement SSH key authentication
- Use strong encryption algorithms

Access Control:

- Restrict directory access
- Implement chroot jails
- Use dedicated file transfer accounts
- Monitor file transfer activity

File Security:

- Scan uploaded files for malware
- Validate file types and sizes
- Encrypt sensitive files at rest

Message Queues (IBM MQ, Kafka)

Authentication:

- Require authentication for all connections
- Use SSL/TLS for message transmission
- Implement client certificates
- Rotate credentials regularly

Authorization:

- Implement queue-level permissions
- Use channel authentication
- Restrict administrative access

Message Security:

- Encrypt sensitive message content
- Implement message signing
- Validate message sources

Defense in Depth Strategy

Layer 1: Network Security

Firewall rules, intrusion detection/prevention, DDoS protection, network segmentation

Layer 2: Authentication

Strong authentication mechanisms, multi-factor authentication, certificate-based authentication

Layer 3: Authorization

Role-based access control, least privilege principle, resource-level permissions

Layer 4: Encryption

TLS/SSL for data in transit, encryption for data at rest, key management

Layer 5: Monitoring and Auditing

Comprehensive logging, real-time monitoring, anomaly detection, SIEM integration

Layer 6: Incident Response

Incident response plan, regular security testing, vulnerability management, patch management

Monitoring and Auditing

What to Monitor

Access Attempts:

- Successful and failed authentication attempts
- Unusual access patterns
- Access from unexpected locations
- Off-hours access

Data Access:

- Sensitive data queries
- Large data exports
- Unusual query patterns
- Privilege escalation attempts

System Changes:

- Configuration changes
- User permission changes
- New user accounts

Logging Best Practices

Comprehensive Logging:

- Log all authentication attempts
- Log all data access
- Log all configuration changes
- Log all errors and exceptions

Log Protection:

- Store logs securely

- Implement log retention policies
- Protect logs from tampering
- Encrypt sensitive log data

Vulnerability Management

Regular Security Assessments

Activities:

- Penetration testing (annually)
- Vulnerability scanning (quarterly)
- Security audits
- Code reviews

Patch Management

Process:

- Monitor for security patches
- Test patches in non-production
- Implement patches promptly
- Document patch status

Priority:

- Critical patches within days
- High-priority patches within weeks
- Regular patches within months

Security Checklist

Authentication and Authorization

- ☐ No cleartext passwords stored anywhere
- ☐ Strong password policies enforced
- ☐ Multi-factor authentication implemented
- ☐ Least privilege principle applied
- ☐ Regular access reviews conducted

Encryption

- ☐ TLS/SSL enabled for all connections
- ☐ TLS 1.2 or higher required
- ☐ Strong cipher suites configured
- ☐ Certificates from trusted CAs
- ☐

Certificate expiration monitoring

Network Security

- ☐ Firewall rules configured
- ☐ Unnecessary ports closed
- ☐ IP whitelisting implemented
- ☐ Network segmentation in place
- ☐ VPN for remote access

Monitoring and Auditing

- ☐ Comprehensive logging enabled
- ☐ Real-time monitoring configured
- ☐ Alerts for suspicious activity
- ☐ Regular log reviews
- ☐ Incident response plan documented

Vulnerability Management

- ☐ Regular security assessments
- ☐ Patch management process
- ☐ Vulnerability scanning
- ☐ Security training for staff

Best Practices Summary

Do:

✓ Use TLS/SSL for all network communication ✓ Implement strong authentication (MFA where possible) ✓ Apply principle of least privilege ✓ Encrypt sensitive data at rest and in transit ✓ Monitor and log all access ✓ Regularly update and patch systems ✓ Use token-based or certificate-based authentication ✓ Implement defense in depth ✓ Use parameterized queries to prevent SQL injection

Don't:

✗ Store passwords in cleartext ✗ Use default credentials ✗ Allow unencrypted connections ✗ Grant excessive permissions ✗ Ignore security logs ✗ Delay security patches ✗ Expose unnecessary services ✗ Trust user input without validation ✗ Neglect security training

Key Takeaways

- Never store passwords in cleartext—use encryption, tokens, or certificates

- Encrypt all data in transit using TLS 1.2 or higher
- Implement strong authentication including multi-factor authentication where possible
- Apply principle of least privilege for all access controls
- Secure network configuration by closing unnecessary ports and implementing firewalls
- Use parameterized queries exclusively to prevent SQL injection
- Implement comprehensive monitoring and logging for all interfaces
- Apply defense in depth with multiple security layers
- Conduct regular security assessments and maintain patch management process
- Implement interface-specific security measures for databases, web services, file transfers, and messaging
- Security is an ongoing process requiring continuous monitoring and improvement

How do I secure APIs with OAuth 2.0 and JWT?

What This Covers

This article explains how to implement enterprise-grade API security for IBM i's Integrated Web Services (IWS) using OAuth 2.0 and JSON Web Tokens (JWT) through the TrustAssociationInterceptor (TAI) framework with custom Java code.

Understanding the Security Challenge

Traditional IBM i security relies on user profiles and passwords, which work well for internal applications but present challenges for modern API consumption:

- **Password exposure**: Sharing credentials with external applications is risky
- **No fine-grained access**: User profiles provide all-or-nothing access
- **Difficult revocation**: Changing passwords affects all applications
- **No expiration**: Credentials don't automatically expire
- **Audit complexity**: Hard to track which application made which API call

OAuth 2.0 and JWT solve these problems by providing token-based authentication and authorization.

OAuth 2.0 Fundamentals

What is OAuth 2.0?

OAuth 2.0 is an industry-standard authorization framework that enables applications to obtain limited access to resources without exposing user credentials.

Key Concepts:

- **Resource Owner**: The user who owns the data (e.g., customer)
- **Client**: The application requesting access (e.g., mobile app)
- **Authorization Server**: Issues access tokens (e.g., Azure AD, Okta, Keycloak)
- **Resource Server**: Your IBM i IWS APIs that validate tokens

OAuth 2.0 Flow (Simplified):

```
1. Client requests authorization from Authorization Server
2. User authenticates and grants permission
3. Authorization Server issues access token
4. Client includes token in API requests to IBM i
5. IBM i validates token and processes request
```

JSON Web Tokens (JWT)

What is JWT?

JWT (JSON Web Token) is a compact, URL-safe token format that contains claims about the user and is digitally signed.

JWT Structure:

```
header.payload.signature
```

Example JWT:

```
eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.
SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

JWT Components:

1. **Header**: Algorithm and token type

```
{
  "alg": "RS256",
  "typ": "JWT"
}
```

2. **Payload**: Claims (user info, permissions, expiration)

```
{
  "sub": "user@example.com",
  "name": "John Doe",
  "roles": ["customer", "premium"],
  "exp": 1735689600,
  "iss": "https://auth.example.com"
}
```

3. **Signature**: Cryptographic signature ensuring integrity

```
RSASHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  privateKey
)
```

JWT Benefits:

- **Self-contained**: All necessary information is in the token
- **Stateless**: No server-side session storage required
- **Tamper-proof**: Signature prevents modification
- **Expirable**: Built-in expiration mechanism
- **Portable**: Works across different systems and languages

IBM i IWS Security Architecture

Traditional IWS Security

By default, IWS uses IBM i user profiles:

```
Client → IWS → Validates IBM i credentials → RPG Program
```

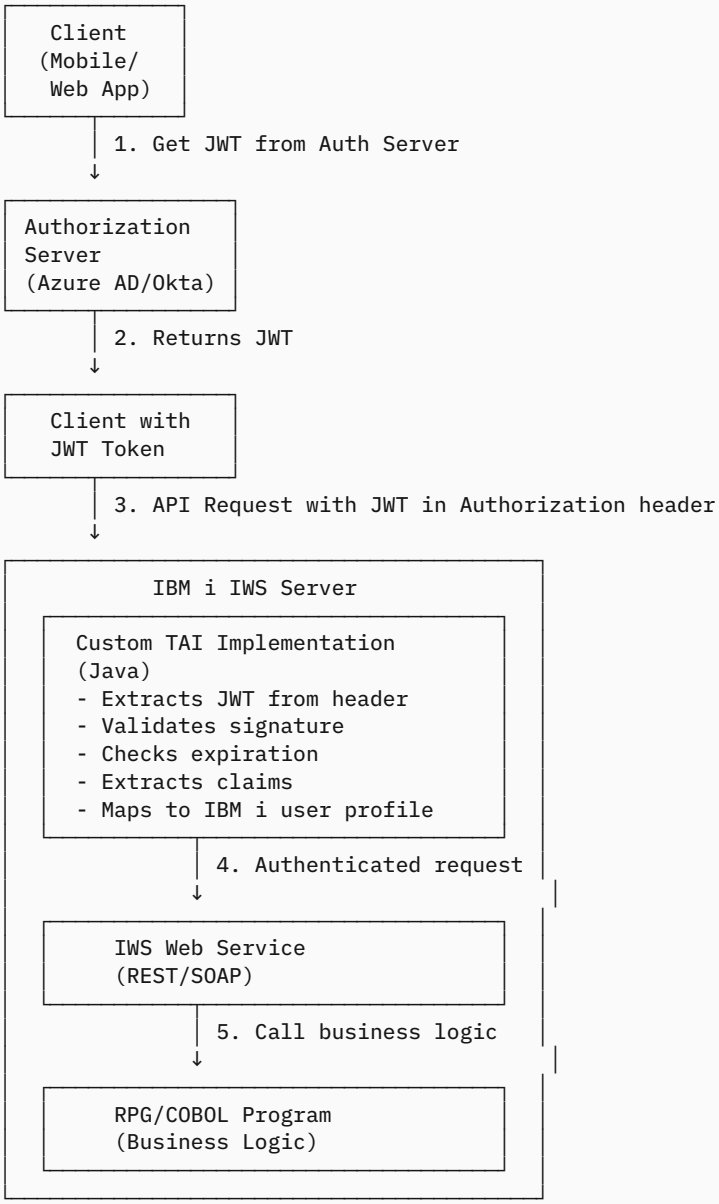
Enhanced Security with TAI

TrustAssociationInterceptor (TAI) is a WebSphere security extension that intercepts requests before they reach your application, allowing custom authentication logic.

```
Client (with JWT) → TAI Interceptor → Validates JWT → Maps to IBM i user → IWS → RPG Program
```

Implementing OAuth 2.0 and JWT with TAI

Architecture Overview



Step 1: Set Up Authorization Server

Choose an OAuth 2.0 provider:

- **Azure Active Directory (Azure AD)**: Microsoft's cloud identity platform
- **Okta**: Enterprise identity management
- **Keycloak**: Open-source identity and access management
- **Auth0**: Developer-friendly authentication platform
- **IBM Security Verify**: IBM's identity solution

Configure your authorization server to:

1. Register your client application
2. Define scopes and permissions
3. Configure JWT signing (RS256 recommended)
4. Set token expiration policies

Step 2: Create Custom TAI Implementation

Create a Java class that implements the TAI interface:

```
package com.ibm.i.security.tai;

import com.ibm.websphere.security.WebTrustAssociationException;
import com.ibm.websphere.security.WebTrustAssociationFailedException;
import com.ibm.wsspi.security.tai.TAIResult;
import com.ibm.wsspi.security.tai.TrustAssociationInterceptor;
import io.jsonwebtoken.*;
import io.jsonwebtoken.security.Keys;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.security.Key;
import java.util.Properties;

public class JWTTrustAssociationInterceptor implements TrustAssociationInterceptor {

    private String issuer;
    private Key publicKey;
    private String userClaimName;

    @Override
    public int initialize(Properties props) throws WebTrustAssociationFailedException {
        // Load configuration from properties
        issuer = props.getProperty("issuer");
        String publicKeyString = props.getProperty("publicKey");
        userClaimName = props.getProperty("userClaimName", "sub");

        // Initialize public key for JWT validation
        publicKey = Keys.hmacShaKeyFor(publicKeyString.getBytes());

        return 0;
    }

    @Override
    public boolean isTargetInterceptor(HttpServletRequest request)
            throws WebTrustAssociationException {
        // Check if request has Authorization header with Bearer token
        String authHeader = request.getHeader("Authorization");
        return authHeader != null && authHeader.startsWith("Bearer ");
    }

    @Override
    public TAIResult negotiateValidateandEstablishTrust(
            HttpServletRequest request,
            HttpServletResponse response)
            throws WebTrustAssociationFailedException {

        try {
            // Extract JWT from Authorization header
            String authHeader = request.getHeader("Authorization");
            String token = authHeader.substring(7); // Remove "Bearer " prefix

            // Parse and validate JWT
            Claims claims = Jwts.parserBuilder()
                .setSigningKey(publicKey)
                .requireIssuer(issuer)
                .build()
                .parseClaimsJws(token)
                .getBody();

            // Extract user identifier from claims
            String username = claims.get(userClaimName, String.class);
```

```
            // Extract roles/groups if present
            @SuppressWarnings("unchecked")
            List<String> roles = claims.get("roles", List.class);

            // Map to IBM i user profile
            String ibmiUser = mapToIBMiUser(username, roles);

            // Create TAI result with authenticated user
            return TAIResult.create(HttpServletResponse.SC_OK, ibmiUser);

        } catch (ExpiredJwtException e) {
            throw new WebTrustAssociationFailedException("Token expired");
        } catch (JwtException e) {
            throw new WebTrustAssociationFailedException("Invalid token: " + e.getMessage());
        } catch (Exception e) {
            throw new WebTrustAssociationFailedException("Authentication failed: " + e.getMessage());
        }
    }

    private String mapToIBMiUser(String username, List<String> roles) {
        // Implement your user mapping logic
        // Options:
        // 1. Direct mapping: username matches IBM i profile
        // 2. Lookup table: Query DB2 table for mapping
        // 3. Default profile: Use a service account with role-based authorization
        // 4. Dynamic profile: Create/use profiles based on roles

        // Example: Simple mapping
        if (username.contains("@")) {
            // Extract username from email
            return username.substring(0, username.indexOf("@")).toUpperCase();
        }
        return username.toUpperCase();
    }

    @Override
    public String getVersion() {
        return "1.0";
    }

    @Override
    public String getType() {
        return "JWT";
    }

    @Override
    public void cleanup() {
        // Cleanup resources if needed
    }
}
```

Step 3: Configure TAI in IWS

1. **Package the TAI implementation**:

```
jar -cvf jwt-tai.jar com/ibm/i/security/tai/*.class
```

2. **Deploy to IBM i**:

```
### Copy JAR to IFS

cp jwt-tai.jar /QIBM/UserData/WebSphere/AppServer/V9/ND/lib/ext/

### Also include JWT library (e.g., jjwt)

cp jjwt-api-0.11.5.jar /QIBM/UserData/WebSphere/AppServer/V9/ND/lib/ext/
cp jjwt-impl-0.11.5.jar /QIBM/UserData/WebSphere/AppServer/V9/ND/lib/ext/
cp jjwt-jackson-0.11.5.jar /QIBM/UserData/WebSphere/AppServer/V9/ND/lib/ext/
```

3. **Configure in WebSphere Admin Console**:

Navigate to: **Security** → **Global security** → **Trust association** → **Interceptors**

Add new interceptor:

- **Class name**: `com.ibm.i.security.tai.JWTTrustAssociationInterceptor`
- **Custom properties**:
  - `issuer`: `https://your-auth-server.com`
  - `publicKey`: Your public key or certificate
  - `userClaimName`: `sub` (or your preferred claim)

4. **Enable TAI**:

- Check "Enable trust association"
- Save and restart the IWS server

Step 4: Configure IWS Web Service

In the IWS deployment descriptor, configure security:

```
<security-constraint>
    <web-resource-collection>
        <web-resource-name>Protected APIs</web-resource-name>
        <url-pattern>/api/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <role-name>authenticated</role-name>
    </auth-constraint>
</security-constraint>

<login-config>
    <auth-method>CLIENT-CERT</auth-method>
</login-config>
```

Step 5: Client Implementation

Clients must obtain a JWT and include it in API requests:

JavaScript Example:

```
// 1. Obtain JWT from authorization server
const response = await fetch('https://auth.example.com/oauth/token', {
    method: 'POST',
    headers: {
        'Content-Type': 'application/x-www-form-urlencoded'
    },
    body: new URLSearchParams({
        grant_type: 'client_credentials',
        client_id: 'your-client-id',
        client_secret: 'your-client-secret',
        scope: 'api.read api.write'
    })
});

const { access_token } = await response.json();

// 2. Call IBM i IWS API with JWT
const apiResponse = await fetch('https://ibmi.example.com/api/customers/12345', {
    method: 'GET',
    headers: {
        'Authorization': `Bearer ${access_token}`,
        'Accept': 'application/json'
    }
});

const customer = await apiResponse.json();
```

Python Example:

```
import requests

### 1. Obtain JWT

auth_response = requests.post(
    'https://auth.example.com/oauth/token',
    data={
        'grant_type': 'client_credentials',
        'client_id': 'your-client-id',
        'client_secret': 'your-client-secret',
        'scope': 'api.read api.write'
    }
)
access_token = auth_response.json()['access_token']

### 2. Call IBM i IWS API

api_response = requests.get(
    'https://ibmi.example.com/api/customers/12345',
    headers={
        'Authorization': f'Bearer {access_token}',
        'Accept': 'application/json'
    }
)
customer = api_response.json()
```

Advanced Features

Token Caching

Implement caching to avoid validating the same token multiple times:

```
private Map<String, Claims> tokenCache = new ConcurrentHashMap<>();

private Claims validateToken(String token) {
    // Check cache first
    if (tokenCache.containsKey(token)) {
        Claims cached = tokenCache.get(token);
        if (cached.getExpiration().after(new Date())) {
            return cached;
        }
        tokenCache.remove(token); // Expired
    }

    // Validate and cache
    Claims claims = Jwts.parserBuilder()
        .setSigningKey(publicKey)
        .build()
        .parseClaimsJws(token)
        .getBody();

    tokenCache.put(token, claims);
    return claims;
}
```

Role-Based Access Control (RBAC)

Extract roles from JWT and enforce authorization:

```
private void checkAuthorization(Claims claims, String requiredRole)
        throws WebTrustAssociationFailedException {
    @SuppressWarnings("unchecked")
    List<String> roles = claims.get("roles", List.class);

    if (roles == null || !roles.contains(requiredRole)) {
        throw new WebTrustAssociationFailedException(
            "Insufficient permissions. Required role: " + requiredRole
        );
    }
}
```

Scope Validation

Validate OAuth scopes for fine-grained access control:

```
private void validateScopes(Claims claims, String... requiredScopes)
        throws WebTrustAssociationFailedException {
    String scopeString = claims.get("scope", String.class);
    Set<String> scopes = new HashSet<>(Arrays.asList(scopeString.split(" ")));

    for (String required : requiredScopes) {
        if (!scopes.contains(required)) {
            throw new WebTrustAssociationFailedException(
                "Missing required scope: " + required
            );
        }
    }
}
```

Audience Validation

Ensure the token is intended for your API:

```
Claims claims = Jwts.parserBuilder()
    .setSigningKey(publicKey)
    .requireIssuer(issuer)
    .requireAudience("https://ibmi.example.com/api") // Add audience check
    .build()
    .parseClaimsJws(token)
    .getBody();
```

Security Best Practices

1. Use Strong Signing Algorithms

- **Recommended**: RS256 (RSA with SHA-256)
- **Avoid**: HS256 for public APIs (symmetric key)
- **Never**: None algorithm

2. Validate All Claims

Always validate:

- **iss** (issuer): Ensure token is from trusted source
- **aud** (audience): Verify token is for your API
- **exp** (expiration): Check token hasn't expired
- **nbf** (not before): Ensure token is valid now
- **iat** (issued at): Verify token age

3. Use HTTPS Only

Always use HTTPS to prevent token interception:

```
https://ibmi.example.com/api/...  ✓
http://ibmi.example.com/api/...   ✗
```

4. Implement Token Expiration

Set reasonable expiration times:

- **Access tokens**: 15-60 minutes
- **Refresh tokens**: Hours to days

- **API keys**: Longer, but rotatable

5. Implement Rate Limiting

Protect against abuse:

```
private void checkRateLimit(String username)
        throws WebTrustAssociationFailedException {
    int requestCount = rateLimiter.getCount(username);
    if (requestCount > MAX_REQUESTS_PER_MINUTE) {
        throw new WebTrustAssociationFailedException("Rate limit exceeded");
    }
    rateLimiter.increment(username);
}
```

6. Log Security Events

Log authentication attempts and failures:

```
logger.info("JWT authentication successful for user: " + username);
logger.warning("JWT authentication failed: " + e.getMessage());
```

7. Rotate Keys Regularly

Implement key rotation strategy:

- Maintain multiple valid public keys
- Rotate signing keys periodically
- Use `kid` (key ID) in JWT header

Troubleshooting

Common Issues:

Token Validation Fails

- Check public key configuration
- Verify issuer matches
- Ensure clock synchronization (NTP)

User Mapping Fails

- Verify IBM i user profile exists
- Check user profile authorities
- Review mapping logic

Performance Issues

- Implement token caching
- Use connection pooling
- Consider token size (keep claims minimal)

Integration Issues

- Verify authorization server configuration
- Check network connectivity
- Review firewall rules

Migration Strategy

Phase 1: Parallel Authentication

- Deploy TAI alongside existing authentication
- Test with subset of users
- Monitor for issues

Phase 2: Gradual Rollout

- Migrate applications incrementally
- Maintain backward compatibility
- Update documentation

Phase 3: Full Migration

- Deprecate old authentication methods
- Remove legacy code
- Enforce OAuth 2.0 only

Key Takeaways

- OAuth 2.0 and JWT provide enterprise-grade security for IBM i IWS APIs without exposing user credentials
- TrustAssociationInterceptor (TAI) enables custom authentication logic in WebSphere, allowing JWT validation before requests reach your RPG programs
- Custom Java TAI implementation validates JWT signatures, checks expiration, extracts claims, and maps to IBM i user profiles
- JWT tokens are self-contained, stateless, and include user identity, roles, and permissions in a tamper-proof format
- Integration with enterprise authorization servers (Azure AD, Okta, Keycloak) provides centralized identity management
- Proper validation of all JWT claims (issuer, audience, expiration) is critical for security
- Token caching and efficient validation improve performance while maintaining security
- This approach enables modern API security patterns while leveraging existing IBM i business logic and security infrastructure

What are IBM i (SQL) Services?

IBM i (SQL) Services, or IBM i Services, or simply SQL Services, are Db2 for i provided interfaces to IBM i operating system detail.

SQL Services are alternatives to well established Control Language (CL) commands and Application Programming Interfaces (APIs). In some cases, SQL Services provide capabilities or access to data that is not found in alternative interfaces.

In all cases, SQL Sevices provide the possibility to gain advanatages:

- **Productivy Boost** - Simply by using SQL, the awesome features found in the SQL language allow the SQL Services consumer to satisfy the business requirement with far less programming effort.
- **Extreme Automation** - With SQL's capability to easily encode business logic, it becomes easy to transform important, but manual, tasks into perfectly automated solutions.
- **Data becomes Information** - Non-SQL interfaces typically return raw data, and the user or application is responsible for transforming it into information. With SQL Services, the data being returned is curated for

consumption and when combined with other SQL services and the SQL language, the goal of reaching information is achieved.

- **Modernization Realized** - When you add up the benefits from the previous bullets, you have achieved modernization, along with its noteworthy benefits and savings.

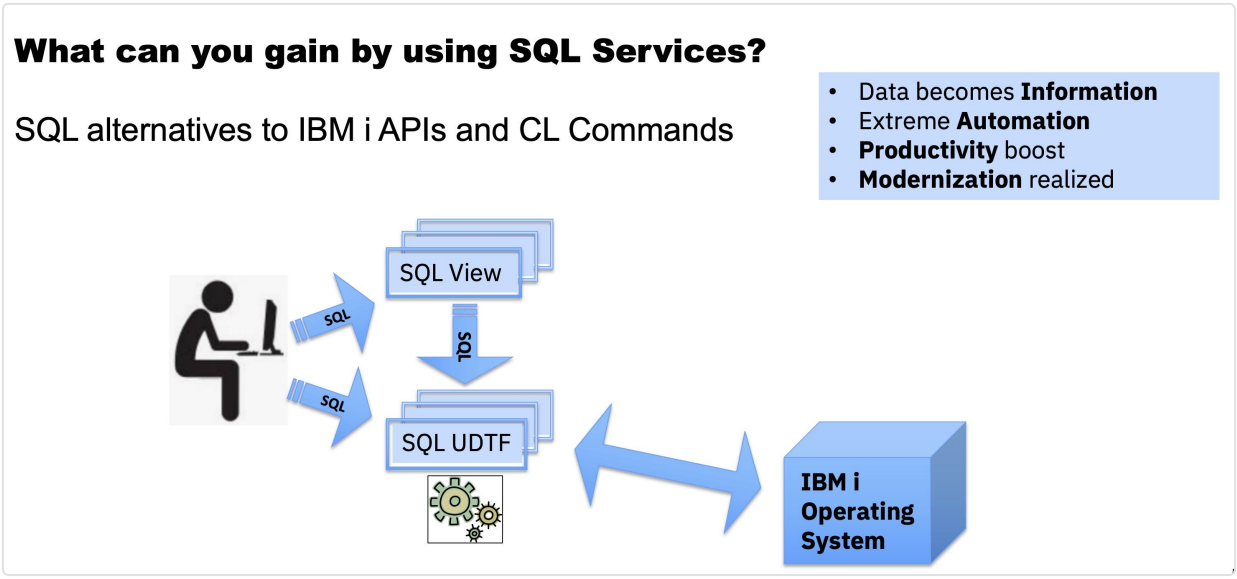


Figure. What can you gain by using SQL Services?

SQL Services – Support Detail

SQL Services are supported interfaces on IBM i, delivered by Db2 for i via the Db2 for IBM i PTF Group.

To understand which SQL Services are supported on a specific IBM i operating system release level, or Db2 for IBM i PTF Group level, see IBM i Services (SQL).

SQL services are located in the following schemas:

- **QSYS2** - These are alternative interfaces to CL Commands or APIs.
- **SYSTOOLS** - These are Db2 for i provided tools. The tools may very well use one or more QSYS2-based SQL Services. The SYSTOOLS services typically allow the user to extract (generate) the SQL source code used to create the tool. In this way, users are able to understand how to tool works, and even have the ability to revise and extend the tool as they see fit, in their own schema.
- **SYSPROC** and **SYSIBMADM** - A couple SQL services are found within the SYSPROC and SYSIBMADM schemas. These schemas are used less frequently, but like QSYS2 and SYSTOOLS, they too are Db2 for i provided schemas.

SQL Services – Form Factor

SQL Services are delivered using the following SQL object types:

- **SQL View** - Easily queried, views typically derive the data the needs to be returned by calling one or more UDTFs.
- **SQL Table Function (UDTF)** - User Defined Table Functions (UDTFs) are the bedrock of SQL Services. UDTFs typically have 1 to many parameters that are used to guide the UDTF processing to find the desired information. Table functions return a single set of rows and columns.
- **SQL Scalar Function (UDF)** - User Defined Functions (UDFs) typically have 1 to many parameters that are used to guide the scalar function processing to find the desired information. Scalar functions return a single value.
- **SQL Procedure** - SQL Services that are packaged within an SQL Procedure typically exist to allow the user to configure or manage a specific topic. As SQL View is used to observe the current details, while the SQL Procedure is used to adjust or change the details.

- **SQL Table** - In some cases, the SQL serivce is comprised of a set of row and column values, located within an SQL table.
- **SQL Global Variable** - Global variables provide the ability to configure a specific, singular, topic.

SQL Services – Landscape Poster

As shown in figure 2, a multi-page poster containing a categorized look at IBM i (SQL) Services and Db2 for i Services can be found at Db2 for i - Technology Updates.

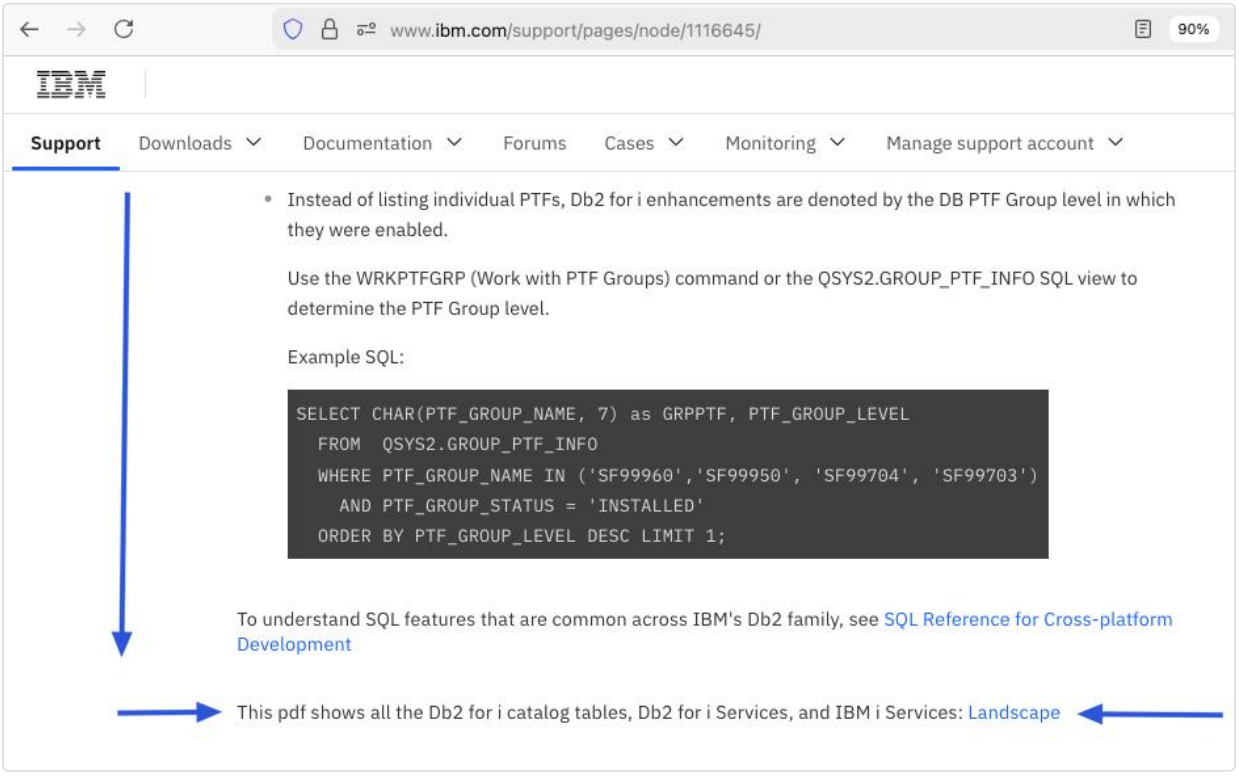


Figure. SQL Services Poster

SQL Services – Documentation

Documentation of IBM i (SQL) Services can be found at IBM i Services.

SQL Services – Resources

There are many places to look for working examples of using SQL services. One of the best practices for success with SQL services is to avoid starting from scratch. Instead, use these and other resources

Here are a few such resources:

- **ACS's Run SQL Scripts - Insert from Examples**
- **Visual Studio Code4i's - Examples**
- **Db2 for i - SQL Tutor** - IBM i Tutorials, Demos, and SQL examples
- **iSee Video Blog** - IBM i Tutorials, Demos, and SQL examples

Documentation of IBM i (SQL) Services can be found at IBM i Services.

What is OpenTelemetry and what can it monitor?

What This Covers

This article explains OpenTelemetry and how to implement it on IBM i using asynchronous telemetry collection with Data Queues, enabling comprehensive monitoring across web applications (including IBM i Integrated Web Services), RPG, COBOL, CL, Java, PHP, Python, and other IBM i applications.

Understanding OpenTelemetry

What is OpenTelemetry?

OpenTelemetry (OTel) is an open-source observability framework that provides a standardized way to collect, process, and export telemetry data (traces, metrics, and logs) from applications and infrastructure.

The Three Pillars of Observability:

1. **Traces**: Track requests as they flow through distributed systems
2. **Metrics**: Measure system performance and resource utilization
3. **Logs**: Capture discrete events and diagnostic information

Why OpenTelemetry Matters for IBM i

IBM i systems often run critical business applications that need comprehensive monitoring:

- **Distributed architectures**: IBM i applications increasingly interact with cloud services, APIs, and microservices
- **Performance optimization**: Identify bottlenecks in RPG programs, database queries, and web services
- **Troubleshooting**: Trace requests across multiple systems and languages
- **Compliance**: Maintain audit trails and performance records
- **Modernization**: Integrate legacy IBM i applications with modern observability platforms

OpenTelemetry on IBM i

Supported Languages and Components

OpenTelemetry can monitor various IBM i components across multiple programming languages:

1. Web Applications

- HTTP request/response times
- Service endpoint performance
- Error rates and status codes
- Request payload sizes
- SOAP and REST service metrics
- Includes IBM i Integrated Web Services (IWS), Apache-hosted applications, and other web frameworks

2. ILE Languages

RPG (RPG IV and Free-Format RPG)

- Program execution times
- Procedure call traces
- Database operation metrics
- Business transaction flows
- Custom business metrics

COBOL

- Program performance monitoring
- Batch job processing metrics
- File I/O operations
- Transaction processing times

CL (Control Language)

- Command execution tracking
- Job submission monitoring
- System command performance
- Workflow orchestration metrics

3. Java Applications

- JVM metrics (heap, threads, garbage collection)
- Application performance monitoring
- Database connection pool metrics
- External API call tracking
- Servlet and JSP performance

4. PHP Applications

- Web request processing
- Database query performance
- Session management metrics
- External service calls
- Custom application metrics

5. Python Applications

- Script execution times
- API endpoint performance
- Data processing metrics
- Integration with IBM i resources
- Machine learning model performance

6. Database Operations (Db2 for i)

- Query execution times
- Row counts and data volumes
- Index usage statistics
- Lock contention monitoring
- Stored procedure performance

7. System Resources

- CPU utilization per job
- Memory usage patterns
- Disk I/O operations
- Network throughput
- Job queue depths
- Subsystem performance

IBM i-Specific Architecture: Asynchronous Telemetry

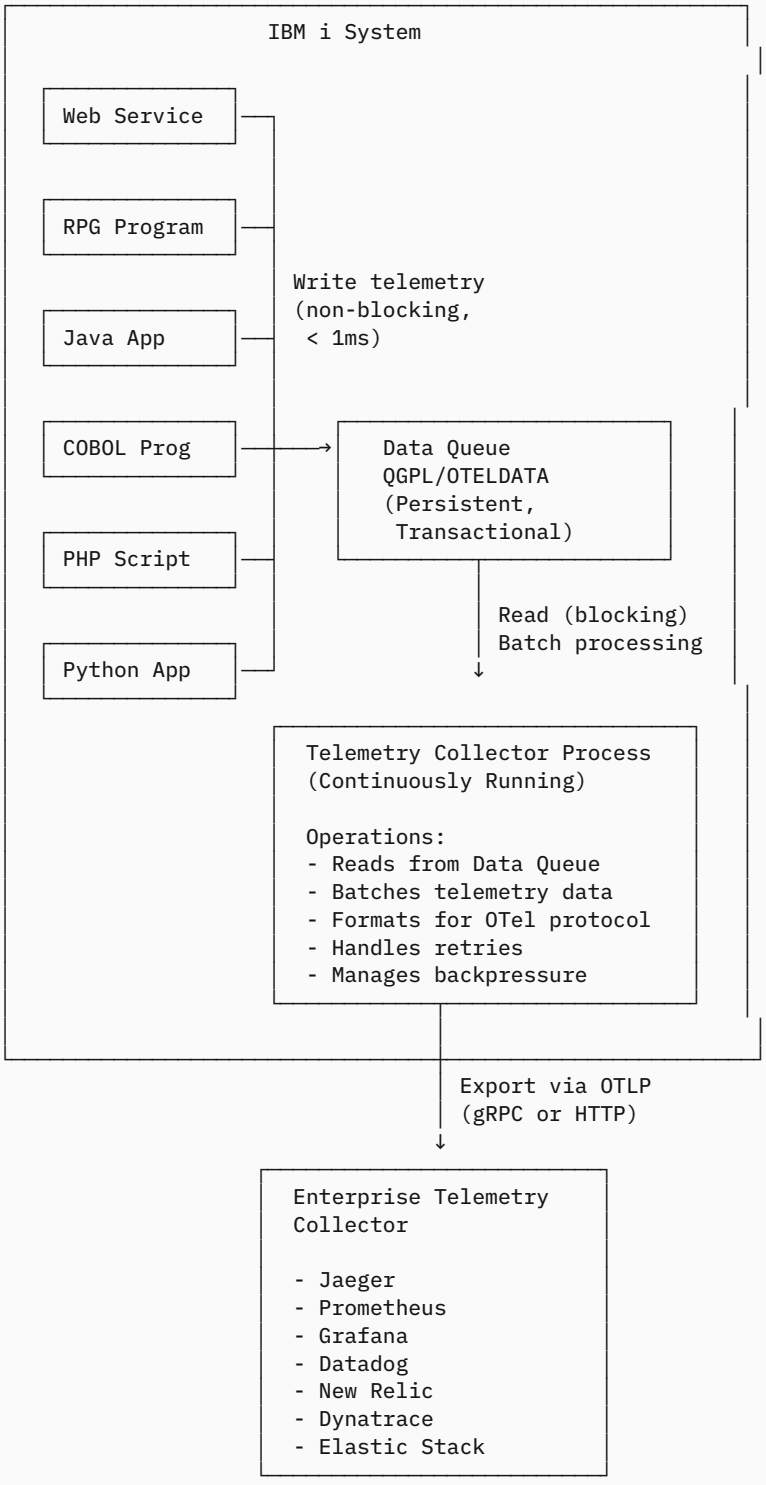
The Challenge

Synchronous telemetry collection can impact application performance. On IBM i, we need a non-blocking approach that doesn't slow down critical business processes. Traditional approaches where applications directly send telemetry to external collectors introduce:

- Network latency in the critical path
- Dependency on external system availability
- Potential for application slowdown
- Complexity in error handling

The Solution: Data Queue-Based Architecture

IBM i Data Queues provide a perfect mechanism for asynchronous telemetry collection. This architecture decouples telemetry generation from telemetry export, ensuring applications remain responsive.



Benefits of This Architecture:

✓ **Non-blocking**: Applications write to Data Queue and continue immediately (typically < 1ms) ✓ **Reliable**: Data Queues are persistent and transactional - no data loss ✓ **Efficient**: Batch processing reduces network overhead and improves throughput ✓ **Scalable**: Multiple collector processes can read from the same queue ✓ **Native**: Leverages IBM i's built-in Data Queue technology ✓ **Resilient**: Telemetry survives collector outages - queued data waits for

processing ✓ **Decoupled**: Applications don't depend on external telemetry infrastructure ✓ **Performance**: Zero impact on application response times

Implementation Approach

Step 1: Create the Telemetry Data Queue

Create a Data Queue to store telemetry data before export:

```
/* Create Data Queue for OpenTelemetry data */
CRTDTAQ DTAQ(QGPL/OTELDATA) +
        MAXLEN(32768) +
        SEQ(*FIFO) +
        SENDERID(*YES) +
        TEXT('OpenTelemetry Data Queue')
```

Key Configuration Decisions:

- **MAXLEN**: Maximum message size (32KB allows for detailed telemetry)
- **SEQ(*FIFO)**: First-in-first-out ensures chronological ordering
- **SENDERID(*YES)**: Captures sender information for debugging

Step 2: Instrument Applications

Each language/platform writes telemetry to the Data Queue using native APIs:

RPG Implementation Approach

Create a service program that provides procedures for:

- Starting and ending trace spans
- Recording metrics
- Logging events
- Writing to the Data Queue asynchronously

The service program uses the `QSNDDTAQ` API to write telemetry data without blocking. Each telemetry record is formatted as JSON for easy parsing by the collector.

COBOL Implementation Approach

Similar to RPG, create copybooks and subprograms that:

- Wrap the `QSNDDTAQ` API call
- Format telemetry data consistently
- Provide simple interfaces for business programs
- Handle errors gracefully without impacting transactions

Java Implementation Approach

For web applications and standalone Java applications:

- Implement servlet filters for automatic HTTP request tracing
- Applies to IWS, Apache-hosted applications, and other Java-based web frameworks
- Use the IBM Toolbox for Java to access Data Queues
- Integrate with OpenTelemetry Java SDK for standard instrumentation
- Write telemetry asynchronously to avoid blocking request threads

PHP Implementation Approach

PHP applications running on IBM i can:

- Use the `ibm_db2` extension for Data Queue access via stored procedures
- Implement middleware for automatic request tracing
- Use OpenTelemetry PHP SDK for standard instrumentation
- Write telemetry through wrapper functions

Python Implementation Approach

Python applications leverage:

- The `pyodbc` or `ibm_db` libraries for Data Queue access
- OpenTelemetry Python SDK for automatic instrumentation
- Decorators for easy function-level tracing
- Asynchronous writes using threading or asyncio

Step 3: Create the Telemetry Collector Process

The collector is a continuously running process that:

Core Responsibilities:

1. **Reads from Data Queue**: Uses blocking reads to efficiently wait for new telemetry
2. **Batches Records**: Accumulates 50-100 records or waits for timeout (e.g., 5 seconds)
3. **Formats Data**: Converts internal format to OpenTelemetry Protocol (OTLP)
4. **Exports Telemetry**: Sends batches to enterprise collectors via gRPC or HTTP
5. **Handles Errors**: Implements retry logic and dead-letter queue for failed exports
6. **Monitors Health**: Tracks queue depth and processing rates

Implementation Considerations:

- Written in Java for OpenTelemetry SDK integration
- Uses IBM Toolbox for Java for Data Queue access
- Implements exponential backoff for retries
- Logs processing statistics for monitoring
- Can run multiple instances for high-volume systems

Step 4: Deploy as a Continuously Running Job

Submit the collector as a batch job that runs continuously:

```
PGM

DCL VAR(&JOBNAME) TYPE(*CHAR) LEN(10) VALUE('OTELCOLL')

/* Submit telemetry collector job */
SBMJOB CMD(JAVA CLASS(com.ibm.i.telemetry.TelemetryCollector) +
           CLASSPATH('/otel/lib/*')) +
       JOB(&JOBNAME) +
       JOBQ(QBATCH) +
       JOBD(QGPL/OTELJOBD) +
       LOG(4 00 *SECLVL)

SNDPGMMSG MSG('OpenTelemetry Collector started') +
          TOPGMQ(*EXT) +
          MSGTYPE(*COMP)

ENDPGM
```

Operational Considerations:

- Configure job description with appropriate resources
- Set up automatic restart on abnormal termination
- Monitor job log for errors
- Implement health check endpoints
- Create alerts for queue depth thresholds

What OpenTelemetry Can Monitor on IBM i

1. Application Performance

Traces:

- End-to-end request flows across web applications, RPG, and database
- Program call hierarchies and execution paths
- Service-to-service communication patterns
- External API calls and integrations
- Cross-language transaction flows (e.g., PHP → RPG → Db2)

Metrics:

- Response times (average, percentiles, max)
- Throughput (requests per second)
- Error rates and failure patterns
- Concurrent users and sessions
- Resource utilization per request

Use Cases:

- Identify slow RPG programs in web service calls
- Track performance degradation over time
- Correlate application performance with system load
- Optimize critical business transactions

2. Database Operations

Traces:

- SQL query execution with full query text
- Stored procedure calls and parameters
- Trigger execution and cascading effects
- Transaction boundaries and commit points

Metrics:

- Query duration (by query pattern)
- Rows processed and data volumes
- Cache hit rates and buffer pool efficiency
- Lock wait times and contention
- Index usage statistics
- Connection pool utilization

Use Cases:

- Identify expensive queries for optimization
- Monitor database connection health
- Track query performance trends
- Detect lock contention issues

3. Business Transactions

Custom Metrics:

- Orders processed per hour
- Invoices generated and amounts
- Inventory movements and adjustments
- Customer interactions and conversions
- Payment processing success rates

Business KPIs:

- Transaction success vs. failure rates
- Processing volumes by type
- Revenue metrics and trends
- SLA compliance percentages
- Customer satisfaction indicators

Use Cases:

- Real-time business dashboards
- Anomaly detection in business metrics
- Capacity planning based on trends
- Performance correlation with business outcomes

4. System Resources

Metrics:

- CPU utilization per job and subsystem
- Memory consumption patterns

- Disk I/O rates and queue depths
- Network throughput and packet loss
- Job queue depths and wait times
- Active job counts by subsystem
- ASP utilization and growth

Use Cases:

- Capacity planning and forecasting
- Resource bottleneck identification
- System health monitoring
- Performance baseline establishment

5. Integration Points

Traces:

- REST API calls to external services
- SOAP web service invocations
- Message queue operations (MQ, Kafka)
- File transfers (FTP, SFTP)
- EDI transaction processing
- Cloud service integrations

Metrics:

- Integration success/failure rates
- Payload sizes and transfer times
- Network latency to external systems
- Retry attempts and backoff patterns
- Circuit breaker state changes

Use Cases:

- Monitor third-party service health
- Track integration reliability
- Identify network issues
- Optimize data transfer patterns

6. Multi-Language Application Flows

Cross-Language Tracing:

- PHP web frontend → RPG business logic → Db2 database
- Python data processing → Java services → COBOL batch
- Web service REST API → RPG procedures → CL commands
- Java application → RPG service program → SQL queries

Benefits:

- Complete visibility across technology stack
- Identify performance bottlenecks regardless of language
- Understand dependencies between components
- Optimize end-to-end transaction flows

Best Practices

1. Asynchronous Collection

Principles:

- Always write to Data Queue without waiting for confirmation
- Never block application threads for telemetry operations
- Handle Data Queue write failures gracefully (log and continue)
- Keep telemetry code separate from business logic

Performance Impact:

- Target < 1ms overhead for telemetry writes
- Monitor application response time impact
- Implement sampling for high-volume operations
- Use conditional compilation for development vs. production

2. Efficient Batching

Batch Configuration:

- Batch size: 50-100 records (balance latency vs. efficiency)
- Timeout: 5-10 seconds (ensure timely export)
- Compression: Enable for network efficiency
- Retry logic: Exponential backoff with maximum attempts

Benefits:

- Reduced network overhead
- Better throughput to telemetry backends
- Lower CPU usage on collector
- Improved reliability with batched retries

3. Data Queue Management

Monitoring:

- Track queue depth continuously
- Alert when depth exceeds thresholds (e.g., 10,000 messages)
- Monitor message age (time in queue)
- Track write and read rates

Maintenance:

```
/* Check Data Queue depth */
DSPQMQRY DTAQ(QGPL/OTELDATA)

/* Clear if necessary (emergency only) */
CLRDTAQ DTAQ(QGPL/OTELDATA)
```

Capacity Planning:

- Size queue based on peak load × retention time
- Consider multiple queues for different priorities
- Implement queue rotation for very high volumes

- Archive old telemetry data periodically

4. Sampling Strategy

For high-volume systems, implement intelligent sampling:

Sampling Approaches:

- **Percentage-based**: Sample 10% of all requests
- **Rate-based**: Sample maximum N requests per second
- **Adaptive**: Increase sampling during errors, decrease during normal operation
- **Priority-based**: Always sample errors and slow requests, sample normal requests less

Implementation:

- Use random number generation for percentage sampling
- Implement time-based sampling for rate limiting
- Always trace requests that exceed thresholds
- Maintain separate sampling rates per service/endpoint

5. Resource Management

Collector Process:

- Limit memory usage with appropriate JVM settings
- Monitor CPU utilization
- Implement graceful shutdown procedures
- Set up automatic restart on failure

Data Queue:

- Implement size limits to prevent runaway growth
- Monitor disk space on ASP
- Set up alerts for abnormal conditions
- Regular health checks

6. Security Considerations

Data Protection:

- Sanitize sensitive data before logging (PII, credentials, tokens)
- Implement field-level redaction for sensitive information
- Use secure connections (TLS) to telemetry backends
- Encrypt telemetry data in transit

Access Control:

- Implement access controls on Data Queues
- Restrict collector process authorities
- Audit telemetry access and exports
- Comply with data retention policies

7. Error Handling

Application-Level:

- Never fail business transactions due to telemetry errors
- Log telemetry failures separately

- Implement circuit breakers for telemetry writes
- Provide fallback mechanisms

Collector-Level:

- Implement dead-letter queue for failed exports
- Retry with exponential backoff
- Alert on persistent failures
- Maintain error metrics

Integration with Enterprise Observability Platforms

The telemetry collector exports data to various enterprise platforms using standard OpenTelemetry Protocol (OTLP):

Jaeger (Distributed Tracing)

- Visualize request flows across services
- Identify performance bottlenecks
- Analyze service dependencies
- Root cause analysis for errors

Prometheus (Metrics)

- Time-series metrics storage
- Powerful query language (PromQL)
- Alerting capabilities
- Long-term metric retention

Grafana (Visualization)

- Custom dashboards for business and technical metrics
- Real-time monitoring
- Alert management and notification
- Multi-source data correlation

Datadog / New Relic / Dynatrace

- Commercial APM solutions
- AI-powered insights and anomaly detection
- Comprehensive monitoring across infrastructure
- Advanced analytics and reporting

Elastic Stack (ELK)

- Log aggregation and analysis
- Full-text search capabilities
- Kibana visualizations
- Machine learning for anomaly detection

Custom Solutions

- Export to internal data warehouses
- Integration with existing monitoring tools
- Custom analytics and reporting
- Compliance and audit systems

Troubleshooting

Data Queue Issues

Queue Full:

```
/* Check Data Queue depth */
DSPQMQRY DTAQ(QGPL/OTELDATA)

/* If queue is backing up, check collector status */
WRKACTJOB JOB(OTELCOLL)

/* Review collector job log */
DSPJOBLOG JOB(OTELCOLL)
```

Symptoms:

- Applications slow down when writing to queue
- Queue depth continuously increasing
- Collector not processing messages

Solutions:

- Verify collector process is running
- Check network connectivity to telemetry backend
- Review collector logs for errors
- Increase collector resources or add instances
- Implement emergency queue clearing if necessary

Collector Not Processing

Diagnostic Steps:

```
/* Check if collector job is running */
WRKACTJOB JOB(OTELCOLL)

/* Review job log for errors */
DSPJOBLOG JOB(OTELCOLL)

/* Check job status */
WRKJOB JOB(OTELCOLL)
```

Common Issues:

- Network connectivity to telemetry backend
- Authentication/authorization failures
- Resource exhaustion (memory, CPU)
- Configuration errors

Performance Impact

Monitoring Guidelines:

- Data Queue writes should be < 1ms
- Collector should process 1000+ records/second
- Application response time impact < 5%
- Queue depth should remain stable under normal load

Performance Tuning:

- Adjust batch sizes for optimal throughput
- Tune JVM settings for collector
- Implement sampling for high-volume operations
- Use multiple collector instances for scale

Data Loss Prevention

Monitoring Strategy:

- Track queue depth trends
- Alert on abnormal growth patterns
- Monitor message age in queue
- Track collector processing rates

Prevention Measures:

- Implement queue size limits
- Set up automatic collector restart
- Use multiple collector instances
- Implement backpressure mechanisms
- Regular health checks and alerts

Migration and Adoption Strategy

Phase 1: Pilot Implementation

- Select one application or service for initial instrumentation
- Implement Data Queue infrastructure
- Deploy collector process
- Validate telemetry data quality
- Measure performance impact

Phase 2: Gradual Rollout

- Instrument additional applications incrementally
- Expand to different languages and platforms
- Refine sampling strategies
- Optimize collector performance
- Build dashboards and alerts

Phase 3: Enterprise Adoption

- Standardize telemetry practices across teams
- Integrate with enterprise observability platforms
- Implement advanced features (custom metrics, business KPIs)
- Establish monitoring and alerting standards
- Train teams on observability best practices

Key Takeaways

- OpenTelemetry provides standardized observability for IBM i applications across traces, metrics, and logs
- IBM i Data Queues enable asynchronous, non-blocking telemetry collection that doesn't impact application performance

- All major languages on IBM i can participate: RPG, COBOL, CL, Java, PHP, Python, and web applications (including IWS)
- A continuously running collector process reads from Data Queues using blocking reads and exports batched telemetry to enterprise platforms
- This architecture is resilient, scalable, and leverages native IBM i capabilities
- OpenTelemetry can monitor application performance, database operations, business transactions, system resources, and integration points across multiple languages
- Proper batching, sampling, and queue management are essential for production deployments
- The approach integrates seamlessly with modern observability platforms like Jaeger, Prometheus, Grafana, and commercial APM solutions
- Asynchronous collection ensures zero impact on critical business processes while providing comprehensive visibility
- Cross-language tracing enables end-to-end visibility across the entire IBM i technology stack

What is Kafka and Why Do I Care?

Apache Kafka is a distributed event streaming platform designed to handle high-throughput, real-time data feeds. Originally developed by LinkedIn and open-sourced in 2011, Kafka has become the de facto standard for building event-driven architectures and real-time data pipelines.

What is Kafka?

At its core, Kafka is a distributed publish-subscribe messaging system that stores streams of records in a fault-tolerant, durable way. Unlike traditional message queues that delete messages after consumption, Kafka retains all published messages for a configurable period, allowing multiple consumers to read the same data independently and at their own pace.

Kafka organizes messages into **topics**—named feeds of records. **Producers** write data to topics, and **consumers** read data from topics. Kafka runs as a cluster of one or more servers called **brokers**, which store and serve the data.

Key Concepts

Topics and Partitions: Topics are divided into partitions, which are ordered, immutable sequences of records.

Producers and Consumers: Producers publish messages to topics without knowing who will consume them.

Durability and Replication: Kafka replicates data across multiple brokers. If a broker fails, another replica takes over, ensuring no data loss.

Ordering Guarantees: Kafka guarantees message order within a partition.

Why IBM i Organizations Should Care

IBM i applications are often the system of record for critical business data—customer orders, inventory levels, financial transactions. As organizations modernize, they need to make this data available to cloud applications, analytics platforms, mobile apps, and microservices **in real time**. Kafka provides the infrastructure to do this reliably and at scale.

Real-Time Integration: Traditional batch ETL processes move data on schedules—hourly, nightly, or weekly. Kafka enables real-time data streaming, allowing downstream systems to react to IBM i events as they happen. When an order is placed or a payment is processed, that event can flow immediately to analytics dashboards, recommendation engines, or customer notification systems.

Decoupling Systems: Kafka acts as a buffer between IBM i and other systems. IBM i applications publish events to Kafka topics without knowing who consumes them.

Event-Driven Architecture: Modern applications are increasingly event-driven. Rather than polling databases for changes, systems react to events as they occur.

Hybrid Cloud and Microservices: As organizations adopt cloud-native architectures and microservices, Kafka provides the messaging infrastructure that connects IBM i to these new environments. IBM i can publish events that trigger cloud functions, or feed machine learning models.

Scalability and Performance: Kafka handles millions of messages per second with low latency. As data volumes grow, Kafka scales horizontally by adding more brokers and partitions.

Common Architecture for IBM i and Kafka

Apache Kafka is a java-based technology that can run on any Cloud, in PASE like every other open source technology on IBM i. The common architecture is a message producer, a message broker and a message consumer. The message producers or consumers can be an IBM i application, the message broker is a Kafka cluster that can run outside or on the IBM i.

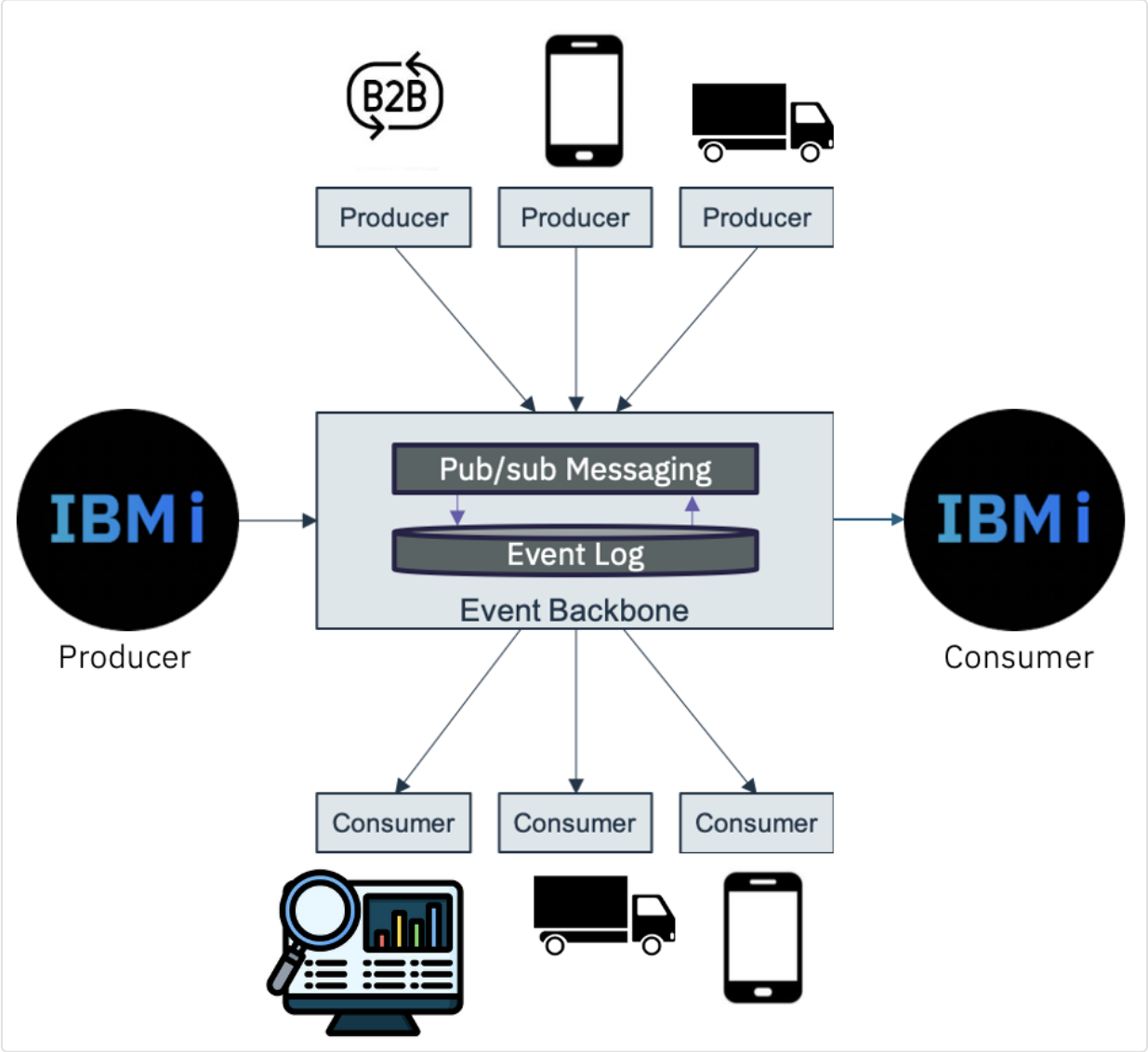


Figure. Kafka architecture diagram

IBM i can interact with Kafka in several ways:

Java-Based Producers and Consumers: Kafka provides Java client libraries that run natively on IBM i. Java applications running in the JVM can produce and consume Kafka messages directly, with optimized performance. On

IBM i, we can cascade such connectors with **Apache Camel** that knows how to fetch data from a data queue, a message queue or a Db2 table, for example.

Open Source Connectors: Kafka Connect is a framework for streaming data between Kafka and other systems. Custom connectors are available to integrate IBM i-specific data sources like Db2 for i. Change Data Capture is a good example of a use case for Kafka Connect.

REST Proxy: For non-Java environments, Kafka REST Proxy provides an HTTP interface to Kafka. IBM i applications written in RPG, or CL can use HTTP APIs to produce and consume messages.

Node.js and Python: Open source runtimes on IBM i (Node.js, Python) have Kafka client libraries. These provide flexible integration options for modern IBM i applications.

Kafka, Apache Camel, MQTT are common open source technologies available on IBM i for a variety of event-driven use cases described in this section.

Please refer to the IBM i Open Source web site to know more about the Kafka installation and usage with/on IBM i.

Common Use Cases for IBM i and Kafka : Event-Driven Architecture

Change Data Capture (CDC): Capture changes from IBM i Db2 tables and stream them to Kafka. Downstream systems consume these changes to maintain synchronized copies, feed analytics platforms, or trigger business processes. CDC tools can monitor journal receivers and publish database changes as Kafka events. An example of CDC tool that works with Kafka is **Debezium** used in conjunction with the native Db2 for i connector.

Order Processing and Fulfillment: When an order is created in an IBM i ERP system, publish an event to Kafka. Warehouse management systems, shipping providers, and customer notification services consume these events to orchestrate fulfillment in real time.

Financial Transactions: Publish payment events, account updates, and transaction records to Kafka. Fraud detection systems, compliance reporting, and real-time dashboards consume these events to monitor financial activity as it happens. Kafka is also used with **predictive analytics and AI (machine learning)**. For example, a bank might use Kafka to stream transaction data to a machine learning model that detects fraudulent activity in real time.

IoT and Sensor Data: Manufacturing environments often use IBM i to manage production data. Sensors and IoT devices can publish data to Kafka, which IBM i applications consume for real-time monitoring, predictive maintenance, and quality control. IoT device and brokers often use the **MQTT protocol**, and IBM i can be used as a MQTT broker, for example, **ActiveMQ on IBM i**. to Kafka.

QSYS2 REST HTTP Functions — Reference Guide

Reference Guide for IBM i Developers

Db2 for i - QSYS2 SQL-based REST Services with call examples

2. QSYS2 REST Function Table

Every QSYS2 REST function follows two naming conventions: the HTTP method (GET, POST, PUT, DELETE, PATCH) and whether it returns the raw response only (scalar) or the response plus the HTTP header (VERBOSE table function).

> **Note:** the `_BLOB` variants for binary payloads was added in IBM i 7.5 TR2 / 7.4 TR8.

Function	Type	Returns
GET		
HTTP_GET	Scalar	CLOB(2G) CCSID 1208 — response body
HTTP_GET_BLOB	Scalar	BLOB(2G) — binary response body
HTTP_GET_VERBOSE	Table	RESPONSE_MESSAGE, RESPONSE_HTTP_HEADER — (CLOB)
HTTP_GET_BLOB_VERBOSE	Table	RESPONSE_MESSAGE (BLOB), RESPONSE_HTTP_HEADER — (CLOB)
POST		
HTTP_POST	Scalar	CLOB(2G) CCSID 1208 — response body
HTTP_POST_BLOB	Scalar	BLOB(2G) — binary response body
HTTP_POST_VERBOSE	Table	RESPONSE_MESSAGE, RESPONSE_HTTP_HEADER — (CLOB)
HTTP_POST_BLOB_VERBOSE	Table	RESPONSE_MESSAGE (BLOB), RESPONSE_HTTP_HEADER — (CLOB)
PUT		
HTTP_PUT	Scalar	CLOB(2G) CCSID 1208 — response body
HTTP_PUT_BLOB	Scalar	BLOB(2G) — binary response body
HTTP_PUT_VERBOSE	Table	RESPONSE_MESSAGE, RESPONSE_HTTP_HEADER — (CLOB)
HTTP_PUT_BLOB_VERBOSE	Table	RESPONSE_MESSAGE (BLOB), RESPONSE_HTTP_HEADER — (CLOB)
DELETE		
HTTP_DELETE	Scalar	CLOB(2G) CCSID 1208 — response body
HTTP_DELETE_BLOB	Scalar	BLOB(2G) CCSID 1208 — binary response body
HTTP_DELETE_VERBOSE	Table	RESPONSE_MESSAGE, RESPONSE_HTTP_HEADER — (CLOB)

Function	Type	Returns
HTTP_DELETE_BLOB_VERBOSE	Table	RESPONSE_MESSAGE (BLOB), RESPONSE_HTTP_HEADER — (CLOB)
PATCH		
HTTP_PATCH	Scalar	CLOB(2G) CCSID 1208 — response body
HTTP_PATCH_BLOB	Scalar	BLOB(2G) — binary response body
HTTP_PATCH_VERBOSE	Table	RESPONSE_MESSAGE, RESPONSE_HTTP_HEADER — (CLOB)
HTTP_PATCH_BLOB_VERBOSE	Table	RESPONSE_MESSAGE (BLOB), RESPONSE_HTTP_HEADER — (CLOB)

Useful Helper functions for URL operations		**---**
Function	Type	Returns
URL_ENCODE	Scalar	VARCHAR — percent-encoded URL string
URL_DECODE	Scalar	VARCHAR — decoded URL string
BASE64_ENCODE	Scalar	VARCHAR — Base64-encoded string
BASE64_DECODE	Scalar	BLOB — decoded binary value

> Scalar functions can be used in `VALUES`, `SELECT`, or `SET` statements. Table functions (VERBOSE) must be with the `SELECT * FROM TABLE(...) X` syntax.

4. HTTP_GET & HTTP_GET_VERBOSE

HTTP GET retrieves a resource from a server. It is the most commonly used HTTP method when consuming REST APIs that return data — for example, fetching a list of records or querying a public data feed.

Syntax

```
-- Scalar (returns CLOB response body only)
VALUES(QSYS2.HTTP_GET( URL, OPTIONS ));

-- Table function (returns response body AND HTTP header)
SELECT * FROM TABLE( QSYS2.HTTP_GET_VERBOSE( URL, OPTIONS ) ) X
```

4.1 Basic GET — Public JSON API

Returns the raw JSON/XML response as a single CLOB value. Usage for quick testing or when you only need the body.

```
-- Fetch a plain-text or JSON response
VALUES QSYS2.HTTP_GET(
    'https://jsonplaceholder.typicode.com/posts',
    '' -- no custom headers
);
```

4.2 GET with JSON_TABLE — Parse Response Inline

Combine `HTTP_GET` with `JSON_TABLE` to shred the JSON response into relational rows inside a single SQL statement. This is the useful way for integrating REST data into IBM i queries for Embedded SQL.

```
-- Fetch a list of posts and project columns directly into a result set
SELECT posts.id,
       posts.title,
       posts.body
FROM JSON_TABLE(
    QSYS2.HTTP_GET('https://jsonplaceholder.typicode.com/posts',
        '{"header":{"Content-Type":"application/json","Accept":"application/json"}}'
    ),
    '$[*]'
    COLUMNS(
        id    INT          PATH '$.id',
        title VARCHAR(128) PATH '$.title',
        body VARCHAR(512)  PATH '$.body'
    )
) AS posts;
```

4.3 GET_VERBOSE — Inspect Status Code & Headers

The VERBOSE variant returns two columns: `RESPONSE_MESSAGE` (body) and `RESPONSE_HTTP_HEADER`. Use this when you need to check the HTTP status code, Content-Type, or other response headers, useful especially during development or error diagnosis.

> **Note:** Sometimes an actual SQL error does get returned but these columns will be NULL , your code will need to handle those cases as well. It can also be surpressed with *signalErrors* option keyword in the SQL call.

```
-- Returns both the body and the full HTTP response header
SELECT RESPONSE_MESSAGE,
       RESPONSE_HTTP_HEADER
FROM TABLE(
    QSYS2.HTTP_GET_VERBOSE(
        'https://jsonplaceholder.typicode.com/postss', -- do notice the extra "s" in order to show the HTT
        '{"header":{"Content-Type":"application/json","Accept":"application/json"}}'
    )
);
```

5. HTTP_POST & HTTP_POST_VERBOSE

HTTP POST submits a body payload to a server endpoint, typically to create a new resource, submit a form, or trigger an action. The request body and content type are both specified in the call.

Syntax

```
-- Scalar
QSYS2.HTTP_POST( URL, REQUEST_BODY, OPTIONS )

-- Table function
SELECT * FROM TABLE( QSYS2.HTTP_POST_VERBOSE( URL, REQUEST_BODY, OPTIONS ) ) X
```

5.1 Basic POST — Create a Resource

```
-- Create a new customer record via REST API
VALUES QSYS2.HTTP_POST(
    'https://api.example.com/customers',
    '{"name":"Acme Corp","email":"info@acme.com","tier":"gold"}',
    '{"header":{"Content-Type":"application/json","Accept":"application/json"}}'
);
```

5.2 POST — Build Request Body from DB2 Table Data

Use `JSON_OBJECT` to construct the POST body dynamically from existing IBM i database columns.

```
-- Build and send a JSON payload from an order record
SELECT QSYS2.HTTP_POST(
    'https://api.example.com/shipments',
    JSON_OBJECT(
        'orderId'   VALUE ORDNUM,
        'customer'  VALUE CUSNAM,
        'shipDate'  VALUE CHAR(SHPDAT)
    ),
    '{"header":{"Content-Type":"application/json","Accept":"application/json"}}'
)
FROM ORDERS
WHERE ORDNUM = 12345;
```

7. HTTP_DELETE & HTTP_DELETE_VERBOSE

HTTP DELETE removes the resource identified by the URL. If the REST Standard is adhered to this would mean that it is also idempotent — a successful first call removes the resource; subsequent calls could return HTTP 404 which would mean the resource was successfully deleted and not found on subsequent calls.

Syntax

```
-- Scalar (no request body required for most DELETE calls)
QSYS2.HTTP_DELETE( URL, OPTIONS )

-- Table function
SELECT * FROM TABLE( QSYS2.HTTP_DELETE_VERBOSE( URL, OPTIONS ) ) X
```

7.1 Basic DELETE

```
-- Delete the order with ID 5001
VALUES QSYS2.HTTP_DELETE(
    'https://api.example.com/orders/5001',
    '{"header":{"Authorization":"Bearer <token>"}}'
);
```

7.2 DELETE_VERBOSE — Confirm Deletion

```
-- Delete and capture the response status
SELECT RESPONSE_MESSAGE,
       RESPONSE_HTTP_HEADER
FROM TABLE(
    QSYS2.HTTP_DELETE_VERBOSE(
        'https://api.example.com/sessions/abc123',
        '{"header":{"Authorization":"Bearer <token>"}}'
    )
);
```

9. BLOB Variants (_BLOB functions)

Added in IBM i 7.5 TR2 / 7.4 TR8, the `_BLOB` variants are functionally identical to their CLOB counterparts but use `BLOB(2G)` as the data type for the request body and response. Use these when interacting with APIs that exchange binary content such as images, PDFs, or compressed files.

Syntax

```
-- Scalar BLOB GET
QSYS2.HTTP_GET_BLOB( URL, OPTIONS ) -- returns BLOB

-- Scalar BLOB POST
SELECT * FROM QSYS2.HTTP_POST_BLOB( URL, REQUEST_BODY_BLOB, OPTIONS )  X   -- returns BLOB

-- (Same pattern for PUT_BLOB, DELETE_BLOB, PATCH_BLOB)

-- VERBOSE BLOB GET (table function)
SELECT * FROM TABLE( QSYS2.HTTP_GET_BLOB_VERBOSE( URL, OPTIONS ) ) X
-- Returns: RESPONSE_MESSAGE BLOB, RESPONSE_HTTP_HEADER CLOB
```

9.1 Download a Binary File (PDF / Image)

```
-- Retrieve a PDF report as a BLOB and write it to the IFS
BEGIN
    DECLARE V_PDF  BLOB(10M);
    SET V_PDF = QSYS2.HTTP_GET_BLOB(
        'https://api.example.com/monthly/PDF/2026-03.pdf',
        '{"header":{"Authorization":"Bearer <token>"}}'
    );
    -- Write the BLOB to an IFS stream file
    CALL QSYS2.IFS_WRITE_BINARY(
        PATH_NAME => '/home/PDF/reports/2026-03.pdf',
        FILE_DATA => V_PDF,
        OVERWRITE => 'REPLACE'
    );
END;
```

9.2 Upload a Binary File

```
-- Read an IFS file and POST it as a BLOB payload
BEGIN
    DECLARE V_FILE BLOB(5M);
    DECLARE V_RESP BLOB(1M);
    -- Read file from IFS into a BLOB variable
    SET V_FILE = QSYS2.IFS_READ_BINARY(
        PATH_NAME => '/home/image/uploads/image.png'
    );
    -- POST the binary data
    SET V_RESP = QSYS2.HTTP_POST_BLOB(
        'https://api.example.com/images',
        V_FILE,
        '{"header":{"Content-Type":"image/png","Authorization":"Bearer <token>"}}'
    );
END;
```

12. Common Errors & Solutions

Error: Timeout — Remote Host Did Not Respond

```
AXISC ERROR: HTTPTransportException: Channel I/O operation timed out.
Failed to open connection to server at host x.x.x.x and port 80.
Error is 3447

AXISC ERROR  : AxisTransportException: Cannot open a channel to the
remote end. Failed to open connection to server at host XXXXX and port 80.
Error is 3401 - Permission denied
```

Possible resolution steps:

- Confirm that the IBM i system has internet/network access.
- Check firewall rules — outbound port 80 (HTTP) or 443 (HTTPS) must be open.
- Verify DNS is configured: `CFGTCP` option 12 (Change TCP/IP Domain).

Error: SSL / Certificate Failure

```
AXISC ERROR: Cannot initialize a channel to the remote end.
Failed to establish SSL connection, gsk_secure_soc_init() failed.

AXISC ERROR : HTTPTransportException: Cannot initialize a channel
to the remote end. Failed to establish SSL connection to server, the operation
gsk_secure_soc_init() failed. GSKit Error is 6000 - Certificate is not signed
by a trusted certificate authority.
```

Possible resolution steps:

- Import the remote server's CA certificate into the DCM `*SYSTEM` store.
- Use DCM **Populate with CAs** to load common public CAs.
- **For testing and troubleshooting SSL errors only**: add `{"sslTolerate":"true"}` to the OPTIONS parameter.

REST Concepts for IBM i Developers

What This Covers

An introduction to REST concepts and how common HTTP methods (GET, POST, PUT, PATCH, DELETE) are used. This part includes some example analogies for an RPGLE developer to could help relate REST operations on familiar patterns like file access and record handling.

What It Is

REST (Representational State Transfer) is a way for systems to communicate over HTTP using standard operations. It treats data as "resources" (like records in a file) that can be created, read, updated, or deleted using specific HTTP methods.

Why It Matters

REST is the foundation of most APIs in the modern web.

Understanding it allows you to:

- Integrate IBM i applications with web and cloud systems
- Replace or extend traditional file-based or program-call interfaces
- Work with external services using a consistent, standardized and predictable method on the modern web
- It is a standard way to talk to other modern systems, so understanding these concepts will help ease confusion normally experienced with structured RPGLE Record access understanding.

Readers Note For an RPGLE developer, REST is essentially a standardized way of performing CRUD operations in APIs normally over the web instead. There could however be more complicated orchistration behind the API , so care and due diligence should be applied to understand the full API detailed documentation on what these operations map to.

How It Works

Best way is to think of a REST API like a remote database file accessed over HTTP. Each URL could represents a "file" or "record set", and HTTP methods act like operations you perform on that file.

GET (Read)

- Retrieves data
- RPGLE analogy: `CHAIN` or `SETLL` + `READE`
- Example: Fetch a customer record

RPGLE Analogy: Like doing a `CHAIN` to a keyed file — you are reading existing data without changing anything.

PUT (Replace)

- Replaces an entire resource
- RPGLE analogy: `UPDATE` with full record
- Example: Overwrite a customer record completely

RPGLE Analogy: Like doing an `UPDATE` where you rewrite the entire record format, even if only one field changed.

DELETE (Remove)

- Deletes a resource
- RPGLE analogy: `DELETE`
- Example: Remove a customer record

Analogy: Exactly like deleting a record from a physical file.

Summary Mapping (RPG vs REST)

REST Method	Purpose	RPG Equivalent
GET	Read data	CHAIN / SETLL+READE
POST	Create new	WRITE
PUT	Replace entire	UPDATE (full record)
PATCH	Partial update	UPDATE (partial)
DELETE	Remove data	DELETE

Key Takeaways

- REST aligns closely with CRUD operations familiar to RPGLE developers.
- Each HTTP method typically has a clear and consistent purpose (Create, Read, Update, Delete).
- GET = Read, POST = Create, PUT/PATCH = Update, DELETE = Remove.
- REST endpoints can be thought of as remote files, and HTTP methods as the operations performed on those files.
- PATCH is more efficient than PUT when only small or specific changes are required.
- PATCH helps preserve data integrity by updating only selected fields rather than overwriting the entire resource.
- REST conventions are widely adopted but not strictly enforced, so real-world APIs may deviate from standard practices.
- Additional HTTP methods and patterns exist, but understanding these core operations provides a strong foundation.

User Interface Modernization

This chapter provides comprehensive guidance on modernizing IBM i user interfaces, from strategic decision-making to practical implementation across multiple frameworks.

Strategic foundations

Why and When Should I Keep the 5250 Interface?

What this covers

In the rush to modernize, there's an unspoken assumption that pervades many IBM i organizations: the 5250 green screen interface is inherently bad and must be replaced at all costs. Vendors reinforce this narrative, positioning their modernization tools as the solution to an obvious problem.

Yet this assumption deserves scrutiny. The 5250 interface has powered business operations reliably for decades, and for certain users and use cases, it remains remarkably efficient. The question isn't whether 5250 is "modern" by contemporary standards, (spoiler alert) it clearly isn't.

The real question is whether replacing it solves a business problem or simply satisfies an aesthetic preference. Understanding when to keep the 5250 interface, when to modernize it, and when to replace it requires moving beyond technology trends to focus on actual user needs, business value, and return on investment.

Why It Matters

Every modernization decision involves trade-offs between cost, risk, and benefit. Modernizing interfaces that do not need modernization wastes resources that could be invested in changes that deliver real business value.

The cost of unnecessary modernization extends beyond the initial project budget; new interfaces need ongoing maintenance, updates for browser compatibility, security patches, and user support. They may introduce performance issues, accessibility problems, or workflow inefficiencies that didn't exist in the original interface. Most importantly, resources spent modernizing interfaces that work well are resources not available for addressing genuine business problems, whether that is integrating with new systems, improving data analytics, or building new capabilities that drive competitive advantage.

Conversely, keeping 5250 interfaces when they genuinely hinder business operations creates its own costs: reduced productivity, difficulty recruiting and retaining staff, lost business opportunities, and competitive disadvantages. The key is making informed decisions based on actual business impact rather than assumptions about what "modern" should look like.

Key Takeaways

1. Expert Users and High-Volume Transaction Processing The 5250 interface excels in scenarios involving expert users performing high-volume, repetitive transactions. Data entry clerks, warehouse operators, customer service representatives, and other power users who work with the same screens hundreds of times daily often achieve remarkable efficiency with 5250 interfaces. These expert users tend to internalize the navigation of menu options and commands, memorize function keys and use macros. In short, they develop a muscle memory that allows them to work at speeds that graphical interfaces struggle to match.

This efficiency isn't nostalgia or resistance to change; it is measurable productivity. A skilled order entry clerk using a well-designed 5250 interface can often process more transactions per hour than the same clerk using a mouse-driven web interface, simply because keyboard navigation is faster than moving a cursor and clicking.

A well designed 5250 interface has minimal visual or network overhead, meaning information can be densely packed into a screen, reducing scrolling and navigation. Its predictable, consistent behavior and support for keyboard buffering eliminate the variability that can slow down repetitive tasks in graphical interfaces.

Before modernizing interfaces used primarily by expert users for high-volume work, measure the actual productivity impact. Pilot the new interface with real users performing real work, and compare transaction times, error rates, and user satisfaction. Check if that modernization reduces rather than improves productivity for these specific use cases. In such situations, keeping the 5250 interface or providing it as an option alongside modern interfaces may be the right business decision. The goal is business value, not universal modernization.

2. Infrequent Use and Low Business Impact Applications Not all applications are created equal. Every organization has applications that are used infrequently. For example, monthly period end processes, annual housekeeping

routines, administrative utilities, or specialized functions accessed by only a handful of users. When these applications work reliably and their 5250 interfaces do not create significant problems, modernizing them often fails basic cost-benefit analysis.

3. Hybrid Approaches and User Choice One of the most overlooked options in modernization discussions is providing both 5250 and modern interfaces for the same application, allowing users to choose based on their needs and preferences. This hybrid approach recognizes that different users have different requirements and that the "best" interface depends on context.

Power users who perform the same tasks repeatedly might prefer the 5250 interface for its speed and efficiency. Occasional users who access the application infrequently might prefer a modern web interface with visual cues and intuitive navigation that does not require memorizing menu options, command and function keys. Mobile users tend to need something other than 5250. Rather than forcing everyone into a single interface paradigm, a hybrid approach lets each user work in the way that makes them most productive.

Modern screen modernization tools and service-oriented architectures make hybrid approaches increasingly practical. You can expose business logic through APIs that serve both traditional 5250 interfaces and modern web or mobile interfaces simultaneously. This allows you to modernize incrementally adding modern interfaces for specific use cases or user groups while maintaining 5250 access for those who prefer it. Over time, as users naturally migrate to modern interfaces or retire, you can phase out 5250 support without forcing disruptive changes.

Hybrid approaches also provide valuable risk mitigation. If the new interface has problems or doesn't meet user needs, the 5250 interface remains available as a fallback. This reduces the risk of modernization projects and provides breathing room to refine new interfaces based on real user feedback.

4. Making the Decision The decision to keep, modernize, or replace 5250 interfaces should be driven by clear business criteria, not technology preferences. Ask specific questions such as: Does the current interface create measurable business problems? Would modernization deliver quantifiable benefits that justify the cost? Are users requesting changes, or is this driven by IT assumptions? Would changing this interface create a lasting drop in productivity? What's the opportunity cost of modernizing this interface versus other potential investments?

For applications where 5250 interfaces work well, serve expert users efficiently, or have low business impact, keeping them (at least in the short term) may be the right choice.

For applications where interfaces genuinely hinder business operations, create recruitment challenges, or prevent important capabilities like mobile access, modernization makes sense.

Mobile design and framework selection

What is Mobile-First Development?

Introduction

Enterprise application design has undergone a significant transformation over the past decade. Traditionally, developers built applications (especially those on platforms such as IBM i) with a desktop-first mindset, assuming users would interact through desktop screens, keyboard-driven inputs, and stable networks. These assumptions shaped not only the user interface but also the overall structure and flow of business applications.

However, the modern enterprise environment tells a different story. Mobile devices have become the primary point of interaction for a large segment of users, including field personnel, warehouse operators, sales teams, and decision-makers. Work is no longer confined to desks; it happens in transit, on-site, and in real time. This shift is reflected in usage patterns, where mobile devices now account for a majority of digital access and user engagement.

This change introduces new challenges, particularly for organizations relying on legacy systems. Applications originally designed for environments such as 5250 green-screen interfaces are often not suitable for mobile usage. Their layouts, navigation models, and interaction patterns do not align with the expectations and constraints of modern mobile devices. Simply exposing these applications through web interfaces is insufficient to deliver an effective user experience.

Mobile-first development addresses this gap by redefining how applications are designed and delivered. It emphasizes building applications with the mobile user in mind from the outset, focusing on simplicity, usability, and performance in constrained environments. This approach ensures that applications are not only functional but also intuitive and efficient across all device types.

In the context of enterprise modernization, adopting a mobile-first strategy enables organizations to align existing systems with current user expectations while preparing for future digital demands. It provides a structured and practical path to delivering responsive, accessible, and user-centric applications without compromising the stability and value of underlying business logic.

Core Principles of Mobile-First Development

Mobile-first development is guided by a set of foundational principles that ensure applications are efficient, user-centric, and adaptable across devices. These principles not only address the constraints of mobile environments but also lead to better overall application design, including for larger screens.

Progressive Enhancement

At the heart of mobile-first development is the concept of **progressive enhancement**. The application is initially built for the most constrained environment "**mobile devices**" and then enhanced for larger screens such as tablets and desktops.

```
Mobile → Tablet → Desktop
```

This approach ensures that **core functionality remains accessible** regardless of device capability, while additional features and visual complexity are layered on as screen size and processing power increase.

Content Prioritization

Mobile devices have limited screen space, which requires a strong focus on what truly matters to the user. Developers and designers must identify and present:

- Essential features
- Primary user actions
- Critical information

Place the most important content upfront, avoiding unnecessary navigation or scrolling. Reveal secondary information progressively using menus, accordions, or step-based workflows.

Implementation strategies include:

- Conducting user research to identify key mobile use cases
- Mapping user journeys with a focus on mobile interactions
- Using analytics to determine frequently accessed features
- Applying progressive disclosure to manage information density

This disciplined approach results in **cleaner, more focused, and more efficient interfaces** across all devices.

Touch-First Interaction Design

Mobile-first apps prioritize touch over mouse and keyboard input.

Key considerations:

- Tap targets should be large enough for accurate interaction (minimum 44×44 pixels)
- Avoid reliance on hover-based interactions, as they are not supported on touch devices
- Use gestures such as swipe and tap where appropriate, ensuring they are intuitive and consistent
- Design layouts with thumb-friendly zones for ease of use
- Provide immediate visual feedback for user actions

- Optimize input fields for mobile keyboards (e.g., numeric for number fields, email keyboard for email fields)

Responsive Layout

Responsive design adapts layouts to different screen sizes and orientations. In a mobile-first approach, the base layout targets small screens, with styles progressively enhanced for larger viewports using CSS media queries with a min-width strategy.

Example:

```
/* Base layout: mobile (single column) */
.content-grid {
  display: flex;
  flex-direction: column;
}

/* Tablet and above: two columns */
@media (min-width: 768px) {
  .content-grid {
    flex-direction: row;
  }
}

/* Desktop and above: three columns */
@media (min-width: 1200px) {
  .content-grid {
    display: grid;
    grid-template-columns: repeat(3, 1fr);
  }
}
```

Performance and Network Awareness

Mobile environments demand careful performance optimization due to unpredictable network conditions and varying device capabilities. Key practices include:

- Minimizing page size by reducing heavy assets (images, scripts)
- Lazy loading non-critical content
- Optimizing API responses to return only necessary data
- Reducing network requests for better data transfer efficiency

In enterprise scenarios (for example, field or warehouse operations), **offline capabilities** may also be required. PWAs and service workers can maintain functionality under limited or intermittent connectivity.

Context Awareness

Mobile devices provide unique capabilities that can enhance application functionality when used effectively. These include:

- GPS for location-based services
- Cameras for scanning and data capture
- Biometric authentication for security
- Sensors for contextual interactions

Designing with context awareness allows applications to deliver **more relevant, efficient, and personalized user experiences**, which are not typically achievable in traditional desktop environments.

Accessibility

Mobile-first design naturally supports accessibility by emphasizing simplicity and clarity. Key considerations:

- Sufficient color contrast
- Clear, descriptive labels
- Logical navigation and focus order
- Large, easy-to-interact UI elements

Following standards like WCAG ensures applications are **inclusive, compliant, and usable by all**, regardless of ability or device.

Why Mobile-First Matters for IBM i Modernization

The shift toward mobile-first development is driven by both evolving user behavior and the practical realities of modern enterprise operations. For organizations relying on IBM i systems, this shift is particularly significant, as it directly impacts how legacy applications are accessed, experienced, and modernized.

The Changing Workforce

Enterprise computing is no longer limited to office environments. Today's workforce extends well beyond the office, relying on mobile devices to perform critical tasks in real time.

Consider common enterprise scenarios:

- **Field technician:** accessing service history and updating work orders on-site
- **Warehouse operator:** scanning inventory and updating stock levels via handheld devices
- **Manager:** approving requests or reviewing reports while traveling

5250 terminals and desktop browsers cannot support these applications practically in these environments because they don't account for mobility, touch interaction, or dynamic work conditions. Mobile-first ensures core business functions are accessible, usable, and efficient on the devices users actually rely on.

Evolving User Expectations

User expectations have been shaped by modern consumer applications that are intuitive, fast, and responsive. These expectations naturally extend to enterprise software.

Applications that require users to zoom, scroll excessively, or interact with small, non-touch-friendly elements create frustration and reduce productivity. Even highly capable backend systems can suffer from low adoption if the user experience is poor.

A mobile-first approach addresses this by delivering:

- Clean and intuitive interfaces
- Optimized navigation for small screens
- Faster and more responsive interactions

This results in **higher user satisfaction, improved adoption rates, reduced training effort, and fewer operational errors**.

Workforce Mobility and Business Agility

Mobile-first development supports a mobile workforce by enabling anytime, anywhere access to systems, improving responsiveness and decision-making.

Key benefits:

- Real-time data access and updates
- Faster execution of business processes
- Improved collaboration across distributed teams

For organizations undergoing digital transformation, especially those modernizing IBM i applications, mobile-first design ensures that systems are aligned with modern operational needs without requiring a complete rewrite of existing logic.

IBM i as a Strong Mobile Back End

IBM i systems are well-suited for mobile-first architectures due to their:

- **Reliable transaction processing** for high volumes of concurrent requests
- **Strong security and data integrity**, critical for public network access
- **Stability**, enabling dependable, long-running services

By exposing IBM i business logic as RESTful APIs (using tools such as Integrated Web Services or API management platforms), organizations can transform existing applications into scalable back-end services.

In this model:

- Mobile or web frontends send lightweight HTTP requests
- IBM i processes the requests using existing business logic
- Responses are returned in formats such as JSON
- Frontend applications render the data based on device context

Efficiency, Lean Design, and Alignment with Digital Transformation Goals

Designing for mobile constraints naturally pushes teams to focus on what truly matters, eliminating feature bloat and reducing unnecessary complexity. This discipline results in cleaner, more modular codebases, faster load times, and better overall performance — benefits that extend across all platforms, not just mobile.

This lean, focused approach also aligns directly with the goals of organizations modernizing IBM i environments. Such initiatives typically aim to deliver modern web and mobile interfaces, improve user experience without rewriting core backend systems, and integrate with cloud and API-driven architectures. Mobile-first development supports all of these objectives by providing a structured, practical path to redesigning interfaces while fully preserving existing backend investments.

Extended Reach and Discoverability

In scenarios where IBM i applications are exposed through web interfaces, mobile-first design also improves visibility and accessibility. Modern search engines prioritize mobile-friendly content, meaning that poor mobile experiences can negatively impact discoverability.

Ensuring a strong mobile experience helps organizations maintain a competitive digital presence while reaching a broader audience.

A Practical Example: Mobile-First Work Order Management

> **Overview:** A real-world walkthrough of modernizing a 5250-based IBM i work order application using a mobile-first approach — without changing any existing business logic.

The Existing Application

- **Type:** 5250-based Work Order Management System built on IBM i
- **Core Functions:**
  - View assigned work orders
  - Record time spent on each task
  - Update work order status
  - Look up parts information

- **Strengths:**
  - Reliable and battle-tested
  - Well-established business logic
- **Pain Points:**
  - Not accessible from mobile devices
  - Technicians must physically walk to a terminal or PC to enter data

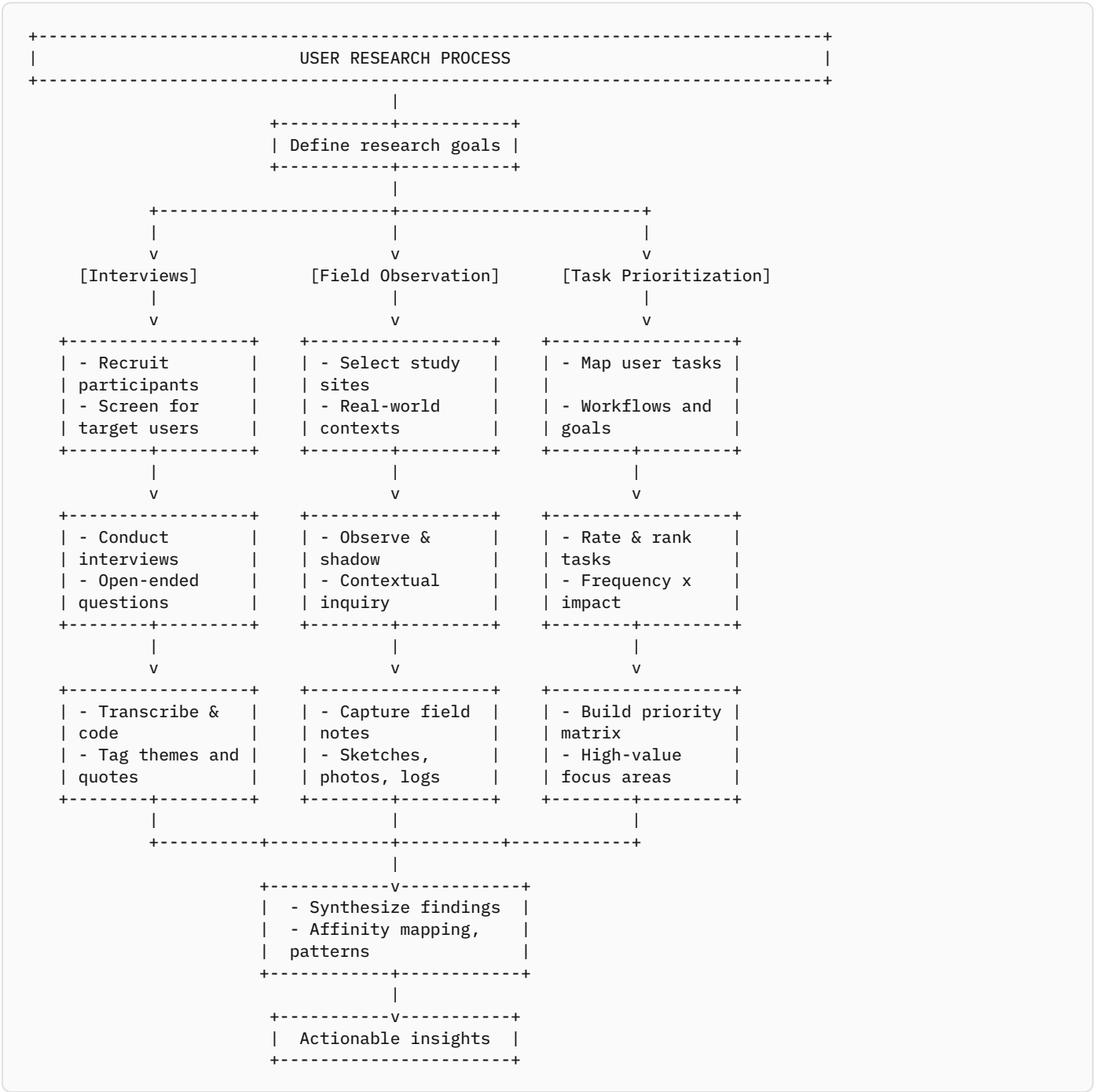
The Mobile-First Redesign

- Conduct interviews with **technicians and managers**
- Observe technicians **in the field** to understand real workflows
- Identify the most common and critical task:

> *"View today's assigned work orders → tap into one → record time → update status to complete"*

- This core task become the **design anchor** — it must work flawlessly on mobile

User Research & Discovery



UI/UX Design Decisions

- Wireframes designed for a **375px-wide mobile screen** (standard iPhone width)
- **Home Screen Layout:**
  - Simple list of assigned work orders
  - Each card shows: Order Number · Equipment ID · Current Status
- **Progressive Disclosure:**
  - Only essential info shown at a glance

- Secondary details (full description, history, parts) hidden until requested

- **Responsive Behavior:**
  - Phone → single-column work order list
  - Tablet/Desktop → list on left, work order detail on right

Back-End Architecture (IBM i — Unchanged)

- **No changes made to existing RPG programs** — all business logic preserved
- Existing programs wrapped as **RESTful Web Services** using the IBM i Integrated Web Services Server
- Exposed API Endpoints:

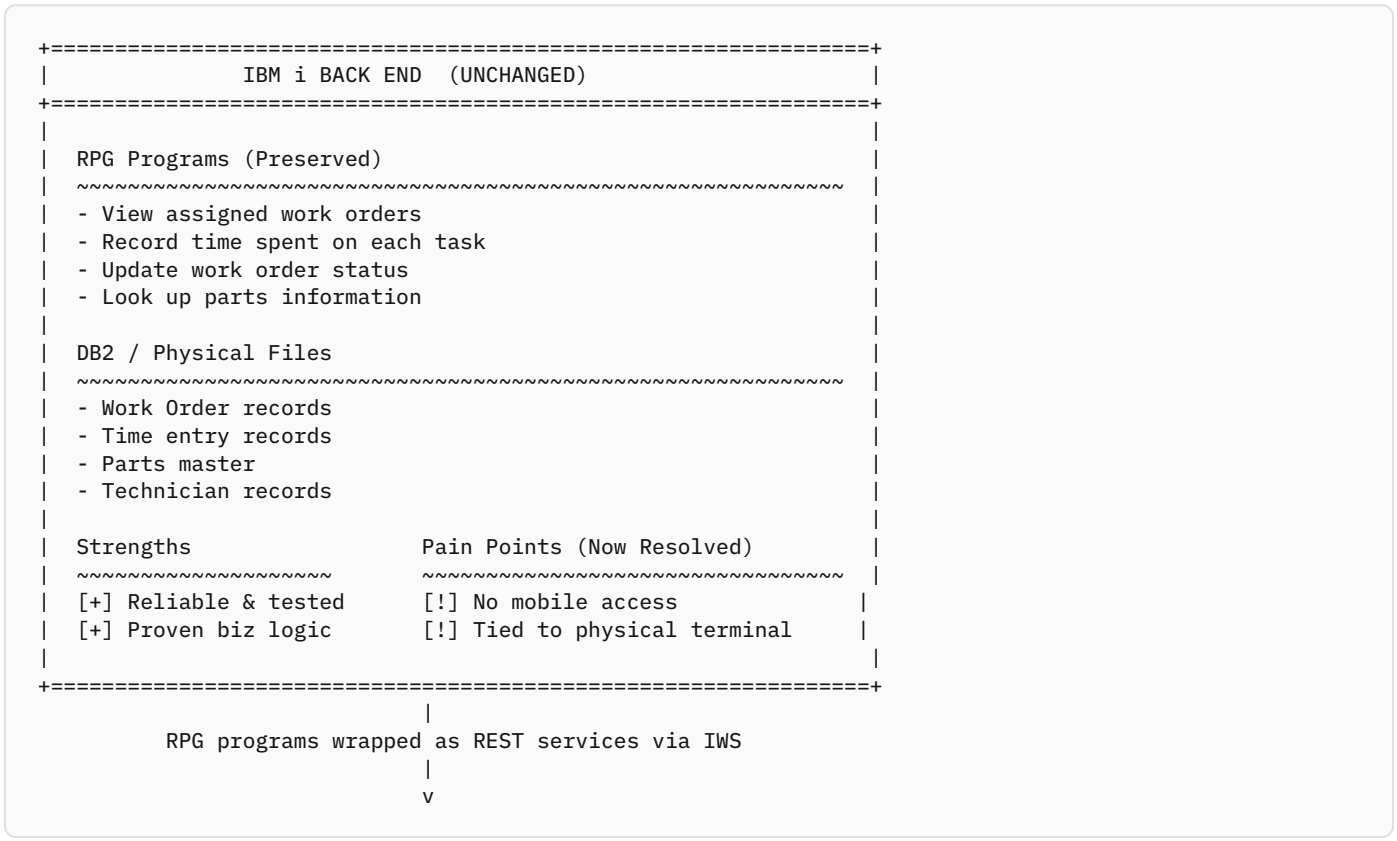
Method	Endpoint	Purpose
GET	/workorders?technician={id}	Retrieve work orders for a technician
GET	/workorders/{id}	Retrieve full detail for a work order
POST	/workorders/{id}/time	Record a time entry
PUT	/workorders/{id}/status	Update work order status

Front-End Stack

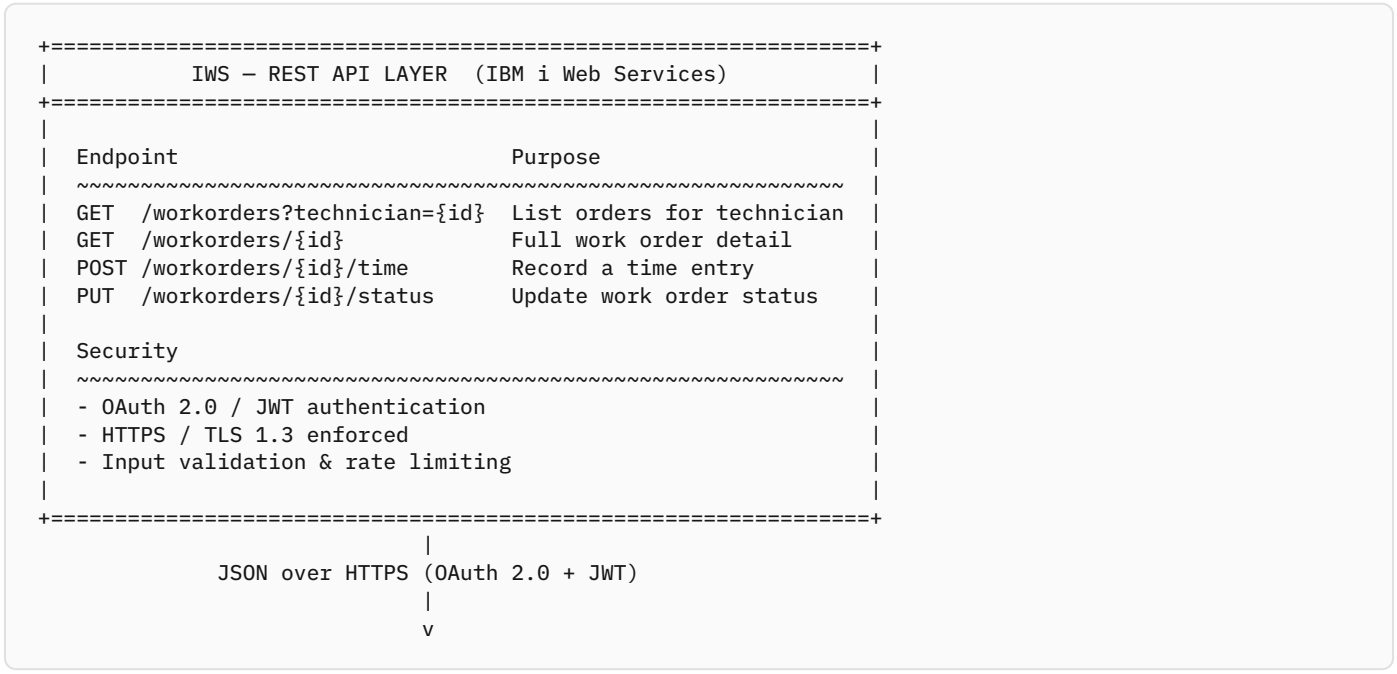
- Built with **React** and **Design System UI** components
- Delivered as a **Progressive Web App (PWA)**:
  - Installable directly on a technician's phone
  - Accessible without opening a browser
  - Works on any device technicians already carry

Basic Architecture

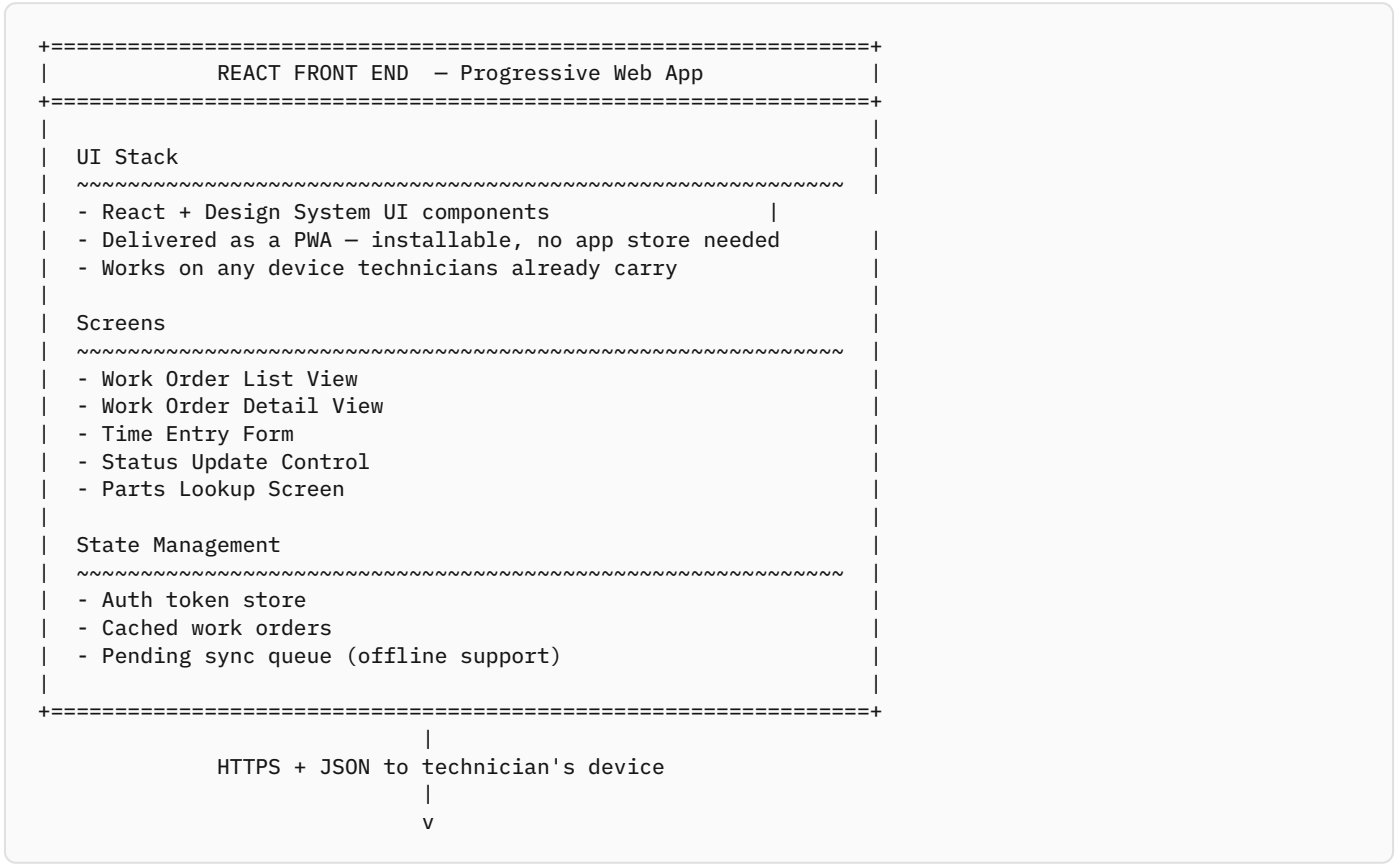
1 — IBM i Back End



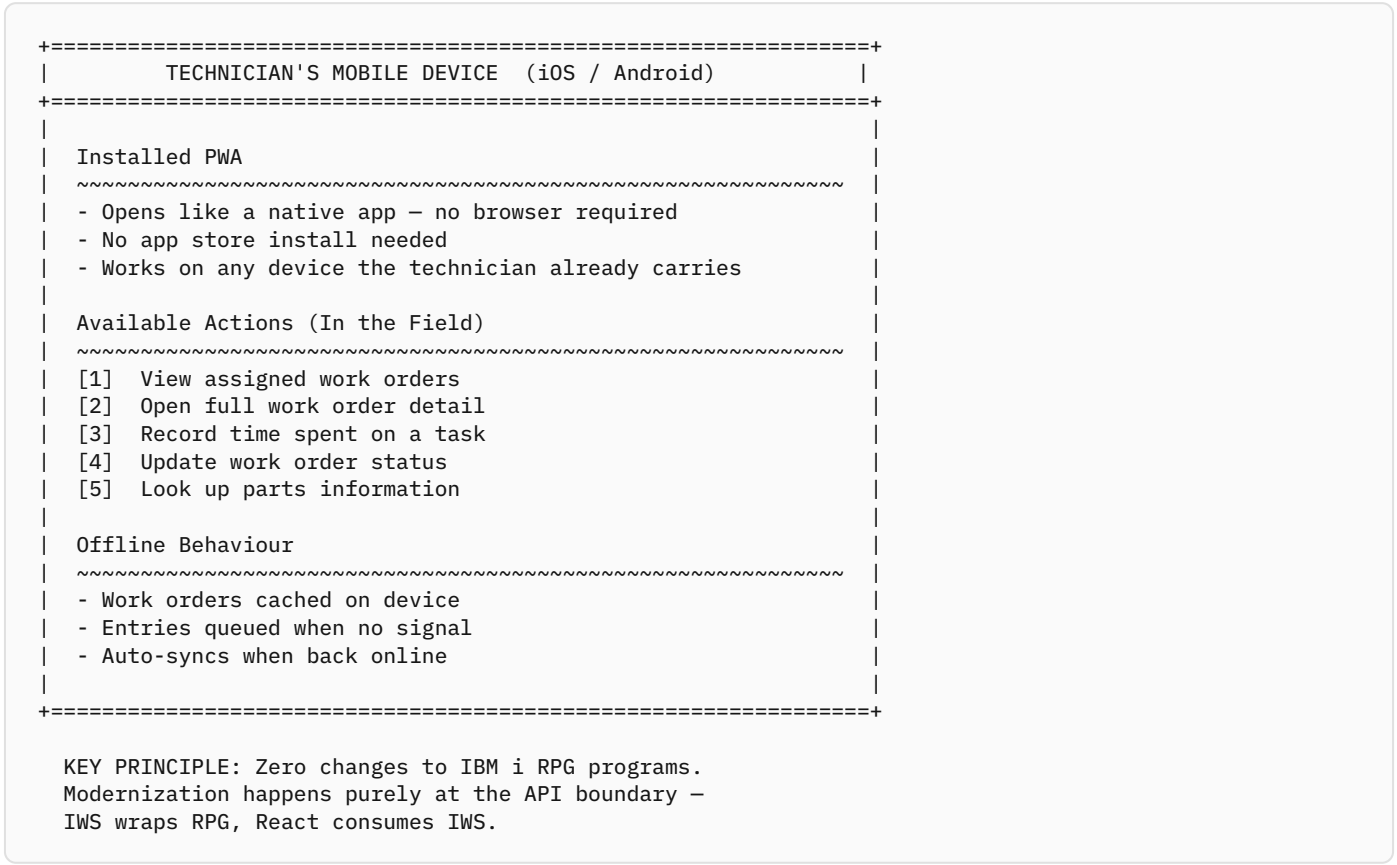
2 — IWS REST API Layer



3 — React Front End (PWA)



4 — Technician's Mobile Device



Outcome & Key Takeaways

The result is an application that technicians can use from anywhere, on the device they already carry, without any changes to the IBM i business logic that the organization has relied on for years.

Key Takeaways:

- Technicians can now manage work orders **from anywhere**, on their own devices
- **Zero changes** to IBM i business logic — the organization's trusted core remains intact
- Modernized experience delivered purely through a **new API + front-end layer**
- Demonstrates that IBM i modernization does **not** require rewriting existing systems

Common Pitfalls to Avoid

Several common mistakes occur when organizations attempt mobile-first development for the first time. Being aware of them can save significant rework.

- **Treating mobile as a reduced feature set permanently:** Mobile-first means designing for mobile constraints first, but it does not mean that mobile users should permanently receive fewer features. As users become comfortable with the mobile experience, they may want access to more functionality on mobile. Design for scalability.
- **Ignoring performance until late in the project:** Performance on mobile networks is a first-class concern. Large JavaScript bundles, unoptimized images, and chatty APIs that make many small requests are all performance problems that are much easier to address if they are considered from the start.
- **Skipping real-device testing:** Browser developer tools are useful but do not fully replicate the experience of using an application on a real device. Include real-device testing in the development process from the beginning.
- **Over-engineering the back-end API:** RESTful services designed to be general-purpose (returning large amounts of data that the client must filter) are less efficient for mobile clients than services designed for specific use cases.

Consider designing APIs around the specific tasks that mobile users need to perform, not around the data model of the back-end application.

- **Neglecting security:** Mobile applications introduce security considerations that do not apply to traditional 5250 or LAN-based applications: data transmitted over public networks, devices that may be lost or stolen, and authentication on small screens where complex passwords are inconvenient. Address authentication (OAuth 2.0 and OpenID Connect are widely used standards), transport security (HTTPS always), and device security policies from the start of the project.

Summary

Mobile-first development is both a design philosophy and a technical strategy, rooted in the recognition that mobile devices are now the primary computing tool for most users. Applications must be built for mobile from the start, not adapted from desktop designs.

Its core principles include: prioritizing essential content and actions, designing for touch interaction, accounting for network and performance constraints, using responsive layouts that scale from small to large screens, and ensuring accessibility throughout.

For IBM i modernization, mobile-first fits naturally with service-oriented architecture. IBM i business logic is exposed as RESTful services consumed by a modern mobile-first frontend without changing the underlying backend.

IBM supports this approach with a focused toolset: Integrated Web Services Server for API exposure, IBM API Connect for enterprise API management, IBM MobileFirst for native and hybrid apps with accessible and responsive frontends.

Ultimately, the shift to mobile-first is not just technical. It changes how organizations think about users, workflows, and application quality. Those who make this shift deliver applications that are more useful, more widely adopted, and better aligned with how their workforce actually works.

Should I Care About Responsive Design?

If you have spent any time developing for the web, you have heard the term *responsive design* thrown around constantly. For a working developer juggling deadlines, feature requests and legacy code, it is reasonable to stop and ask: does this actually matter for my project, or is it just another buzzword to feel guilty about ignoring?

The honest answer is: it depends, but the cases where you can safely ignore it are shrinking fast.

What Responsive Design Actually Means

Responsive design is the practice of building web interfaces that adapt their layout and presentation to the screen size and capabilities of the device rendering them.

Rather than maintaining separate desktop and mobile versions of a site, a responsive design uses fluid grids, flexible images and CSS media queries to serve a single codebase that works well everywhere.

Today your users might be on a 27-inch desktop monitor, a 13-inch laptop, a tablet in landscape or portrait mode, a phone or even a smart TV. The web has never been a fixed canvas.

The Case For Caring

The numbers are hard to argue with. Mobile devices now account for roughly 60% of global web traffic. In some sectors, retail, news, social, that figure is even higher.

If your site is not usable on a phone, you are turning away the majority of your potential audience before they have read a single word.

Search engines care. Google has used mobile-first indexing as its default since 2019, meaning it primarily uses the mobile version of your content for ranking.

A site that works poorly on mobile will rank worse in search results, regardless of how well-optimised the desktop experience is.

Users have no patience. Research consistently shows that mobile users abandon sites that are difficult to use on their device within a few seconds. A layout that requires horizontal scrolling, text that is too small to read without zooming, or buttons that are too close together to tap accurately, these are not minor inconveniences. Users will stop using your site.

Accessibility overlaps with responsiveness. Many of the practices that make a site responsive, flexible text sizing, touch-friendly tap targets, logical content flow also make it more accessible to users with disabilities. Responsive design is not just about screen size; it is about reducing assumptions about how someone is interacting with your page.

The Case Against

There are genuine situations where full responsive design is a lower priority.

Internal tooling is a legitimate exception. If you are building a back-office admin panel, a warehouse management interface, or an internal reporting dashboard that will only ever be used by staff sitting at desks with large monitors, spending significant effort on mobile layouts may be poor use of your time. Know your user base!

Complexity has a cost. A fully responsive design is harder to build and maintain than a fixed-width layout. Components that look great at 1200px sometimes behave unexpectedly at 375px. If you are a solo developer or a small team with limited resources, it is worth being honest about what you can sustain.

Some content does not translate well to small screens. Complex data tables, multi-panel dashboards and rich interactive maps are genuinely difficult to make usable on a phone. In these cases, a graceful degradation, showing a simplified mobile view or a clear message directing users to a desktop, may be more honest than a poorly executed attempt at full responsiveness.

A Practical Middle Ground

Rather than treating responsive design as an all-or-nothing commitment, most developers are better served by thinking in tiers.

Start with a mobile-first approach in your CSS. Write your base styles for small screens and add complexity for larger ones.

This tends to produce leaner stylesheets than the reverse and forces you to prioritise content over decoration.

Adopt a component library or CSS framework that handles responsiveness for you where possible. Tools like Tailwind CSS, Bootstrap or CSS Grid with sensible breakpoints mean you do not have to solve every layout problem from scratch.

Test on real devices early, not as an afterthought!

Browser developer tools are useful but they do not replicate the full experience of tapping a button with a thumb or reading text in sunlight.

The Verdict

Responsive design is not a trend you can wait to go away. It is a reflection of how people actually use the web today. For public-facing projects, ignoring it means a worse experience for the majority of your users, lower search rankings and a codebase that will feel increasingly dated.

That said, you do not have to apply it universally. Apply it proportionally. Understand your audience, be honest about your constraints and make deliberate trade-offs rather than defaulting to either extreme.

The developers who get this right are not the ones who follow every guideline, they are the ones who know when and why the guidelines exist.

Andy Youens March 2026

What Mobile Framework Works with IBM i?

Introduction

A common question in IBM i modernization projects is: *"Which mobile framework do I need to use with IBM i?"* It's understandable — the wrong choice can mean wasted effort, poor maintainability, and a weak user experience. Many IBM i systems host core business logic written in RPG, COBOL, or CL, traditionally built for green-screen (5250) interfaces. As businesses adopt mobile-first strategies, framework selection becomes a critical decision.

However, this question often stems from a misconception: that IBM i restricts front-end choices. In reality, a properly modernized IBM i environment is **framework-agnostic**. Through REST APIs, SOAP services, and standard database connectivity, IBM i exposes business logic over HTTP/HTTPS, making it a viable backend for any modern mobile framework. Any framework that can make an HTTP request can consume those APIs.

Framework selection should therefore be driven by team skills, target devices, and user experience goals — not by IBM i itself. This section explores the frameworks most commonly used in IBM i modernization, their compatibility with the platform, and best practices for building mobile-enabled solutions that modernize the interface while preserving existing backend investments.

The IBM i Back End as a Service Provider

IBM i Exposes REST APIs

Modern IBM i applications expose business logic as RESTful web services, typically through the **IBM i Integrated Web Services (IWS) Server**, which publishes RPG and COBOL programs as HTTP endpoints with JSON or XML payloads. Organizations may also use IBM API Connect, Java-based application servers like Apache Tomcat (for deploying Spring Boot or servlet-based APIs), or open-source middleware like Node.js on IBM i's PASE environment.

The result is a set of endpoints — for example, `https://ibmi-host/api/orders` — that any HTTP client can call. The mobile frontend sends a request, IBM i processes it through existing business programs and returns a JSON response, which the frontend then renders.

Separation of Concerns

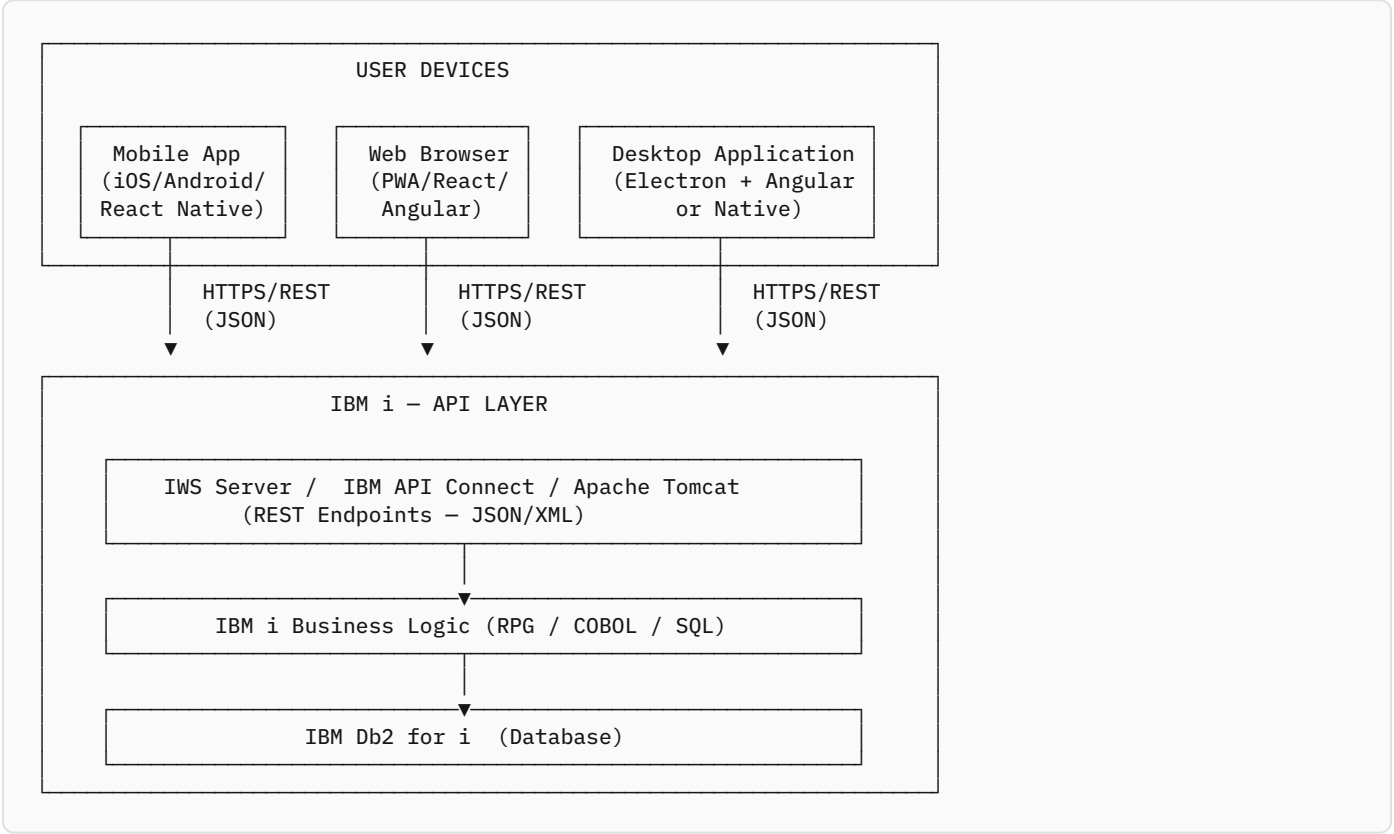
This architecture cleanly separates the **back end** (IBM i: business logic, data, security) from the **front end** (mobile framework: UI, navigation, interaction), with communication only through the API. This means:

- The mobile framework can be replaced without touching the IBM i backend
- The IBM i backend can be upgraded without breaking the frontend, provided the API contract is maintained
- Multiple frontends (mobile, web, desktop) can consume the same IBM i APIs simultaneously

Authentication and Security

Mobile applications authenticate with IBM i APIs using **OAuth 2.0** and **JWT (JSON Web Tokens)**. The app authenticates via IBM Security Verify or another enterprise identity provider, receives a token, and includes it in every API request's `Authorization` header. IBM API Connect or IWS validates the token before granting backend access.

This keeps IBM i off the public internet — the API gateway handles authentication, rate limiting, and routing, while IBM i operates securely within the internal network.



Multiple front-end frameworks consuming the same IBM i REST APIs. The front-end choice does not affect the IBM i backend.

Mobile Frameworks for IBM i Modernization

Choosing the right mobile framework is one of the most consequential decisions in any IBM i modernization project, as it determines the development toolchain, team skill requirements, deployment model, and long-term maintenance burden.

Regardless of framework, all modern mobile frameworks communicate with IBM i through the same mechanism: HTTPS requests to IBM i REST APIs. The framework choice affects the front-end experience and development workflow, not the backend integration model.

Progressive Web Apps (PWA)

A PWA is a web application built with HTML, CSS, and JavaScript, enhanced to behave like a native mobile app. PWAs can be installed from the browser to a device's home screen, run offline via service workers, and deliver push notifications — without app store distribution.

Connecting to IBM i: PWAs issue standard HTTPS `fetch` requests to IBM i REST APIs. No special middleware or IBM-specific SDK is required.

Best fit when:

- The team has existing web skills (Angular, React, Vue)
- A single codebase must target iOS, Android, and desktop
- App store distribution is not required
- Feature requirements are met by the modern web platform

PWAs are not ideal when deep hardware integration (Bluetooth, NFC, advanced file system) is needed beyond what the web platform supports.

Hybrid Mobile Apps

Hybrid apps are packaged as native binaries for iOS and Android, but the UI is built with HTML, CSS, and JavaScript running inside a native web view. Frameworks like **Apache Cordova** and **Ionic** wrap web apps in a native shell and expose JavaScript plugins for native device APIs unavailable to browser-based apps.

Connecting to IBM i: Hybrid apps call IBM i REST APIs over HTTPS identically to PWAs. Native packaging does not alter the integration model.

Ionic Framework is the most widely adopted hybrid framework for IBM i modernization. It compiles to native iOS, Android, or PWA targets from a single codebase, integrates with Angular, React, or Vue, and is compatible with Material or IBM Carbon Design System.

Best fit when:

- App store distribution is required
- Native device hardware access is needed beyond web platform capabilities
- The team has web skills and time-to-market is the priority

Cross-Platform Native Frameworks

Cross-platform native frameworks compile a single codebase to actual native code (*not a web view*) for each target platform, delivering native performance and hardware access without maintaining separate iOS and Android codebases.

React Native

React Native, from Meta, uses React but renders actual native UI components (not HTML) on iOS and Android, giving apps a native look, feel, and performance.

Connecting to IBM i: Uses the standard JavaScript `fetch` API or `axios` to make HTTPS requests to IBM i REST APIs — identical to a React web project.

Strengths for IBM i projects:

- Teams already using React for IBM i web modernization can transfer skills with moderate additional learning
- Design systems such as IBM Carbon, Material and Fluent provide community and enterprise-supported component patterns for React Native, enabling visual consistency across web and mobile
- Node.js running natively on IBM i can serve as the API layer, allowing one team to own the full JavaScript stack

Best suited for field service, logistics, and other applications requiring complex data handling and a high-quality native experience.

Flutter

Flutter, from Google, uses the Dart programming language and draws all UI elements using its own rendering engine (Skia/Impeller) rather than platform-native components. This delivers pixel-perfect visual consistency across iOS, Android, web, and desktop.

Connecting to IBM i: Uses the `http` or `dio` Dart packages to make **HTTPS requests** to IBM i REST APIs. Dart's strong typing maps cleanly to IBM i's strongly typed data structures.

Best fit when:

- Pixel-perfect visual consistency across all platforms is required
- The application targets both mobile and desktop (Flutter supports Windows, macOS, and Linux)
- High-fidelity animations or custom visualizations are needed
- The team is willing to adopt Dart — a shallow learning curve for those familiar with Java, Kotlin, or TypeScript

Backend Integration Patterns

The API layer between the mobile app and IBM i can use several backend technologies, independent of the mobile framework choice.

- **Java:** A proven enterprise option with Spring Boot, Jakarta REST, MicroProfile, and JDBC for DB2 access. IBM Java SDK provides optimized on-platform performance.
- **Node.js:** Well-suited for concurrent mobile connections due to its non-blocking I/O model. Runs natively on IBM i, simplifying infrastructure. Native JSON support and rich library ecosystem including DB2 and ODBC connectors.
- **.NET (ASP.NET Core):** Suitable for Microsoft-invested organizations. Integrates with Azure, supports Entity Framework over DB2, and offers strong security features.
- **Python:** Appropriate for lighter workloads. Flask and FastAPI are common choices; IBM i Access Client Solutions includes Python support with a DB2 connector.

Framework Selection Summary

Framework	Language	Rendering	IBM i API Model	Best Fit
PWA	HTML/CSS/JS	Web browser	HTTPS REST	Web-first teams; no app store needed
Ionic (Hybrid)	HTML/CSS/JS	Web view in native shell	HTTPS REST	Existing web skills; app store distribution needed
React Native	JavaScript / React	Native components	HTTPS REST	React teams; performance-sensitive native apps
Flutter	Dart	Custom rendering engine	HTTPS REST	Pixel-perfect consistency; multi-platform (mobile + desktop)

All four frameworks consume IBM i REST APIs over HTTPS and require no special IBM i client libraries on the device. The decision should be driven by team skills, performance requirements, and distribution model.

Architecture Reference

```
Mobile Application (PWA / Ionic / React Native / Flutter)
        |
        |  HTTPS REST (JSON)
        |
  API Middleware Layer (Node.js / Spring Boot / ASP.NET Core / FastAPI)
        |
        |  JDBC / ODBC / Native Program Call
        |
   IBM i Backend (RPG / CL Programs, DB2 for i)
```

The middleware layer handles authentication, authorization, request validation, and translation between mobile-friendly JSON and IBM i data structures. Keeping it thin and stateless simplifies scaling and testing.

Enterprise Design Systems: The UI Foundation

Enterprise design systems deserve particular attention in the context of IBM i UI modernization. They are not simply visual themes — they are complete design systems that provide components, patterns, and guidance specifically for enterprise applications. They are the preferred UI foundation for IBM i modernization projects that target web and mobile web delivery.

Popular options include:

- **Google Material Design** (via MUI for React)
- **Microsoft Fluent UI**
- **Ant Design**
- **IBM Carbon Design System**
- **Atlassian Design System**

What a Design System Provides

Modern enterprise design systems provide the following building blocks for mobile-first IBM i front ends:

Grid system. Responsive grid layouts adapt from mobile (single column) to tablet (multi-column) to desktop (full width), following mobile-first CSS principles with `min-width` media queries.

Components. Most design systems include 40+ pre-built, accessible components: buttons, forms, data tables, modals, notifications, navigation menus, date pickers, and more. Each component is designed for accessibility (WCAG 2.1 AA compliance) and works on touch screens as well as mouse/keyboard.

Themes. Design systems typically provide light and dark themes and support custom theming through CSS custom properties. Organizations can apply their own brand colors while retaining the system's layout and accessibility characteristics.

Patterns. Beyond individual components, design systems document common interaction patterns for enterprise applications — how to design a data table with filtering and pagination, how to structure a multi-step form, how to handle error states. These patterns encode accumulated experience building enterprise UIs at scale.

Icons. Most design systems include a comprehensive icon library in SVG format, designed for use at small sizes on high-density (Retina) screens.

Installing a Design System in a React Project

The examples below show installation and basic setup for each of the major design systems. All follow the same pattern — install the package, import styles or a provider, and use components. Choose the one that best fits your organization's standards or existing IBM i front-end conventions.

MUI (Material Design)

```
npm install @mui/material @emotion/react @emotion/styled
```

```
import { Button, Container, Typography } from '@mui/material';

function App() {
  return (
    <Container maxWidth="lg">
      <Typography variant="h4" gutterBottom>Work Orders</Typography>
      <Button variant="contained" color="primary" onClick={fetchWorkOrders}>
        Refresh
      </Button>
    </Container>
  );
}
```

Microsoft Fluent UI

```
npm install @fluentui/react-components
```

```
import { FluentProvider, webLightTheme, Button } from '@fluentui/react-components';

function App() {
  return (
    <FluentProvider theme={webLightTheme}>
      <h1>Work Orders</h1>
      <Button appearance="primary" onClick={fetchWorkOrders}>Refresh</Button>
    </FluentProvider>
  );
}
```

Ant Design

```
npm install antd
```

```
import { Button, Typography } from 'antd';
import 'antd/dist/reset.css';

function App() {
  return (
    <>
      <Typography.Title level={2}>Work Orders</Typography.Title>
      <Button type="primary" onClick={fetchWorkOrders}>Refresh</Button>
    </>
  );
}
```

IBM Carbon Design System

```
npm install @carbon/react
```

```
import '@carbon/react/index.scss';
import { Button } from '@carbon/react';

function App() {
  return (
    <div className="cds--grid">
      <div className="cds--row">
        <div className="cds--col-sm-4 cds--col-md-8 cds--col-lg-12">
          <h1>Work Orders</h1>
          <Button kind="primary" onClick={fetchWorkOrders}>Refresh</Button>
        </div>
      </div>
    </div>
  );
}
```

Calling IBM i REST APIs from a Design System React App

Calling IBM i APIs from a React application uses the standard JavaScript `fetch` function regardless of which design system is in use. The fetch logic and data-mapping pattern are identical across all options — only the table component names and props differ. The IBM i back-end requires no changes in any case; it simply returns JSON from its existing RPG or SQL business logic.

The following example shows how to retrieve a list of work orders from an IBM i REST endpoint and display them in a Carbon data table:

```
import { useEffect, useState } from 'react';
import {
  DataTable,
  TableContainer,
  Table,
  TableHead,
  TableRow,
  TableHeader,
  TableBody,
  TableCell,
  Loading,
  InlineNotification
} from '@carbon/react';

const headers = [
  { key: 'orderId',     header: 'Order ID'    },
  { key: 'description', header: 'Description' },
  { key: 'status',      header: 'Status'      },
  { key: 'assignedTo',  header: 'Assigned To' },
];

function WorkOrderList() {
  const [rows, setRows]       = useState([]);
  const [loading, setLoading] = useState(true);
  const [error, setError]     = useState(null);

  useEffect(() => {
    fetch('https://ibmi-host/api/workorders', {
      headers: {
        'Authorization': `Bearer ${sessionStorage.getItem('token')}`,
        'Content-Type': 'application/json',
      },
    })
      .then(res => {
        if (!res.ok) throw new Error(`HTTP ${res.status}`);
        return res.json();
      })
      .then(data => setRows(data))
      .catch(err => setError(err.message))
      .finally(() => setLoading(false));
  }, []);

  if (loading) return <Loading description="Loading work orders..." />;
  if (error)
    return (
      <InlineNotification
        kind="error"
        title="Could not load work orders"
        subtitle={error}
      />
    );

  return (
    <DataTable rows={rows} headers={headers}>
      {({ rows, headers, getTableProps, getHeaderProps, getRowProps }) => (
        <TableContainer title="Work Orders">
          <Table {...getTableProps()}>
            <TableHead>
              <TableRow>
                {headers.map(h => (
                  <TableHeader {...getHeaderProps({ header: h })} key={h.key}>
                    {h.header}
                  </TableHeader>
```

```
                ))}
              </TableRow>
            </TableHead>
            <TableBody>
              {rows.map(row => (
                <TableRow {...getRowProps({ row })} key={row.id}>
                  {row.cells.map(cell => (
                    <TableCell key={cell.id}>{cell.value}</TableCell>
                  ))}
                </TableRow>
              ))}
            </TableBody>
          </Table>
        </TableContainer>
      )}
    </DataTable>
  );
}
```

This component renders a fully responsive, accessible data table populated with data from IBM i. On a mobile screen, the table reflows appropriately. On a desktop, it displays in full multi-column format. The IBM i back end requires no changes — it simply returns JSON from its existing RPG or SQL business logic.

Choosing the Right Mobile Framework for IBM i Modernization

Selecting a mobile framework is not a purely technical decision. It involves a realistic assessment of team capabilities, project constraints, business priorities, and long-term maintainability. This section provides a structured set of criteria to guide that decision, along with a concrete recommendation for teams starting their first IBM i mobile modernization project.

Evaluation Criteria

Technical Considerations

Performance Requirements

Different application types carry different performance expectations. A field service app that processes barcode scans and syncs orders has different demands than an executive dashboard that loads a few charts. Before evaluating frameworks, define the following:

- Target response times for common user interactions (screen loads, form submissions, data lookups)
- Expected concurrent user load on the IBM i REST API layer
- Volume and complexity of data transactions per session
- Network conditions users will operate in (reliable Wi-Fi, mobile data, intermittent connectivity)

React Native and Flutter are the strongest choices when native rendering performance is non-negotiable. PWAs and hybrid apps perform well for most business workflows but may show limitations in animation-heavy or computationally intensive scenarios.

Offline Support

All four major framework categories support offline operation, but the implementation differs. PWAs use service workers to cache assets and data. React Native and Flutter rely on local database libraries (such as SQLite or Realm). Ionic-based hybrid apps can use either approach. If the application must function reliably in low-connectivity environments (for example, warehouse floors, field sites, or remote locations), offline support should be an explicit design requirement, not an afterthought.

Access to Native Device APIs

The web platform now exposes camera, geolocation, push notifications, and basic Bluetooth on most devices. If the project requires deeper hardware integration like NFC, barcode scanner SDKs, enterprise device management APIs,

or advanced file system access, then a hybrid or cross-platform native framework is required. PWAs are not suitable for these cases.

IBM i API Integration

All frameworks connect to IBM i through standard HTTPS REST API calls. There is no framework-specific IBM i SDK or connector required on the mobile side. The integration approach is uniform: the mobile app makes an HTTPS `fetch`, `axios`, or equivalent HTTP call to an API endpoint exposed by IBM i or a middleware layer. This means IBM i API compatibility should not be a differentiating factor in the framework decision.

Team and Skill Considerations

The available skill set within the development team is one of the most reliable predictors of project success. A technically superior framework delivered by an inexperienced team will underperform a simpler framework used by a skilled one.

Existing expertise is the primary lens. Teams with strong React and JavaScript backgrounds can move to React Native with moderate additional learning. Teams with general web skills (HTML, CSS, JavaScript) can be productive in Ionic or PWA development immediately. Flutter requires learning Dart, which is a real investment — although Dart has a shallow learning curve for developers familiar with Java, Kotlin, or TypeScript.

Long-term maintainability matters as much as initial development velocity. Choose a framework the team can support without sustained external dependency. A Flutter application maintained by a team that never fully adopted Dart will accrue technical debt over time.

Business Considerations

Distribution Model

PWAs are deployed like any web application — users access them via a URL and can install them to their home screen without going through an app store. This simplifies deployment, eliminates app store review cycles, and allows updates to be pushed instantly. However, PWAs are not listed in the App Store or Google Play, which may be a requirement in some organizational contexts.

Ionic, React Native, and Flutter all produce app store-distributable binaries. If corporate policy, end-user expectations, or IT governance requires an installed app, these frameworks are appropriate.

Total Cost of Ownership

Initial development cost is one component; ongoing maintenance is often larger. Key factors include:

- Framework update cycles and backward compatibility — React Native and Flutter both have active release schedules with occasional breaking changes
- Third-party library dependencies and their maintenance status
- Training and onboarding cost for new team members
- Infrastructure required for the API backend (on-platform Node.js or Java, or a separate middleware tier)

Time-to-Market

PWAs and Ionic hybrid apps typically offer the fastest path from existing web skills to a deployable application. React Native requires more ramp-up for teams new to React, but the investment pays off in application quality. Flutter has the longest initial learning curve but can achieve high productivity once the team is fluent in Dart and the widget model.

Recommended Starting Point

For most IBM i modernization teams, the recommended starting point is a **Progressive Web App built with React and UI Design System**.

This recommendation is grounded in practical realities rather than theoretical merit:

Low barrier to entry. Web development skills — HTML, CSS, JavaScript, and React — are widely available in the market, and the majority of IBM i modernization teams already have some exposure to these technologies through earlier web-facing work.

UI Design alignment. UI Design System is built and optimized for React. It provides a comprehensive, accessible, branded component library that significantly accelerates development and ensures visual consistency with other IBM i web interfaces in the modernization portfolio.

No app store friction. PWAs are deployed and updated like any web application. Users access them via a URL and can add them to their home screen without visiting an app store. This removes the overhead of app store review cycles and allows the team to iterate quickly.

Straightforward IBM i integration. A React PWA connects to IBM i REST APIs through standard HTTPS `fetch` calls — the simplest possible integration path with no additional client libraries required.

A natural upgrade path. If the project later requires native app distribution, deeper device hardware access, or integration with enterprise mobile device management platforms, the team's React knowledge transfers directly to React Native. Many component patterns and business logic from the PWA codebase can be carried forward with minimal rework.

When to Move Directly to React Native

If project requirements clearly indicate a native app from the outset — offline barcode scanning, integration with hardware not accessible through web APIs, mandatory app store distribution, or corporate mobile device management enrollment — **React Native** is the natural choice for teams already working in React and JavaScript. It avoids the intermediate PWA step and delivers a production-quality native application from the start.

When to Consider Flutter

Flutter is the right choice when pixel-perfect visual consistency across mobile and desktop is a hard requirement, or when the application roadmap includes Windows, macOS, or Linux desktop targets in addition to iOS and Android. It is also appropriate when the organization is making a deliberate long-term investment in building Dart expertise within the development team.

Framework Comparison

The table below consolidates the key characteristics of each framework across the criteria discussed above.

Criterion	PWA	Ionic / Cordova	React Native	Flutter
Team skills needed	Web / JS	Web / JS	React / JS	Dart
App store distribution	No	Yes	Yes	Yes
Offline support	Yes (service workers)	Yes	Yes	Yes
Native device API access	Limited	Yes	Yes	Yes
Rendering performance	Good	Good	Excellent	Excellent
Single codebase (iOS + Android)	Yes	Yes	Yes	Yes
IBM i API integration	fetch / REST	fetch / REST	fetch / REST	http / dio package
Learning curve	Low	Low – Medium	Medium	High
Development speed	High	High	Medium – High	Medium
Flexibility	Medium	High	High	High

IBM MobileFirst Platform Foundation

For organizations that require a more comprehensive enterprise mobile management solution, **IBM MobileFirst Platform Foundation** provides a server-side platform specifically designed to support mobile applications connected to enterprise back ends, including IBM i.

IBM MobileFirst provides:

Adapters. Adapters are server-side components that proxy and transform requests between mobile clients and back-end systems. An adapter can connect to IBM i REST APIs (or directly to IBM i data through JDBC), apply business logic and data transformation, and return the result to the mobile client. Adapters can also handle authentication, caching, and error handling at the server level.

Security framework. MobileFirst includes an OAuth 2.0 based security framework with support for multiple authentication flows and challenge handlers. This simplifies the implementation of authentication in the mobile client.

Push notifications. MobileFirst provides a unified API for sending push notifications to iOS, Android, and Windows devices, using a single server-side call regardless of platform.

Analytics. MobileFirst collects analytics on application usage, crashes, and performance, giving the team visibility into how the mobile application is being used in production.

Device management. MobileFirst tracks enrolled devices and can enforce policies (such as blocking access from compromised devices).

IBM MobileFirst is particularly relevant when IBM i applications need to be extended to mobile in a large enterprise environment with complex security requirements, many users, and multiple back-end systems. For smaller projects, a direct-to-REST-API approach without MobileFirst is simpler and may be sufficient.

Integration Considerations

Regardless of framework, the following integration considerations apply to all IBM i mobile projects:

Security. All communication between the mobile front end and IBM i should occur over HTTPS. API endpoints should be protected with appropriate authentication mechanisms — OAuth 2.0 and JWT-based tokens are the most common patterns for REST APIs. Avoid exposing internal IBM i interfaces directly to the public internet.

Performance. Design APIs to return only the data required by the mobile interface. Avoid adapting existing green-screen data structures directly to API responses without review. Mobile networks are variable in bandwidth and latency; APIs should be designed with pagination, filtering, and minimal payload sizes in mind.

Testing. A clean API boundary between IBM i and the mobile front end simplifies testing significantly. Back-end APIs can be tested independently using tools such as Postman or automated API test suites. Front-end components can be developed and tested against mock API responses before full integration.

Language and Runtime Support. The IBM i API layer can be implemented using a variety of server-side runtimes beyond native IBM i web services, including Node.js, Java, Python, and .NET. The choice of runtime depends on the team's expertise and the complexity of the integration logic required.

Summary

IBM i's open, API-first approach to integration makes it a durable and adaptable platform for mobile application development. By exposing business logic as REST services, organizations retain the ability to adopt new front-end technologies as they mature — without rewriting back-end systems.

For most IBM i modernization projects, the recommended starting point is a React-based Progressive Web App using IBM Carbon Design System, integrated with IBM i through REST APIs. From this foundation, teams can extend to React Native for native distribution, or evaluate Flutter and IBM MobileFirst for specific performance or enterprise management requirements.

The most important architectural decision is not which framework to choose, but ensuring that IBM i business logic is consistently delivered as clean, well-documented services — making the system ready to support whatever mobile technology comes next.

Which UI Approach Should I Choose?

Modernizing your IBM i user interface is not a simple decision. The right approach depends on your organization's goals, resources, timeline, and long-term strategy. Some organizations need quick wins to demonstrate progress. Others require strategic transformation that positions them for the next decade.

This article examines the primary UI modernization approaches available for IBM i applications—from keeping your existing 5250 interface to building custom web solutions or adopting turnkey ISV products. Each approach has distinct advantages and limitations. Understanding these trade-offs is essential to making an informed decision.

The UI Modernization Spectrum

Staying with 5250

The 5250 green screen interface remains a viable option for certain scenarios. If your users are experienced, your workflows are stable, and your business requirements haven't changed significantly, maintaining the existing interface may be the most pragmatic choice. You can also consider have both options, a modern web and the 5250 interface.

When it makes sense: In today's world, keeping 5250 as your only option is not a viable solution. It may be acceptable to keep 5250 for high-speed data entry activities, but also have a more modern interface.

Limitations: Difficult to attract new talent, limited integration with modern systems, poor mobile support, and increasingly out of step with user expectations.

Screen Scraping

Screen scraping converts terminal-based applications into window-based, web, or mobile interfaces by intercepting the 5250 data stream. The scraper extracts data and generates a new interface while the underlying application remains completely unchanged. It's a tactical solution that shows progress quickly, and can then allow a more modern interface to be built.

Advantages:

- Extremely fast implementation—days or weeks, not months
- Works without source code access
- No changes to existing programs
- Low risk to stable business logic
- Immediate visual improvement

Limitations:

- Limited flexibility in redesigning workflows
- Performance overhead from translation layer
- Green screen logic translated directly often creates awkward user experiences
- Does not address underlying technical debt
- Menu-driven workflows don't translate well to modern UI patterns

Screen scraping is tactical, not strategic. It demonstrates progress quickly but leaves fundamental modernization challenges unaddressed.

Rational Open Access (RPG Edition)

IBM Rational Open Access operates differently than screen scraping. Instead of intercepting the data stream, it directly accesses the RPG program's I/O buffer, allowing you to process data by field name or through data structures.

This provides more control than screen scraping. You can modify UI behavior from RPG code and pass data between your new interface and existing business logic.

> **Strategic Caution:** While Open Access is mentioned here for completeness, it is not recommended as a strategic modernization approach. It remains tightly coupled to the RPG program structure and does not address underlying maintainability challenges. Open Access is best viewed as a tactical option for specific scenarios rather than a foundation for long-term modernization.

Custom Web Solutions

Building custom web interfaces using modern frontend frameworks represents a more comprehensive approach. Organizations can develop new UI components using technologies like Angular, React, or Vue.js, creating truly modern user experiences unconstrained by legacy screen layouts.

Frontend Technologies: Modern JavaScript frameworks provide rich component libraries, responsive design, accessibility standards, and sophisticated user interactions. AI-assisted tools can accelerate development by reading 5250 screen descriptions and generating corresponding frontend components automatically.

IBM i Backend Options: Custom web solutions typically integrate with IBM i through REST APIs and web services. Several technologies enable this on IBM i:

- **Java with Liberty:** Enterprise-grade, robust, extensive ecosystem
- **Node.js:** JavaScript on the server, rapid development, large package ecosystem
- **Python:** Clean syntax, strong data processing capabilities, growing IBM i support
- **PHP:** Mature web development platform, straightforward IBM i integration

Advantages:

- Complete control over user experience
- Access to modern web development ecosystem

- Decoupled architecture enables independent evolution
- Attractive to developers with modern skillsets
- Mobile-responsive by design

Limitations:

- Significant time investment
- Requires modern development expertise
- Backend programs remain unchanged (technical debt persists)
- Ongoing maintenance of custom code
- Integration complexity

Custom web solutions modernize the frontend experience without addressing backend technical debt. For comprehensive modernization, this approach should be paired with backend refactoring.

ISV Solutions

The IBM i community includes numerous Independent Software Vendors offering turnkey UI modernization solutions. Companies like Profound Logic, CNX, Fresche, and others provide proven platforms that accelerate the modernization process.

Advantages:

- Proven technology with established track records
- Professional support and ongoing updates
- Faster than building custom solutions
- Best practices built into the platform
- Training and documentation included
- Community of users sharing knowledge

Limitations:

- Licensing costs (though often justified by reduced development time)
- Vendor dependency for updates and support
- May require adapting to platform conventions
- Less flexibility than fully custom solutions

ISV solutions represent a middle ground—faster than custom development, more sophisticated than screen scraping, with professional support throughout the journey.

Decision Framework

Choosing the right approach requires evaluating several factors:

Timeline and Resources:

- Need results in weeks? Consider screen scraping or ISV solutions
- Have 6-12 months and skilled developers? Custom web solutions become viable
- Limited budget? 5250 maintenance or screen scraping may be necessary

Strategic Goals:

- Quick wins to build momentum? Screen scraping or ISV solutions
- Long-term competitive advantage? Custom web solutions with backend modernization
- Attracting modern development talent? Avoid Open Access, favor web technologies

Technical Considerations:

- Source code unavailable? Screen scraping is your only option
- Complex workflows requiring redesign? Custom solutions or sophisticated ISV platforms
- Integration with other systems? Web-based approaches provide better options

Organizational Factors:

- Internal development expertise? Influences custom vs. ISV decision
- Risk tolerance? Screen scraping and ISV solutions are lower risk
- Vendor relationship preferences? May favor or discourage ISV solutions

The Strategic Perspective: Remember that UI modernization alone does not modernize your application. The backend code, data access patterns, and business logic remain unchanged with refacing approaches. For true modernization, plan to address the backend as well—either immediately or as a subsequent phase.

Making Your Choice

There is no universally correct answer. The right choice depends on your specific situation, constraints, and objectives.

Start by honestly assessing your organization's capabilities, timeline, and strategic goals. Consider whether you need a quick tactical win or are ready for strategic transformation. Evaluate your team's skills and your budget for both initial implementation and ongoing maintenance.

Most importantly, make a decision and start. Paralysis helps no one. Even tactical approaches like screen scraping can demonstrate progress, build organizational confidence, and create momentum for deeper modernization efforts.

The worst choice is no choice at all. Your competitors are modernizing. Your users expect modern interfaces. Your future employees want to work with current technologies. Choose an approach that fits your situation and begin the journey.

Security for modern interfaces

How to Make a User Interface More Secure

As IBM i applications modernize using browser and mobile interfaces, the attack surface expands. Yet increased security does not mean sacrificing usability or accessibility. The key is to implement security in every layer of your modernization strategy, combining encryption, multi-factor authentication, input sanitization and a tiered architecture, you can create interfaces that are both secure and practical.

The key is understanding that security is not a single feature you add, but rather a comprehensive approach that is implemented at every layer, it should limit access to only what is necessary and maintain comprehensive logs and constant monitoring which send alerts whenever something unexpected occurs.

Why It Matters

Every security breach starts somewhere, and user interfaces represent one of the most common entry points. Attackers exploit weak authentication, inject malicious code through unsanitized inputs, or leverage excessive permissions to access data they shouldn't reach. The consequences can extend far beyond technical problems, as data breaches damage reputation, trigger regulatory penalties, and erode customer trust.

The business case for interface security is straightforward: prevention costs far less than remediation. Implementing proper security controls during development adds only a minimal overhead, whereas responding to a breach can impact a business for weeks or months.

Encryption is Your First Line of Defense

All communication between users and your application must be encrypted using TLS. This is optional as unencrypted connections expose credentials, session tokens, and sensitive data to interception which can then be used in replay attacks where unauthorized third parties sell this data or use this data to access your systems

Multi-Factor Authentication is Essential

Multi-factor authentication (MFA) adds a critical second layer by requiring something the user knows (password) plus something they have (authentication app, SMS code, or hardware token). This combination dramatically reduces the risk of compromised credentials leading to unauthorized access. Implement MFA not just at initial login, but also for sensitive operations for example, when a user attempts to delete, modify security settings, access particularly sensitive data, etc. At such times you should require them to re-authenticate with an additional MFA code.

Consider geolocation and time-based restrictions where feasible. If your users typically access the system from specific countries or regions, flag or block access attempts from unexpected locations. If business operations occur during specific hours, restrict access outside those times or require additional authentication. These contextual controls add security without significantly impacting legitimate users while making attackers' jobs considerably harder.

Input Sanitization

Never trust user input. Every field, parameter, and value submitted through your interface must be validated and sanitized before processing. SQL injection attacks—where malicious code is inserted through input fields—remain one of the most common and dangerous vulnerabilities. Use parameterized queries or prepared statements that separate SQL code from data, making injection attacks impossible. Validate that inputs match expected formats, lengths, and character sets, rejecting anything suspicious.

Tiered Architecture Protect Your Data Layer

Implement a tiered architecture that separates your user interface layer from your data layer, with an application logic layer in between. The UI layer should never connect directly to your database. Instead, it communicates with an application layer that enforces business rules and security policies before accessing data. This separation provides multiple benefits; it limits the blast radius if the UI layer is compromised, enables consistent security enforcement, and makes it easier to audit and monitor data access.

Implement Firewall Between Application Layers

Place firewalls between each tier of the application and create rules to only allow each application layer to communicate to required network addresses, using the specific ports and protocols needed for that specific interaction, anything else should be blocked and reported. For example, the application layer should only be able to reach the data layer through defined channels. This network segmentation means that even if an attacker compromises one layer, they face additional barriers before reaching sensitive data.

For best security, these firewalls should be configured to deny everything by default, allowing only the bear minimum communications your application requires to function. As your application evolves you then add more rules to permit the new network traffic and remove the older rules that permitted the now obsolete traffic.

Implement Least Privilege Needed for Service Accounts

The service accounts that connect your application tiers must operate under the principle of least privilege, they should have only the minimum permissions necessary to perform their function, and nothing more. If the application layer only needs to read customer data and update order status, its database credentials should allow exactly that. So, for example, they should not update customer data.

Create separate service accounts for different functions. Do not use a single account with broad permissions. Instead, use specific accounts for order processing, reporting, customer service, and other functions, each with permissions tailored to its needs. These service accounts should not be able to modify security settings that could be used to escalate their privileges or data access authority. Implementing these limitations means that even if an attacker compromises one layer, they do not automatically gain access to the others.

Continuous Monitoring and Alerting

Implement comprehensive auditing and alerting throughout your architecture. Log all authentication attempts, especially failures. Record all data access, particularly for sensitive information. Monitor for unusual patterns access from unexpected locations, attempts to access data outside normal business hours, repeated failed authentication attempts, or queries that return unusually large result sets. Configure these alerts to integrate with company SEIM solutions, and where appropriate directly notify security team.

Framework implementation guides

How Do I Use React Native with IBM i?

Introduction

Enterprise mobility is now a baseline expectation. Everyone in an enterprise environment requires access to core systems via smartphones or tablets. IBM i provides reliability, transactional integrity, and high throughput, but was not designed for mobile-first experiences. The goal is not to replace existing applications, but to introduce a modern presentation layer that interacts with the backend while keeping the core system unchanged.

React Native enables this by allowing developers to build mobile apps using JavaScript and React, compiling to true native iOS and Android components. Unlike hybrid frameworks, it does not rely on web views, instead iOS uses real `UIButton` components, and Android uses real `RecyclerView` elements. This delivers native performance while maintaining a single codebase.

In this architecture, React Native acts as the **System of Engagement** (UI layer), while IBM i remains the **System of Record**, running RPG/COBOL logic with DB2 for i managing data. A secure API middle layer exposes IBM i functionality as REST services, consumed by React Native over HTTPS. A key advantage is that the IBM i backend remains unchanged which means all your existing RPG programs, service programs, and DB2 schemas continue to operate as-is. Only the presentation layer is modernized, without requiring new programming languages.

For teams already using React on IBM i web applications, adopting React Native is straightforward. The same component model, state management, API integration, and utility logic apply, enabling a single JavaScript skillset to support both web and mobile platforms.

Why React Native for IBM i Modernization?

The choice of mobile development framework is critical when building applications that connect to enterprise systems like IBM i. React Native stands out as a particularly effective solution for several compelling reasons, especially in the context of enterprise mobility.

React Native offers several advantages:

- **Single Codebase** for iOS and Android
- **Native Performance** using native components
- **Strong Ecosystem** and community support
- **Rapid Development** with hot reloading

For IBM i environments:

- Reduces cost of mobile development
- Accelerates UI modernization initiatives
- Integrates seamlessly with API-driven backends

Architectural Foundation: The Bridge to Power

React Native does not directly interact with IBM i. Instead, it consumes APIs exposed via middleware (Java, Node.js, etc.).

This separation is not a limitation; it is the correct architecture for mobile enterprise applications. It means:

- The IBM i backend can serve multiple clients simultaneously — the React Native mobile app, a React web app, and a desktop application can all use the same APIs
- The mobile app does not need to know or care whether the data lives in IBM Db2 for i, an Oracle database, or a cloud service
- Security controls (authentication, authorization, rate limiting) are applied at the API layer, not in the mobile app
- The IBM i backend can be upgraded or refactored without releasing a new version of the mobile app, as long as the API contract is maintained.

The diagram below shows how React Native fits into the IBM i modernization architecture.

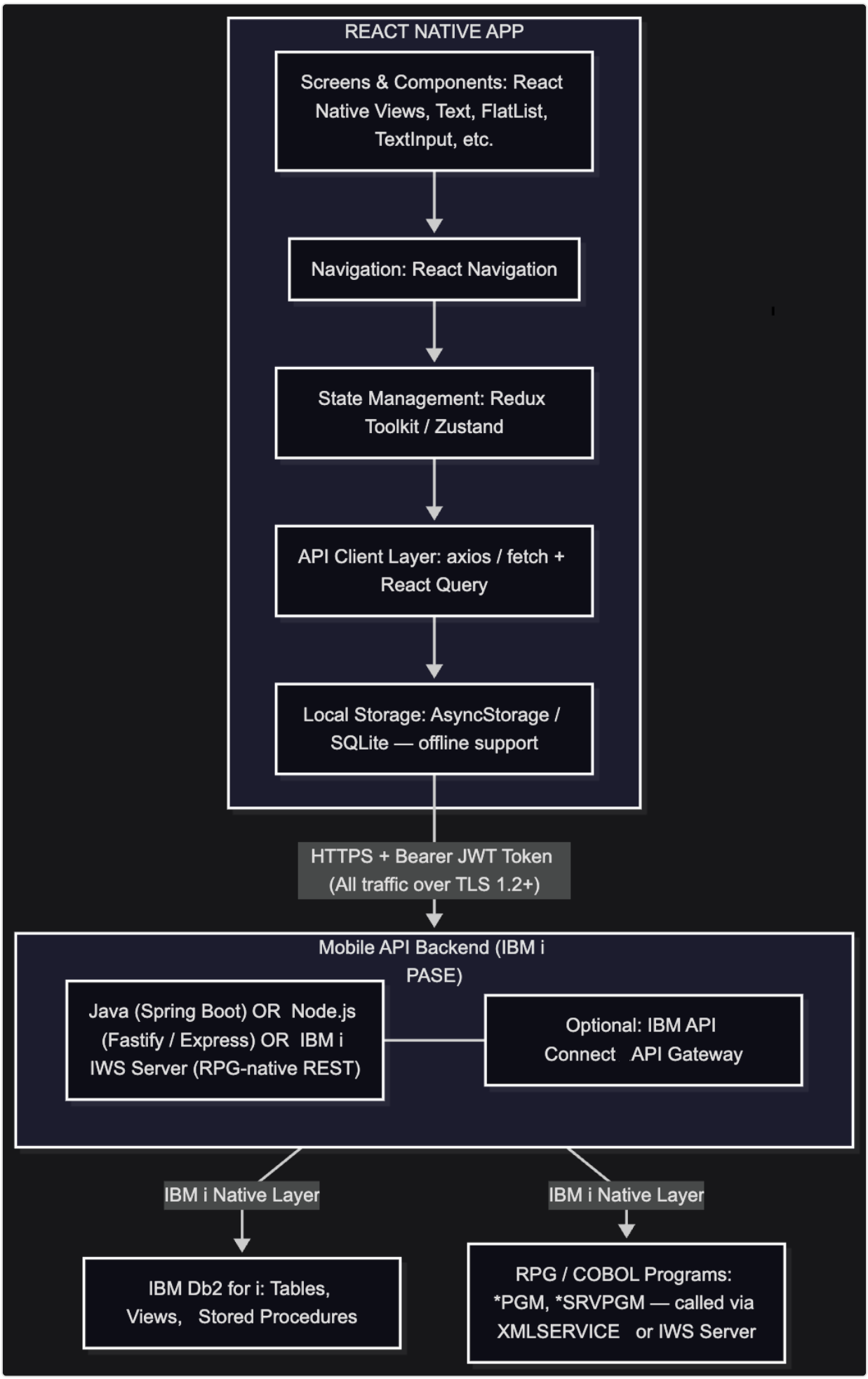

REACT NATIVE APP
Screens & Components: React Native Views, Text, FlatList, TextInput, etc.
Navigation: React Navigation
State Management: Redux Toolkit / Zustand
API Client Layer: axios / fetch + React Query
Local Storage: AsyncStorage / SQLite — offline support
HTTPS + Bearer JWT Token (All traffic over TLS 1.2+)
Mobile API Backend (IBM i PASE)
Java (Spring Boot) OR Node.js (Fastify / Express) OR IBM i IWS Server (RPG-native REST)
Optional: IBM API Connect API Gateway
IBM i Native Layer
IBM i Native Layer
IBM Db2 for i: Tables, Views, Stored Procedures
RPG / COBOL Programs: *PGM, *SRVPGM — called via XMLSERVICE or IWS Server

Figure. React Native Basic App Architecture

The Role of the API Backend

The API backend (hosted on the IBM i) acts as the translator. It receives JSON requests from the React Native app and performs the following:

1. **Authentication:** Validates the user's JWT (JSON Web Token).
2. **Data Orchestration:** Calls RPG programs via `itoolkit` or queries DB2 for i.
3. **Transformation:** Converts EBCDIC-based data or complex DS (Data Structures) into clean JSON.

Setting Up the React Native Development Environment

Setting up a React Native development environment requires several tools depending on your operating system and whether you are targeting iOS, Android, or both.

Below are the general prerequisites categorized by their role in the development process.

Core Prerequisites for React Native

Category	Tool	Description
Runtime	Node.js	The JavaScript runtime required to run the packager and manage dependencies. (LTS version recommended).
Package Manager	npm / Yarn	Used to install libraries and manage project packages.
Engine	Watchman	(macOS/Linux) A tool by Meta for watching file changes. Highly recommended for performance.
Language	Java (JDK)	Specifically the OpenJDK 17 (or 11, depending on version) is required for Android builds.
Android Tooling	Android Studio	Includes the Android SDK, SDK Platform-Tools, and Android Virtual Device (Emulator).
iOS Tooling	Xcode	(macOS only) Includes the iOS SDK, Simulators, and Command Line Tools. Required for iOS development.
iOS Dependencies	CocoaPods	The dependency manager for iOS (written in Ruby) used to link native libraries.
CLI	React Native CLI	The command-line interface used to initialize, build, and run projects.

Recommended Versions & Tips

1. **Node.js**: Ensure you use an even-numbered version (like 18 or 20). Use a version manager like `nvm` to switch versions easily.
2. **Environment Variables**: On Windows and macOS, you must manually set your `ANDROID_HOME` and `PATH` variables to point to the Android SDK location, or the CLI will not be able to find your emulator.

3. **Expo vs. CLI**:
    - **Expo Go**: If you are a beginner, you only need Node.js and a mobile phone. You don't need Android Studio or Xcode immediately.
    - **React Native CLI**: Required if you need to write custom Native Modules or have a complex app architecture. This requires all the tools listed in the tables above.

Creating a New Project

React Native projects can be created using the React Native CLI (for full control) or **Expo** (for a managed workflow that abstracts native build tooling). For IBM i enterprise projects, **React Native CLI** is recommended because enterprise deployments often require custom native modules, specific security configurations, and MDM (Mobile Device Management) integration — all of which are handled more predictably with the bare CLI workflow.

```
## Install React Native CLI globally

npm install -g react-native-cli

## Create the project

npx react-native@latest init IBMiMobileApp --template react-native-template-typescript

## Navigate into the project

cd IBMiMobileApp

## Start the Metro bundler (JavaScript bundler for React Native)

npx react-native start

## Run on iOS simulator (macOS only, requires Xcode)

npx react-native run-ios

## Run on Android emulator (requires Android Studio + running emulator)

npx react-native run-android
```

TypeScript Project

The `--template react-native-template-typescript` flag creates a TypeScript project. TypeScript is strongly recommended for IBM i enterprise mobile applications

Why TypeScript?

- **Catches integration errors at compile time** — not at runtime on a user's device
- Common errors it prevents:
    - Mismatched API response shapes
    - Wrong field names
    - Incorrect data types

Installing Key Dependencies

```
## Navigation

npm install @react-navigation/native @react-navigation/native-stack
npm install @react-navigation/bottom-tabs
npm install react-native-screens react-native-safe-area-context

## HTTP client and server state management

npm install axios
npm install @tanstack/react-query    # React Query — server state management

## Client state management

npm install @reduxjs/toolkit react-redux

## Secure token storage (replaces AsyncStorage for sensitive data)

npm install react-native-keychain

## Offline-capable local storage

npm install @react-native-async-storage/async-storage

## Form handling and validation

npm install react-hook-form zod @hookform/resolvers

## UI component library

npm install react-native-paper       # Material Design components
## OR

npm install @carbon/react-native    # IBM Carbon Design System (if available)

## Date handling

npm install date-fns

## Network status (for offline detection)

npm install @react-native-community/netinfo

## Icons

npm install react-native-vector-icons

## iOS-specific: link native modules

cd ios && pod install && cd ..
```

Recommended Project Structure

```
IBMiMobileApp/
├── src/
│   ├── api/
│   │   ├── client.ts           # Axios instance + interceptors
│   │   ├── workOrders.ts       # Work order API calls
│   │   ├── auth.ts             # Authentication API calls
│   │   └── types.ts            # TypeScript types for API responses
│   ├── components/
│   │   ├── common/
│   │   │   ├── LoadingSpinner.tsx
│   │   │   ├── ErrorMessage.tsx
│   │   │   ├── EmptyState.tsx
│   │   │   └── StatusBadge.tsx
│   │   └── workOrders/
│   │       ├── WorkOrderCard.tsx
│   │       └── TimeEntryForm.tsx
│   ├── screens/
│   │   ├── auth/
│   │   │   └── LoginScreen.tsx
│   │   ├── workOrders/
│   │   │   ├── WorkOrderListScreen.tsx
│   │   │   ├── WorkOrderDetailScreen.tsx
│   │   │   └── AddTimeEntryScreen.tsx
│   │   └── profile/
│   │       └── ProfileScreen.tsx
│   ├── navigation/
│   │   ├── AppNavigator.tsx    # Root navigation (auth vs. main)
│   │   ├── MainTabNavigator.tsx # Bottom tab navigation
│   │   └── WorkOrderNavigator.tsx # Work order stack
│   ├── store/
│   │   ├── index.ts            # Redux store setup
│   │   └── authSlice.ts        # Authentication state slice
│   ├── hooks/
│   │   ├── useWorkOrders.ts    # React Query hooks for work orders
│   │   ├── useAuth.ts          # Authentication hook
│   │   └── useNetworkStatus.ts # Offline detection hook
│   ├── utils/
│   │   ├── dateUtils.ts        # Date formatting utilities
│   │   ├── errorUtils.ts       # Error message extraction
│   │   └── constants.ts        # App-wide constants
│   └── theme/
│       ├── colors.ts           # Color palette
│       ├── typography.ts       # Font sizes and weights
│       └── spacing.ts          # Spacing scale
├── android/                    # Android native project
├── ios/                        # iOS native project
├── .env                        # Environment variables (not committed)
├── .env.production             # Production environment
├── babel.config.js
├── tsconfig.json
└── package.json
```

Configuring the API Client

The API client serves as the centralized communication layer between the mobile application and the IBM i backend. By consolidating configuration, authentication, and error handling in one place, the application avoids duplication and ensures consistent behavior across all API interactions.

This approach is especially valuable in enterprise environments, where multiple screens and modules depend on the same backend services.

Centralized HTTP Client with Interceptors

A single HTTP client instance is configured to manage:

- Base URL and request timeouts
- Default headers for all API calls
- Automatic attachment of authentication tokens
- Centralized error handling and retry logic

Key Responsibilities

1. **Request Enrichment:** Every outgoing request automatically includes the JWT access token retrieved from the application state (Redux store).
2. **Token Refresh Handling:** When the server responds with an authentication error (HTTP 401), the client should:
   - Attempts to refresh the access token using a stored refresh token
   - Retries the failed request transparently (if explicitly instructed)
   - Queues concurrent requests during refresh to avoid duplicate calls
3. **Error Normalization:** All API errors are transformed into a consistent structure, simplifying error handling in UI components.

Simplified Example

This design ensures that authentication and retry logic remain completely transparent to the rest of the application.

```
const apiClient = createHttpClient({
  baseURL: BASE_URL,
  timeout: 15000,
});

// Attach token to every request
apiClient.onRequest((config) => {
  const token = getAccessTokenFromStore();
  if (token) {
    config.headers.Authorization = `Bearer ${token}`;
  }
  return config;
});

// Handle 401 and refresh token
apiClient.onResponseError(async (error) => {
  if (error.status === 401) {
    const newToken = await refreshAccessToken();
    retryOriginalRequestWith(newToken);
  }
  return normalizeError(error);
});
```

Type Definitions for API Contracts

Strong typing is critical when integrating with IBM i systems, where strict data structures and field consistency are essential.

TypeScript interfaces define the structure of:

- API responses
- Request payloads
- Domain models (e.g., Work Orders)
- Error formats

Key Design Principles

- **Consistency with backend JSON structure**
- **Early detection of mismatches at compile time**
- **Improved developer productivity through IntelliSense**

Simplified Example

By enforcing these contracts, the frontend remains aligned with backend expectations, reducing runtime errors.

```
interface WorkOrderSummary {
  id: string;
  description: string;
  status: 'OPEN' | 'IN_PROGRESS' | 'COMPLETE';
  dueDate: string;
}

interface PagedResponse<T> {
  data: T[];
  pagination: {
    page: number;
    totalPages: number;
    hasNext: boolean;
  };
}

interface ApiError {
  code: string;
  message: string;
  status?: number;
}
```

Work Order API Module

The API layer is organized into domain-specific modules. For example, all Work Order–related operations are grouped together.

Responsibilities

- Encapsulate HTTP calls
- Provide clean, reusable functions
- Abstract endpoint details from UI components

Common Operations

- Fetch paginated work orders
- Retrieve work order details
- Create new work orders
- Update status
- Add time log entries

Simplified Example

This abstraction ensures that UI components interact with clean, business-oriented functions rather than raw HTTP calls.

```
/**
 * Fetch a paginated list of work orders from the IBM i backend.
 * Used by the Work Order List screen.
 */
export async function fetchWorkOrders(
  params: WorkOrderListParams = {}
): Promise<PagedResponse<WorkOrderSummary>> {
  const response = await apiClient.get<PagedResponse<WorkOrderSummary>>(
    '/workorders',
    { params }
  );
  return response.data;
}

/**
 * Fetch full detail for a single work order.
 * Used by the Work Order Detail screen.
 */
export async function fetchWorkOrderById(id: string): Promise<WorkOrderDetail> {
  const response = await apiClient.get<WorkOrderDetail>(`/workorders/${id}`);
  return response.data;
}

/**
 * Create a new work order.
 */
export async function createWorkOrder(
  data: CreateWorkOrderRequest
): Promise<WorkOrderDetail> {
  const response = await apiClient.post<WorkOrderDetail>('/workorders', data);
  return response.data;
}

/**
 * Update work order status.
 * If status is COMPLETE, the IBM i backend calls the RPG completion program.
 */
export async function updateWorkOrderStatus(
  id: string,
  data: UpdateStatusRequest
): Promise<WorkOrderDetail> {
  const response = await apiClient.patch<WorkOrderDetail>(
    `/workorders/${id}/status`,
    data
  );
  return response.data;
}

/**
 * Add a time log entry to a work order.
 */
export async function addTimeEntry(
  workOrderId: string,
  data:        AddTimeEntryRequest
): Promise<TimeLogEntry> {
  const response = await apiClient.post<TimeLogEntry>(
    `/workorders/${workOrderId}/timelog`,
    data
  );
  return response.data;
}
```

Authentication: Login and Token Management

Authentication in the mobile application is designed to be secure, centralized, and easy to manage across all screens. It combines **Redux** for in-memory state management with **secure device storage** for sensitive credentials such as refresh tokens.

This separation ensures:

- Fast access to authentication state within the app (via Redux)
- Secure persistence of long-lived credentials (via device keychain)

Redux Authentication State

The Redux authentication slice acts as the **single source of truth** for the user's authentication status within the application.

Responsibilities

The authentication state maintains the following core elements:

- **Access Token** A short-lived token used to authenticate API requests.
- **User Information** Basic user profile data required across the application (e.g., name, roles).
- **Authentication Status** A boolean flag indicating whether the user is currently authenticated.

This centralized approach ensures that all components can reliably determine whether the user is logged in and access relevant identity information.

Key Actions

- **loginSuccess** → Stores token and user data
- **setTokens** → Updates access token after refresh
- **logout** → Clears all authentication data

Design Considerations

- **Minimal State Surface** Only essential authentication data is stored in Redux to reduce complexity.
- **Separation of Concerns**
  - Redux handles runtime state
  - Secure storage handles persistent secrets (refresh tokens)
- **Scalability** This structure easily supports future enhancements such as role-based access control or multi-session handling.

Simplified Example

This centralized state enables consistent authentication handling across the application.

```
const authState = {
  accessToken: null,
  user: null,
  isAuthenticated: false,
};

function loginSuccess(state, payload) {
  state.accessToken = payload.accessToken;
  state.user = payload.user;
  state.isAuthenticated = true;
}
```

Login Screen Workflow

The login screen is responsible for:

1. Collecting user credentials
2. Validating input
3. Calling the authentication API
4. Storing tokens securely
5. Updating application state

Key Behaviors

- **Form Validation** ensures required fields are provided
- **Secure Storage** stores the refresh token using device keychain
- **Redux Update** stores access token and user profile
- **Error Feedback** displays meaningful messages on failure

Simplified Example

This flow ensures both security and a smooth user experience.

```
async function handleLogin(credentials) {
  const response = await apiClient.post('/auth/login', credentials);

  storeRefreshTokenSecurely(response.refreshToken);
  dispatch(loginSuccess(response.accessToken, response.user));
}
```

Sample Authentication Flow

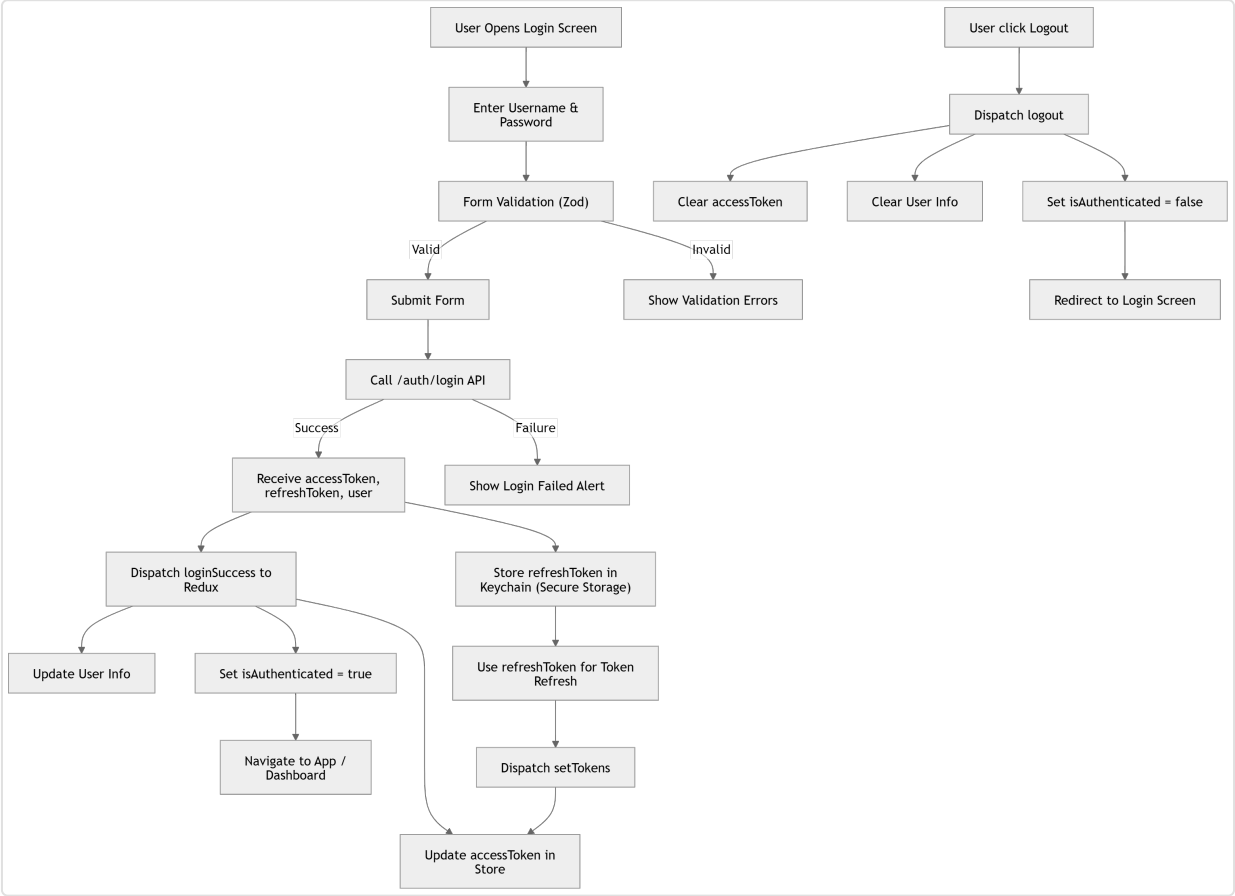


Figure. React Sample Authentication Flow

Core Screens: Work Order Management

The application's primary functionality revolves around viewing and managing work orders.

Data Fetching with React Query

Managing server-side data efficiently is a critical requirement for mobile applications that interact with IBM i systems. Instead of handling API calls, caching, and loading states manually in each screen, the application uses **React Query** to centralize and simplify server state management.

React Query abstracts the complexity of data synchronization between the client and backend, allowing developers to focus on business logic rather than infrastructure concerns.

React Query provides a robust set of features that significantly improve data handling:

- **Data Fetching:** Automatically handles API calls and tracks loading and success states.
- **Caching:** Stores previously fetched data in memory, reducing redundant API calls and improving performance.
- **Background Updates:** Refreshes stale data in the background without interrupting the user experience.
- **Error Handling:** Captures and exposes errors in a consistent way, enabling clean UI-level handling.

Key Advantages

- Eliminates manual state management
- Provides built-in caching
- Supports pagination and infinite scrolling
- Automatically retries failed requests

Simplified Example

```
function useWorkOrders(params) {
  return useQuery(['workOrders', params], () =>
    fetchWorkOrders(params)
  );
}
```

Work Order List Screen

The Work Order List screen is the primary entry point for users to view and interact with work orders retrieved from the IBM i backend. It is designed to efficiently handle large datasets while providing a smooth and responsive user experience.

To achieve this, the screen combines **pagination**, **filtering**, and **infinite scrolling**, all powered by React Query.

Key Features

The screen provides several essential capabilities:

- **Status-Based Filtering:** Users can filter work orders by status (e.g., Open, In Progress), allowing them to quickly focus on relevant tasks.
- **Pull-to-Refresh:** Users can manually refresh the list to retrieve the latest data from the backend.
- **Infinite Scrolling:** Additional records are automatically loaded as the user scrolls, eliminating the need for manual pagination controls.
- **Navigation to Detail Screen:** Each work order item is interactive and navigates to a detailed view for further actions.

Data Handling Approach

Instead of loading all records at once, the screen retrieves data in **pages**. This approach:

- Reduces initial load time
- Minimizes memory usage on mobile devices
- Improves responsiveness on slower networks

React Query's infinite query mechanism manages this process seamlessly.

Core Flow

The overall data flow of the screen follows a structured sequence:

1. **Fetch Paginated Data:** The screen initiates a query to retrieve the first page of work orders from the backend.
2. **Flatten Results for Display:** Since data is received in multiple pages, it is combined into a single list structure for rendering.
3. **Load Additional Pages on Scroll:** When the user scrolls near the end of the list, the next page is automatically requested and appended.
4. **Refresh Data on User Action:** A pull-to-refresh gesture triggers a fresh API call, ensuring the list reflects the latest backend state.

Simplified Example

```
const { data, fetchNextPage, refetch } = useInfiniteWorkOrders();

// Combine all pages into a single array
const workOrders = data?.pages.flatMap(page => page.data) ?? [];

// Trigger next page load when user scrolls
function handleLoadMore() {
  fetchNextPage();
}

// Refresh list manually
function handleRefresh() {
  refetch();
}
```

Work Order Detail Screen

The Work Order Detail screen provides a comprehensive and structured view of a selected work order. It serves as the primary interface for users to review detailed information, track progress, and perform key operational actions.

This screen plays a critical role in bridging the mobile application with backend business processes running on the IBM i system.

Key Capabilities

The screen is designed to support both **information visibility** and **actionable workflows**:

- **Display Detailed Information:** Presents complete work order data, including description, assigned technician, equipment details, location, priority, and due date.
- **Time Log Visualization:** Displays historical time entries associated with the work order, enabling users to track effort and progress over time.
- **Status Management:** Allows users to update the current status of the work order (e.g., marking it as complete).
- **Backend Process Integration:** Certain actions (such as completion) trigger backend business logic on the IBM i system, such as invoking RPG programs or updating operational records.

Status Update Flow

Updating the work order status is a controlled and user-driven process to prevent accidental changes and ensure backend consistency.

Step-by-Step Flow

1. **User Initiates Action:** The user selects an action such as *"Mark Complete"*.
2. **Confirmation Dialog:** A confirmation prompt is displayed to ensure the user intends to proceed with the operation.
3. **API Request Execution:** Upon confirmation, a request is sent to the backend to update the work order status.
4. **UI Synchronization via Cache Invalidation:** After a successful update:
    - The local cache is updated or invalidated
    - The latest data is fetched automatically
    - The UI reflects the updated status immediately

Simplified Example

```
function handleMarkComplete(id) {
  confirmAction(() => {
    updateWorkOrderStatus(id, { status: 'COMPLETE' });
  });
}
```

Offline Support and Network Awareness

Mobile environments often experience unreliable connectivity. The application is designed to handle such scenarios gracefully.

Network Monitoring

A custom hook monitors network status and provides real-time updates.

Responsibilities

- Detect connection availability
- Identify network type
- Enable UI adaptation

Simplified Example

```
function useNetworkStatus() {
  const [isConnected, setIsConnected] = useState(true);

  useEffect(() => {
    subscribeToNetworkChanges(setIsConnected);
  }, []);

  return { isConnected };
}
```

Offline Behavior

When the device is offline:

- Cached data is displayed
- API calls are paused or retried later
- UI indicators inform the user

Offline Indicator Example

```
function OfflineBanner() {
  const { isConnected } = useNetworkStatus();

  if (!isConnected) {
    return <Text>No connection — showing cached data</Text>;
  }
}
```

Best Practices for React Native with IBM i

API and Data

- **All IBM i access goes through REST APIs.** Never attempt to connect to IBM i directly from the device. Keep that logic in the backend layer.

- **Use React Query for all server state.** Avoid storing API response data in Redux — React Query's caching, background refresh, and retry logic handles server state better. Reserve Redux for application state that does not come from the server (authentication, UI preferences).
- **Type everything.** Define TypeScript interfaces for all API response shapes and request bodies. When the IBM i backend changes a field name, TypeScript will catch it at compile time.
- **Use infinite scroll, not pagination controls,** for list screens on mobile. `useInfiniteQuery` with `FlatList`'s `onEndReached` provides a seamless load-more experience.

Authentication and Security

- **Store the refresh token in the device keychain** (`react-native-keychain`), not in `AsyncStorage`. `AsyncStorage` is not encrypted; `Keychain` is backed by iOS Keychain and Android Keystore, which are hardware-backed secure enclaves on modern devices.
- **Implement silent token refresh** in the Axios response interceptor, as shown in Section 3.1. Users should never see a login screen because their access token expired in the background.
- **Validate SSL certificates strictly.** React Native's fetch and Axios do not bypass SSL validation by default — keep it that way. Never disable certificate validation in production builds.
- **Use biometric authentication** where available (`react-native-keychain` supports `BIOMETRY_ANY_OR_DEVICE_PASSCODE`) to protect app access on shared or MDM-managed devices.

Performance

- **Use `useCallback` and `React.memo` on list item components.** `FlatList` renders many items; every unnecessary re-render of list items degrades scroll performance. The `renderWorkOrderCard` example in Section 5.2 demonstrates this.
- **Use `getItemLayout` on `FlatList`** if all items have the same height — it allows FlatList to skip measuring items and scroll instantly to any position.
- **Limit API response size.** Mobile screens show 20–50 items at a time. Request only as many items as fit on the screen, with pagination or infinite scroll for the rest. Large responses waste bandwidth and slow the initial render.
- **Avoid `console.log` in production.** `console.log` in React Native is surprisingly expensive at high frequency. Strip it in production builds using Babel plugins (`babel-plugin-transform-remove-console`).
- **Image Optimization.** Use the fast-image library for better image loading and caching.
- **List Performance.** Use React.memo and useMemo to prevent unnecessary re-renders in list components.
- **Bundle Optimization.** Use code splitting with React.lazy for screens that aren't loaded immediately.

Accessibility

- Add `accessibilityRole`, `accessibilityLabel`, and `accessibilityHint` to all interactive elements. IBM i enterprise apps are subject to corporate accessibility policies and, in many regions, legal accessibility requirements.
- Test with iOS VoiceOver and Android TalkBack during development — not just before release.
- Ensure touch targets are at least 44×44 points in size. Respect device font size settings by using relative font sizes where appropriate.

Testing

Test the API integration layer (`src/api/`) with mocked Axios responses using `axios-mock-adapter`. Test screen components with React Native Testing Library (`@testing-library/react-native`). End-to-end test critical flows (login, view work order, mark complete) with **Detox**, which supports both iOS and Android.

```
## Run unit tests (Jest — configured out of the box by React Native CLI)

npx jest

## Run tests in watch mode during development

npx jest --watch

## Generate test coverage report

npx jest --coverage
```

Summary

React Native enables IBM i modernization teams to deliver high-quality, native mobile experiences on iOS and Android from a single JavaScript codebase. They will use the same language for React web frontends. React Native does not connect to IBM i directly; it connects to a REST API backend that exposes IBM i data and business logic over HTTPS. This separation keeps the IBM i backend independent of the mobile framework and allows both layers to evolve independently.

A well-structured React Native IBM i project leverages TypeScript to ensure type safety across API boundaries, Axios with JWT interceptors and silent token refresh for secure API communication, React Query for efficient server state management and caching, Redux Toolkit for handling authentication state, React Navigation for managing screen flow, and `react-native-keychain` for secure token storage. Together, these libraries create a production-ready foundation that effectively addresses key concerns such as authentication, offline support, pagination, error handling, and network awareness which are essential for any IBM i mobile application.

The IBM i platform's role in this architecture is as a reliable, high-throughput service provider. It's data and programs are made accessible through a REST API layer, and React Native consumes that API exactly as any well-built mobile application consumes a modern backend. The result is a mobile application that brings the power of IBM i to the device in every worker's pocket.

How Do I Use Flutter with IBM i?

Introduction

Delivering a high-quality mobile experience to IBM i users has historically been a challenge. Green-screen applications designed for 5250 terminals are functional, but they offer no practical path to mobile access. Screen-scraping solutions carry significant technical debt. Responsive web applications help, but they stop short of the native platform behaviors like smooth animations, biometric authentication, offline operation, device hardware access etc. that users now take for granted.

Flutter is a cross-platform application framework from Google that takes a different approach to mobile development. Rather than mapping components to platform-native UI elements, Flutter brings its own high-performance rendering engine built on Skia (and more recently Impeller) and draws every pixel itself. This makes Flutter applications pixel-perfect across iOS, Android, web, and desktop from a single codebase, with consistent appearance and behavior regardless of platform. Flutter is written in **Dart**, a strongly typed, compiled language that compiles to native ARM code on mobile and to JavaScript or WebAssembly on web.

For IBM i modernization, Flutter fits into the same service-oriented architecture used by React Native and other mobile frameworks: the IBM i backend exposes data and logic through REST APIs, and the Flutter application communicates with those APIs over HTTPS. Flutter has no special IBM i integration layer and requires none, it uses standard HTTP communication, standard JSON parsing, and standard secure storage, all of which are well-supported in Flutter's package ecosystem.

When Flutter Is the Right Choice

Flutter is ideal when:

- Pixel-perfect consistency across platforms is required

- Desktop support is part of the roadmap
- Teams are open to adopting Dart

For teams already using React, React Native may be easier to adopt. However, for organizations prioritizing visual consistency and multi-platform reach, Flutter is a strong and mature option for IBM i modernization.

Flutter and IBM i: The Architectural Relationship

This is the most important design principle to state clearly: **Flutter does not connect to IBM i directly**. There is no Dart package that speaks directly to IBM Db2 for i, no Flutter plugin that calls RPG programs by name. Flutter is a UI framework. Its job is to render screens, handle user input, and communicate with a backend through standard HTTP.

The IBM i backend which built with Java Spring Boot, Node.js, IBM i's Integrated Web Services (IWS) Server, or any other HTTP-capable technology should be exposing business logic and data as RESTful APIs. Flutter calls those APIs, receives JSON responses, and renders the data. The IBM i platform does not know or care that the client is a Flutter application; it sees only standard HTTPS requests.

This architecture cleanly separates the Flutter UI layer from the IBM i business layer. Both can evolve independently. The Flutter app can be redesigned, the navigation restructured, or the visual design updated without any changes to the IBM i backend. The IBM i backend can be refactored, its libraries reorganized, or its underlying programs updated without breaking the Flutter app, **as long as the API contract remains stable**.

Architecture Overview

Flutter communicates exclusively through HTTPS REST APIs to a backend running on IBM i. The IBM i native layer having Db2 for i and RPG/COBOL programs, is unchanged and unaware of the Flutter frontend. As shown in the diagram below:

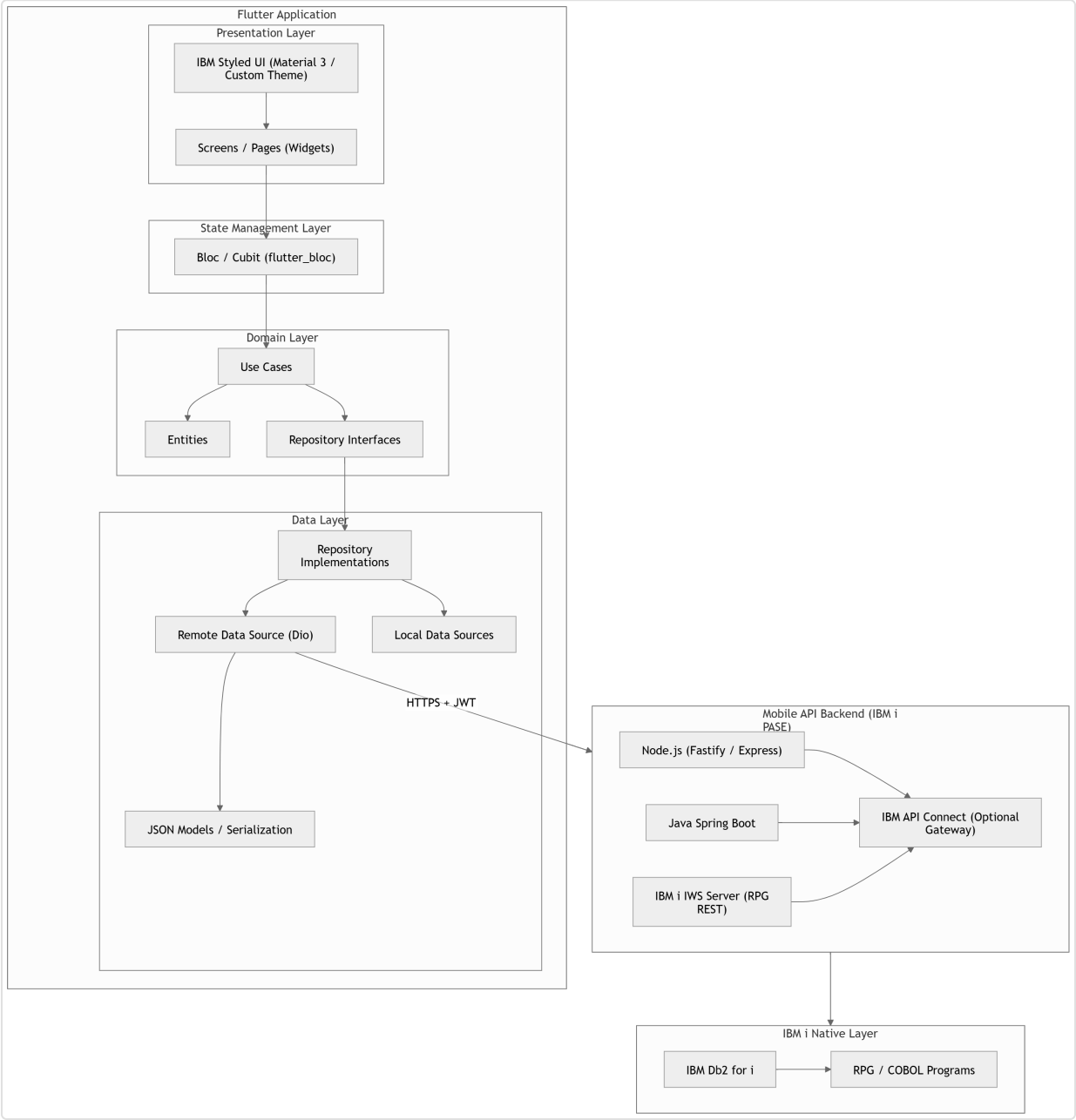


Figure. Flutter Basic App Architecture

Setting Up the IBM i Backend

Before writing a single line of Flutter code, the IBM i must be prepared to serve data in a format Flutter understands: **JSON via RESTful APIs**.

The Middleware Options

To bridge the gap between Flutter and ILE, several middleware technologies are commonly used on IBM i:

- **Java (Spring Boot):** Ideal for organizations with existing Java expertise and complex business rules.
- **Node.js:** Highly popular due to its asynchronous nature and the `itoolkit` for calling RPG programs.
- **IBM Integrated Web Services (IWS):** A native choice for quickly exposing RPG/COBOL procedures as REST services without adding external language runtimes.

Setting Up the Flutter Development Environment

Prerequisites

Flutter development takes place entirely on the developer's workstation — not on IBM i. IBM i is involved only at runtime, through the REST API backend. Flutter applications for iOS are built on macOS; Android builds can be performed on macOS, Windows, or Linux.

```
Developer Workstation Requirements
┌──────────────────────────────────────────────────────────┐
│  Tool                    Purpose                         │
├──────────────────────────────────────────────────────────┤
│  Flutter SDK             Framework, CLI, Dart runtime    │
│  Dart SDK                Bundled with Flutter SDK        │
│  Xcode (macOS only)      iOS simulator + iOS build tools │
│  Android Studio          Android emulator + SDK + AVD    │
│  VS Code / IntelliJ      IDE with Flutter + Dart plugins │
│  CocoaPods (macOS)       iOS dependency manager          │
│  Git                     Version control                 │
└──────────────────────────────────────────────────────────┘
```

Installing Flutter and Creating the Project

```
## 1. Download and install the Flutter SDK from flutter.dev

## Add the Flutter bin directory to your PATH

## 2. Verify the installation and check for missing dependencies

flutter doctor

## 3. Create a new Flutter project

flutter create ibmi_mobile_app --org com.yourcompany --platforms ios,android

## 4. Navigate to the project directory

cd ibmi_mobile_app

## 5. Run on a connected device or emulator

flutter run
```

The `flutter doctor` command checks for all prerequisites — Xcode, Android SDK, connected devices, and licenses — and reports any missing components with instructions to resolve them. Address all `flutter doctor` issues before beginning development.

Adding Dependencies

Flutter dependencies are managed through `pubspec.yaml`. The following packages form the recommended foundation for an IBM i mobile backend integration:

```
## pubspec.yaml

name: ibmi_mobile_app
description: IBM i Mobile Application — UI Modernization

environment:
  sdk: '>=3.0.0 <4.0.0'

dependencies:
  flutter:
    sdk: flutter

  # ── HTTP Client ──────────────────────────────────────────────
  dio: ^5.4.0                      # HTTP client with interceptors + retry

  # ── State Management ─────────────────────────────────────────
  flutter_bloc: ^8.1.4             # BLoC / Cubit state management
  equatable: ^2.0.5                # Value equality for BLoC states/events

  # ── Dependency Injection ─────────────────────────────────────
  get_it: ^7.6.7                   # Service locator

  # ── Secure Storage ───────────────────────────────────────────
  flutter_secure_storage: ^9.0.0   # Encrypted keychain / keystore storage

  # ── Local Persistence ────────────────────────────────────────
  shared_preferences: ^2.2.2       # Simple key-value storage (non-sensitive)
  sqflite: ^2.3.2                  # SQLite — for offline data caching
  path: ^1.9.0                     # Path utilities (used with sqflite)

  # ── JSON Serialization ───────────────────────────────────────
  json_annotation: ^4.8.1          # Annotations for code generation

  # ── Form Validation ──────────────────────────────────────────
  formz: ^0.7.0                    # Validated form input objects

  # ── Navigation ───────────────────────────────────────────────
  go_router: ^13.2.0               # Declarative routing for Flutter

  # ── Connectivity ─────────────────────────────────────────────
  connectivity_plus: ^6.0.3        # Network status monitoring

  # ── Date / Time ──────────────────────────────────────────────
  intl: ^0.19.0                    # Date formatting, localization

  # ── UI Utilities ─────────────────────────────────────────────
  cached_network_image: ^3.3.1     # Image loading with cache
  shimmer: ^3.0.0                  # Loading skeleton placeholders

dev_dependencies:
  flutter_test:
    sdk: flutter
  build_runner: ^2.4.8             # Code generation runner
  json_serializable: ^6.7.1        # JSON model code generation
  bloc_test: ^9.1.6                # BLoC testing utilities
  mocktail: ^1.0.3                 # Mocking for Dart tests
  flutter_lints: ^3.0.0            # Lint rules
```

After editing `pubspec.yaml`:

```
## Install all dependencies

flutter pub get

## Run the code generator (for JSON serialization)

dart run build_runner build --delete-conflicting-outputs
```

Recommended Project Structure

Flutter projects for IBM i enterprise applications benefit from a clean-architecture-inspired structure that separates data access, business logic, and UI into distinct layers:

```
lib/
├── main.dart                        # App entry point
├── app.dart                         # MaterialApp + theme + routing
├── injection_container.dart         # Dependency injection setup (get_it)
│
├── core/
│   ├── constants/
│   │   ├── api_constants.dart       # Base URLs, endpoint paths
│   │   └── app_constants.dart       # App-wide constants
│   ├── errors/
│   │   ├── exceptions.dart          # Custom exception types
│   │   └── failures.dart            # Failure sealed classes
│   ├── network/
│   │   ├── dio_client.dart          # Configured Dio instance
│   │   ├── auth_interceptor.dart    # JWT attach + token refresh interceptor
│   │   └── network_info.dart        # Connectivity check
│   ├── theme/
│   │   ├── app_theme.dart           # Material 3 theme config
│   │   ├── app_colors.dart          # IBM-aligned color palette
│   │   └── app_text_styles.dart     # Typography
│   └── utils/
│       ├── date_utils.dart          # Date formatting helpers
│       └── validators.dart          # Reusable field validators
│
├── features/
│   ├── auth/
│   │   ├── data/
│   │   │   ├── models/
│   │   │   │   └── auth_model.dart  # JSON - Dart model
│   │   │   ├── datasources/
│   │   │   │   └── auth_remote_datasource.dart
│   │   │   └── repositories/
│   │   │       └── auth_repository_impl.dart
│   │   ├── domain/
│   │   │   ├── entities/
│   │   │   │   └── auth_user.dart   # Pure Dart entity
│   │   │   ├── repositories/
│   │   │   │   └── auth_repository.dart  # Abstract interface
│   │   │   └── usecases/
│   │   │       └── login_usecase.dart
│   │   └── presentation/
│   │       ├── bloc/
│   │       │   ├── auth_bloc.dart
│   │       │   ├── auth_event.dart
│   │       │   └── auth_state.dart
│   │       └── pages/
│   │           └── login_page.dart
│   │
│   └── work_orders/
│       ├── data/
│       │   ├── models/
│       │   │   ├── work_order_model.dart
│       │   │   └── time_entry_model.dart
│       │   ├── datasources/
│       │   │   ├── work_order_remote_datasource.dart
│       │   │   └── work_order_local_datasource.dart  # SQLite cache
│       │   └── repositories/
│       │       └── work_order_repository_impl.dart
│       ├── domain/
│       │   ├── entities/
│       │   │   ├── work_order.dart
│       │   │   └── time_entry.dart
│       │   ├── repositories/
```

```
│       │   │   └── work_order_repository.dart
│       │   └── usecases/
│       │       ├── get_work_orders_usecase.dart
│       │       ├── get_work_order_detail_usecase.dart
│       │       ├── update_work_order_status_usecase.dart
│       │       └── add_time_entry_usecase.dart
│       └── presentation/
│           ├── bloc/
│           │   ├── work_order_list_bloc.dart
│           │   ├── work_order_detail_bloc.dart
│           │   └── work_order_detail_event.dart
│           ├── pages/
│           │   ├── work_order_list_page.dart
│           │   ├── work_order_detail_page.dart
│           │   └── add_time_entry_page.dart
│           └── widgets/
│               ├── work_order_card.dart
│               ├── status_badge.dart
│               └── time_entry_tile.dart
│
test/
├── features/
│   ├── auth/
│   └── work_orders/
└── core/
```

Configuring the IBM i API Client

This section describes how the Flutter application is structured to interact with the IBM i backend. It focuses on theming, HTTP communication, data modeling, and architectural patterns that ensure scalability and maintainability in an enterprise environment.

Application Theme — IBM-Aligned Design

Before implementing UI screens, it is important to establish a consistent visual identity. The application adopts IBM's Carbon Design System color palette to align with IBM enterprise tooling.

Key Concept: A centralized color and theme configuration ensures:

- Consistency across all screens
- Easier maintenance and updates
- Alignment with enterprise branding standards

Color Definition: The `AppColors` class acts as a single source of truth for all color tokens used in the application.

Simplified Example:

```
class AppColors {
  static const Color ibmBlue = Color(0xFF0F62FE);
  static const Color gray100 = Color(0xFF161616);
  static const Color white   = Color(0xFFFFFFFF);
  static const Color red60   = Color(0xFFDA1E28);
}
```

Theme Configuration: The `AppTheme` class applies these colors across Material components such as AppBar, buttons, inputs, and cards.

Simplified Example:

```
ThemeData get lightTheme => ThemeData(
  colorScheme: ColorScheme.fromSeed(
    seedColor: AppColors.ibmBlue,
  ),
  appBarTheme: AppBarTheme(
    backgroundColor: AppColors.gray100,
    foregroundColor: AppColors.white,
  ),
);
```

Practical Insight: By centralizing UI styling:

- Developers avoid hardcoding colors across widgets
- UI changes can be implemented quickly
- The application maintains a professional, IBM-aligned look

HTTP Communication Using Dio with JWT Interceptors

Efficient communication with the IBM i backend requires secure and reliable HTTP handling. The application uses the Dio library due to its support for interceptors and advanced request handling.

Key Responsibilities: The HTTP layer is designed to:

1. Attach JWT tokens to every request
2. Automatically refresh expired tokens
3. Retry failed requests transparently
4. Handle authentication failures centrally

Authentication Interceptor: The interceptor acts as middleware between the application and backend.

Core Flow

1. Add JWT token to request headers
2. Detect `401 Unauthorized` responses
3. Trigger token refresh
4. Retry failed requests
5. Logout user if refresh fails

Simplified Example:

```
class AuthInterceptor extends Interceptor {
  @override
  void onRequest(options, handler) async {
    options.headers['Authorization'] = 'Bearer <token>';
    handler.next(options);
  }

  @override
  void onError(err, handler) async {
    if (err.response?.statusCode == 401) {
      // Refresh token logic
      // Retry request
    }
    handler.next(err);
  }
}
```

Key Design Considerations:

- A separate Dio instance is used for token refresh to avoid infinite loops
- Failed requests are queued during refresh
- Secure storage is used for token persistence

Dio Client Configuration

The Dio client is configured once and reused across the application.

Simplified Example:

```
Dio createDioClient() {
  final dio = Dio(BaseOptions(
    baseUrl: 'https://api.example.com',
    connectTimeout: Duration(seconds: 10),
  ));

  dio.interceptors.add(AuthInterceptor());
  return dio;
}
```

Practical Insight: This approach ensures:

- Clean separation of networking concerns
- Reusable and testable HTTP logic
- Enterprise-grade authentication handling

API Constants and Data Models

A well-defined contract between frontend and backend is essential for maintainability.

API Constants

All API endpoints are centralized in a constants class.

Simplified Example:

```
class ApiConstants {
  static const baseUrl   = 'https://ibmi-host/api';
  static const login     = '/auth/login';
  static const workOrders = '/workorders';
}
```

This avoids:

- Hardcoded URLs scattered across the codebase
- Errors during endpoint updates

Data Models vs Domain Entities

The application follows a layered architecture:

Layer	Responsibility
Model	JSON parsing
Entity	Business logic
UI	Presentation

Work Order Model

The model handles JSON serialization from API responses.

Simplified Example:

```
class WorkOrderModel extends WorkOrder {
  factory WorkOrderModel.fromJson(Map<String, dynamic> json) {
    return WorkOrderModel(
      id: json['id'],
      description: json['description'],
    );
  }
}
```

Domain Entity

The entity represents the core business object used across the application.

Simplified Example:

```
class WorkOrder {
  final String id;
  final String status;
  final String dueDate;

  bool get isOverdue =>
      DateTime.parse(dueDate).isBefore(DateTime.now());
}
```

Practical Insight: This separation ensures:

- UI remains independent of backend changes
- Business logic is reusable and testable
- Cleaner architecture aligned with enterprise standards

State Management with BLoC

The BLoC (Business Logic Component) pattern separates UI from business logic, making the application predictable and testable.

Work Order List BLoC

Core Components: A BLoC consists of:

- **Events** → It defines user or system actions.

Simplified Example:

```
class WorkOrderListLoaded extends WorkOrderListEvent {}
class WorkOrderListLoadMore extends WorkOrderListEvent {}
```

- **States** → States represent UI conditions.

Simplified Example:

```
class WorkOrderListLoading extends WorkOrderListState {}

class WorkOrderListSuccess extends WorkOrderListState {
  final List<WorkOrder> workOrders;
}
```

- **Logic** → The BLoC reacts to events and fetches data accordingly.

Simplified Example:

```
on<WorkOrderListLoaded>((event, emit) async {
  emit(WorkOrderListLoading());
  final result = await getWorkOrders();
  emit(WorkOrderListSuccess(result));
});
```

Practical Insight: This pattern:

- Makes state transitions explicit
- Simplifies debugging
- Improves unit test coverage

Work Order Detail BLoC

This BLoC handles:

- Fetching a single work order
- Updating its status

Simplified Example:

```
on<WorkOrderDetailRequested>((event, emit) async {
  emit(WorkOrderDetailLoading());
  final data = await getWorkOrder(event.id);
  emit(WorkOrderDetailLoaded(data));
});
```

Building Core Screens

Work Order List Page

This screen displays a paginated list of work orders with filtering and refresh capabilities.

Key Features

- Infinite scrolling
- Pull-to-refresh
- Status-based filtering
- Error and empty states

UI Rendering with BLoC: The UI reacts to state changes using `BlocBuilder`.

Simplified Example:

```
BlocBuilder<WorkOrderListBloc, WorkOrderListState>(
  builder: (context, state) {
    if (state is WorkOrderListLoading) {
      return CircularProgressIndicator();
    }
    if (state is WorkOrderListSuccess) {
      return ListView.builder(
        itemCount: state.workOrders.length,
        itemBuilder: (_, i) => Text(state.workOrders[i].description),
      );
    }
    return Container();
  },
);
```

Work Order Card Widget

Each work order is displayed using a reusable card component.

Key Responsibilities:

- Display summary information
- Navigate to detail screen
- Highlight overdue items

Simplified Example:

```
Card(
  child: ListTile(
    title: Text(workOrder.description),
    subtitle: Text(workOrder.technicianName),
    onTap: () => navigateToDetail(),
  ),
);
```

Status Badge

A visual indicator showing work order status using color coding.

Simplified Example:

```
Color getStatusColor(String status) {
  switch (status) {
    case 'OPEN': return Colors.blue;
    case 'COMPLETE': return Colors.grey;
    default: return Colors.black;
  }
}
```

Navigation with go_router

`go_router` is the Flutter team's recommended navigation package. It provides declarative, URL-based routing that works consistently across mobile, web, and desktop — all platforms that Flutter targets.

Key Features

- URL-based routing
- Nested routes
- Authentication-based redirection

Routing Example

```
GoRouter _buildRouter(AuthState authState) => GoRouter(
  initialLocation: '/workorders',
  redirect: (context, state) {
    final isLoggedIn = authState is AuthAuthenticated;
    final isOnLogin  = state.matchedLocation == '/login';

    if (!isLoggedIn && !isOnLogin) return '/login';
    if (isLoggedIn  &&  isOnLogin) return '/workorders';
    return null;
  },
  routes: [
    GoRoute(
      path:    '/login',
      builder: (_, __) => const LoginPage(),
    ),
    GoRoute(
      path:    '/workorders',
      builder: (_, __) => const WorkOrderListPage(),
      routes: [
        GoRoute(
          path:    ':id',
          builder: (context, state) => WorkOrderDetailPage(
            workOrderId: state.pathParameters['id']!,
          ),
          routes: [
            GoRoute(
              path:    'timelog',
              builder: (context, state) => AddTimeEntryPage(
                workOrderId: state.pathParameters['id']!,
              ),
            ),
          ],
        ),
      ],
    ),
  ],
);
```

Practical Insight: This approach ensures:

- Consistent navigation across platforms
- Better deep-linking support
- Clear route hierarchy

Offline Support with SQLite Caching

IBM i enterprise mobile applications must handle intermittent network connectivity. A SQLite-backed local cache ensures that users can view the most recently fetched work order list even when the IBM i backend is unreachable.

Caching Strategy

1. Fetch data from API when online
2. Store results in SQLite

3. Retrieve cached data when offline

Local Data Storage

Simplified Example:

```
Future<void> cacheWorkOrders(List<WorkOrderModel> workOrders) async {
  for (final wo in workOrders) {
    db.insert('work_orders', {
      'id': wo.id,
      'data': jsonEncode(wo.toJson()),
    });
  }
}
```

Reading Cached Data

```
Future<List<WorkOrderModel>> getCachedWorkOrders() async {
  final rows = await db.query('work_orders');
  return rows.map((r) => WorkOrderModel.fromJson(jsonDecode(r['data']))).toList();
}
```

Repository Fallback Logic: The repository determines whether to use API or cache.

Simplified Example:

```
if (hasConnection) {
  final data = await fetchFromApi();
  cache(data);
  return data;
} else {
  return getCachedData();
}
```

Practical Insight: This design ensures:

- Continuous usability during network outages
- Improved performance for repeat access
- Better user experience in field environments

Best Practices for Flutter with IBM i

Architecture and Code Quality

- **Follow clean architecture layering.** Keep the domain layer (entities, use cases, repository interfaces) free from Flutter and Dio dependencies. This makes the business logic independently testable without needing to mock the network or the UI.
- **Use sealed classes for failures.** Dart 3 sealed classes make API error handling exhaustive — the compiler ensures that every failure type is handled. Use `Either<Failure, T>` from the `dartz` or `fpdart` package as the repository return type.
- **Run** `dart run build_runner build` after any change to JSON-serializable models. Generated files (`*.g.dart`) must be regenerated when the model class changes, or the JSON parsing will be out of sync.
- **Use** `const` **constructors everywhere possible.** Dart can optimize widget rebuilds more aggressively when widgets are declared as `const`. This improves scroll performance on list screens, which are common in IBM i mobile applications.

Security

- **Store tokens in `flutter_secure_storage`.** This package uses iOS Keychain and Android Keystore — hardware-backed secure storage. Never store access tokens or refresh tokens in `SharedPreferences`, which is unencrypted.
- **Pin the IBM i API certificate** in production builds for applications that carry sensitive business data. Certificate pinning prevents man-in-the-middle attacks on corporate Wi-Fi networks. Dio supports this through `SecurityContext`.
- **Use `--dart-define` for environment-specific configuration.** Never hardcode the IBM i backend URL or other environment-specific values in the source code. Use compile-time constants: `flutter run --dart-define=API_BASE_URL=https://ibmi-host:3000/api/v2`.
- **Obfuscate release builds.** Flutter supports code obfuscation for release builds: `flutter build apk --obfuscate --split-debug-info=./debug-info`. This makes reverse engineering of the API client harder.

Performance

- **Use `const` widgets aggressively.** The Flutter framework skips rebuilding `const` widgets during state changes, which reduces the work done during BLoC state emissions.
- **Use `ListView.builder` or `CustomScrollView` with slivers** for all long lists — they render only the visible items. `Column` with a `children` list renders all items at once, which causes performance issues with large IBM i data sets.
- **Avoid heavy computation on the main isolate.** JSON parsing of large IBM i API responses can cause frame drops. For responses with hundreds of records, use `compute()` to run `fromJson` in a background isolate.
- **Monitor performance with Flutter DevTools.** The Flutter DevTools timeline view shows frame render times, widget rebuild counts, and memory usage — use them during development to catch performance issues early.
- **Image Optimization.** Use the `cached_network_image` package for better image loading and caching.
- **List Performance.** Use `ListView.builder` and `AutomaticKeepAliveClientMixin` for optimal list performance.
- **Bundle Optimization.** Use code splitting with deferred components for screens that aren't loaded immediately.

Testing

```
## Run all tests

flutter test

## Run tests with coverage

flutter test --coverage

## View coverage report (requires lcov)

genhtml coverage/lcov.info -o coverage/html
open coverage/html/index.html
```

- **Unit test use cases and repositories** with mocked data sources using `mocktail`.
- **BLoC test** state transitions with `bloc_test`'s `blocTest` helper, which provides a clean DSL for specifying input events and asserting output states.
- **Widget test** key screen components — work order cards, status badges, form validation — using `flutter_test`'s widget testing utilities.
- **Integration test** critical user flows (login → view list → view detail → mark complete) using Flutter's `integration_test` package, which runs on a real device or emulator.

Summary

Flutter provides a compelling path for IBM i UI modernization when pixel-perfect cross-platform consistency, strong static typing, and the ability to target iOS, Android, web, and desktop from a single Dart codebase are priorities. Like

all mobile frameworks in IBM i modernization, Flutter connects to IBM i exclusively through REST APIs served by a backend on IBM i.It make sure that the IBM i native layer is unchanged and unaware of the Flutter frontend.

A well-architected Flutter IBM i application uses clean architecture layering (presentation, domain, data) with BLoC for state management, Dio with JWT interceptors for secure API communication, `flutter_secure_storage` for token security, SQLite for offline caching, and `go_router` for navigation. Dart's strong typing, combined with `json_serializable` for compile-time JSON model generation, catches integration errors between the Flutter app and the IBM i API at build time — not at runtime on a user's device.

The architecture described in this topic makes the Flutter layer completely independent of the IBM i backend technology choice. Whether the IBM i backend is Spring Boot, Node.js, or the IBM i IWS Server, the Flutter application communicates through the same HTTP/JSON interface, with the same authentication flow, and with the same offline behavior. The IBM i platform contributes its greatest strengths such as reliability, transactional integrity, and decades of accumulated business logic, while Flutter contributes pixel-perfect, performant, and maintainable mobile interfaces that meet the expectations of the modern workforce.

How do I use Angular with IBM i?

Introduction

IBM i systems have long been the backbone of enterprise business applications, offering unmatched reliability, transactional integrity, and performance. However, traditional user interfaces—such as green screens or legacy web layers—often fall short of modern user expectations.

Angular, a powerful front-end framework maintained by Google, provides a robust platform for building dynamic, responsive, and enterprise-grade web applications. It is particularly well-suited for UI modernization initiatives where IBM i remains the system of record.

In an **IBM i–first architecture**, Angular serves as the **presentation layer**, consuming APIs exposed from IBM i through middleware such as Java (Spring Boot) or Node.js. This enables organizations to modernize their UI without rewriting core business logic.

Why Angular for IBM i Modernization?

- Angular is a **complete, opinionated front-end framework** developed and maintained by Google
- Unlike lightweight JS libraries, Angular provides a **full application platform out of the box:**
  - Component architecture
  - Dependency injection
  - Routing
  - HTTP client
  - Reactive forms
  - Comprehensive CLI (code generation, testing, deployment)
- Fewer architectural decisions and fewer third-party libraries to evaluate and maintain
- Ideal for teams building **large-scale enterprise applications**

A Framework Built for Enterprise Scale

TypeScript-First Design

- Angular is **built entirely on TypeScript** — distinguishing it from JavaScript-based frameworks
- Benefits of strong static typing:
  - Catches errors **before runtime**
  - Improved IDE support — refactoring, autocompletion, inline documentation
  - Large codebases are easier to **navigate and maintain over time**
- Especially valuable for IBM i projects — typically **long-lived applications** maintained by evolving teams

Component-Based Architecture

- Enforces a clear **separation of concerns** between:
  - UI layer
  - Business logic layer
  - Data access layer
- Components are **self-contained, testable units** with well-defined inputs and outputs
- Architecture **scales naturally** as the application expands across:
  - Additional IBM i workflows
  - New user roles
  - More screens

Built-In Scalability Features

- **Dependency injection** — makes services easy to share across components
- **Module system** — provides natural boundaries for organizing large codebases

Stability and Long-Term Support

- Angular has been in **production use for over a decade**
- Google maintains strong commitment to:
  - Backward compatibility
  - Predictable release cycles
  - Supported upgrade paths with each major version
- **Long-Term Support (LTS)** releases extend coverage further
- Key advantage for enterprise environments:
  - Upgrades can be **carefully planned and tested**
  - More stable than frameworks with volatile release histories

Integration with IBM i

How the Integration Works

- Angular connects to IBM i via a **service-oriented model** — the same model used by all modern front-end frameworks
- IBM i exposes business logic and data through **REST or GraphQL APIs**
- Angular's built-in `HttpClient` module consumes those APIs over **HTTPS**
- **No IBM i-specific client library required**

`HttpClient` Capabilities Relevant to IBM i

Feature	Benefit
Typed responses	Ensures API data matches expected structure
Observable-based reactive patterns	Handles async IBM i data streams cleanly
Request & response interceptors	Centralized authentication handling
Error management	Structured handling of IBM i API errors

Architectural Blueprint for Angular + IBM i

Angular does not run on the IBM i operating system; it runs in the end-user's web browser. Therefore, the architecture is fundamentally **decoupled**. The IBM i serves as the "API Provider," and the Angular application serves as the "Consumer."

Choosing the API Middleware

To allow Angular to access IBM i data, you must choose a middleware technology to expose your business logic as RESTful endpoints.

- **Java (Spring Boot):** Preferred for high-volume enterprise environments where Java is already the standard for backend services.
- **Node.js:** The most natural fit for Angular teams. Since both use TypeScript/JavaScript, developers can work seamlessly across the stack.
- **Integrated Web Services (IWS):** A native IBM i feature that allows RPG procedures to be exposed as APIs without installing additional languages.

High-Level Architecture

Diagram shows that Angular runs in the browser and communicates with the IBM i back-end exclusively through HTTPS REST APIs. The IBM i native layer is decoupled from the Angular front end and unaware of it.

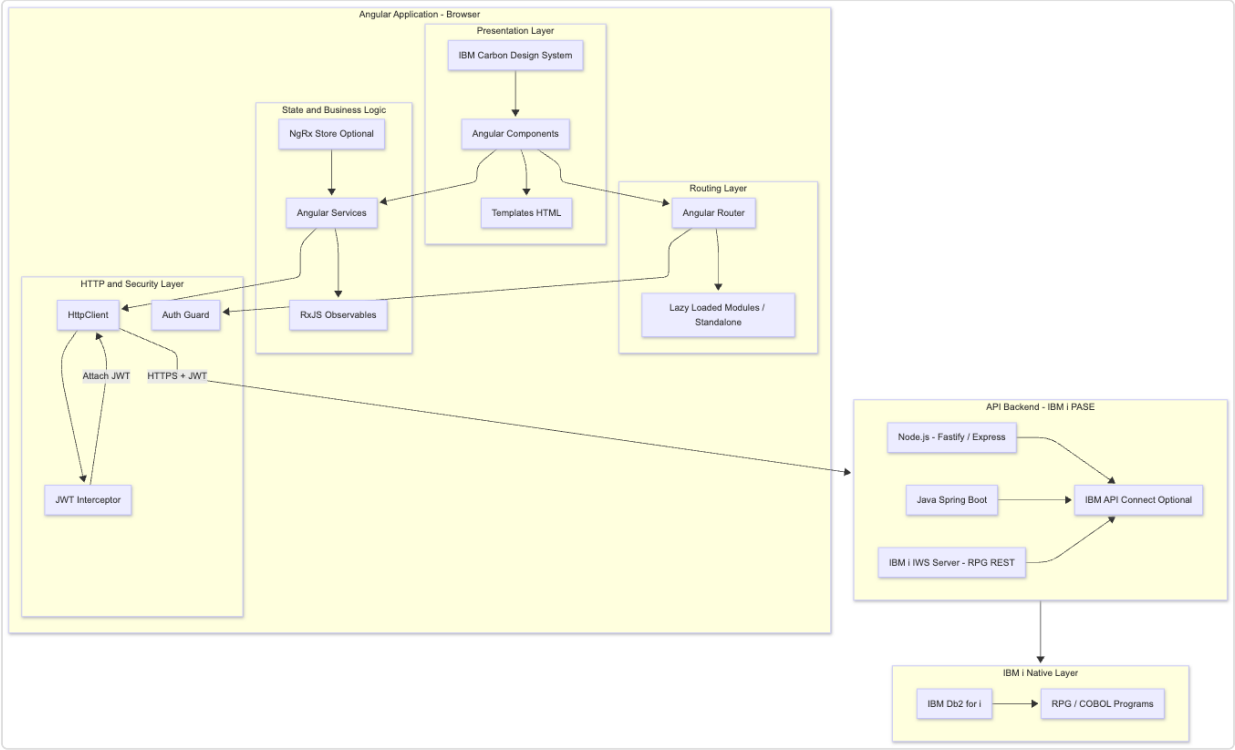


Figure. Angular App Basic Architecture

Deployment Model

Angular applications are built into a set of static files such as HTML, CSS, and JavaScript bundles, by the Angular CLI (`ng build`). Because Angular is entirely static after the build, the web server only needs to serve files — it does not need to understand Angular, execute JavaScript, or maintain sessions. All dynamic behavior happens in the browser through REST API calls to the IBM i back end.

To host your Angular app on an IBM i partition:

1. **Build the app:** Run `ng build --configuration production`.
2. **Upload to IFS:** Transfer the contents of the `dist/` folder to a directory in the Integrated File System (IFS), such as `/www/modernapp/htdocs`.
3. **Configure Apache:** Update the HTTP configuration file to point to this directory and handle "Single Page Application" routing (redirecting 404s back to `index.html`).

Setting Up the Angular Project

Installing the Angular CLI

```
## Install the Angular CLI globally

npm install -g @angular/cli

## Verify the installation

ng version

## Expected output shows Angular CLI version, Node.js version,

## and the versions of key Angular packages
```

Creating the Project

```
## Create a new Angular project with routing and SCSS styling

ng new ibmi-portal \
  --routing=true \
  --style=scss \
  --strict=true \
  --standalone=false   # Use NgModule architecture (recommended for large enterprise apps)
                       # Change to --standalone=true for the newer standalone API

cd ibmi-portal

## Start the development server

ng serve --open
```

The `--strict=true` flag enables TypeScript strict mode and Angular-specific strict template checking. Strict mode catches a wide range of integration errors — particularly mismatched API response types — at compile time rather than at runtime, and is strongly recommended for IBM i enterprise projects.

Installing Key Dependencies

```
## IBM Carbon Design System for Angular

npm install carbon-components-angular carbon-components

## HTTP and reactive utilities (included with Angular, no separate install needed)

## @angular/common/http, rxjs — both included in the Angular package

## Authentication: JWT decoding

npm install @auth0/angular-jwt

## Form utilities

npm install @angular/forms   # Included with Angular

## Date utilities

npm install date-fns

## Environment-based configuration

npm install @ngx-translate/core @ngx-translate/http-loader  # i18n (optional)

## State management (optional — for large applications)

npm install @ngrx/store @ngrx/effects @ngrx/entity @ngrx/store-devtools

## Development utilities

npm install --save-dev @types/node
```

Recommended Project Structure

Angular's modular architecture works best when features are organized into feature modules (or, in newer Angular, standalone component groups), each containing its own components, services, and routing:

```
src/
├── app/
│   ├── app.module.ts                  # Root module
│   ├── app.component.ts               # Root component (shell + nav)
│   ├── app-routing.module.ts          # Root routes + lazy-load config
│   │
│   ├── core/                          # Singleton services, guards, interceptors
│   │   ├── core.module.ts             # Imported once in AppModule
│   │   ├── auth/
│   │   │   ├── auth.service.ts        # Login, logout, token management
│   │   │   ├── auth.guard.ts          # Route protection
│   │   │   └── auth.interceptor.ts    # Attach JWT to all requests
│   │   ├── http/
│   │   │   ├── api.service.ts         # Base HTTP wrapper
│   │   │   └── error.interceptor.ts   # Global HTTP error handling
│   │   └── models/
│   │       └── api-response.model.ts  # Shared API response types
│   │
│   ├── shared/                        # Reusable components, pipes, directives
│   │   ├── shared.module.ts
│   │   ├── components/
│   │   │   ├── status-badge/
│   │   │   ├── page-header/
│   │   │   ├── loading-spinner/
│   │   │   ├── empty-state/
│   │   │   └── error-message/
│   │   └── pipes/
│   │       ├── ibmi-date.pipe.ts      # Format IBM i date strings
│   │       └── duration.pipe.ts       # Format minutes as "2h 30m"
│   │
│   └── features/
│       ├── work-orders/               # Work order feature module
│       │   ├── work-orders.module.ts
│       │   ├── work-orders-routing.module.ts
│       │   ├── services/
│       │   │   └── work-order.service.ts
│       │   ├── models/
│       │   │   ├── work-order.model.ts
│       │   │   └── time-entry.model.ts
│       │   └── components/
│       │       ├── work-order-list/
│       │       ├── work-order-detail/
│       │       └── add-time-entry/
│       │
│       ├── technicians/
│       └── equipment/
│
├── environments/
│   ├── environment.ts                 # Development config
│   └── environment.production.ts      # Production config
│
├── styles/
│   ├── _variables.scss                # CSS custom properties + IBM colors
│   └── _carbon-overrides.scss         # Carbon component theme overrides
│
└── assets/
    └── i18n/                          # Translation files (if i18n used)
```

Environment Configuration and IBM i API Base URL

Environment configuration in Angular plays a critical role in separating development and production behavior without changing application code. This is especially important for IBM i integrations, where API endpoints, authentication servers, and deployment environments differ significantly between local development and production systems.

Environment Files Overview

Angular uses environment-specific configuration files to define runtime variables such as API endpoints, authentication URLs, and application metadata. These files are automatically replaced during the build process based on the target environment.

A simplified version of the environment configuration is shown below:

```
export const environment = {
  production: false,
  apiBaseUrl: 'http://localhost:3000/api/v2',
  authBaseUrl: 'http://localhost:3000/auth',
  tokenKey: 'ibmi_access_token',
};
```

Key Configuration Properties

- `production`: Indicates whether the application is running in production mode.
- `apiBaseUrl`: Base URL for all IBM i backend API calls.
- `authBaseUrl`: Dedicated endpoint for authentication-related operations.
- `tokenKey` / `refreshKey`: Keys used to store JWT tokens in browser storage.

Development vs Production

- In **development**, APIs typically point to a local or proxy server (`localhost`) that forwards requests to the IBM i backend.
- In **production**, endpoints point directly to the IBM i host (often running in PASE or behind a reverse proxy).

This separation ensures:

- No hardcoding of environment-specific values
- Safer deployments
- Easier testing and debugging

Proxy Configuration for Local Development

When developing locally, browser security policies (CORS) can block API calls to external servers such as IBM i. Angular solves this using a proxy configuration that transparently forwards API requests.

A minimal proxy configuration looks like this:

```
{
  "/api": {
    "target": "http://ibmi-dev-host:3000",
    "secure": false
  }
}
```

How It Works

- Requests made to `/api` from the Angular app are redirected to the IBM i backend.
- The browser treats the request as same-origin, avoiding CORS issues.

- No changes are required in the application code.

To enable the proxy, Angular CLI is configured as follows:

```
"serve": {
  "options": {
    "proxyConfig": "proxy.conf.json"
  }
}
```

This setup is essential for a smooth developer experience when working with IBM i APIs.

TypeScript Models for IBM i API Responses

Strong typing is a key advantage of Angular and TypeScript. By defining API response structures explicitly, applications gain compile-time validation, improved maintainability, and better developer productivity.

Generic API Response Models

IBM i APIs commonly return paginated responses and structured error objects. These can be modeled using reusable TypeScript interfaces.

A simplified structure is shown below:

```
export interface PagedResponse<T> {
  data: T[];
  pagination: {
    page: number;
    totalItems: number;
  };
}
```

Benefits

- Ensures consistent handling of paginated data across the application
- Enables type-safe API consumption
- Reduces runtime errors caused by incorrect assumptions about response shape

Error Handling Model

APIs typically return structured error responses. Standardizing this structure allows centralized error handling:

```
export interface ApiError {
  code: string;
  message: string;
}
```

This approach ensures that all services and components handle errors uniformly.

Work Order Domain Models

The Work Order module demonstrates how API contracts are translated into TypeScript models.

Core Concepts

Two levels of models are typically defined:

1. **Summary Model** – Lightweight representation used in lists
2. **Detail Model** – Full representation used in detailed views

Simplified Example

```
export interface WorkOrderSummary {
  id: string;
  description: string;
  status: 'OPEN' | 'IN_PROGRESS';
  dueDate: string;
}
```

Key Design Considerations

- **Enums via union types**: Restrict values such as status and priority
- **ISO date strings**: Maintain backend compatibility
- **Separation of concerns**: Keep list and detail payloads distinct

Extended Models

The detailed model typically includes:

- Technician information
- Equipment details
- Time logs and notes
- Audit fields (created/updated timestamps)

This layered modeling approach improves performance (smaller payloads for lists) while still supporting rich detail views.

Authentication: JWT Service and HTTP Interceptors

Authentication in Angular applications interacting with IBM i backends is typically implemented using JWT (JSON Web Tokens). The architecture combines a centralized service with HTTP interceptors to ensure secure and seamless communication.

Authentication Service

The authentication service is responsible for managing the entire authentication lifecycle, including login, logout, token storage, and session restoration.

A simplified version of the service structure is shown below:

```
@Injectable({ providedIn: 'root' })
export class AuthService {
  private currentUser$ = new BehaviorSubject<any>(null);

  login(username: string, password: string) {
    return this.http.post('/auth/login', { username, password });
  }

  logout() {
    localStorage.clear();
  }
}
```

Responsibilities

- **Login**: Authenticate users against the IBM i backend
- **Token Storage**: Persist access and refresh tokens securely
- **Session Restoration**: Reload user session on app startup
- **User State Management**: Provide reactive access to user data via Observables

Key Design Patterns

- **BehaviorSubject**: Maintains the current authentication state
- **Observable streams**: Allow UI components to react to login/logout events
- **JWT decoding**: Extract user information from tokens

JWT Authentication Interceptor

The authentication interceptor ensures that every HTTP request includes the required JWT token and handles token expiration automatically.

Core Responsibilities

1. Attach JWT token to outgoing requests
2. Detect unauthorized (401) responses
3. Refresh tokens silently
4. Retry failed requests after refresh

A reduced version of the interceptor logic:

```
intercept(req, next) {
  const authReq = req.clone({
    setHeaders: { Authorization: `Bearer token` }
  });
  return next.handle(authReq);
}
```

Token Refresh Strategy

- When a `401 Unauthorized` response is received:
  - A refresh token request is triggered
  - New tokens are stored
  - Pending requests are retried automatically

Benefits

- Transparent authentication handling
- Improved user experience (no forced re-login)
- Centralized security logic

Global Error Interceptor

To ensure consistent error handling across the application, a dedicated interceptor transforms all HTTP errors into a standard format.

Simplified Flow

```
catchError((error) => {
  return throwError(() => ({
    code: 'ERROR',
    message: error.message
  }));
});
```

Key Features

- Normalizes backend error responses

- Handles network failures (e.g., IBM i unreachable)
- Provides a consistent error contract for services and UI

Typical Error Scenarios

- Backend validation errors
- Authentication failures
- Network connectivity issues

Route Guard for Protected Navigation

Route guards prevent unauthorized users from accessing protected routes.

Simplified Example

```
canActivate(): boolean {
  if (this.authService.isAuthenticated) {
    return true;
  }
  this.router.navigate(['/login']);
  return false;
}
```

Responsibilities

- Verify authentication before route activation
- Redirect unauthenticated users to the login page
- Preserve the intended navigation URL for post-login redirection

Registering Interceptors in the Core Module

Angular allows multiple HTTP interceptors to be chained together. These are typically registered in a centralized `CoreModule`.

```
providers: [
  { provide: HTTP_INTERCEPTORS, useClass: AuthInterceptor, multi: true },
  { provide: HTTP_INTERCEPTORS, useClass: ErrorInterceptor, multi: true }
]
```

Important Considerations

- **Order matters**: Authentication should run before error handling
- **Singleton module**: CoreModule must be imported only once (usually in `AppModule`)
- **Separation of concerns**: Each interceptor handles a specific responsibility

Work Order Service — Calling IBM i APIs with RxJS

Overview

In Angular applications, services act as the primary layer for communicating with backend systems such as IBM i. The `WorkOrderService` encapsulates all HTTP interactions and exposes them as **RxJS Observables**, enabling a reactive and composable data flow.

This approach allows components to:

- React to asynchronous data streams
- Chain API calls
- Handle errors consistently

- Apply transformations declaratively

Service Structure

The service is defined as an injectable singleton and uses Angular's `HttpClient` to communicate with REST APIs exposed from IBM i (typically backed by Db2 for i and RPG programs).

A simplified version of the service structure is shown below:

```
@Injectable({ providedIn: 'root' })
export class WorkOrderService {

  private readonly baseUrl = '/api/workorders';

  constructor(private http: HttpClient) {}

  getWorkOrders(params): Observable<any> {
    return this.http.get(this.baseUrl, { params });
  }

  getWorkOrderById(id: string): Observable<any> {
    return this.http.get(`${this.baseUrl}/${id}`);
  }

  updateStatus(id: string, data): Observable<any> {
    return this.http.patch(`${this.baseUrl}/${id}/status`, data);
  }
}
```

Key Responsibilities

Retrieving Work Orders

The service constructs query parameters dynamically and sends them to the backend. The IBM i system processes these parameters and returns paginated JSON data.

- Supports filtering (status, technician)
- Supports pagination (page, pageSize)
- Returns a structured response with metadata

CRUD Operations

The service exposes methods for:

- Creating work orders (`POST`)
- Retrieving details (`GET`)
- Updating status (`PATCH`)
- Adding time entries (`POST`)

Each method returns an Observable, allowing the caller to decide when and how to subscribe.

RxJS-Based Chaining

A common enterprise pattern is chaining API calls. For example, updating a work order and then reloading it:

```
completeAndReload(id: string): Observable<any> {
  return this.updateStatus(id, { status: 'COMPLETE' }).pipe(
    // switchMap(() => this.getWorkOrderById(id))
  );
}
```

This demonstrates how RxJS enables sequential execution without deeply nested callbacks.

Building Core Components

Work Order List Component

Purpose

This component displays a paginated, filterable list of work orders. It demonstrates a **reactive UI pattern**, where changes in user input automatically trigger backend calls.

Reactive Data Flow

Instead of directly calling the service on every user action, the component uses a `BehaviorSubject` to represent the current query state.

```
private loadParams$ = new BehaviorSubject({ page: 0, pageSize: 20 });

ngOnInit(): void {
  this.loadParams$
    .pipe(
      debounceTime(150),
      switchMap(params => this.service.getWorkOrders(params))
    )
    .subscribe(response => this.handleResponse(response));
}
```

Key Concepts

Debouncing and Change Detection:

- `debounceTime` prevents excessive API calls during rapid user input
- `distinctUntilChanged` ensures calls are made only when parameters actually change

State Management: The component maintains a local state object:

- `workOrders` → data to display
- `pagination` → metadata from backend
- `loading` → UI feedback
- `error` → error handling
- `activeStatus` → current filter

This centralized state simplifies UI rendering.

User Interaction

User actions such as filtering or pagination do not directly call the API. Instead, they update the parameter stream:

```
onStatusFilterChange(status: string | null): void {
  this.loadParams$.next({ status, page: 0 });
}
```

This keeps the logic declarative and consistent.

UI Structure

The HTML template is structured into logical UI states:

- **Loading state** → spinner
- **Error state** → notification
- **Empty state** → no results message

- **Data state** → table with pagination

The table leverages **IBM Carbon components**, ensuring consistency with enterprise UI standards.

Work Order Detail Component

Purpose

This component displays detailed information for a single work order and allows status updates.

Route-Driven Data Loading

The component reacts to route changes using Angular's `ActivatedRoute`:

```
this.route.paramMap
  .pipe(
    switchMap(params => this.service.getWorkOrderById(params.get('id')))
  )
  .subscribe(data => this.workOrder = data);
```

This ensures:

- Automatic reload when navigating between records
- Clean separation of routing and data logic

Updating Work Order Status

When marking a work order as complete:

```
onMarkComplete(): void {
  this.service.updateStatus(this.workOrder.id, { status: 'COMPLETE' })
    .subscribe(updated => this.workOrder = updated);
}
```

Key Features

- Uses **OnPush change detection** for performance
- Displays success notifications using Carbon UI services
- Handles loading and error states explicitly
- Supports confirmation workflows before critical actions

Reactive Form: Add Time Entry

Overview

Angular Reactive Forms provide a structured and testable way to handle form input. This is particularly useful in enterprise IBM i applications where validation and data integrity are critical.

Form Initialization

The form is created using `FormBuilder`:

```
this.form = this.fb.group({
  minutesWorked: [null, [Validators.required]],
  notes: ['']
});
```

Custom Validation

A custom validator ensures that the entered time is valid:

```
function minutesValidator(control: AbstractControl) {
  const value = Number(control.value);
  return value > 0 && value <= 1440 ? null : { invalid: true };
}
```

Form Submission

On submission:

```
onSubmit(): void {
  if (this.form.invalid) return;

  this.service.addTimeEntry(this.workOrderId, this.form.value)
    .subscribe(() => this.router.navigate(['/work-orders', this.workOrderId]));
}
```

Key Features

- Strong validation (required fields, limits)
- Clear error messaging
- Controlled submission flow
- Seamless navigation after success

UI Behavior

The template includes:

- Inline validation messages
- Character count for notes
- Disabled submit button during processing
- Loading indicators for better UX

Routing with Lazy Loading

Overview

Angular routing is designed to support **modular and scalable applications**. Lazy loading ensures that only the required modules are loaded when needed, improving performance.

Root Routing Configuration

The main routing module defines application-level routes:

```
const routes: Routes = [
  { path: '', redirectTo: 'work-orders', pathMatch: 'full' },
  {
    path: 'work-orders',
    loadChildren: () => import('./work-orders.module')
      .then(m => m.WorkOrdersModule)
  }
];
```

Key Concepts

Lazy Loading

Feature modules are loaded only when their route is accessed:

- Reduces initial bundle size

- Improves application startup time

Route Guards

Protected routes use authentication guards:

```
{
  path: 'work-orders',
  canActivate: [AuthGuard],
  loadChildren: ...
}
```

This ensures that only authenticated users can access IBM i data.

Feature-Level Routing

Each feature module defines its own routes:

```
const routes: Routes = [
  { path: '', component: WorkOrderListComponent },
  { path: ':id', component: WorkOrderDetailComponent }
];
```

This modular approach:

- Improves maintainability
- Keeps routing concerns localized
- Supports independent feature development

Fallback Handling

A wildcard route ensures graceful handling of unknown URLs:

```
{ path: '**', redirectTo: 'work-orders' }
```

Shared Utility Pipes for IBM i Data

Angular pipes transform data in templates. Two pipes are particularly useful for IBM i applications: one for formatting IBM i date strings and one for formatting minutes as a human-readable duration.

```
// src/app/shared/pipes/ibmi-date.pipe.ts

import { Pipe, PipeTransform } from '@angular/core';
import { format, parseISO, isValid } from 'date-fns';

/**
 * Formats ISO 8601 date strings returned by IBM i REST APIs
 * into human-readable dates.
 *
 * Usage: {{ workOrder.dueDate | ibmiDate }}           → "Nov 15, 2024"
 *        {{ workOrder.createdAt | ibmiDate:'long' }}  → "November 15, 2024 at 2:30 PM"
 */
@Pipe({ name: 'ibmiDate' })
export class IbmiDatePipe implements PipeTransform {
  transform(
    value:  string | null | undefined,
    style:  'short' | 'long' | 'time' = 'short',
  ): string {
    if (!value) return '—';
    const date = parseISO(value);
    if (!isValid(date)) return value;

    switch (style) {
      case 'long':  return format(date, 'MMMM d, yyyy \'at\' h:mm a');
      case 'time':  return format(date, 'h:mm a');
      default:      return format(date, 'MMM d, yyyy');
    }
  }
}
```

```
// src/app/shared/pipes/duration.pipe.ts

import { Pipe, PipeTransform } from '@angular/core';

/**
 * Formats a number of minutes into a human-readable duration.
 *
 * Usage: {{ 90 | duration }}   → "1h 30m"
 *        {{ 45 | duration }}   → "45m"
 *        {{ 480 | duration }}  → "8h"
 */
@Pipe({ name: 'duration' })
export class DurationPipe implements PipeTransform {
  transform(minutes: number | null | undefined): string {
    if (minutes == null || isNaN(minutes)) return '—';
    const h = Math.floor(minutes / 60);
    const m = minutes % 60;
    if (h === 0) return `${m}m`;
    if (m === 0) return `${h}h`;
    return `${h}h ${m}m`;
  }
}
```

Best Practices for Angular with IBM i

TypeScript and API Integration

- **Define TypeScript interfaces for every IBM i API endpoint's request and response shape.** When the IBM i backend team changes a field name, the TypeScript compiler surfaces every affected location immediately.

- **Use `strict` mode in `tsconfig.json`.** Angular's `--strict=true` flag enables strict null checks, strict property initialization, and strict template type checking. These catch the vast majority of integration bugs at compile time.
- **Never use `any` for HTTP response types.** Always type `HttpClient` calls: `this.http.get<WorkOrderDetail>('/workorders/123')`. Using `any` defeats the entire purpose of TypeScript in an IBM i integration context.
- **Keep API call logic in services, not components.** Components should not call `HttpClient` directly. Services own the IBM i API integration; components subscribe to service Observables and render data.

RxJS and Reactive Patterns

- **Always unsubscribe in `ngOnDestroy`** using a `Subject` and `takeUntil`, or use the `async` pipe in templates (which handles unsubscription automatically). Memory leaks from unclosed subscriptions are a common Angular bug.
- **Prefer the `async` pipe in templates** for Observable data — it subscribes and unsubscribes automatically, and triggers `OnPush` change detection correctly.
- **Use `switchMap` for dependent IBM i API calls** (for example, when a route param change should load a new resource). `switchMap` cancels the previous in-flight request when a new param arrives, preventing stale data from loading out of order.
- **Use `shareReplay(1)`** when multiple components need the same IBM i data — it multicasts the Observable so only one HTTP request is made regardless of how many subscribers exist.

Performance

- **Use `ChangeDetectionStrategy.OnPush`** on all components. This tells Angular to only re-render a component when its `@Input()` references change or when an Observable emits through the `async` pipe — significantly reducing the number of change detection cycles in complex IBM i data views.
- **Use `trackBy` in `*ngFor`** loops over IBM i data lists. Without `trackBy`, Angular re-renders the entire list DOM when any item changes. With `trackBy`, it re-renders only the changed items.
- **Lazy-load every feature module.** Feature modules loaded lazily reduce the initial JavaScript bundle size, improving the time to first interaction — important when the IBM i application is accessed over slower corporate VPN connections.
- **Preload lazy modules after the initial load** using Angular's `PreloadAllModules` strategy (set in `RouterModule.forRoot`) so that subsequent navigation feels instant.

Security

- **Store JWT tokens in `sessionStorage`, not `localStorage`,** for applications with high security requirements. `sessionStorage` is cleared when the browser tab closes, limiting the window of exposure if a device is left unattended.
- **Never log JWT tokens or API responses containing sensitive IBM i data** to the browser console in production. Use Angular environment checks (`environment.production`) to gate debug logging.
- **Configure CORS correctly on the IBM i backend.** The Angular app's origin must be explicitly allowed — a wildcard `Access-Control-Allow-Origin: *` is not appropriate for APIs that accept Bearer tokens.
- **Use Angular's built-in XSS protection.** Angular's template binding (`{{ value }}`) automatically HTML-encodes values. Never use `[innerHTML]` with unsanitized IBM i data — use Angular's `DomSanitizer` if raw HTML rendering is genuinely needed.

Testing

```
## Run unit tests (Karma + Jasmine, configured by Angular CLI)

ng test

## Run tests once with coverage report

ng test --watch=false --code-coverage

## Run end-to-end tests (Cypress or Playwright — replace default Protractor)

npx cypress run       # Cypress
npx playwright test   # Playwright
```

- **Unit test every service** that calls IBM i APIs, using `HttpClientTestingModule` and `HttpTestingController` to mock requests and assert that the correct URLs and payloads are sent.
- **Unit test form validators** thoroughly — edge cases in IBM i data validation (padded strings, numeric range checks, date boundaries) are common sources of production bugs.
- **Component tests** with `ComponentFixture` verify that components render correctly for all states: loading, success with data, empty, and error.
- **E2E test the full login → view list → view detail → update status flow** with Cypress or Playwright. This flow exercises the entire stack from Angular through the IBM i API to Db2 for i.

Summary

Angular provides a complete, opinionated, and production-proven platform for building the UI layer of IBM i modernization projects. Its strong TypeScript foundation, built-in HTTP client with interceptors, reactive programming model through RxJS, powerful Reactive Forms module, and lazy-loaded routing make it particularly well-suited for large enterprise applications where long-term maintainability and team consistency matter as much as immediate delivery speed.

Angular connects to IBM i exclusively through REST APIs served by an API Server and the IBM i native layer is unaware of and unchanged by the Angular frontend. TypeScript interfaces model IBM i API responses precisely, interceptors handle JWT authentication and token refresh transparently, services encapsulate IBM i API integration and expose RxJS Observables to components, and IBM Carbon Design System for Angular provides a visual language that is consistent with IBM's broader enterprise tooling portfolio.

The architecture is clean, testable, and maintainable: IBM i handles data integrity, business rules, and decades of accumulated logic; Angular handles rendering, navigation, form validation, and the user experience. When these two layers communicate through a well-designed, versioned REST API, the result is a modernized IBM i application that meets the expectations of the modern workforce. It is accessible from any browser, on any device, without changing a line of the IBM i business logic that organizations have built and relied on for decades.

Modern RPG

Modern RPG Fundamentals

What defines modern RPG?

Modern RPG represents the current, preferred way IBM i developers write, organize, and maintain RPG programs to keep applications reliable, readable, and adaptable. It builds on the RPG IV foundation while incorporating free-format syntax, ILE modularity, data-centric techniques, and clear coding practices that align with how professionals work in

2026. These characteristics separate maintainable, future-ready code from legacy patterns that increase defects and slow down enhancements. The following topics are described in this section:

- Core language and syntax foundation
- Free-format presentation
- ILE modularity and organization
- Data-centric practices and built-in functions
- Coding style and readability
- Interface and integration support
- Modernization spectrum and incremental approach
- Role of AI-assisted tools in 2026

Core language and syntax foundation

Modern RPG requires the RPGLE source type (or **FREE for fully free-format members since IBM i 7.1). Older member types such as RPG, RPG36, or RPG38 lack subprocedures, service programs, qualified data structures, and most built-in functions introduced since V5R1. Without these features, developers stay trapped in outdated patterns that raise defect rates and make routine maintenance unnecessarily difficult.

Free-format presentation

Free-format syntax eliminates the rigid column positions that made older RPG source error-prone and hard to read. With the **FREE directive (IBM i 7.1 and later), the entire member starts in column 1, allowing natural indentation, longer descriptive names, and complex expressions without alignment issues. Shops that adopt free format experience fewer entry mistakes, faster code reviews, and easier comprehension by team members.

ILE modularity and organization

Modularity through the Integrated Language Environment (ILE) provides the structural foundation for modern RPG applications. Developers extract reusable business logic—such as pricing rules, tax calculations, customer validation, and inventory checks—into service programs that export procedures. These procedures bind directly into any program or other service program that requires them, eliminating the need to duplicate logic across standalone programs or copy source code between members.

In a typical IBM i shop, ILE modularity delivers concrete, day-to-day advantages:

- **Single source of truth for business rules.** A change to a pricing formula, discount logic, or validation rule occurs in one procedure and immediately affects every caller—no searching dozens of programs to locate every instance.
- **Reuse across multiple interfaces without modification.** The same CalcTax procedure serves a 5250 order-entry screen, a REST API endpoint, a mobile app service, a batch update job, or an integration with an external system.
- **Focused and reliable unit testing.** Individual procedures can be tested in isolation using simple test harnesses or modern unit-test frameworks, rather than requiring a full program cycle with specific input files and database states.
- **Better dependency management and isolation.** Activation groups and binding directories control resource sharing and prevent the side effects (unintended global variable changes) that plague monolithic programs.
- **Lower risk and faster evolution.** When business requirements change—adding a new discount type, compliance check, or rounding rule—the update happens once in the service program, shrinking testing scope, deployment effort, and the chance of inconsistent behavior across the application.

Within individual modules, subprocedures replace traditional subroutines for internal code organization. Subprocedures offer:

- Parameter passing by value or reference
- Local variables with proper scope (no accidental global interference)
- Structured control flow with meaningful return values

- Clearer debugging because behavior depends only on inputs and local state

These capabilities remove many longstanding problems with EXSR and global variables, such as unexpected side effects, hard-to-trace bugs, and fragile code that breaks when one part changes.

Data-centric practices and built-in functions

Modern RPG favors database strengths and built-in functions to handle data efficiently. Developers use embedded SQL for result-set processing, multi-table joins, and aggregates when DB2 optimization provides clear advantages over native READ/CHAIN loops. Built-in functions replace custom routines for routine tasks:

- String handling: %TRIM, %REPLACE, %SCAN, %XLATE
- Date/time operations: %DATE, %DIFF, %SUBDT, %TIMESTAMP
- JSON/XML processing: DATA-INTO and DATA-GEN (since IBM i 7.3)

Qualified data structures (LIKEDS, LIKEREC) and dynamic arrays (%ALLOC, %REALLOC) produce clearer, less error-prone code than positional fields or fixed-length limits. These techniques cut code volume, improve performance through indexes and constraints, and bring RPG closer to industry-standard data practices.

Coding style and readability

Modern coding style makes code self-explanatory. Developers use descriptive names (OrderTotal, CustNumber instead of OTOT, CUSNBR), qualify subfields (CustRec.Name instead of CUSTNM), group logic with blank lines and block comments, and replace numbered indicators with named indicators or direct conditions. Consistent indentation and logical flow reduce mental effort during maintenance and debugging, especially in mixed teams that include developers familiar with other languages.

Interface and integration support

Modern RPG supports multi-channel delivery through RPG Open Access handlers and service-oriented design. Open Access enables custom I/O for browser, mobile, or API front ends while preserving core business logic. RESTful services built around ILE procedures expose rules externally without duplication. This approach extends the life of existing applications and enables new delivery channels.

Modernization spectrum and incremental approach

Modern RPG exists on a practical continuum. A program can be meaningfully modern even if it retains some native I/O, provided it uses free-format logic, extracts reusable procedures, and avoids numbered indicators. The recommended path begins with small, high-value steps:

- Convert calculations to free format
- Extract internal subprocedures from subroutines
- Move shared logic into service programs
- Replace custom routines with built-in functions
- Adopt embedded SQL for complex data access

Each step improves readability, reduces defects, and simplifies future work.

Role of AI-assisted tools in 2026

In 2026, AI-assisted tools such as IBM Bob™ analyze RPG source in real time within editors. They suggest replacements for outdated constructs (MOVE to assignment, indicator logic to structured blocks), recommend qualified structures for legacy records, explain legacy sections, and generate modern free-format equivalents from descriptions. This assistance speeds incremental modernization without requiring complete rewrites.

Modern RPG combines free-format clarity, ILE modularity, data-centric techniques, and readable style to produce code that is easier to understand, test, extend, and hand off. These characteristics lower maintenance costs, reduce production incidents, shorten enhancement cycles, and support multi-channel strategies while protecting decades of

business logic. For more details on free-format usage, see sections 7.12 through 7.14; for ILE modularity, see 7.4 and 7.11; for preferred design and coding practices, see 7.2 and 7.3.

SECTION COMPLETE: IBMI-RPG-CONCEPT-079 — What defines modern RPG?

Key topics covered:

- Defining characteristics of modern RPG organized into clear categories with practical, shop-focused explanations
- Significantly expanded ILE modularity subsection with bulleted lists for benefits and subprocedure advantages to improve readability
- Emphasis on incremental modernization as an achievable, step-by-step process

Connections for future sections:

- Free-format foundation → expanded in why use fully free-format RPG (7.12), how to write it (7.13), and how to convert (7.14)
- ILE modularity, service programs, and procedure benefits → detailed in What is ILE modularization? (7.4) and How do I modularize RPG programs? (7.11)
- Data-centric practices and built-ins → built upon in embedded SQL (7.15), modern data structures and arrays (7.16), and JSON processing (7.17)

NEXT SECTION: IBMI-RPG-CONCEPT-080 — What is the business value of modern RPG? To start: "Write section IBMI-RPG-CONCEPT-080 — What is the business value of modern RPG?. Previous section covered the defining characteristics of modern RPG with expanded, bulleted explanation of ILE modularity benefits, free-format advantages, data-centric practices, and the incremental modernization path."

(Approximate word count: 1,280)

RPG Coding Best Practices

Coding best practices are key to any program that you develop. When you implement best practices in your shop, you enable your team to be consistent enough without squashing creativity that makes sure one developer can look at anyone's code developed by any member of the team.

Don't hard code

Hardcoding was a practice done by developers in the past. You will encounter it but, dont ever add to it. Imagine what you will have to do if you inherited a series of programs that checked whether they were on the production system or not. The developer before you decided to add logic that checked for the system name/serial number of the IBM i you are developing on. Now imagine needing to switch systems and that system name/serial number changes. You now have a ton of programs that wont work in production because the system or serial number was changed. You never want to hard code a value, if you need to add logic lets say to check whether you are on the production system or not, create a utility service program that has a procedure that returns a boolean, that will tell you if you are on production or not by doing a lookup to a utility table that has the system name soft coded. You want to softcode as much as possible.

API keys, login information

Today's RPG programs won't be limited to languages native on IBM i. You will most likely integrate with newer technologies via APIs and they will be secured either by using a secret key, user name and password. You don't ever want to hard code those credentials in your programs. Put them in a database file that has limited access and masking so you can't see the values in plain text.

User profiles, emails

It was very common for a long time to hard code user profiles and/or email addresses for notifications. Now you have employees who leave and their profiles have to be left for one reason or another, emails that will start bouncing once they are deactivated. Create an email table and soft code any notification email addresses and create a lookup program to do the work.

Business logic

Business logic like item numbers, customer numbers, warehouses etc. You will see them hard coded in many places. There are many different ways to handle this logic and the conditioning you need to do, instead of logic like

```
if ( warehouse = 1 );
   itemNumber = 'ABC123';
endif;
```

You can create a procedure in a service program that will do the work. The procedure could look like this.

```
dcl-proc isWarehouseOne;
  dcl-pi *n ind;
     warehouse int( 5 );
  end-pi;

   dcl-s count int( 5 );

   exec sql
      select count( * ) into :count
      from warehouseMast
      where whs = :warehouse and whs = 1;

   // returns true if count > 0, otherwise false.
   return ( count > 0 ) ;
end-proc;
```

Now instead of the if block above, you have

```
if ( isWarehouseOne( whsNum ));
   item = whsOneItem();
endif;
```

The action of hard coding key business logic that you know will be used in multiple places is a hard habit to break. You want to create service programs and procedures that will keep logic in one place.

The list of things not to hard code is vast but, the gist of it is softcode where you can, avoid hard coding like the plague and you will create less work for yourself and others in the future.

Don't be ambiguous with naming

Having naming conventions that require you to have to stop and think to understand what they mean needs to go away. For some they might think if they figure out IT12 = Item12 that they are like a hacker in a hacker movie but the reality is, you won't remember what IT12 means 3 weeks after your change goes live, let alone when another developer has to look at your code. Be purposeful with your names. If you have an item number variable, make that variable make sense. It could be itemNumber, itmNum, ItemNum etc. You want to be able to understand what the variable is by a quick glance.

Naming is not limited to just variables. Procedures, variables, constants are all up for grabs. You want to be able to look at something and know exactly what its doing. For procedures use getXX for returns, setXX for setting something with no return values, is for booleans, has if is does not make sense. For example getItemNumbers, setItemNumbers, isItemNumberActive, hasItemBalance all tell you exactly what the procedure is doing.

Don't write subroutines

Subroutines are logic that is just in a separate spot in the program and can be called from multiple places. They use global variables and are hard to scope for that reason; use internal procedures to your program or service program procedures if available and applicable.

Limit global variables

Global variables can be touchy but, you need to be purposeful with the use of them. In main RPG programs, global variables are a big part of process but, that prevents you from limiting their scope. Or worse, put a variable (not template, explained below) in a copy book and its global use now just jumped from one program to many. If you have a new RPG program, scope out the logic to use internal procedures with local variables.

For example instead of

```
dcl-s itemDescription varchar( 50 );
```

at the beginning of your program, create a procedure that uses local variables.

```
dcl-proc getItemDescription;
   dcl-pi *n varchar( 50 );
      itemNumber int( 5 );
   end-pi;

   dcl-s itemDescription varchar( 50 );

   exec sql
      select itmDesc
      into :itemDescription
      from myItemTable
      where itemNum =  :itemNumber;

   return itemDescription;
end-proc;
```

Use service programs procedures when you can where you can.

The rule of thumb is if you need the same logic in more than one place then it needs to go into a service program procedure. The first time you are developing the code and it exists in one place only then you can have it as mainline code or an internal procedure to that program but, as soon as you need that logic, extract the code and put it into a service program procedure.

Make use of error handling

RPG gives you monitor/end-mon with on-excp. It is pretty typical in generalized program to have some error handling in place, its a good practice to make use of monitor blocks to prevent your program from having a hard crash.

Comments should make sense

How many times have you read code that has limited, out of date or meaningless comments or worse no comments at all. Program and procedures should have an explanation; this may be as short as a sentence or as long as a few paragraphs. Explain its interactions with the outside world and/or how it should be used; don’t explain its actions with pseudocode. Use proper English grammar.

Be Consistent

Consistency is key; come up with an internal or team standard that makes sense. How you name your variables, comment your code, indent your code, etc. should all be consistent. It's very easy to indent one block of code two places in and the next three places in for no reason other than you just coded it that way. Find what works for you and what your team's standards are and make sure to keep applying it in all your coding.

Use like or template where applicable.

Template is a important keyword you can apply to variables. They make the template variable unchangable but, you are able to define another variable like the template you created.

```
// template variable
dcl-s myItemTemplate  varchar( 50 ) template;

dcl-s item like myItemTemplate;
```

like is a powerful keyword, it helps you define variables, procedure return values, input values etc like something else. This is helpful when you know you are going to have the same variable in multiple places and instead of setting the length in all places, you can have a template variable and reference that variable in all of the places you need to.

Conclusion

Coding best practices can vary based on your teams standards. If standards dont exist then create some for yourself and keep with them using the best practices that were given here. For some places the standard and best practice is to use SQL only. While most will agree with that standard, there are some places where that cannot be a hard and fast rule. Also note that there are best practices that wont ever change like dont hardcode but best practices that can change like how you create a data structure today might not look the same 10 years from now. Best practices can change like the technology we work with, just make sure to change with it. --- title: "What Are RPG Program Design Best Practices?" type: "Concept" product: "Modern RPG" audience: "Intermediate" time: "5 minutes"

Why Use Fully Free Format RPG

For decades, RPG has been a cornerstone of application development on IBM i. However, traditional fixed-format RPG, rooted in its punch-card era constraints, imposes rigid column-based rules that make code difficult to read, write and maintain. With the introduction of fully free format RPG in IBM i 7.1, developers are no longer bound by these limitations. Instead, they can write code in a modern, flexible syntax that closely resembles contemporary programming languages like Java, C#, and Python.

This evolution is not just about syntax, it represents a fundamental shift in how developers can approach coding on IBM i, unlocking new levels of productivity, maintainability, and integration capabilities. By removing the burden of column positioning and enabling clear declarations (`ctl-opt`, `dcl-s`, `dcl-ds`, `dcl-pr`), natural indentation, and longer meaningful variable names, it allows developers to focus on business logic rather than formatting rules. This leads to cleaner code that is easier to understand and maintain.

Better readability

Readable code is cheaper to maintain. Most of the life of an IBM i application developer is spent enhancing, debugging, and maintaining existing code rather than writing new code. Fully free format RPG improves readability in several ways:

- It eliminates position-based syntax and most column-related mistakes.
- It allows for meaningful variable names such as `OrderTotal` or `TransactionStatus`, making the code self-explanatory.
- It supports natural indentation which enhances the visual structure of the code and makes it easier to follow the flow of logic.
- It enables the use of modern constructs like `if/else`, `for` loops, and `procedure` calls that are familiar to developers from other languages.

Consider the difference:

Fixed Format:

```
C     FLD1          IFEQ      'VALUE'
C                   EXSR      SUBR1
C                   ELSE
C                   EXSR      SUBR2
C                   ENDIF
```

Fully Free Format:

```
if FLD1 = 'VALUE';
  exsr SUBR1;
else;
  exsr SUBR2;
endif;
```

The free format version is immediately recognizable to developers familiar with modern programming languages.

Easier Maintenance and lower risk

Fully free format does not change the business rules in a program, but it makes them easier to understand and modify. When code is clearer, developers can make changes with more confidence. Reviews are faster, debugging is simpler and the chance of introducing new bugs is reduced. This leads to lower maintenance costs and faster turnaround times for enhancements and fixes. This is very important for IBM i applications, which often have long lifecycles and are maintained by multiple developers over time.

Stronger fit with modern development practices

Modern RPG encompasses fully free format syntax combined with advanced language features that align with contemporary programming practices. Key characteristics include:

- **ILE modularity**: Service programs, exported procedures, and subprocedures are easier to define and organize.
- **Built-in Functions**: Operations such as `%trim`, `%date`, `%timestamp` and `%replace` read more naturally in free format.
- **Qualified Data Structures**: Modern declaration are clearer and less error-prone than older position-based definitions.
- **Integration Capabilities**: Enables seamless integration of SQL, JSON, and web services in free form source.
- **IDE integration**: Leverages modern compilation and binding techniques within the ILE environment or IDEs like VS Code, which provide features such as content assist, syntax checking, and debugging for RPG development.

Easier onboarding for new developers

One of the biggest practical benefits of fully free format RPG is that it lowers the entry barrier for new developers. Programmers having knowledge of modern programming languages such as Java, C#, or Python will find the free format syntax more familiar and easier to learn. This can help organizations attract new talent to work on their IBM i applications and reduce the time required for training and onboarding. The modern syntax allows new developers to quickly become productive and contribute to the codebase without struggling with the quirks of fixed-format RPG.

Better tooling and source control integration

Fully free format RPG is better suited for modern development tools and source control systems. The clear syntax and structure of free format code make it easier to use with editors like VS Code and version control systems like Git, enabling better collaboration among developers. It also makes code reviews more efficient. Additionally, modern IDEs can provide enhanced features such as syntax highlighting, code completion, and refactoring tools that work more effectively with free format code. This leads to improved developer productivity and a more efficient development workflow.

Practical modernization path

Another very good reason to use fully free form is that adoption can be gradual. Developers do not need to rewrite their entire fixed-format codebase at once. New development can be done in free format while existing fixed-format code continues to work without modification. Over time, as developers become more comfortable with the new syntax, they can choose to convert existing code to free format using available conversion tools or as part of regular maintenance activities. This allows organizations to modernize their applications incrementally without disrupting ongoing development or maintenance efforts.

IBM continues to enhance fully free format RPG with new features and capabilities. By adopting this format, developers ensure their code remains current and can take advantage of future language improvements. Fixed-format RPG, while still supported, is unlikely to receive major new features.

Conclusion

Fully free format RPG is worth using because it improves readability, lowers maintenance risk, aligns with modern development practices, allows easier knowledge transfer and makes it easier to onboard new developers. It helps developers spend less time wrestling with formatting rules and more time focusing on business logic. The shift to fully free format is not just about syntax; it's about embracing modern development practices that will serve your organization for years to come. Not only should you use free format, but there are also great options for moving from fixed to free—tools like IBM Bob can easily help you with this transition. The best part: the core function of the code does not change, just the readability to the developer.

Modularization and Code Organization

What is ILE modularization?

ILE modularization changes how RPG applications are structured and maintained. Traditional RPG programs are often large, standalone units with duplicated logic, global variables, and subroutines that make changes risky and time-consuming. The Integrated Language Environment (ILE) lets developers break applications into smaller, reusable pieces—modules and service programs—that encapsulate specific responsibilities. This approach reduces duplication, isolates changes to a single place, and allows the same business logic to serve multiple interfaces without modification.

In a working IBM i shop, modularization delivers measurable benefits. A pricing rule or tax calculation lives in one service program; update it once, and every calling program benefits without recompilation (as long as the interface stays compatible). Service programs load once and stay resident, lowering overhead. Binding catches interface errors at creation time. Activation groups control resource scope. These features make code easier to test, debug, and extend—critical when the same rules support 5250 applications, web services, APIs, or mobile front ends.

The following topics are described in this section:

- Core ILE concepts: modules, programs, and service programs
- Subprocedures as the foundation of modular code
- Building and exporting procedures in service programs
- Binding and activation groups in modular designs
- The program entry procedure requirement
- Practical organization patterns for modular applications

Core ILE concepts: modules, programs, and service programs

An ILE module is the unit produced by the CRTRPGMOD command. Modules used should be cycle-less using the NOMAIN keyword in the control specification. NOMAIN modules are ideal for reusable components because they avoid unnecessary cycle overhead.

Modules alone are not executable. Bind them into a program object (*PGM) with CRTPGM or into a service program (*SRVPGM) with CRTSRVPGM. A program object has a single entry point and runs as a standalone unit. A service program functions as a shared library: it exports procedures that bound programs call, with the code loaded once and shared across callers.

Service programs provide the primary reuse benefit. Change exported logic once, and all bound programs see the update immediately, provided the procedure prototype remains unchanged. This eliminates the need for copy/paste or /COPY members that require recompiling every consumer when rules change.

Subprocedures as the foundation of modular code

Subprocedures replace subroutines in modular RPG. Subroutines rely on global variables and cannot be exported cleanly. Subprocedures offer local scope, typed parameters, return values, and export capability.

A subprocedure starts with DCL-PROC and ends with END-PROC. The procedure interface (DCL-PI) defines parameters and return type. Add EXPORT to make the subprocedure reusable from other modules.

Here is an exported subprocedure that calculates age in years:

```
**free
ctl-opt nomain;

// Prototype (typically in a /COPY member)
dcl-pr AgeInYears int(10);
  BirthDate date(*iso) const;
end-pr;

// Implementation
dcl-proc AgeInYears export;
  dcl-pi *n int(10);
    BirthDate date(*iso) const;
  end-pi;

  dcl-s Today date(*iso);
  dcl-s Years int(10);

  Today = %date();
  Years = %diff(Today : BirthDate : *years);

  return Years;
end-proc;
```

This subprocedure is self-contained and reusable. A caller invokes it simply:

```
CustAge = AgeInYears(CustRec.BirthDate);
```

Subprocedures promote small, focused units that are straightforward to understand and test.

Building and exporting procedures in service programs

Service programs collect related subprocedures into a reusable library. They let you package business logic once and call it from multiple programs, green-screen or otherwise.

Follow these steps to create one:

1. Write the subprocedures in a source member, marking those for external use with EXPORT.

Here is a basic example of a reusable utility procedure in fully free-format RPG:

```
**free
ctl-opt nomain;

// Prototype (place in a /COPY member for callers)
dcl-pr FormatPhone varchar(20);
  PhoneNum char(10) const;
end-pr;

// Exported implementation
dcl-proc FormatPhone export;
  dcl-pi *n varchar(20);
    PhoneNum char(10) const;
  end-pi;

  dcl-s Formatted varchar(20);

  if PhoneNum = *blanks;
    return '';
  endif;

  Formatted = '(' + %subst(PhoneNum:1:3) + ') ' +
              %subst(PhoneNum:4:3) + '-' +
              %subst(PhoneNum:7:4);

  return Formatted;
end-proc;
```

This procedure formats a 10-digit phone number with parentheses and a hyphen. NOMAIN keeps it cycle-less and lightweight for reuse.

2. Compile the source to a module object: `CRTRPGMOD MODULE(REDBOOKCDE/PHONEUTILS) SRCFILE(REDBOOKCDE/QRPGLESRC) DBGVIEW(*SOURCE)`
3. Define the export list in binder source (strongly preferred over EXPORT(*ALL) for signature stability):

   ```
   STRPGMEXP PGMLVL(*CURRENT)
     EXPORT SYMBOL("FormatPhone")
   ENDPGMEXP
   ```

   This controls exactly which procedures are visible to callers and helps manage version compatibility.
4. Create the service program: `CRTSRVPGM SRVPGM(REDBOOKCDE/PHONEUTILS) MODULE(REDBOOKCDE/PHONEUTILS) EXPORT(*ALL) ACTGRP(*CALLER)`

A caller program bound to this service program accesses the procedure naturally:

```
CustPhone = FormatPhone(CustRec.Phone);
```

Use binding directories to manage dependencies cleanly:

Create the Binding Directory `CRTBNDDIR BNDDIR(REDBOOKCDE/APPBNDDIR) TEXT('Redbook Binding Directory')`

Add the service program once `ADDBNDDIRE BNDDIR(REDBOOKCDE/APPBNDDIR) OBJ((REDBOOKCDE/PHONEUTILS))`

Then reference the directory when building programs:

`CRTBNDRPG PGM(REDBOOKCDE/CUSTMAINT) SRCFILE(REDBOOKCDE/QRPGLESRC) BNDDIR(REDBOOKCDE/APPBNDDIR) DFTACTGRP(*NO)`

This approach centralizes dependencies and keeps compile commands short.

Binding and activation groups in modular designs

Binding resolves procedure calls at creation time using prototypes and binding directories. Use DFTACTGRP(*NO) for modern ILE programs to avoid the default activation group.

Activation groups control runtime isolation:

- **ACTGRP(CALLER)* — Service program runs in the caller's activation group; simplest and most common choice.
- Named activation group (e.g., ACTGRP(REDBOOKCDE)) — Enables persistence across calls; useful for caching.
- **ACTGRP(NEW)* — Creates a fresh group per call for strict isolation, but with higher overhead.

Select *CALLER for most service programs unless persistence or separation is required. Activation groups also govern commitment control and open data path behavior in modular database access.

The program entry procedure requirement

Every *PGM object requires a program entry procedure (PEP)—the starting point for execution. Cycle-main modules (compiled without NOMAIN) provide a PEP automatically through the RPG cycle. NOMAIN modules (containing only subprocedures) do not provide a PEP by default.

Attempting to bind only NOMAIN modules into a *PGM triggers CPD5D1A: "No program entry procedure was found." Service programs do not need a PEP—they exist as libraries.

To create a program:

- Include at least one module with a main procedure (cycle-main or MAIN(proc-name)).
- Specify ENTMOD(*FIRST) on CRTPGM if needed to designate the entry module.

Practical organization patterns for modular applications

Organize modular RPG effectively:

- Group procedures by domain: utility service programs (DATEUTILS, STRINGUTILS) and business service programs (ORDERPROC, CUSTRULES).
- Use NOMAIN on all service program modules.
- Store prototypes in /COPY members for consistent interfaces.
- Avoid global variables; pass data explicitly through parameters.
- Limit procedures to a single responsibility (ideally one screen or less).
- Document exported procedures in prototypes.
- Test subprocedures independently (consider RPGUnit).

These patterns make large applications comprehensible. Developers locate logic quickly in well-named service programs instead of monolithic code.

ILE modularization makes RPG a flexible, component-based language that preserves existing investments while supporting modern needs. For transitioning legacy code to this structure, see Section 7.3, "How do I transition to modern RPG?" For syntax that enables cleaner subprocedures, see Section 7.5, "Free format RPG."

What is cyclomatic complexity?

Cyclomatic complexity provides an objective measure of how complicated a program's control flow really is. Introduced by Thomas J. McCabe in 1976, this metric counts the number of linearly independent paths through a section of code—typically a procedure or module. In RPG terms, it tells you how many different ways the logic can execute based on decisions, loops, and conditions. Lower values indicate simpler, more predictable code that is easier to understand, test, and maintain. Higher values signal potential trouble spots where bugs hide and future changes become risky.

In a modern IBM i shop, cyclomatic complexity matters because legacy RPG programs often accumulate nested IFs, multiple WHEN groups in SELECT, DO loops with embedded conditions, and scattered branching logic. These structures multiply execution paths quickly. A single procedure with a complexity of 30 or more can require dozens of

test cases to achieve reasonable coverage, and even then, subtle defects slip through. When maintenance programmers encounter high-complexity code, they hesitate to make changes, leading to workarounds, duplicated logic, or patches that increase technical debt further.

The practical value of tracking cyclomatic complexity lies in guiding refactoring decisions. During modernization efforts, teams use this metric to prioritize which procedures or service programs to break apart first. A high score flags code that should be split into smaller, focused subprocedures—each responsible for a single decision branch or loop. This refactoring reduces risk, improves unit testability, and makes the overall application more responsive to business changes. Tools that compute the metric automatically help enforce team standards: many organizations set a target of 10 or less per procedure, with anything above 20 requiring review before promotion.

How cyclomatic complexity is calculated

The formal definition uses a program's control flow graph, where nodes represent blocks of sequential code and edges represent transfers of control (jumps caused by conditions, loops, or calls). The McCabe formula is:

Cyclomatic complexity (CC) = E - N + 2P

- **E** = number of edges (control transfers)
- **N** = number of nodes (sequential code blocks)
- **P** = number of connected components (usually 1 for a single procedure)

For most RPG subprocedures, which have a single entry and exit, P = 1, so the formula simplifies to E - N + 2. In practice, developers rarely draw graphs. Instead, they count decision points (predicate nodes) and add 1:

CC ≈ number of decision points + 1

In RPG IV and later, each of the following adds one to the count:

- Each **IF** statement (regardless of nesting level)
- Each **ELSEIF** (or **WHEN** in a SELECT group)
- Each **DO** / **DOW** / **DOU** / **FOR** loop
- Each **AND** / **OR** in a complex condition (some analyzers count these separately)
- **CASxx** operations in older code
- **MONITOR** / **ON-ERROR** blocks (each ON-ERROR adds a path)

A simple procedure with no decisions has CC = 1. Adding one IF increases it to 2 (true and false paths). Nesting another IF inside raises it to 3, and so on.

Consider a basic validation routine:

```
**free
ctl-opt main('ValidateOrder');

dcl-proc ValidateOrder;
  dcl-pi *n;
    OrderQty   packed(5:0) const;
    StockAvail packed(5:0) const;
    Price      packed(9:2) const;
  end-pi;

  dcl-s IsValid    ind;
  dcl-s TotalPrice packed(11:2);

  IsValid = *on;

  if OrderQty <= 0;
    IsValid = *off;
  elseif StockAvail < OrderQty;
    IsValid = *off;
  endif;

  TotalPrice = OrderQty * Price;
  if TotalPrice > 10000.00;
    IsValid = *off;
  endif;

  if not IsValid;
    // log error or throw exception
  endif;

  return;
end-proc;
```

This procedure has three decision points: two in the first IF-ELSEIF chain (treated as two branches) and one more IF for the total price check. Cyclomatic complexity is therefore 4 (3 decisions + 1 base path). Testing needs to cover four scenarios: valid order, zero quantity, insufficient stock, and high total price.

Now examine a more complex version with nested logic:

```
**free
ctl-opt main('ProcessPayment');

dcl-proc ProcessPayment;
  // ... declarations ...

  if PaymentType = 'CC';
    if CardNumberValid(CardNumber);
      if Amount <= CreditLimit;
        if AuthorizeCC(CardNumber: Amount);
          ProcessOrder();
        else;
          SendDecline();
        endif;
      else;
        SendLimitExceeded();
      endif;
    else;
      SendInvalidCard();
    endif;
  elseif PaymentType = 'ACH';
    if BankAccountValid(Account);
      if FundsAvailable(Account: Amount);
        TransferACH();
      else;
        SendNSF();
      endif;
    else;
      SendInvalidAccount();
    endif;
  else;
    SendUnknownType();
  endif;

  // ... cleanup ...
end-proc;
```

Here, the outer IF-ELSEIF creates two main branches, but each contains nested IFs. Counting decisions: outer IF (1), inner IF for CC (2), another for limit (3), authorize (4); elseif branch has two more IFs (5,6). Total decisions ≈ 6–8 depending on exact analyzer. Complexity easily reaches 7–9, meaning 7–9 independent paths. This single procedure now demands extensive testing, and a small change (such as adding a fraud check) risks breaking unrelated paths.

Interpreting complexity scores

McCabe's original guidance suggested:

- 1–10: simple, low risk
- 11–20: moderate complexity, manageable with care
- 21–50: high complexity, refactor recommended
- > 50: very high, serious maintenance risk

In RPG modernization projects, teams often adopt stricter targets. A service program procedure with CC > 15 typically gets flagged for splitting. The Software Engineering Institute and many quality standards recommend keeping individual modules under 10–15 to ensure reliable testing and low defect rates.

High complexity correlates strongly with defect density. Studies show that procedures above 20 have significantly higher bug rates because paths interact in unexpected ways. In IBM i environments, where many programs run

unattended for years, hidden defects in high-complexity code can cause costly production failures when data patterns change.

Using cyclomatic complexity in practice

During code reviews, calculate or display CC for each subprocedure. Many modern tools integrated with Rational Developer for i, VS Code extensions, or open-source analyzers provide this automatically. When a score exceeds the threshold, refactor by:

- Extracting nested logic into separate subprocedures
- Replacing deep nesting with early returns or guard clauses
- Converting complex SELECT/WHEN groups to lookup tables or polymorphic calls
- Simplifying conditions with helper procedures

As discussed in Section 7.9, "How do I measure and reduce complexity?", tracking CC over time shows progress in a modernization effort. A downward trend in average procedure complexity indicates cleaner, more maintainable code.

Cyclomatic complexity is not perfect—it ignores data complexity and external calls—but it provides a reliable, language-independent starting point for identifying trouble spots. In modern RPG, keeping procedures simple enough that their complexity stays low makes unit testing practical, onboarding faster, and long-term maintenance far less painful. Every shop should incorporate this metric into its quality gates to prevent legacy-style spaghetti logic from creeping back in.

How do I transition to modern RPG?

Transitioning to modern RPG is a practical, step-by-step process that lets any IBM i shop improve code quality, reduce maintenance effort, and increase flexibility without halting business operations or requiring a complete rewrite. The previous section explained the business value: lower long-term costs, faster response to new requirements, easier onboarding of developers familiar with contemporary languages, and stronger support for modern integration points. This section provides a realistic roadmap for making that transition in controlled increments so that benefits appear early and risks stay low.

The guiding principle is to apply modern practices wherever code is already being changed—whether for bug fixes, new features, or regulatory updates—and to use those same techniques on all new development. Over multiple maintenance cycles, the entire application gradually adopts free-format syntax, modular structure, and cleaner logic. This approach makes modernization achievable even in shops with limited resources or large legacy codebases.

The following topics are described in this section:

- Taking stock of the current codebase
- Starting with free-format calculations
- Shifting from subroutines to subprocedures
- Moving shared logic into service programs
- Replacing legacy patterns as opportunities arise
- Incorporating supporting practices and tools

Taking stock of the current codebase

Begin with a quick, honest assessment of the existing RPG sources. Examine:

- Which programs still use older source member types such as RPG, RPG36, or RPG38
- The extent of numbered indicator usage
- How much business logic is duplicated across programs
- The ratio of subroutines to subprocedures
- Reliance on the traditional RPG cycle

Focus first on programs that change frequently or contain core business rules. These deliver the fastest return on effort. Less frequently touched utilities or archival reports can wait. This inventory helps set realistic priorities without trying to boil the ocean.

Starting with free-format calculations

The single most accessible improvement is moving calculation logic into free format. Free-format calculations have been supported for many years and immediately improve readability while reducing common syntax mistakes.

On programs that need maintenance, convert the calculation specifications first, the **FREE directive on releases that support fully free-format sources. Test the change thoroughly in the normal development cycle, then promote it.

The free-format version eliminates rigid column positions, allows longer, more descriptive names, and supports natural operators and expressions. Within a few maintenance cycles, most active programs shift to free-format logic. New programs should use free format from the start.

Shifting from subroutines to subprocedures

Subroutines were effective for many years, but subprocedures provide better organization, explicit parameter interfaces, and proper variable scoping. Begin by converting one well-defined subroutine at a time—especially those that perform a single task such as calculating a total or validating a field.

The conversion makes dependencies clear through parameters instead of relying on global variables. After each conversion, compile, test, and deploy as part of the normal change process. Over time, the program becomes more modular and easier to understand without a disruptive rewrite.

Moving shared logic into service programs

The greatest long-term benefit comes from eliminating duplicated code. Identify business rules that appear in multiple places—tax calculations, credit checks, pricing adjustments, date formatting—and extract them into service programs.

Group related procedures into a module, compile it, and bind it into a service program object. Programs that need the logic bind to the service program rather than carrying their own copy. A single change to the service program immediately corrects or enhances behavior across the entire application.

Start with one or two high-value functions that are used widely and are likely to change over time. As confidence grows, expand the service program with additional procedures. This practice is one of the strongest drivers of reduced maintenance cost and improved consistency.

Replacing legacy patterns as opportunities arise

Whenever a program is opened for enhancement or correction, replace outdated elements with modern equivalents:

- Numbered indicators with named indicators or structured control blocks
- MOVE and MOVEL operations with EVAL, %CHAR, %EDITC, or other built-ins
- CALL statements with CALLP for bound procedures
- Cycle-based logic with linear main procedures

These replacements are low-risk when made in small, tested increments. Tools such as IBM Bob, available in 2026 with IBM i 7.6, can review code in real time and suggest these modern replacements directly in the editor. For example, Bob can propose converting a fixed-format MOVE to an EVAL or replacing indicator logic with an ON-EXCEPT block, significantly reducing the manual effort required.

Incorporating supporting practices and tools

Modern RPG development benefits from editors that understand free-format syntax and ILE binding structures. Source control systems track changes reliably. Unit testing frameworks verify behavior after refactoring. These supporting practices become routine as the team works with free-format and modular code.

The transition does not demand perfection from the beginning. Each maintenance window applies one or more of these techniques. Over months or years, the application moves steadily from legacy patterns to modern RPG practices. The result is code that is easier to read, test, extend, and integrate with new technologies.

For more information on the modular design principles that support subprocedures and service programs, see 7.4, "What is ILE modularization?"

How do I modularize RPG programs?

Modularizing RPG programs means breaking large, tightly coupled monolithic applications into smaller, independent, reusable components that communicate through well-defined interfaces. This approach, made practical by the Integrated Language Environment (ILE), reduces maintenance effort, minimizes defects when changes occur, eliminates redundant code across applications, and makes unit testing far more feasible. In a typical IBM i shop, modularization turns a single 5,000-line order-entry program into a main driver that calls shared business logic housed in service programs—logic that the same shop can then reuse for web APIs, mobile back-ends, or batch reporting without duplication.

The following topics are described in this section:

- Moving from subroutines to subprocedures
- Organizing logic into modules
- Packaging reusable procedures in service programs
- Creating a module for the main application logic and binding it to service programs
- Accelerating the process with IBM Bob

Move from subroutines to subprocedures

Legacy RPG programs organize internal reusable code with subroutines (BEGSR/ENDSR). Subroutines operate in the global scope of the program, share all variables, and rely on indicators or global fields to pass information. This design creates hidden dependencies: changing a variable in one subroutine can break unrelated logic elsewhere. Subprocedures solve these problems by providing local scope, typed parameters, return values, and explicit interfaces.

Begin the transition by identifying a self-contained subroutine. Determine what data it needs as input and what it must produce as output. Move those values into parameters and return the result instead of modifying globals.

Consider a typical subroutine that validates an order total against credit limits:

```
C     CHKCRD        BEGSR
C                   IF        ORDAMT > CRTLIM
C                   MOVE      'Y'           CREDIT_ERR
C                   ENDIF
C                   ENDSR
```

This code depends on global fields ORDAMT, CRTLIM, and CREDIT_ERR. Any other part of the program can accidentally change these fields before or after the subroutine runs.

Replace it with a subprocedure:

```
**free
// Source member: CREDIT_H.RPGLE (prototype and implementation example)
// Usually placed in a /COPY member named CREDIT_H.RPGLE for prototypes
ctl-opt nomain;

dcl-pr CheckCreditLimit ind;
  OrderAmount  packed(11:2) const;
  CreditLimit  packed(11:2) const;
end-pr;

dcl-proc CheckCreditLimit export;
  dcl-pi *n ind;
    OrderAmount  packed(11:2) const;
    CreditLimit  packed(11:2) const;
  end-pi;

  return (OrderAmount > CreditLimit);

end-proc;
```

Call the subprocedure like this:

```
if CheckCreditLimit(Order.Total : Customer.CreditLimit);
  // handle credit limit exceeded
endif;
```

The subprocedure is now self-contained. It cannot be affected by unrelated code, and its behavior is predictable based only on the parameters passed. This change alone makes debugging easier and enables isolated testing of the credit-check logic.

Organize logic into modules

A module (*MODULE) is a compiled object that contains one or more subprocedures. When created with the NOMAIN keyword, the module has no main procedure and exists solely to provide reusable code.

Group related subprocedures into a single module based on responsibility. For example, create a module named CUSTUTIL for all customer-related operations:

```
**free
// Source member: CUSTUTIL.RPGLE in REDBOOKCDE/QRPGLESRC
ctl-opt nomain;

dcl-proc GetCustomerName export;
  dcl-pi *n varchar(50);
    CustNo zoned(6:0) const;
  end-pi;
  // retrieve and format name logic here
  return 'John Doe';
end-proc;

dcl-proc ValidateCustomerStatus export;
  dcl-pi *n ind;
    CustNo zoned(6:0) const;
  end-pi;
  // status check logic here
  return *on;
end-proc;
```

Compile the source to a module:

```
CRTRPGMOD MODULE(REDBOOKCDE/CUSTUTIL) SRCFILE(REDBOOKCDE/QRPGLESRC) SRCMBR(CUSTUTIL)
```

Modules become the basic unit of reuse. Keep each module focused: one module for customer functions, another for order calculations, another for date utilities. Cohesive modules with 5–20 subprocedures strike a good balance between manageability and reduced overhead.

Package reusable procedures in service programs

Service programs (*SRVPGM) collect one or more modules into a single object that multiple applications can bind to and share at runtime. This is the key to eliminating copy-paste duplication of business rules across green-screen, web, and API programs.

Create the service program from the module using the simplest form (which exports all procedures by default):

```
CRTSRVPGM SRVPGM(REDBOOKCDE/CUSTSRV) MODULE(REDBOOKCDE/CUSTUTIL)
          TEXT('Customer utility service program')
```

This command produces a service program that exports every exported subprocedure from the module. For introductory modularization, this approach avoids extra complexity while still delivering the core benefit: shared, callable business logic.

Creating a module for the main application logic and binding it to service programs

In a fully modular ILE application, even the entry-point logic belongs in a module. The ORDENTRY module contains the main procedure that drives the application and calls subprocedures from service programs. Create ORDENTRY as a module with a main procedure (specified via the MAIN keyword), then bind it to the service program when creating the final *PGM object.

Here is the ORDENTRY module source. The DSPly operation sends an information message to the message queue of the user running the program (typically QSYSOPR or the user's own message queue). You can view these messages by typing DSPMSG at a command line. Prototypes are shown inline for this example; in practice, move them to a separate prototype member.

```
**free
// Source member: ORDENTRY.RPGLE in REDBOOKCDE/QRPGLESRC
ctl-opt main(Entry);

dcl-pr GetCustomerName varchar(50);
  CustNo zoned(6:0) const;
end-pr;

dcl-pr ValidateCustomerStatus ind;
  CustNo zoned(6:0) const;
end-pr;

dcl-s CustomerNumber zoned(6:0) inz(123456);
dcl-s CustName varchar(50);
dcl-s Msg varchar(52);

// Main entry procedure
dcl-proc Entry;
  CustName = GetCustomerName(CustomerNumber);

  if ValidateCustomerStatus(CustomerNumber);
    Msg = 'Valid customer: ' + CustName;
    dsply Msg '*REQUESTER';
  else;
    dsply 'Invalid customer status' '*REQUESTER';
  endif;
end-proc;
```

Compile ORDENTRY to a module:

```
CRTRPGMOD MODULE(REDBOOKCDE/ORDENTRY) SRCFILE(REDBOOKCDE/QRPGLESRC) SRCMBR(ORDENTRY)
```

Create the final program by binding the ORDENTRY module to the CUSTSRV service program:

```
CRTPGM PGM(REDBOOKCDE/ORDENTRY) MODULE(REDBOOKCDE/ORDENTRY)
       BNDSRVPGM(REDBOOKCDE/CUSTSRV) ACTGRP(*CALLER)
```

This ORDENTRY module focuses on application flow while delegating business rules to the service program. The DSPly operation with '*REQUESTER' sends messages to the interactive user's message queue, making them visible with DSPMSG. In production code, replace DSPly with a display file or dedicated message handling for more robust output.

Binding by reference to service programs keeps program objects small and allows the service program to be updated independently (as long as the exported procedures remain compatible). This separation is especially valuable in shops that deliver changes frequently.

Incremental modernization and practical tips

Start modularization on high-value areas: pricing logic, tax calculations, credit checks, or address validation—routines duplicated in multiple programs. Extract one routine at a time into a subprocedure, move it to a module, build the service program, then create and bind the entry-point module.

Adopt qualified data structures inside subprocedures to avoid name collisions and improve readability.

Monitor binding errors during testing to catch interface mismatches early. Use DSPPGMREF and DSPMOD to document dependencies.

Modular RPG code becomes the foundation for cleaner design, easier testing, faster onboarding of new developers, and flexible deployment across interfaces. The effort pays off quickly in reduced defects and faster response to business requests.

Accelerating modularization with IBM Bob

IBM Bob, available since IBM i 7.6, brings AI assistance directly into the VS Code-based IDE and significantly speeds up the modularization process described in this section. Bob understands full RPG programs—including legacy fixed-format, ILE structures, subroutines, data flows, and dependencies—and can analyze large monolithic sources to identify modularization opportunities.

Highlight a subroutine in your editor and ask Bob to extract it as a subprocedure. Bob automatically:

- Identifies input and output fields.
- Converts globals to typed parameters (using CONST where safe).
- Generates the DCL-PROC wrapper, local variables, and return value.
- Produces a matching prototype for /COPY inclusion.
- Suggests meaningful names if the original variables are cryptic.

For broader refactoring, provide a prompt like "Analyze this order-entry program and extract all pricing logic into a new service program named PRICING." Bob maps the relevant subroutines, consolidates duplicated code, generates the module source, and updates the main program to call the new service program via prototyped calls.

Bob also performs impact analysis before changes, showing which programs reference the affected logic, and generates basic unit-test stubs or integrates with RPGUNIT. This reduces manual risk when extracting high-value routines.

In practice, Bob turns weeks of careful manual extraction into hours of assisted work with review steps. Shops report faster progress on modularization projects because Bob handles repetitive pattern recognition and boilerplate while preserving business intent. Use Bob in combination with the incremental approach: start with one duplicated routine, let Bob propose the extraction, review and test, then bind the result into existing applications.

For background on the ILE concepts that enable this approach, see 7.x, "What is ILE modularization?"

Embedded SQL in RPG

How do I use embedded SQL in RPG programs?

What This Covers

This procedure covers embedded SQL in RPGLE — host variables, cursors, error handling, dynamic SQL with parameter markers, NULL indicators, and transaction control. If you're writing new RPG and hitting the database, embedded SQL is how you do it.

What It Is

Embedded SQL lets you write SQL statements directly inside your RPGLE source. The SQL precompiler extracts them, builds access plans, and wires the results into RPG host variables. No CALL to a stored procedure, no API — the SQL is inline with your program logic.

Two compilation paths:

Command	What It Does	When to Use
CRTBNDRPG	Compiles RPG with embedded SQL, creates a bound program	Standard for programs that contain SQL
CRTSQLRPGI	SQL precompile + RPG compile in one step	Legacy path — still works, but CRTBNDRPG is the modern default

On 7.5 and 7.6, `CRTBNDRPG` handles the SQL precompile automatically. Use it.

Why It Matters

Native RPG file I/O (`CHAIN`, `READ`, `SETLL`) works fine for keyed single-row access against a physical file. But the moment you need a join, a computed column, an aggregate, or filtering beyond a simple key — you're fighting the language instead of using it.

Embedded SQL gives you the full power of Db2 for i inside your RPG program. You get:

- Joins across multiple files in a single statement
- `GROUP BY`, `HAVING`, `ORDER BY` without building your own sort
- Scalar and aggregate functions (`COALESCE`, `TRIM`, `SUM`, `COUNT`)
- Access to SQL views, CTEs, and table functions
- Consistent NULL handling instead of RPG's "blank means null... maybe" convention

The performance story is straightforward: the SQL optimizer builds access plans at precompile time and caches them. For complex access patterns, SQL will outperform chained READ loops with runtime filtering.

How It Works

All examples use this customer master table:

```
-- DDL for CUSTMAST
CREATE OR REPLACE TABLE CUSTDB.CUSTMAST (
    CMCUSTID  CHAR(10)       NOT NULL,
    CMNAME    VARCHAR(50)    NOT NULL,
    CMCITY    VARCHAR(30)    DEFAULT 'UNKNOWN',
    CMSTATE   CHAR(2)        DEFAULT 'XX',
    CMBALANCE DECIMAL(11,2)  DEFAULT 0,
    CMSTATUS  CHAR(1)        DEFAULT 'A',
    CMLASTORD DATE           DEFAULT CURRENT_DATE,
    CONSTRAINT CUSTMAST_PK PRIMARY KEY (CMCUSTID)
);

CREATE INDEX CUSTDB.CUSTMAST_X1 ON CUSTDB.CUSTMAST (CMSTATE, CMSTATUS);
```

A. Host variables — moving data between SQL and RPG

Host variables are RPG variables prefixed with `:` inside SQL statements. They're how data crosses the boundary.

```
**FREE
CTL-OPT DFTACTGRP(*NO)       // Must be *NO — DFTACTGRP(*YES) doesn't support SQL
        ACTGRP(*CALLER);      // Run in caller's activation group

DCL-PI GetCustName;
    pCustId   CHAR(10) CONST;   // :pCustId becomes the WHERE = value
    pCustName VARCHAR(50);       // :pCustName receives the SELECT result
END-PI;

// SELECT INTO — single row only. If the query returns more than one row, you get a negative SQLCODE
Exec SQL
    SELECT CMNAME INTO :pCustName
    FROM CUSTDB.CUSTMAST
    WHERE CMCUSTID = :pCustId;  // :pCustId passed as a parameter — no concatenation

IF SQLCODE = 0;
    // Row found — pCustName is populated with CMNAME
ELSEIF SQLCODE = 100;
    pCustName = '';   // No row matched — not an error, just not found
ELSE;
    pCustName = '*ERROR';   // Negative SQLCODE — authority, table not found, truncation, etc.
ENDIF;

RETURN;
```

Rules for host variables:

- Prefix with `:` inside SQL, use bare name in RPG
- RPG data types map automatically — `CHAR(10)` to `CHAR(10)`, `PACKED(11:2)` to `DECIMAL(11,2)`
- `VARCHAR` RPG variables map to SQL `VARCHAR` — length and data travel together
- `SELECT INTO` is for single-row results only. If the query might return multiple rows, use a cursor (section C)

B. SQLCODE and SQLSTATE — knowing what happened

Every SQL statement sets two status fields. Check them.

Field	Type	Key Values
SQLCODE	Integer	0 = success, 100 = not found / end of data, negative = error
SQLSTATE	CHAR(5)	'00000' = success, '02000' = not found, anything starting with '01' = warning

`SQLCODE` is faster to check for the common cases. `SQLSTATE` is portable across databases and more precise for error diagnosis.

```
DCL-S sqlMsg VARCHAR(100);

Exec SQL
    SELECT CMNAME INTO :pCustName
    FROM CUSTDB.CUSTMAST
    WHERE CMCUSTID = :pCustId;

// SQLCODE and SQLSTATE are implicitly declared — the precompiler injects them
SELECT;
    WHEN SQLCODE = 0;
        // Success — carry on
    WHEN SQLCODE = 100;
        // Row not found — SQLSTATE '02000'. Normal condition, not an error
        sqlMsg = 'Customer ' + %TRIM(pCustId) + ' not found';
    WHEN SQLCODE < 0;
        // Real error — SQLSTATE gives you the specific cause
        sqlMsg = 'SQL error: SQLSTATE=' + SQLSTATE;
        // Log it, don't just ignore it — swallowing negative SQLCODEs is how silent corruption happens
OTHER;
    // SQLCODE > 0 but not 100 — a positive warning. Rare but worth knowing about
    sqlMsg = 'Unexpected SQLCODE: ' + %CHAR(SQLCODE);
ENDSL;
```

A common mistake: checking only `SQLCODE = 0` and treating everything else as "not found." Negative SQLCODEs are real errors — authority failures, table not found, data truncation. Don't swallow them.

C. Cursors — processing multiple rows

When a query returns more than one row, you need a cursor. Declare it, open it, fetch in a loop, close it.

```
**FREE
CTL-OPT DFTACTGRP(*NO) ACTGRP(*CALLER);

DCL-S wCustId   CHAR(10);
DCL-S wName     VARCHAR(50);
DCL-S wBalance  PACKED(11:2);
DCL-S wCount    INT(10) INZ(0);

// DECLARE is processed at precompile time — it's a definition, not an executable statement.
// The cursor name (C_ACTIVECUST) must be unique within the program.
Exec SQL
    DECLARE C_ACTIVECUST CURSOR FOR
        SELECT CMCUSTID, CMNAME, CMBALANCE
        FROM CUSTDB.CUSTMAST
        WHERE CMSTATUS = 'A'
          AND CMBALANCE > 0
        ORDER BY CMBALANCE DESC;     // Highest balance first

// OPEN executes the query and builds the result set
Exec SQL OPEN C_ACTIVECUST;

// DOW 1=1 is intentional — SQLCODE isn't set until the first FETCH,
// so testing SQLCODE in the DOW condition requires a priming fetch. Don't do that.
DOW 1 = 1;
    Exec SQL
        FETCH C_ACTIVECUST INTO :wCustId, :wName, :wBalance;

    IF SQLCODE = 100;   // End of result set — normal exit
        LEAVE;
    ENDIF;

    IF SQLCODE < 0;     // Error mid-cursor — log before leaving
        // Something broke — log and bail
        LEAVE;
    ENDIF;

    wCount += 1;
    // Row is ready — wCustId, wName, wBalance are populated
    // Process the row — write to a report, call a subprocedure, whatever
ENDDO;

// Always close the cursor — releases server-side resources and locks
Exec SQL CLOSE C_ACTIVECUST;

// wCount now has the number of rows processed
RETURN;
```

The `DOW 1 = 1` / `LEAVE` pattern is cleaner than `DOW SQLCODE = 0` because SQLCODE isn't set until after the first FETCH. You'd need a priming fetch otherwise, which means duplicating the FETCH statement. Don't.

Scrollable cursors — if you need to move backward or jump to a specific row:

```
// SCROLL keyword enables bidirectional navigation — costs more because
// Db2 for i materializes the full result set in a temp spool
Exec SQL
    DECLARE C_SCROLL SCROLL CURSOR FOR
        SELECT CMCUSTID, CMNAME
        FROM CUSTDB.CUSTMAST
        WHERE CMSTATE = 'AZ'
        ORDER BY CMNAME;

Exec SQL OPEN C_SCROLL;

// ABSOLUTE n — jump directly to row n (1-based). SQLCODE 100 if n > row count
Exec SQL FETCH ABSOLUTE 10 FROM C_SCROLL INTO :wCustId, :wName;

// PRIOR — step backward one row. SQLCODE 100 if already at row 1
Exec SQL FETCH PRIOR FROM C_SCROLL INTO :wCustId, :wName;

Exec SQL CLOSE C_SCROLL;
```

Scrollable cursors cost more — the database builds a temp result set to support random access. Use them only when you genuinely need to navigate, not as a default.

D. INSERT, UPDATE, DELETE — modifying data

DML statements use host variables the same way:

```
// INSERT — check SQLCODE for failures: duplicate primary key, authority, referential constraint, etc.
Exec SQL
    INSERT INTO CUSTDB.CUSTMAST
        (CMCUSTID, CMNAME, CMCITY, CMSTATE, CMBALANCE, CMSTATUS)
    VALUES (:wCustId, :wName, :wCity, :wState, 0, 'A');

IF SQLCODE < 0;
    // Insert failed — duplicate key? Authority failure? Check SQLSTATE for the cause
ENDIF;

// UPDATE — SQLCODE = 0 does NOT mean a row was changed.
// It only means the statement executed without error.
Exec SQL
    UPDATE CUSTDB.CUSTMAST
    SET CMBALANCE = CMBALANCE + :wPayment,
        CMLASTORD = CURRENT_DATE
    WHERE CMCUSTID = :wCustId;

// SQLERRD(3) is the actual row count — the only reliable way to know if anything changed
IF SQLERRD(3) = 0;
    // No rows matched — customer doesn't exist or WHERE filtered everything out
ENDIF;

// DELETE with a date range — CURRENT_DATE is a special Db2 for i register, no host var needed
Exec SQL
    DELETE FROM CUSTDB.CUSTMAST
    WHERE CMSTATUS = 'I'
      AND CMBALANCE = 0
      AND CMLASTORD < CURRENT_DATE - 2 YEARS;  // 2 years back from today

// SQLERRD(3) holds the delete count here too — log it if auditing is a concern
```

`SQLERRD(3)` is the row-affected count for INSERT, UPDATE, and DELETE. Always check it — a successful SQLCODE of 0 doesn't mean any rows were actually changed. An UPDATE that matches zero rows still returns SQLCODE 0.

E. NULL handling with indicator variables

NULLs in SQL don't map to any RPG value — there's no concept of "absent" in RPG. Indicator variables bridge the gap.

```
DCL-S wCity     VARCHAR(30);
DCL-S wCityInd  INT(5);   // Indicator: 0 = not null, -1 = null, >0 = truncated (VARCHAR overflow)

// Note the syntax: :wCity :wCityInd — no comma between host var and indicator var
// If you add a comma by mistake, the precompiler interprets them as two separate SELECT targets
Exec SQL
    SELECT CMCITY INTO :wCity :wCityInd
    FROM CUSTDB.CUSTMAST
    WHERE CMCUSTID = :pCustId;

IF wCityInd < 0;
    // CMCITY is NULL in the database — wCity is undefined at this point, don't use it
    wCity = '(not on file)';
ENDIF;
```

The indicator variable sits right after the host variable — no comma between them. That syntax trips people up the first time.

You can also use indicators on INSERT to set a column to NULL:

```
wCityInd = -1;  // -1 tells Db2 for i to store NULL — the value of wCity itself is ignored

// The indicator variable goes right after its host variable — same no-comma rule as SELECT
Exec SQL
    INSERT INTO CUSTDB.CUSTMAST
        (CMCUSTID, CMNAME, CMCITY, CMSTATE, CMBALANCE, CMSTATUS)
    VALUES (:wCustId, :wName, :wCity :wCityInd, :wState, 0, 'A');
// CMCITY will be stored as NULL regardless of what's in wCity
```

Indicator arrays for data structures — when you're fetching into a DS with multiple fields, you can declare a single indicator array instead of one indicator per field:

```
DCL-DS custRec;
    custId   CHAR(10);
    custName VARCHAR(50);
    city     VARCHAR(30);
    state    CHAR(2);
    balance  PACKED(11:2);
END-DS;

// DIM must match the number of fields in the DS exactly — one indicator per field, in order
DCL-S custInd INT(5) DIM(5);

Exec SQL
    SELECT CMCUSTID, CMNAME, CMCITY, CMSTATE, CMBALANCE
    INTO :custRec :custInd          // DS and indicator array — one INTO clause covers all 5 fields
    FROM CUSTDB.CUSTMAST
    WHERE CMCUSTID = :pCustId;

// custInd(1)=custId, custInd(2)=custName, custInd(3)=city, custInd(4)=state, custInd(5)=balance
IF custInd(3) < 0;   // city is NULL
    city = '(not on file)';
ENDIF;
```

F. Dynamic SQL with parameter markers

When you don't know the SQL at compile time — building WHERE clauses based on user input, for instance — use dynamic SQL with parameter markers. Never concatenate user input into SQL strings.

```
DCL-S wStmt     VARCHAR(500);
DCL-S wState    CHAR(2);
DCL-S wMinBal   PACKED(11:2);
DCL-S wCustId   CHAR(10);
DCL-S wName     VARCHAR(50);

wState  = 'AZ';
wMinBal = 1000.00;

// Build the SQL string with ? placeholders — never inject wState or wMinBal directly into the string
// e.g., 'WHERE CMSTATE = ''' + wState + '''' opens the door to SQL injection
wStmt = 'SELECT CMCUSTID, CMNAME FROM CUSTDB.CUSTMAST '
      + 'WHERE CMSTATE = ? AND CMBALANCE >= ? '
      + 'ORDER BY CMNAME';

// PREPARE parses the SQL and builds an access plan — S1 is the statement name
Exec SQL PREPARE S1 FROM :wStmt;

// DECLARE references the prepared statement by name — still a definition, not execution
Exec SQL DECLARE C_DYN CURSOR FOR S1;

// OPEN executes the plan, binding ? markers to host variables in left-to-right order
// wState fills the first ?, wMinBal fills the second ?
Exec SQL OPEN C_DYN USING :wState, :wMinBal;

DOW 1 = 1;
    Exec SQL FETCH C_DYN INTO :wCustId, :wName;
    IF SQLCODE <> 0;   // 100 = EOF, negative = error — either way, stop fetching
        LEAVE;
    ENDIF;
    // wCustId and wName populated — process the row
ENDDO;

Exec SQL CLOSE C_DYN;
```

Parameter markers (`?`) are placeholders filled at OPEN time via the `USING` clause. The database treats them as parameters, not as SQL text — so there's no SQL injection risk, and the optimizer can reuse access plans across different parameter values.

EXECUTE IMMEDIATE — for DDL or one-shot statements where you don't need a result set:

```
DCL-S wStmt VARCHAR(200);

// EXECUTE IMMEDIATE is for one-and-done statements — DDL, SET commands, CALL
// It parses, executes, and discards in one step — no PREPARE, no cursor
wStmt = 'CREATE INDEX CUSTDB.CUSTMAST_X2 '
      + 'ON CUSTDB.CUSTMAST (CMLASTORD)';

Exec SQL EXECUTE IMMEDIATE :wStmt;

IF SQLCODE < 0;
    // Object already exists? Not authorized? Check SQLSTATE for the specific cause
ENDIF;
```

Don't use EXECUTE IMMEDIATE for SELECT or DML with host variables — that's what PREPARE + cursor is for.

G. Transaction control — COMMIT and ROLLBACK

By default, RPG programs run under `COMMIT(*NONE)` — every SQL statement auto-commits. For multi-step operations that must succeed or fail together, you need explicit commit control.

```
**FREE
// Named activation group gives us a commit scope boundary.
// ACTGRP(*CALLER) would share the caller's boundary — any ROLLBACK issued by the caller
// would undo our work too. Use a named group to own your own transaction.
CTL-OPT DFTACTGRP(*NO) ACTGRP('ORDERAPP')
        COMMIT(*CS);  // Cursor stability: reads don't lock, writes lock until COMMIT

DCL-PI ProcessPayment;
    pCustId  CHAR(10) CONST;
    pAmount  PACKED(11:2) CONST;
    pResult  CHAR(1);  // 'S' = success, 'F' = failure
END-PI;

// MONITOR catches RPG exceptions (MCH, CPF). ROLLBACK here prevents partial commits
// if something non-SQL blows up mid-transaction.
MONITOR;

    // Step 1: Debit the customer balance — the WHERE includes CMBALANCE >= :pAmount
    // so the UPDATE itself refuses to go negative. If it matches 0 rows, we roll back.
    Exec SQL
        UPDATE CUSTDB.CUSTMAST
        SET CMBALANCE = CMBALANCE - :pAmount,
            CMLASTORD = CURRENT_DATE
        WHERE CMCUSTID = :pCustId
          AND CMBALANCE >= :pAmount;    // Guard against overdraft at the SQL level

    IF SQLERRD(3) = 0;
        // 0 rows updated: customer not found OR balance too low — either way, abort
        Exec SQL ROLLBACK;
        pResult = 'F';
        RETURN;
    ENDIF;

    // Step 2: Record the payment in the ledger — if this fails, undo step 1 too
    Exec SQL
        INSERT INTO CUSTDB.PAYMENT_LOG
            (PLCUSTID, PLAMOUNT, PLDATE, PLTYPE)
        VALUES (:pCustId, :pAmount, CURRENT_DATE, 'PMT');

    IF SQLCODE < 0;
        Exec SQL ROLLBACK;   // Undo the balance update from step 1
        pResult = 'F';
        RETURN;
    ENDIF;

    // Both steps succeeded — make it permanent
    Exec SQL COMMIT;
    pResult = 'S';

ON-ERROR;                    // RPG exception (not SQL) — still need to clean up
    Exec SQL ROLLBACK;
    pResult = 'F';
ENDMON;

RETURN;
```

Key points on commit control:

- `COMMIT(*CS)` on `CTL-OPT` enables cursor stability — reads don't lock, updates lock the row until commit
- `COMMIT(*CHG)` locks only changed rows — lighter, but doesn't prevent non-repeatable reads
- `COMMIT(*NONE)` means no transaction boundaries at all — fine for read-only programs, dangerous for multi-step writes
- The journal (`JRN`) must be configured on the library/files — `COMMIT` without journaling gives you a negative SQLCODE at runtime
- Named activation groups (`ACTGRP('ORDERAPP')`) give you a commit scope boundary. `ACTGRP(*CALLER)` shares the caller's commit scope, which can cause surprises when a calling program issues ROLLBACK

Key Takeaways

- **Use embedded SQL for anything beyond simple keyed access** — joins, aggregates, computed columns, filtered queries all belong in SQL, not chained READ loops
- **Always check SQLCODE after every Exec SQL** — 0 is success, 100 is not found, negative is an error. Don't just check for "not zero"
- `SQLERRD(3)` **is your row count** — for INSERT, UPDATE, DELETE, check it. SQLCODE 0 doesn't mean rows were affected
- **Use parameter markers for dynamic SQL** — never concatenate user values into SQL strings. `?` placeholders prevent injection and improve plan reuse
- **NULL needs indicator variables** — RPG has no NULL concept. The indicator variable (placed after the host variable, no comma) is the bridge
- **COMMIT/ROLLBACK for multi-step writes** — if two operations must succeed together, wrap them in a transaction. Set `COMMIT(*CS)` on CTL-OPT and configure journaling on the files
- `DOW 1 = 1` **/ LEAVE for cursor loops** — avoids the priming-fetch problem and keeps the FETCH in one place
- **Compile with** `CRTBNDRPG` — it handles the SQL precompile automatically on 7.5 and 7.6

How do I use modern RPG data structures and arrays

Modern RPG makes extensive use of data structures and arrays to organize and manipulate business data efficiently. This section shows you how to define qualified data structures, create arrays of those structures, and—most importantly—populate them directly from your IBM i database using only embedded SQL statements. You will also see the newest array capabilities introduced in IBM i 7.4 and still available in 7.6: variable-dimension arrays. These techniques eliminate tedious looping and manual field-by-field assignment, reduce the chance of errors, and produce code that is dramatically easier to read and maintain.

The following topics are described in this section:

- Defining qualified data structures for business records
- The two main types of variable-dimension arrays: `*VAR` and `*AUTO`
- Creating and populating array data structures with SQL
- Using `EVAL-CORR` to move data between differently named or typed structures
- Sorting and searching arrays after population
- Best practices and performance considerations

Why modern data structures and arrays matter

A data structure in modern RPG is far more than a collection of fields. When you qualify it and combine it with arrays, you create a self-documenting container that mirrors the shape of your database tables or result sets. This approach brings three immediate benefits to any IBM i shop:

1. **Readability** — Long, descriptive subfield names replace cryptic positions or numbered fields.

2. **Performance** — SQL can fetch hundreds or thousands of rows into an array in a single operation, replacing slow RPG loops.
3. **Maintainability** — Changes to a table column automatically flow through the data structure when you use `EXTNAME` or `LIKEDS`.

As discussed in 7.15, “How to use embedded SQL,” modern RPG treats the database as a first-class citizen. Data structures and arrays are the natural bridge between that database and your business logic.

Defining qualified data structures

Start every modern data structure with the `QUALIFIED` keyword. This forces you to use the dot notation (`Cust.CUSTID`) and eliminates the risk of field-name collisions.

You can define a data structure manually or base it directly on a database table or view with `EXTNAME`. The `EXTNAME` approach is preferred when the structure should stay in sync with the physical file.

```
// Example 7-20 Data structure based directly on the CUSTOMER table
Dcl-Ds Cust ExtName('REDBOOKCDE/CUSTOMER') Qualified End-Ds;
```

Any future change to the NAME column length is automatically reflected the next time you recompile.

7.16.3 The two types of variable-dimension arrays

IBM i 7.4 introduced variable-dimension arrays that remove the need to guess a safe fixed size. There are two flavors:

- **DIM(VAR : maximum)* — The actual number of elements used is controlled by your program (you set it explicitly with %ELEM). This is ideal when you know exactly how many rows you want to fetch or when you want full control over the active size.
- **DIM(AUTO : maximum)* — The array grows automatically as you assign elements beyond the current size, up to the declared maximum. This is perfect for situations where the number of items is truly unknown until run time, such as processing JSON input or variable query results.

Both types are fully compatible with SQL multi-row FETCH and with EVAL-CORR.

Here are the two styles shown side-by-side using order data from the provided tables:

```
// Example 7-21 SQL fetch into a *VAR array (fixed number of rows expected)
Dcl-Ds OrderVarDS Qualified Dim(*VAR : 20);
  OrderNo    Char(7);
  CustId     Char(10);
  CustName   Varchar(50);
  OrderTotal Char(20);   // Different type for illustration
End-Ds;

// Example 7-22 Host processing array using *AUTO (grows as needed)
Dcl-Ds OrdersAutoDS Qualified Dim(*AUTO : 500);
  OrderNo    Char(7);
  CustId     Char(10);
  CustName   Varchar(50);
  OrderTotal Zoned(11:2);
End-Ds;
```

Notice the intentional differences in subfield names and data types. These differences are common when one structure is optimized for SQL fetch and another for host-program processing.

7.16.4 Populating array data structures with SQL

The real power appears when you combine these arrays with embedded SQL. You can load the entire array in one statement—no DO loops, no manual indexing.

Single-row fetch into a data structure

```
// Example 7-23 Single-row fetch into a qualified data structure
Dcl-S InputCustId Char(10) Inz('CUST000123');

Exec Sql
  SELECT CUSTID, NAME
    INTO :Cust
    FROM REDBOOKCDE.CUSTOMER
   WHERE CUSTID = :InputCustId
     WITH NC;
```

Multi-row fetch into a variable-dimension array

SQL can deliver rows directly into either *VAR or *AUTO arrays. Use GET DIAGNOSTICS to learn exactly how many rows were returned.

Moving data between structures with EVAL-CORR

When subfield names or types differ, EVAL-CORR is the modern way to copy matching fields. It automatically handles name-based matching and skips fields that do not exist in the target or have incompatible types.

Complete working example

The following complete program demonstrates all the concepts in a realistic business scenario. It fetches aggregated order information from the ORDERS, CUSTOMER, and ORDERDTL tables into a *AUTO array, moves data with EVAL-CORR, and then sorts and searches the results—all using modern RPG techniques.

```
**FREE
// Example 7-24 Complete program to retrieve orders using varying-dimension arrays
// Demonstrates DIM(*AUTO), multi-row FETCH, GET DIAGNOSTICS, EVAL-CORR,
// and sorting/searching with %SORTA and %LOOKUP

Ctl-Opt DftActGrp(*No) Option(*NoDebugIO:*SrcStmt:*NoUnRef);

// ============================================================
// Data Structures - RPG data types
// ============================================================
// SQL fetch into a Fixed Structure
Dcl-Ds OrderFixedDS Qualified Dim(20);
  OrderNo    Char(7);
  CustId     Char(10);
  Name       Varchar(50);
  OrderTotal Zoned(11:2);
End-Ds;

// SQL fetch into a variable structure
Dcl-Ds OrderVarDS Qualified Dim(*VAR: 20);
  OrderNo    Char(7);
  CustId     Char(10);
  CustName   Varchar(50);
  OrderTotal Char(20);   // Different type for illustration
End-Ds;

// Host processing array - varying dimension (*AUTO)
Dcl-Ds OrdersAutoDS Qualified Dim(*AUTO : 500);
  OrderNo    Char(7);
  CustId     Char(10);
  CustName   Varchar(50);
  OrderTotal Zoned(11:2);
End-Ds;

Dcl-S NumOrders   Int(10) Inz(0);
Dcl-S ArrayIDX    Int(10);
Dcl-S RowsToFetch Int(10) Inz(20);

// ============================================================
// Main Procedure
// ============================================================
Exec Sql
  DECLARE C1 CURSOR FOR
    SELECT O.ORDERNO,
           O.CUSTID,
           C.NAME,
           Coalesce(Sum(D.QTY * D.PRICE - D.DISCAMT), 0) As ORDERTOTAL
      FROM REDBOOKCDE.ORDERS O
      JOIN REDBOOKCDE.CUSTOMER C On O.CUSTID = C.CUSTID
 LEFT JOIN REDBOOKCDE.ORDERDTL D On O.ORDERNO = D.ORDERNO
     GROUP BY O.ORDERNO, O.CUSTID, C.NAME, O.ORDERDATE
     ORDER BY O.CUSTID Desc
     FETCH FIRST 20 ROWS ONLY;

Exec Sql Open C1;

// Fetch 20 rows into the *AUTO array
Exec Sql
  FETCH C1 FOR :RowsToFetch ROWS
    INTO :OrdersAutoDS;

// Great way to check results of SQL Statements
Exec Sql Get Diagnostics :NumOrders = ROW_COUNT;
```

```
Exec Sql Close C1;

// Moves all field data with the exception of the CustName to Name fields
// (inconsistent naming between structures)
Eval-Corr OrderFixedDS = OrdersAutoDS;

// Must manually set the dimension for the *VAR array before EVAL-CORR
%Elem(OrderVarDS) = NumOrders;

// Moves all field data with the exception of OrderTotal (wrong data type)
Eval-Corr OrderVarDS = OrdersAutoDS;

// Sorting and searching an array of orders after the SQL fetch
// Sort by customer ID descending, then order total descending
// (only the active portion of the *AUTO array - shown with 4 rows for brevity)
SortA(D) %SubArr(OrdersAutoDS : 1 : 4) %Fields(OrderNo : OrderTotal);

// Find the first order for a specific customer
ArrayIDX = %Lookup('CUST000003' : OrdersAutoDS(*).CustId : 1 : NumOrders);

If ArrayIDX > 0;
  Dsply ('Found order ' + OrdersAutoDS(ArrayIDX).OrderNo
         + ' for customer ' + OrdersAutoDS(ArrayIDX).CustId);
EndIf;

*InLr = *On;
Return;
```

After the multi-row FETCH, the *AUTO array OrdersAutoDS contains exactly the number of rows returned (up to the requested 20). GET DIAGNOSTICS captures that count in NumOrders.

EVAL-CORR then copies matching subfields:

- From OrdersAutoDS to OrderFixedDS — the CustName field is skipped because the target uses Name.
- From OrdersAutoDS to OrderVarDS — OrderTotal is skipped because the target field is defined as Char(20) instead of Zoned(11:2).

For the *VAR array, you must explicitly set %ELEM(OrderVarDS) before the EVAL-CORR; otherwise only the first element would be affected. The *AUTO array needs no such step because its active size is managed automatically by the FETCH.

The program then demonstrates sorting with %SORTA(D) (descending on the listed fields) limited to the active rows via %SUBARR, and searching with %LOOKUP on the CustId subfield. These operations show how modern arrays become powerful in-memory structures after the initial SQL population.

Sorting and searching arrays

Once your array data structure is populated, you frequently need to reorder or locate specific entries without additional database calls. Modern RPG provides efficient built-in functions that operate directly on arrays of data structures.

%SORTA sorts the entire array (or a portion of it) on one or more subfields. For multiple keys, use the %FIELDS built-in function and list the subfields in priority order. You can sort only the active portion of a variable-dimension array with %SUBARR. The (D) extender on %SORTA requests descending sequence.

%LOOKUP quickly finds an element by value and returns its index. The (*) notation on the search argument tells RPG to search the active elements of the array.

These operations complete the modern array pattern: fetch once with SQL, move or transform with EVAL-CORR, sort or search in memory, then process or pass the array to a subprocedure. The combination is far faster and cleaner than legacy RPG looping with READ and manual swaps.

Best practices and performance considerations

- Always qualify data structures and arrays.
- Use EXTNAME or LIKEDS whenever possible so the structure stays synchronized with the database.
- Prefer *AUTO when the number of rows is truly variable; use *VAR when you want explicit control over the dimension.
- Keep the maximum dimension generous but realistic.
- When you only need a subset of columns, declare the data structure with exactly those columns rather than using SELECT *.
- Test with realistic data volumes. A 20-row fetch is instantaneous; larger result sets may require commitment control or a larger SQL cursor hold setting.
- Combine these techniques with the fully free-format coding style covered in 7.13, "How do I write fully free format RPG?" The result is code that looks more like modern business applications written in any other language.

Shops that adopt array data structures populated by SQL routinely report 30–70 % fewer lines of code in their data-retrieval routines and dramatically lower maintenance costs. New developers who know Java, Python, or JavaScript immediately recognize the pattern: a list of objects fetched from the database. The same business rules can now be reused from green-screen programs, REST services, or mobile front ends without duplication. Modern RPG data structures and arrays are not cosmetic improvements. They are the foundation that lets you write concise, high-performance code that is easy to test, easy to hand off, and ready for whatever interface technology comes next.

How do I process JSONs within RPGLE programs?

What This Covers

This procedure covers the options available on IBM i to process JSON payloads.

What It Is

REST APIs, message queues, webhook callbacks — most integrations your RPG programs touch today exchange data as JSON. Your programs need to parse, extract from, and generate JSON.

Three options exist (this procedure focuses on SQL JSON functions — see the references at the end of this section for the other two):

	SQL JSON Functions	DATA-INTO with JSON parsers	YAJL - External JSON library
Best for	Set-based processing, direct Db2 for i inserts	Mapping JSON directly into RPG data structure hierarchies	Streaming large docs, highly dynamic schemas
External dependency	None	Yes - needs a parser program (e.g. YAJLINTO, QOAR/ JSON_PARSE)	Yes - YAJL
Learning Curve	Low - SQL devs pick it up quickly	Medium - need to understand DS mapping	Medium - new API to learn
Performance	High — Db2 for i engine handles parsing natively, no RPG overhead	Medium - Depends on the complexity of the JSON and the efficiency of the parser	Variable - fast C parser, but RPG ↔ C call overhead adds up on many small calls
Flexibility	Medium - handles fixed/known JSON schemas well; struggles with highly variable or deeply nested structures	High - Can handle complex JSON structures with custom parsing logic	High - Can handle any JSON structure as long as the library supports it
Use Cases	Ingesting API payloads directly into Db2 for i tables	Populating RPG data structures from JSON for in-program processing (e.g. config files, lookup data)	Streaming or incrementally parsing very large JSON docs, or when you need to build JSON dynamically at runtime
Standards	ISO SQL standard - syntax transfers to PostgreSQL, Oracle, SQL Server, etc.	IBM i RPG-specific op-code	Platform-specific C library wrapper
Support	Yes	No - leverages an external non-support component	No, Open Source

For more info on the other approaches, check out the following references:

- DATA-INTO with JSON parsers
- YAJL - Open source JSON library

Why It Matters

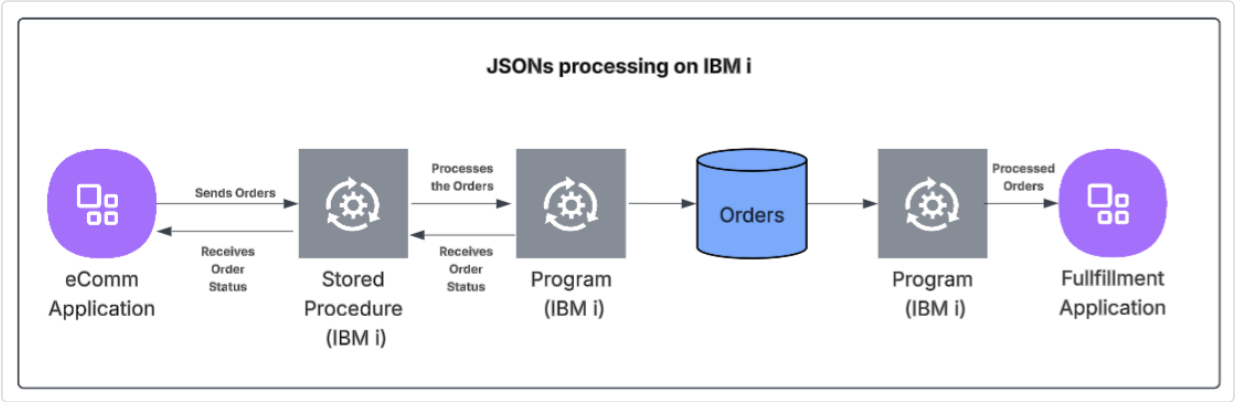


Figure. Native JSON handling eliminates middleware translation layer

If your RPG programs can't consume or produce JSON, you'll need middleware just to translate data formats - adding latency, another moving part, and a dependency on a team that may not understand your business logic.

Native JSON handling on IBM i eliminates that translation layer. Your RPG program receives a JSON order, shreds it into Db2 for i rows, processes the business rules, and returns a JSON response - all without leaving the platform. That's fewer hops, fewer failure points, and business logic that stays where your team knows it best.

There's a portability angle, too. These SQL/JSON functions are part of the ISO SQL standard, so the skills transfer across PostgreSQL, Oracle, and SQL Server - unlike DATA-INTO and YAJL, which are IBM i-specific.

How It Works

Use Case:

- An eCommerce application exchanges Orders with IBM i — all requests and responses are JSON (API payloads or event stream messages)
- We will demonstrate how to:
  - A. Extract a single value with JSON_VALUE
  - B. Parse a JSON payload into rows with JSON_TABLE
  - C. Transform JSON arrays into relational tables
  - D. Return JSON responses
  - E. Read JSON from IFS files

Sample JSON payload for an Order

```
{
  "orderNumber": "W7484738837",
  "customer": {
    "name": "John Doe",
    "email": "jdoe@aol.com",
    "phone": "480-746-7342"
    },
  "products": [
    {
      "productId": "P1001",
      "quantity": 2,
      "price": 19.45
    },
    {
      "productId": "P1022",
      "quantity": 1,
      "price": 24.50
    }
    ],
    "orderTotal": 63.40
}
```

A. Extract Customer Email for validation purposes using JSON_VALUE

```
Dcl-Pr getCustomerEmail EXTPGM('GETCUSTEMAIL');
    jsonData CLOB(500K) CCSID 1208 Const;
    email Varchar(256);
End-Pr;
Dcl-Pi getCustomerEmail ;
    jsonData CLOB(500K) CCSID 1208 Const;
    email Varchar(256);
End-Pi;

// Extract the customer's email from the incoming JSON payload
Exec SQL
    VALUES JSON_VALUE(:jsonData, '$.customer.email' Returning Varchar(256) )
    Into :email;

If Sqlcode = 0;
    // Proceed with email validation logic
Else;
    // Handle error - invalid JSON or missing email
Endif;
Return;
```

B. Parse the incoming JSON Payload and insert into Order Header and Order Detail tables

RPGLE Program to process the Order

```
Dcl-Pr processOrder EXTPGM('PROCESSORDER');
    jsonData CLOB(500K) CCSID 1208 Const;
End-Pr;

Dcl-Pi processOrder ;
    jsonData CLOB(500K) CCSID 1208 Const;
End-Pi;

Dcl-DS orderData Qualified;
    orderNumber Varchar(20);
    customerName Varchar(100);
    customerEmail Varchar(256);
    customerPhone Varchar(20);
    orderTotal Packed(10:2);
End-DS;

Dcl-DS productData Qualified;
    productId Varchar(20);
    quantity Int(10);
    price Packed(10:2);
End-DS;

// Parse the JSON payload
Exec SQL
    Declare ordersCursor Cursor For
      Select
        coalesce(orderNumber, '') as orderNumber,
        coalesce(customerName, '') as customerName,
        coalesce(customerEmail, '') as customerEmail,
        coalesce(customerPhone, '') as customerPhone,
        coalesce(orderTotal, 0) as orderTotal,
        coalesce(productId, '') as productId,
        coalesce(quantity, 0) as quantity,
        coalesce(price,0) as price
    From (
        Select
            x.* from JSON_TABLE(:jsonData,
            '$'
            COLUMNS (
                orderNumber Varchar(20) PATH '$.orderNumber',
                customerName Varchar(100) PATH '$.customer.name',
                customerEmail Varchar(256) PATH '$.customer.email',
                customerPhone Varchar(20) PATH '$.customer.phone',
                orderTotal Dec(10,2) PATH '$.orderTotal',
                NESTED '$.products[*]' COLUMNS (
                    productId Varchar(20) PATH '$.productId',
                    quantity Int PATH '$.quantity',
                    price Dec(10,2) PATH '$.price'
                )
            )) as x ) as y;

Exec SQL
    Open ordersCursor;

DoU Sqlcode = 100;

    Exec SQL
        Fetch Next From ordersCursor Into
            :orderData.orderNumber,
            :orderData.customerName,
            :orderData.customerEmail,
            :orderData.customerPhone,
```

```
            :orderData.orderTotal,
            :productData.productId,
            :productData.quantity,
            :productData.price;

    If Sqlcode = 0;
        // Insert into Order Header and Order lines tables
    Endif;

Enddo;

Exec SQL
    Close ordersCursor;
```

Example B gives you row-by-row control when you need validation or transformation logic in RPG before inserting. If you just need a straight insert with no intermediate processing, Example C is simpler.

C. Extract the Product details and save in Order Details table using the JSON_TABLE function

```
// Transform the JSON array of products into rows in a relational table
Exec SQL
    Insert Into OrderDetails (orderId, productId, quantity, price)
    Select orderNumber, productId, productQty, productPrice
    From JSON_TABLE(:jsonData,
        '$'
        COLUMNS (
            orderNumber Varchar(20) PATH '$.orderNumber',
            NESTED '$.products[*]' COLUMNS (
                productId Varchar(20) PATH '$.productId',
                productQty Int PATH '$.quantity',
                productPrice Dec(10,2) PATH '$.price'
            )
        )) as products;
```

D. Return JSON responses from RPGLE programs

```
Dcl-C cComplete 'C';

// For an OrderID received as input, return the Product details as a JSON response
Exec SQL
   Declare orderResponseJSON Cursor For
    Select JSON_ARRAYAGG(
        JSON_OBJECT(
            'productId' : productId,
            'quantity'  : productQty,
            'price'     : productPrice
        )
    ) as data
    From OrderDetails
    Where orderId = :orderId and
          orderStatus = :cComplete;

Exec SQL
    Open orderResponseJSON;

If Sqlcode <> 0;
    // Handle error
EndIf;

Exec SQL
 Set result sets Cursor orderResponseJSON;
```

> **Note:** This RPG program would typically be wrapped in a stored procedure - the standard way to expose RPG business logic to external callers over SQL, HTTP, or ODBC/JDBC. The stored procedure provides a clean call interface, handles authorization and supports returning result sets (e.g. JSON responses) directly to the caller (in this case the eCommerce application).

E. Read JSON from IFS files

```
// Read a JSON document from an IFS file and parse it
Dcl-S jsonData CLOB(500K) CCSID 1208;
Dcl-S filePath Varchar(256) Value('/home/api/in/W7484738837.json');

// Read the JSON document from the IFS file into a CLOB variable
Exec SQL
    Select LISTAGG(LINE, '')
    Into :jsonData
    From Table(QSYS2.IFS_READ_UTF8(
        PATH_NAME => :filePath
    ));

// Now you can parse the jsonData variable using the techniques shown in the previous examples
```

The job profile needs `*R` on the file and `*RX` on every parent directory — missing authority surfaces as `SQL7064`.

Before passing `jsonData` to any of the JSON functions in the earlier examples, validate that it's well-formed:

```
// JSON_VALUE returns NULL and sets SQLCODE = 0 on a valid-but-missing path.
// It sets SQLCODE = -171 when the first argument is not valid JSON at all.
// A quick probe before the real parse catches malformed input early.
Dcl-S isValid Varchar(5);

Exec SQL
    VALUES JSON_VALUE(:jsonData, '$' Returning Varchar(5))
    Into :isValid;

If Sqlcode = -171;      // SQL0171 - jsonData is not valid JSON
    // Handle error - log, signal caller, return early
Endif;
```

This is worth adding to any RPG program that accepts JSON from an external caller. Malformed JSON passed straight to `JSON_TABLE` raises `SQL0104` with a message that points at the SQL statement, not the bad input — which makes it hard to diagnose in a production job log.

Key Takeaways

- **Pick the right tool for the shape of your work** — SQL JSON functions for Db2 for i-bound data, DATA-INTO for RPG variable processing
- **Always specify CCSID 1208** on CLOB parameters receiving external JSON
- **JSON_TABLE replaces pages of parsing logic** with one SQL statement
- **Wrap your RPG in a stored procedure** — it provides the standard call interface for external consumers and supports returning JSON result sets
- **Validate JSON before parsing** — probe with `JSON_VALUE` first; `SQLCODE -171` means the input isn't valid JSON; catching it there gives you a clean error path before `JSON_TABLE` or `JSON_ARRAYAGG` see the data
- **IFS reads require IFS authority** — the job profile needs `*R` on the file and `*RX` on every parent directory; missing authority surfaces as `SQL7064`, not a permission message

Data-Centric Programming on IBM i

Database Definition and Design

How do I define tables with SQL?

A SQL table is the container for storing a physical data set. Tables are defined using the CREATE TABLE statement. The CREATE TABLE statement is part of the SQL Data Definition Language (DDL). SQL DDL is a subset of SQL used to define, modify, and manage the structure of database objects.

A table is SQL's equivalent of an IBM i physical file object. While both objects can used to store data, SQL tables have a clear advantage in terms of features and functions over a physical file. The following CREATE TABLE example demonstrates one of the biggest differences in that SQL table names are not limited to 10 character object names like a physical file. This example uses a descriptive table name, EMPLOYEE_MASTER, which is 15 characters in length.

```
CREATE TABLE EMPLOYEE_MASTER (
  EMPNO CHAR(6) NOT NULL,
  FIRST_NAME VARCHAR(20) NOT NULL,
  MIDINIT CHAR(1) NOT NULL WITH DEFAULT,
  LAST_NAME VARCHAR(30) NOT NULL,
  DEPT_NUMBER CHAR(3) DEFAULT NULL,
  HIRE_DATE DATE DEFAULT NULL,
  JOB_CODE CHAR(8) DEFAULT NULL,
  GENDER CHAR(1) NOT NULL,
  BIRTH_DATE DATE NOT NULL,
  SALARY DECIMAL(9, 2) NOT NULL WITH DEFAULT 50000
)
```

While SQL table names can be up to 128 characters in length, it's best for performance to limit your SQL table names to 30 characters or less. SQL supporting object names longer than 10 characters brings up an interesting challenge because IBM i system commands such as Save Object (SAVOBJ) are limited to 10 characters.

Due to the fact that the table name in this example (i.e., EMPLOYEE_MASTER) is greater than 10 characters in length, Db2 for i will generate a system name for usage with IBM i system commands. Db2 system name generation algorithm takes the first 5 characters of the table (i.e., EMPLO) and then uses a 5 digit number for the suffix. The 5 digit number is incremented for each SQL object that starts with the same 5 characters to guarantee that the system name will be unique within the library or schema that will contain the SQL table.

Assuming that the EMPLOYEE_MASTER is the first table created within the database, the generated system name will be: EMPLO00001. This generated name can be used on IBM i commands and will be used on interfaces or commands that return a list of objects such as the Display Library (DSPLIB) command. Obviously, this generated system name is not user friendly. In addition, the generated system name can vary from system to system depending on the order your SQL objects get created. Let's say the employee database has a second table named EMPLOYEE_EMAIL in addition to EMPLOYEE_MASTER. On the development system, EMPLOYEE_MASTER was created first and EMPLOYEE_EMAIL table created second. The system names for these two tables respectively would be EMPLO00001 and EMPLO00002. Let's assume that on the test system, the table creation process is reversed, and the EMPLOYEE_EMAIL table is created first. As a result, the EMPLOYEE_EMAIL table will have a system name of EMPLO00001 on the test system and EMPLO00002 on the development system.

This potential difference in system names is not ideal, so it's strongly recommended to provide a system name for Db2 to use instead of having Db2 for i generate a name. A system name can be provided using the FOR SYSTEM NAME clause on your SQL CREATE statements:

```
CREATE TABLE employee_master FOR SYSTEM NAME empmast …
```

Supplying your own system names has two benefits. First, that name is almost certain to be more user-friendly. Second, that system name will always be used for that object.

While both the SQL table name and system name can be used when retrieving or changing data in table with SQL, it's recommended that application programs use the longer, more descriptive SQL table name instead of the system name.

Column names in an SQL table have the same length advantage. The FOR COLUMN clause can be used to supply a name (e.g., PHONE_NUMBER FOR COLUMN PHONENO CHAR(4)) instead of relying on Db2 to generate a name. Supplying a shorter name for long SQL column names is only beneficial for those high-level language programs that need to access an SQL-created table using record-level access and require column names that are more user-friendly than the system generated names.

Here's an updated version of the DDL for the EMPLOYEE_MASTER example which supplies system names for the table and column names instead of relying on generated system names.

```
CREATE TABLE EMPLOYEE_MASTER FOR SYSTEM NAME EMPMAST (
  EMPNO CHAR(6) NOT NULL,
  FIRST_NAME VARCHAR(20) NOT NULL,
  MIDINIT CHAR(1) NOT NULL WITH DEFAULT,
  LAST_NAME VARCHAR(30) NOT NULL,
  DEPT_NUMBER FOR COLUMN DEPTNUM CHAR(3) DEFAULT NULL,
  HIRE_DATE DATE DEFAULT NULL,
  JOB_CODE CHAR(8) DEFAULT NULL,
  GENDER CHAR(1) NOT NULL,
  BIRTH_DATE DATE NOT NULL,
  SALARY DECIMAL(9, 2) NOT NULL WITH DEFAULT 50000
)
```

Using this example DDL, one can also see how columns are defined in an SQL table. After the column name, the data type and length (if applicable) for the column are specified. After the type information is the default clause. The default clause not only allows the specification of a default value for a column but it also controls if the column is allowed to store null values.

By default, any column in an SQL table is allowed to contain null values unless the NOT NULL clause is specified. In this case, EMPNO is a column which cannot store null values. In addition, a value must always be provided on an insert operation because a WITH DEFAULT clause is not included on this column definition. In contrast, the MIDINIT column doesn't allow nulls, but Db2 will generate a default value based on the data type (e.g., blanks for character type) when a value is not provided for the column on an insert. The SALARY column uses the WITH DEFAULT clause to provide a user-supplied default value of 50000 when a value is not provided on an insert.

When designing which SQL tables to create to support your application, the database table implementation should be normalized to at least third normal form.

What data type should I use for dates?

Before you learn about the concept of data-centric programming, it's helpful to understand the traditional approach for developing applications on IBM i – a program-centric approach. With this methodology, business rule enforcement and data access methods are embedded within the application program logic.

The developer must decide which data access method (e.g., keyed-access or sequential) to use for each of the required files and add code to request that method. If there are multiple files being processed, the program is also responsible for determining the order that the files are processed in. In addition, program code must be written to enforce business rules associated with the data files being processed. An example of business rules that might be enforced by the program are ensuring the requested order quantity is within the allowed quantity range for the item being ordered or that an order record references a valid customer record (i.e., referential integrity).

Monolithic applications almost employ a program-centric design, but even modular applications can include a program-centric approach. The following figure shows an example of the structure of an application that is designed using a program-centric approach. The gears in this figure represent the data access and business logic encapsulated within the program object.

Data-Centric Programming Concepts

What is data-centric programming?

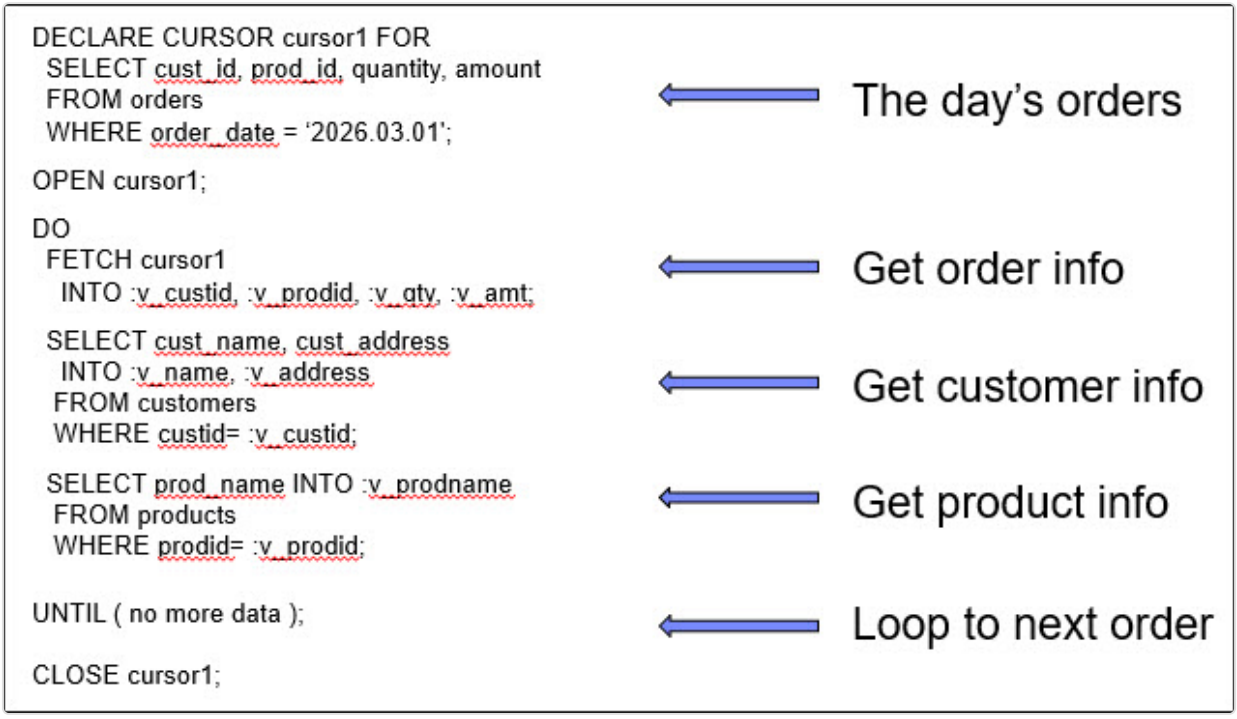


Figure. Program-centric approach

As you might notice, the program-centric approach has many disadvantages:

- The application is not taking advantage of database management system (DBMS) functions to reduce the amount of application code that needs to be written by the developer.
- Business rules can be easily bypassed with any tools or interfaces that directly change data in the files. This exposure can lead to data integrity issues in the application database.
- If a new interface (mobile, web, etc.) needs to be supported by the applications, the data access and business rule logic may have to be duplicated in multiple programs.
- Performance can be sub-optimal if the developer chooses an inefficient data access method.

Conversely, data-centric programming involves moving more of the business rules and data access logic into the DBMS. With the data centric approach, DBMS features are utilized to enforce business rules at the database layer instead of the program layer. Figure 2 shows the data-centric programming approach at a high-level.

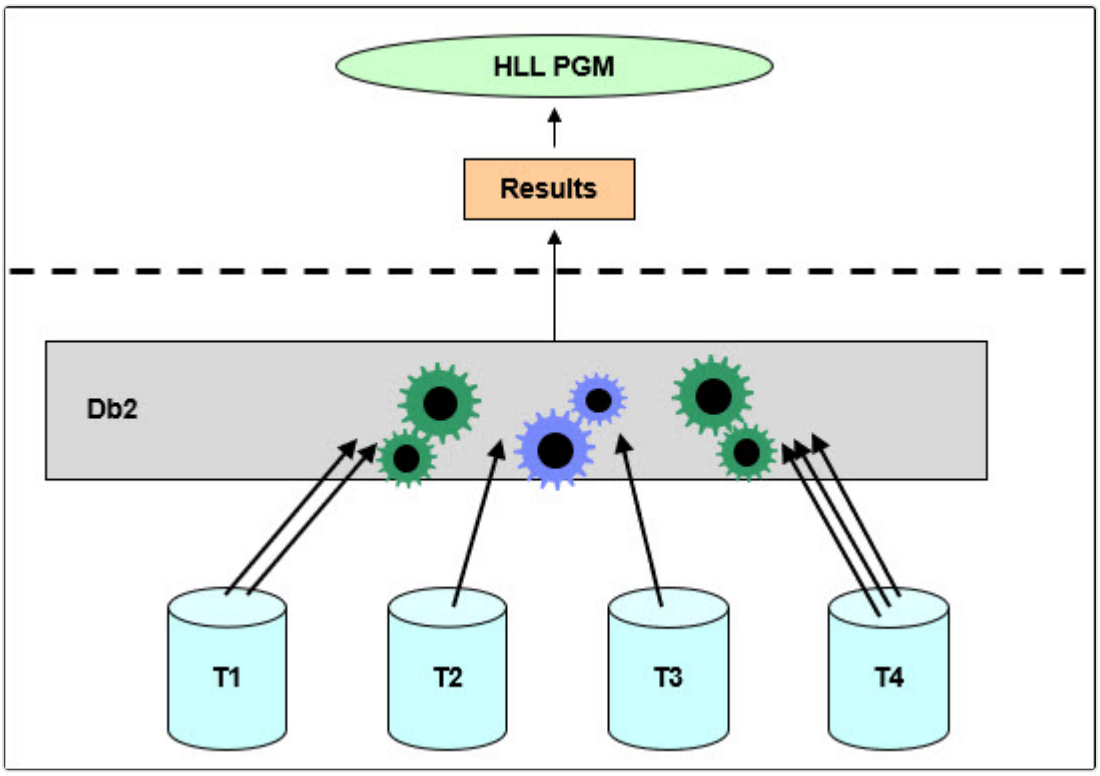


Figure. Data-centric approach

Data-centric programming aims to take advantage of the DBMS to reduce the amount of program code that needs to be written, specifically with the following items:

- When data access is performed with SQL statements, the Db2 query optimizer is responsible for determining the most efficient data access method. The query optimizer also can advise the creation of indexes to improve data access efficiency.
- The usage of constraints allows you to move the enforcement of simple business rules from applications to Db2. For example, a check constraint can be used to enforce the allowing range of values that can be stored in an order quantity column.
- DBMS triggers allow you to move more complex business rules from the application layer to Db2 as well. With a database trigger, a program is called automatically by Db2 before or after database change operation to enforce a more complex business rule.
- Multiple Db2 features can be utilized to generate the next value for a business identifier such as order number or to generate internal keys for a primary key in a normalized database.

A data-centric approach is more easily implemented on Db2 for i database using the SQL interface because some of the highlighted Db2 features are only supported by SQL. The goal of data-centric programming is to use SQL to have Db2 do more work on behalf of the application.

Data Integrity and Constraints

How to Maintain Data Integrity with Constraints

What This Covers

You'll learn how to use database constraints to protect data integrity in IBM i applications, understand the three types of constraints available, and implement a "do no harm" approach when adding constraints to existing systems.

Before You Start

- ☐ Understand your existing business rules and how they're currently enforced in application code
- ☐ Review your database tables to identify relationships between data
- ☐ Verify that existing data doesn't violate the constraints you plan to implement
- ☐ Have access to SQL or DDS commands to create and modify constraints

Understanding Constraints

Constraints are rules created and managed by the database manager that protect data integrity regardless of how applications access the database. They integrate business logic directly into the database, making it the foundation for data-centric programming.

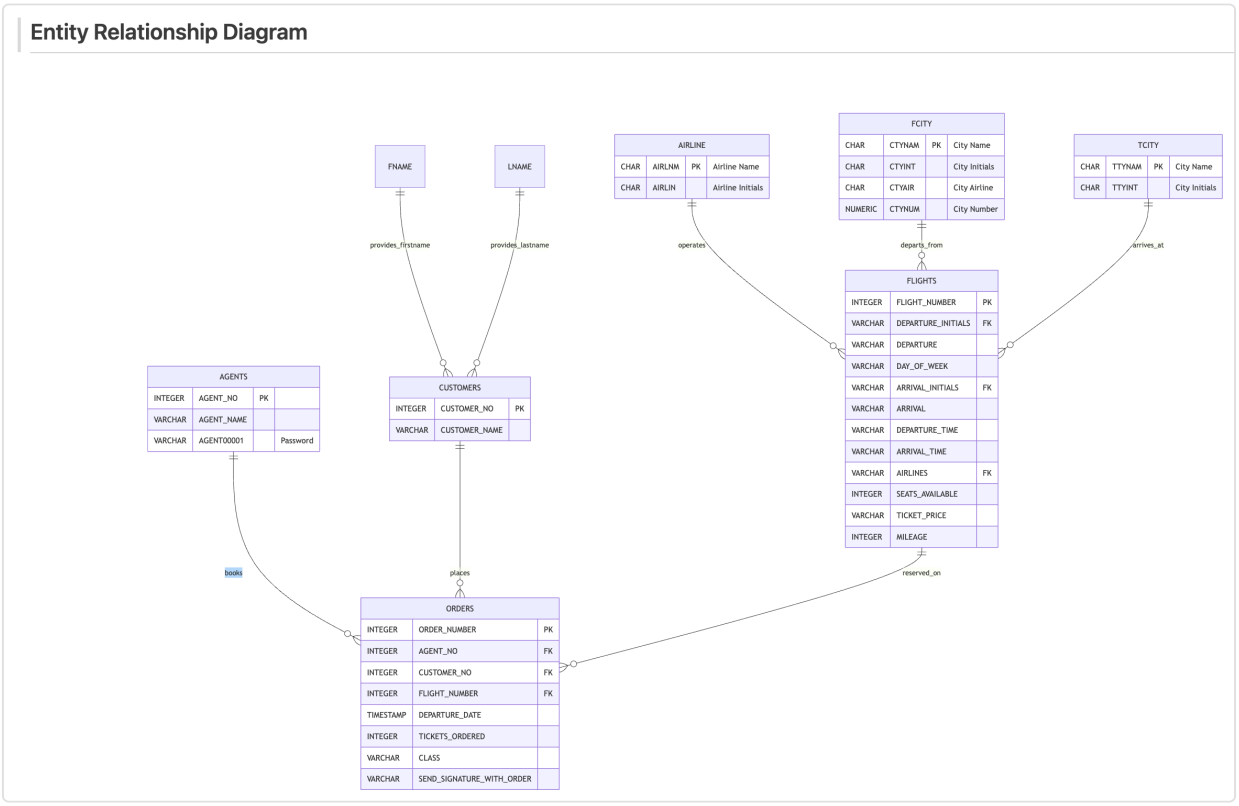


Figure. Referential integrity

Three Types of Constraints

1. Primary and Unique Key Constraints These prevent duplicate entries in columns. Primary keys don't allow NULL values, while unique constraints do. They ensure that no matter what application modifies the file, the database never contains duplicate keys.

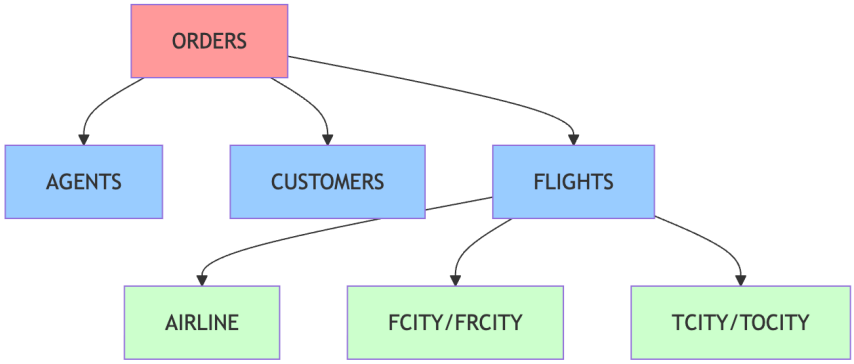
2. Referential Integrity Constraints These define relationships between tables and ensure they remain valid. For example, they prevent orders from being inserted for non-existent customers. A foreign key in one table must match a parent key (primary or unique key) in another table.

Table Relationships

Primary Relationships

ORDERS (Central Transaction Table)

The ORDERS table is the central transaction table that links all core entities:



Relationship Details:

1. **ORDERS → AGENTS**
   - Foreign Key: `AGENT_NO`
   - Relationship: Many-to-One
   - Business Rule: Each order must be booked by an agent
2. **ORDERS → CUSTOMERS**
   - Foreign Key: `CUSTOMER_NO`
   - Relationship: Many-to-One
   - Business Rule: Each order belongs to one customer
3. **ORDERS → FLIGHTS**
   - Foreign Key: `FLIGHT_NUMBER`
   - Relationship: Many-to-One
   - Business Rule: Each order is for one specific flight

Figure. Referential integrity concepts

3. Check Constraints These ensure column data doesn't violate defined rules or restrict values to an allowed set. For example, you can enforce that order quantities are always greater than zero and less than 1000, or ensure state codes are valid.

Steps

1. **Define Primary and Unique Keys**
   - Create primary keys for tables that need unique identifiers with no NULL values
   - Use unique constraints when NULL values are acceptable
   - Example SQL syntax:

```
CREATE TABLE customers (
  customer_no INTEGER,
  customer_name VARCHAR(64),
  CONSTRAINT customer_no PRIMARY KEY (customer_no)
)
```

- For existing tables, use ALTER TABLE to add constraints:

```
ALTER TABLE customers
ADD CONSTRAINT customers_unq_customer_number
UNIQUE KEY(customer_no);
```

2. **Implement Referential Integrity**
   - Identify parent-child relationships between tables
   - Define foreign key constraints linking child tables to parent tables
   - Choose a delete rule:
     - **RESTRICT**: Prevents deletion of parent rows if dependent rows exist
     - **CASCADE**: Deletes all dependent rows when parent is deleted
   - The parent key must be unique, ascending, and cannot contain null values
   - Foreign key values must match parent key values or be null
   - Example: Link orders to customers so orders can't exist without valid customer numbers
3. **Add Check Constraints for Business Rules**
   - Identify well-defined, consistently enforced business rules
   - Create check constraints to validate data at the database level
   - Focus on rules that are critical to data integrity
   - Examples include:
     - Numeric ranges (quantity > 0 AND quantity < 1000)
     - Valid value sets (state codes, status values)
     - Date validations (end_date > start_date)
4. **Apply the "Do No Harm" Approach**
   - Start with constraints for well-defined, consistently enforced business rules
   - Avoid imposing constraints if existing data contains violations
   - Leave existing application validation code intact initially to avoid breaking functionality
   - Accept the slight performance penalty of checking twice (once in constraint, once in application)
   - Consider removing redundant application code only after constraints are proven stable
   - For cascading constraints that perform actions (like deleting related rows), remove duplicate application code to avoid conflicts
5. **Handle Constraint Violations in Applications**
   - Decide whether to check for violations proactively or handle them reactively
   - If already accessing related data (like chaining to a master file), handle validation using existing methods
   - For new applications, consider checking for constraint violations instead of traditional validation
   - Be aware that SQL, RPG, and COBOL signal constraint violations, but determining which specific constraint failed requires using the Receive Program Message API
   - Remember that violations are reported when the first one is encountered, which can complicate validating entire panels of information
6. **Enable or Disable Constraints as Needed**
   - Use the Change PF Constraint (CHGPFCST) command to enable or disable referential integrity constraints
   - This flexibility allows you to temporarily disable constraints during data migrations or bulk operations

    - Always re-enable constraints after completing operations that required them to be disabled

Verify It Worked

- Test insert, update, and delete operations that should be blocked by constraints
- Verify that constraint violations are properly signaled to applications
- Confirm that valid operations complete successfully
- Check that cascading deletes (if implemented) remove all dependent rows
- Review application logs to ensure constraint violations are handled appropriately
- Validate that existing applications continue to function correctly with constraints in place

Key Takeaways

- Constraints protect data integrity regardless of how applications access the database, even if applications are bypassed or contain errors
- Start with well-defined business rules that are consistently enforced across all applications
- The "do no harm" approach means keeping existing validation code intact while adding database-level constraints
- Constraints make business rules accessible to applications on other platforms and written in different languages
- Primary and unique keys prevent duplicate data, referential integrity maintains relationships, and check constraints enforce business rules
- Even if you never check for constraint violations in application code, implementing constraints improves data protection and application portability
- Constraints are a standard database capability that improves overall system reliability and data quality

SQL Error Handling and Services

How do I handle SQL status and errors via RPG?

What This Covers

This procedure explains how to properly handle SQL status codes and errors (SQLCOD != 0) when using embedded SQL in free-form RPG programs. You'll learn to detect, interpret, and respond to SQL execution results to build robust database applications.

Before You Start

- Basic embedded SQL syntax in RPG
- Understanding of SQLCOD (return code) and SQLSTATE (standardized status)

Steps

1. Declare SQL Communication Area (SQLCA)

- Include the SQLCA data structure in your program's definition specifications
- The SQLCA is automatically updated after each SQL statement execution
- Key fields: SQLCOD (return code) and SQLSTATE (standardized status)

```
// SQLCA is implicitly available when using embedded SQL
// No explicit declaration needed in modern RPG
```

2. Execute SQL Statement and Check SQLCOD

- Place your SQL statement within EXEC SQL and END-EXEC delimiters
- Immediately after execution, check the SQLCOD value
- SQLCOD = 0 indicates successful execution

- SQLCOD > 0 indicates a warning condition
- SQLCOD < 0 indicates an error condition

```
dcl-s customerName varchar(50);
dcl-s customerId int(10);

customerId = 12345;

exec sql
  SELECT CUSTNAME INTO :customerName
  FROM CUSTOMERS
  WHERE CUSTID = :customerId;

if SQLCOD <> 0;
  // Handle non-successful execution
endif;
```

3. Implement Basic Error Handling Logic

- Use conditional logic to branch based on SQLCOD value
- Handle three scenarios: success (0), warning (> 0), and error (< 0)
- Common positive codes: +100 (no data found), +0 (success with warnings)
- Common negative codes: -204 (object not found), -803 (duplicate key)

```
if SQLCOD = 0;
  // Success - continue processing
  dsply ('Customer found: ' + customerName);
elseif SQLCOD = 100;
  // No data found - handle gracefully
  dsply 'Customer not found';
elseif SQLCOD < 0;
  // Error occurred - log and handle
  dsply ('SQL Error: ' + %char(SQLCOD));
endif;
```

4. Use SQLSTATE for Standardized Error Handling

- SQLSTATE provides a 5-character standardized error code
- First 2 characters indicate error class (00=success, 01=warning, 02=not found)
- More portable across different database systems than SQLCOD
- Recommended for new development

```
exec sql
  UPDATE CUSTOMERS
  SET CUSTNAME = :customerName
  WHERE CUSTID = :customerId;

select;
  when SQLSTATE = '00000';
    // Success
    dsply 'Update successful';
  when SQLSTATE = '02000';
    // No rows updated
    dsply 'Customer not found';
  when %subst(SQLSTATE:1:2) = '23';
    // Integrity constraint violation
    dsply 'Constraint violation';
  other;
    // Other errors
    dsply ('SQL Error: ' + SQLSTATE);
endsl;
```

5. Create a Centralized Error Handling Procedure

- Build a reusable procedure to handle SQL errors consistently
- Log errors with context information for troubleshooting

```
dcl-proc handleSQLError;
  dcl-pi *n ind;
    operation varchar(100) const;
  end-pi;

  if SQLCOD <> 0;
    // Log to job log and error table
    exec sql
      INSERT INTO ERRORLOG (TIMESTAMP, PROGRAM, OPERATION,
                            SQLCODE, SQLSTATE)
      VALUES (CURRENT_TIMESTAMP, 'MYPGM', :operation,
              :SQLCOD, :SQLSTATE);
    return *off;
  endif;
  return *on;
end-proc;
```

6. Handle Common Error Scenarios

Common SQLCOD values: -204 (object not found), -803 (duplicate key), -530 (foreign key violation), -551 (authorization failure), +100 (no data found)

```
exec sql
  INSERT INTO CUSTOMERS (CUSTID, CUSTNAME)
  VALUES (:customerId, :customerName);

select;
  when SQLCOD = 0;
    dsply 'Customer added successfully';
  when SQLCOD = -803;
    dsply 'Customer ID already exists';
  when SQLCOD = -530;
    dsply 'Invalid reference to related table';
  other;
    handleSQLError('INSERT CUSTOMERS');
endsl;
```

7. Use SQL Services for Enhanced Error Information

IBM i provides diagnostic resources for detailed error information:

- **SYSTOOLS.SQLCODE_INFO** - Table function returning message text and cause for SQLCODE values
- **QSYS2.SQLSTATE_INFO** - Table containing standardized SQLSTATE descriptions

```
dcl-proc logSQLError;
  dcl-pi *n;
    operation varchar(100) const;
  end-pi;

  dcl-s sqlcodeMsg varchar(1000);
  dcl-s sqlstateMsg varchar(1000);

  // Get detailed error descriptions
  exec sql
    SELECT MESSAGE_TEXT INTO :sqlcodeMsg
    FROM TABLE(SYSTOOLS.SQLCODE_INFO(:SQLCOD));

  exec sql
    SELECT MESSAGE_TEXT INTO :sqlstateMsg
    FROM QSYS2.SQLSTATE_INFO
    WHERE SQLSTATE = :SQLSTATE;

  // Log comprehensive error information
  exec sql
    INSERT INTO ERRORLOG (TIMESTAMP, PROGRAM, OPERATION,
                          SQLCODE, SQLCODE_MSG, SQLSTATE, SQLSTATE_MSG)
    VALUES (CURRENT_TIMESTAMP, 'MYPGM', :operation,
            :SQLCOD, :sqlcodeMsg, :SQLSTATE, :sqlstateMsg);
end-proc;
```

Use these resources to provide user-friendly messages while logging technical details:

```
if SQLCOD <> 0;
  exec sql
    SELECT MESSAGE_TEXT INTO :technicalDetails
    FROM TABLE(SYSTOOLS.SQLCODE_INFO(:SQLCOD));

  // Show user-friendly message, log technical details
  dsply 'An error occurred. Please contact support.';
  logSQLError('INSERT ORDERS: ' + technicalDetails);
endif;
```

Verify It Worked

- Compile your RPG program without SQL precompiler errors
- Test with valid data - verify SQLCOD = 0 and successful execution
- Test with invalid data (non-existent keys) - verify SQLCOD = 100 is handled
- Test constraint violations - verify negative SQLCOD values are caught
- Review job log or error log table for proper error recording
- Confirm program doesn't abnormally end on SQL errors

Key Takeaways

- Always check SQLCOD or SQLSTATE after every SQL statement execution
- SQLCOD = 0 means success; > 0 is warning; < 0 is error
- SQLSTATE provides standardized, portable error codes across platforms
- Implement centralized error handling for consistency and maintainability
- Log errors with context for easier troubleshooting and debugging
- Handle common scenarios explicitly (not found, duplicates, constraints)
- Use WHENEVER for automatic error handling when appropriate
- Leverage SYSTOOLS.SQLCODE_INFO (table function) for detailed SQLCODE descriptions
- Use QSYS2.SQLSTATE_INFO (table) for standardized SQLSTATE information
- Combine both resources to provide user-friendly error messages and comprehensive logging
- Test error paths as thoroughly as success paths in your applications

What are Db2 for i (SQL) Services?

Why is SQL better than DDS?

What It Is

Dates are (too) often stored as NUMERIC(8,0), DECIMAL(7,0), or CHAR(xx) in DDS files, but SQL offers a native DATE data type.

Why It Matters

Using proper DATE data types provides access to numerous built-in SQL date functions, eliminates custom date calculation code, and ensures consistency with other DBMS platforms. This best practice also applies to TIMESTAMPs and other data types. You will benefit from using proper data types and aligning the data model and applications to use identical data types and precisions.

How It Works

Solution: Define date columns as DATE data type in SQL tables.

Benefits:

- Convert dates to other formats (Julian, ISO, etc.)
- Calculate days from current date
- Automatic leap year handling
- Date comparisons and arithmetic
- Faster execution than custom functions
- Cross-platform compatibility

Takeaways

- Modernization plans should include this topic
- If the data model cannot be changed, a proper data type can be provided via an SQL VIEW

Why is SQL better than DDS?

Db2 for IBM i supports SQL Data Definition Language (DDL) and Data Description Specifications (DDS) as the programming techniques for defining a data model on IBM i. This section compares and contrasts these two techniques, making clear the advantages provided by SQL DDL.

These SQL DDL versus DDS comparison points will be examined in the following sections.

Why is SQL better than DDS:

1. **Features**
2. **Industry Standard**
3. **Tooling**
4. **Data Type Integrity**

Features

DDS has existed on IBM i longer than SQL DDL, and there's no reason why IBM i would deprecate support for this popular technology. That being said, DDS is not being enhanced by Db2 for i. SQL is the strategic interface for Db2 for i.

As a result, the SQL capabilites in Db2 for i are continually being enhanced. The Db2 for i team has delivered a steady stream of SQL enhancements in IBM i operating system releases, and via the IBM i Technology Refreshes. Db2 for i enhancements can be seen on the 'Db2 for i - Technology Updates' website: https://www.ibm.com/support/pages/node/1116645/

Some of the numerous Db2 for i enhancements pertain to the data model, and are only available if SQL is being used. Here are some examples of those features:

- **Column types**: BOOLEAN, BLOB, CLOB, DBCLOB, XML
- **Column attributes**: Generated and Hidden columns
- **AI**: Watson Geospatial Analytics user-defined types

If an IBM i client is using only DDS-based data models, they have a lower return on investment posture, because they're unable to use features that are available within Db2 for i.

Standards

SQL is an industry standard supported by Db2 for i and most other relational database products. Although not every relational database conforms to the SQL standards like Db2 for i, SQL skills and code can be easily transferred from one database to another. Embracing an SQL industry standards provides many tangible benefits to the business:

- Easier to find and on-board new staff members because SQL skills are readily available in the industry.
- Faster delivery on IT requirements because modern SQL features allow the Db2 engine to do more work on behalf of the application
- Ability to leverage more database tools and integrate with Open Source, Ansible, and other technologies.

Tooling

IBM i Access Client Solutions (ACS), Visual Studio Code4i, and Db2 for i Services provide extensive support for managing and tuning SQL-based data models.

Industry database tooling would typically expect SQL data models, resulting is a less-than-ideal user experience when pointed to a DDS-based data model on IBM i.

As shown in the following figure, the ACS Schema dialog includes powerful capabilities for the Database Engineer and Application Developers.

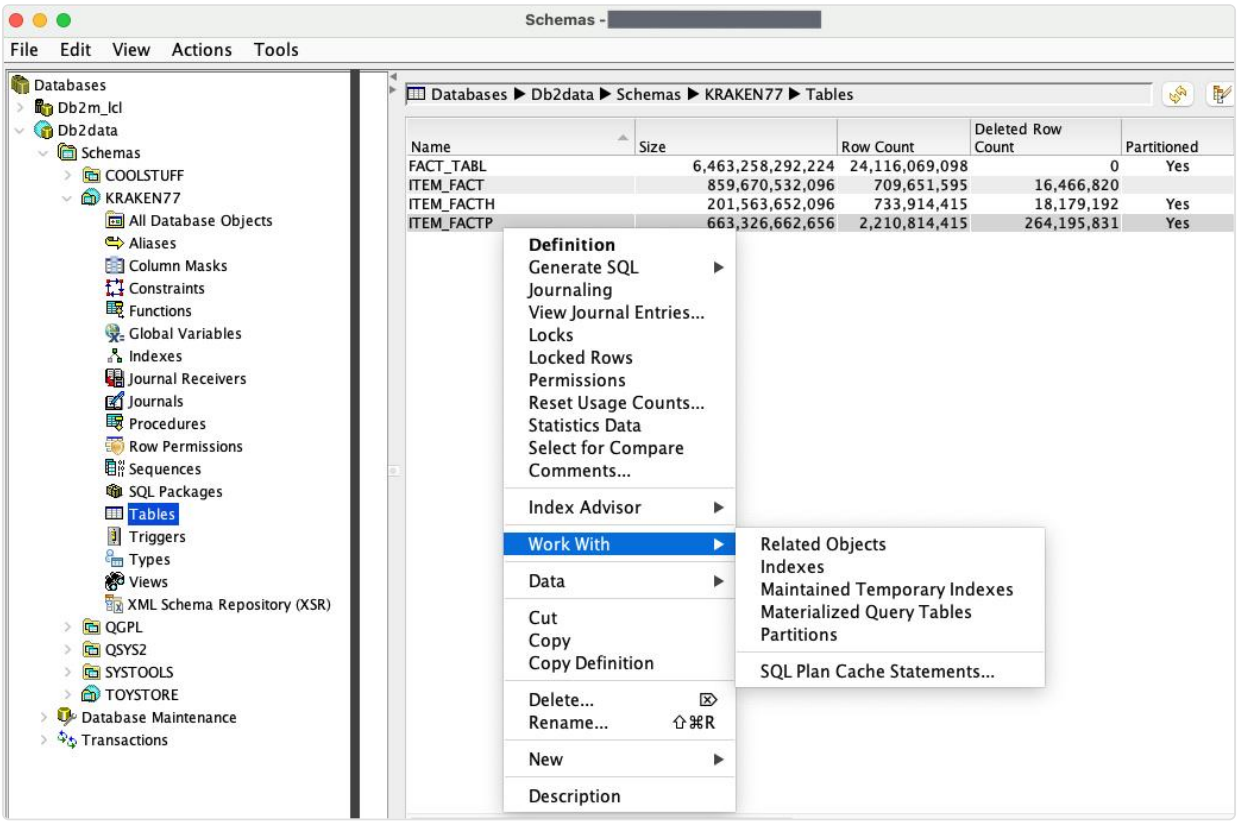


Figure. ACS Schema Dialog

Data Type Integrity

SQL protects the integrity of business data being written to columns in your tables.

The SQL Standards prohibit data from being inserted or updated into an SQL table, when the data does not adhere to the rules of the target column's data type. DDS-created physical files have no innate protection within the database, leaving the task of adherance to data type validity solely with the responsibility of the application. For example, an application using record-level access is allowed to write dirty decimal data to decimal columns in a physical file. This write operation from the same application would fail if the target was an SQL table.

Database Re-engineering for IBM i

Database Re-engineering Concepts

What is Re-engineering vs Re-representing?

Introduction

As IBM i environments evolve, organizations face increasing pressure to modernize their database structures while maintaining stability, performance, and business continuity. Many systems running today were originally designed decades ago using DDS-based physical and logical files, tightly coupled RPG programs, and record-level access patterns. While these systems remain reliable and performant, they often lack the flexibility required for modern integration, analytics, API-driven applications, and cloud-connected architectures.

Database modernization efforts on IBM i typically fall into two major categories: **re-representing** the database or **re-engineering** the database. Although these terms are sometimes used interchangeably, they represent fundamentally different strategies with different levels of impact, risk, and long-term value.

Understanding the difference between re-representing and re-engineering is critical for architects, developers, and technical leaders who are planning database modernization initiatives. Choosing the wrong approach can lead to unnecessary risk, excessive cost, or limited modernization benefits.

This section explains the distinction between these two approaches and provides guidance on when each strategy should be used.

What is Database Re-representing?

Re-representing is the process of changing how a database is defined or exposed without significantly altering its underlying structure or business logic.

The goal of re-representing is to modernize the database interface while preserving existing behavior and application compatibility. This approach focuses on improving accessibility, readability, and interoperability while minimizing disruption to existing RPG programs and business processes.

Common Re-representing Activities

- Converting DDS physical and logical files to SQL tables and views
- Adding SQL DDL definitions while keeping the same record structure
- Creating SQL views over existing physical files
- Adding constraints and metadata without changing application logic
- Exposing data through services or APIs
- Adding meaningful labels and descriptions to every field and column
- Enabling modern tools such as ACS Run SQL Scripts and SQL Services

In this approach, the database still behaves the same way from an application perspective. RPG programs continue to work with minimal or no changes, and the data model remains largely intact.

Key Characteristics

- Low risk
- No required application changes
- Preserves existing database structure
- Focus on SQL enablement and modernization
- Improves integration and tooling support
- Faster implementation timeline

Re-representing does not require application changes because the underlying database structure and access patterns remain intact. Existing RPG programs continue to operate against the same physical structures, while SQL definitions and metadata provide a modern representation of the data. Organizations may optionally update applications to use SQL views or a virtual data model over time, but this is not required for re-representing to be successful. Re-representing is often the first step in IBM i database modernization because it allows organizations to move from DDS to SQL while maintaining operational stability.

What is Database Re-engineering?

Re-engineering is the process of redesigning the database structure to improve architecture, performance, scalability, and maintainability.

Unlike re-representing, re-engineering involves fundamental changes to how data is structured, stored, and accessed. This approach focuses on aligning the database with modern relational design principles and business requirements.

Common Re-engineering Activities

- Redesigning tables and relationships
- Normalizing or denormalizing data
- Removing legacy constraints and dependencies
- Eliminating redundant fields and files
- Introducing surrogate keys and referential integrity
- Breaking large monolithic files into logical entities
- Redesigning access patterns
- Refactoring RPG programs and SQL logic
- Aligning the database with domain-driven design
- Optimizing for APIs, microservices, and analytics

In this approach, the database structure changes significantly, and applications must be updated to work with the new design.

Key Characteristics

- Greater risk, greater reward
- Requires application changes
- Structural database redesign
- Improved long-term maintainability
- Enables modern architectures
- Longer implementation timeline
- Requires detailed planning and testing

Re-engineering is typically performed when the existing database structure limits business growth, performance, or modernization efforts.

Core Difference Between Re-representing and Re-engineering

The simplest way to understand the difference is:

> **Re-representing changes how the database looks Re-engineering changes how the database works**

Aspect	Re-representing	Re-engineering
Database structure	Mostly unchanged	Redesigned
Application impact	Minimal	Significant
Risk level	Low	Higher
Timeline	Shorter	Longer
Business disruption	Minimal	Moderate to high
Goal	Modern interface and SQL enablement	Modern architecture and design
RPG program changes	Rare	Common
Long-term transformation	Limited	High

Re-representing focuses on **modern visibility and accessibility**, while re-engineering focuses on **modern architecture and sustainability**.

When to Use Re-representing

Re-representing is the best choice when:

- The database structure is stable and reliable
- RPG applications must remain unchanged
- The organization wants to adopt SQL gradually
- Risk must be minimized
- Modern tools and integrations are needed
- DDS conversion is the primary goal
- Quick wins are required

This approach allows organizations to modernize their database environment without disrupting core business operations.

It is often used as a **foundation for future re-engineering efforts**.

When to Use Re-engineering

Re-engineering is the better choice when:

- The database design limits performance
- Data structures are difficult to maintain
- Large physical files contain mixed business entities
- Integration with modern systems is difficult
- Analytics and reporting are constrained
- Business logic is tightly coupled to file structures
- Long-term modernization is a strategic priority

This approach provides the greatest long-term value but requires strong governance, planning, and executive support.

Recommended Approach

Most successful IBM i modernization strategies follow a phased model rather than choosing only one approach.

Phased Strategy

1. Re-represent DDS files into SQL
2. Stabilize and validate the environment
3. Introduce constraints and metadata
4. Gradually re-engineer high-value database areas
5. Align applications with the new design

This hybrid model reduces risk while enabling meaningful modernization.

- Re-representing creates stability and visibility
- Re-engineering creates transformation and sustainability
- Together they provide a practical modernization path

Summary

Database modernization on IBM i requires a clear understanding of the difference between re-representing and re-engineering.

- Re-representing focuses on SQL enablement and minimal disruption
- Re-engineering focuses on structural redesign and long-term improvement
- Both approaches play important roles in modernization
- A phased strategy provides the best balance of risk and value

Organizations that clearly distinguish between these two strategies are better positioned to modernize their IBM i databases in a controlled, predictable, and sustainable way.

Transaction Management and Data Access

How Commitment Control Works

What This Covers

Understanding the core functionality of commitment control, what guarantees it provides, and how to start using it with the STRCMTCTL command.

Before You Start

- ☐ Understanding of basic commitment control concepts
- ☐ Knowledge of database transactions
- ☐ Access to IBM i system with appropriate permissions

Core Functionality

Commitment control ensures that either the entire group of individual changes occurs on all systems that participate or that none of the changes occur.

Commitment control allows you to complete the following tasks:

- Ensure that all changes within a transaction are completed for all resources that are affected.
- Ensure that all changes within a transaction are removed if processing is interrupted.

- Remove changes that are made during a transaction when the application determines that a transaction is in error.

Application Restart Capability

You can also design an application so that commitment control can restart the application if a job, an activation group within a job, or the system ends abnormally. With commitment control, you can have assurance that when the application starts again, no partial updates are in the database because of incomplete transactions from a prior failure.

Starting Commitment Control

The Start Commitment Control (STRCMTCTL) command makes it possible for programs that run within a job to make changes under commitment control. When commitment control is started by using the STRCMTCTL command, the system creates a commitment definition.

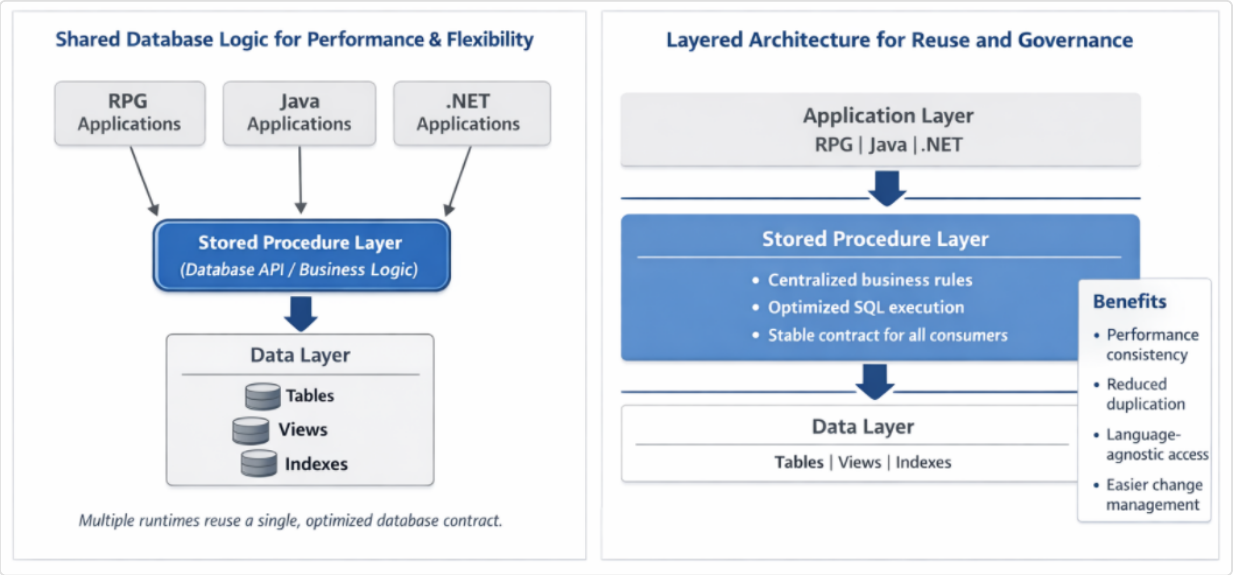


Figure. Commitment control diagram

Commitment Definitions

Each commitment definition is known only to the job that issued the STRCMTCTL command. The commitment definition contains information pertaining to the resources that are being changed under commitment control within that job. The commitment control information in the commitment definition is maintained by the system as the commitment resources change. The commitment definition is ended by using the End Commitment Control (ENDCMTCTL) command.

Transaction Isolation Levels

DB2 for IBM i provides four isolation levels that control how transactions interact with concurrent data access. You specify the level using the LCKLVL parameter on STRCMTCTL:

- **(CHG) (Uncommitted Read)* - Allows reading uncommitted changes from other transactions. Best performance but least data integrity. Use for reports where approximate data is acceptable.
- **(CS) (Cursor Stability)* - Default level. Prevents dirty reads by only reading committed data. Holds read locks only while cursor is on the row. Good balance for most OLTP applications.
- **(ALL) (Read Stability)* - Prevents dirty reads and non-repeatable reads. Holds read locks on all accessed rows until commit/rollback. Use when consistent result sets are required.
- **(RR) (Repeatable Read)* - Highest isolation level. Prevents all read anomalies including phantom reads. Holds all locks until transaction completes. Use for financial transactions requiring absolute consistency.

COMMIT(*NC) - IBM i Unique Feature

DB2 for IBM i uniquely supports COMMIT(*NC) (No Commit), which allows file operations to run without commitment control even when STRCMTCTL is active. This is specified on individual file opens (OVRDBF or program-level). Files opened with COMMIT(*NC) are not protected by commitment control—their changes are immediately permanent and cannot be rolled back. This provides flexibility to mix committed and non-committed operations within the same job, useful when certain files need immediate updates while others require transactional protection.

Choose your isolation level based on the balance between data consistency requirements and performance/concurrency needs.

Using SQL with Commitment Control

SQL applications on IBM i handle commitment control differently than native record-level access programs:

- **No STRCMTCTL/ENDCMTCTL needed** - SQL automatically manages commitment control through connection properties or SQL statements
- **SET TRANSACTION** - Use `SET TRANSACTION ISOLATION LEVEL` to set isolation level (READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, SERIALIZABLE)
- **COMMIT and ROLLBACK** - Use SQL's `COMMIT` and `ROLLBACK` statements instead of CL or RPG commands
- **Connection-based** - Each SQL connection has its own commitment control scope, independent of CL-based commitment definitions
- **WITH NC** - SQL supports `WITH NC` (No Commit) clause on individual statements for operations outside commitment control

SQL's approach provides more granular control and follows standard SQL conventions, making it easier for developers familiar with other database platforms.

Journaling Requirement

Commitment control requires that all tables involved in a transaction are attached to the same journal. Journals record copies of all database changes (adds, updates, deletes), which enables the system to roll back uncommitted changes if needed. Without journaling, commitment control cannot track or reverse database modifications.

For detailed information on setting up and managing journals, see IBM's documentation: Journal Management

Usage and Testing

For comprehensive information on implementing, using, and testing commitment control including practical examples and step-by-step procedures, see IBM's documentation: Practice Problem: Commitment Control

Key Takeaways

- Commitment control guarantees all-or-nothing transaction processing
- It protects against interrupted processing and application errors
- Applications can be designed to restart safely after abnormal termination
- The STRCMTCTL command creates a commitment definition for the job
- Each commitment definition tracks resources being changed under commitment control
- Use ENDCMTCTL to end commitment control when no longer needed
- Journaling is required—all tables in a transaction must be attached to the same journal
- See IBM documentation for detailed information on journal setup and management

How can I use Result Sets?

What This Covers

This guide explains how to use SQL result sets, including how to define, associate, allocate, and process result sets returned by other RPG programs or other types of stored procedures.

A result set is the output of a SQL query (typically a SELECT statement) that is kept open in memory through a cursor, allowing you to process the data row-by-row or in batches without loading all rows into variables at once.

By default, an SQL or External procedure returns zero result sets. The procedure definer has the ability to return between zero and 32767 unique result sets to its caller or client.

Why It Matters

Result sets provide a powerful way to pass data between RPG programs without the size limitations of arrays. Since result sets are stored in memory and accessed via pointers, they can handle large amounts of data efficiently, making them ideal for complex data processing scenarios.

How It Works

Basic Result Set Processing

1. Define a RESULT_SET_LOCATOR field

```
D MyResultSet     S             SQLTYPE(RESULT_SET_LOCATOR)
```

2. Call the RPG program or Stored Procedure

```
CALL GetMyResultSets();
```

3. Associate the RESULT_SET_LOCATOR with the returned result set

```
Exec SQL
  ASSOCIATE RESULT SET LOCATORS
    (:MyResultSet)
  WITH PROCEDURE GetMyResultSets;
```

> **Note:** For multiple result sets, separate RESULT_SET_LOCATORS with commas:
>
> ```
> (:MyResultSets, :rsCustomers, :rsOrders)
> ```

4. Allocate a CURSOR for the RESULT_SET_LOCATOR

```
Exec SQL
  ALLOCATE MyCursor CURSOR
  FOR RESULT SET :MyResultSet;

Exec SQL
  Open MyCursor;
```

5. Process data from CURSOR

```
Dow SQLcode = 0;
  Exec SQL FETCH next from MyCursor into MyDataStructure;
  // Do something to row...
Enddo;

Exec SQL close MyCursor;
```

Returning Result Sets: RETURN TO CALLER vs RETURN TO CLIENT

When creating SQL or External (i.e. RPG) stored procedures that return result sets, you must specify how the result set should be returned using either `RETURN TO CALLER` or `RETURN TO CLIENT`.

RETURN TO CALLER

Use `RETURN TO CALLER` when the result set will be consumed by another SQL procedure or RPG program (not directly by a client application):

```
CREATE PROCEDURE GetMyResultSets()
  DYNAMIC RESULT SETS 1
  LANGUAGE SQL
BEGIN
  DECLARE MyCursor CURSOR WITH RETURN TO CALLER FOR
    SELECT * FROM MyTable;

  OPEN MyCursor;
END;
```

- Result set remains available to the calling program/procedure
- Allows the caller to use `ASSOCIATE RESULT SET LOCATORS`
- Required when chaining result sets between procedures
- Default behavior if neither option is specified

RETURN TO CLIENT

Use `RETURN TO CLIENT` when the result set will be consumed directly by a client application (JDBC, ODBC, etc.):

```
CREATE PROCEDURE GetCustomerData()
  DYNAMIC RESULT SETS 1
  LANGUAGE SQL
BEGIN
  DECLARE CustomerCursor CURSOR WITH RETURN TO CLIENT FOR
    SELECT * FROM Customers;

  OPEN CustomerCursor;
END;
```

- Result set is sent directly to the client application, effectively bypassing any programs that are on the stack
- Cannot be intercepted by intermediate procedures

> **Important:** If you need an RPG program to process a result set, the stored procedure must use `RETURN TO CALLER`. Using `RETURN TO CLIENT` will prevent the RPG program from accessing the result set via `ASSOCIATE RESULT SET LOCATORS`.

Using Multiple Occurrence Data Structures or Arrays

For bulk operations (inserts, updates, or deletes), you can fetch multiple rows at once:

1. Define a multiple occurrence data structure or array

```
Dcl-ds MyAgentStructure Extname(AGENTS)
  Occurs(1000) End-ds;
Dcl-s rsAGENTS SQLTYPE(RESULT_SET_LOCATOR);
Dcl-s NextRecords Zoned(9);
```

2. Fetch multiple rows into the data structure

```
NextRecords = %Elem(MyAgentStructure);

GETAGENTS();

Exec SQL
  ASSOCIATE RESULT SET LOCATORS
    (:rsAGENTS)
  WITH PROCEDURE GETAGENTS;

Exec SQL
  ALLOCATE AgentsCursor CURSOR
  FOR RESULT SET :rsAGENTS;

Exec SQL
  FETCH NEXT from AgentsCursor
  for :NextRecords
  into :MyAgentStructure;
```

Processing fetched rows:

- Individual rows: `%occurs(MyAgentStructure) = indexnumber`
- Individual columns: `MyAgentStructure(indexnumber).ColumnName`

Key Takeaways

- Result sets overcome array size limitations when passing data between RPG programs
- `RESULT_SET_LOCATOR` is a pointer to in-memory result set data
- Result sets can be processed one row at a time or in bulk using multiple occurrence data structures
- Multiple result sets can be returned and associated from a single procedure call
- Bulk fetching into arrays or data structures is efficient for batch operations
- The cursor does not need to be explicitly opened, but doing so improves code readability

Database Re-engineering Implementation

IBMI-DBRE-PROCEDURE-114

How do I prepare for database re-engineering?

Introduction

With the decision to re-engineer established, the next step is preparation. This is where many efforts succeed or fail.

Re-engineering is not a task that begins with changing tables. It begins with understanding the current system, defining the future state, and building a controlled path between the two. On IBM i, this preparation is critical due to the deep coupling between Db2 for i database objects and RPG applications that depend on them.

Poor preparation leads to broken programs, data inconsistencies, and extended outages. Strong preparation creates a predictable, low-risk transition to a modern and maintainable database architecture.

This section outlines the practical steps required to prepare for database re-engineering in an IBM i environment.

Perform a Full Database Assessment

Before designing anything new, you need a complete understanding of what exists today.

Schema Inventory

Create a full inventory of database objects:

- Tables (physical files)
- Indexes (logical files and SQL indexes)
- Views
- Stored procedures and user-defined functions
- Triggers and constraints

On IBM i, this can be done using system catalogs or IBM APIs. The goal is to eliminate assumptions and work from facts.

Data Analysis

Assess the quality and structure of your data:

- Identify duplicate or redundant data
- Detect inconsistent or invalid values
- Review field usage and data types
- Validate referential integrity

This step is often underestimated. If data quality issues are not addressed early, they will be carried into the new design.

Dependency Mapping

This is one of the most important preparation steps in an IBM i environment.

- Identify all RPG programs using each table or field
- Detect native I/O usage versus SQL access
- Map reports, batch jobs, and integrations

Without this, changes to the database will break applications. With it, you can plan controlled transitions.

Design the Target Architecture

Preparation requires a clear definition of the future state before any changes begin.

Logical Design

Define the business structure of the data:

- Clear entity relationships
- Consistent naming conventions
- Meaningful labels for all fields

On IBM i, many legacy field names lack clarity. This is the opportunity to improve readability without necessarily breaking compatibility.

Physical Design

Translate the logical model into Db2 for i structures:

- Tables with proper keys
- Indexes aligned with access patterns
- Consideration for large data volumes and performance

Design for how the data is used, not just how it is stored.

Data Access Strategy

This is where modernization becomes real.

Define how applications will access the database going forward:

- Direct table access (legacy)
- Views for abstraction
- Stored procedures as the primary interface

A strong recommendation is to standardize on stored procedures as the access layer. This reduces coupling and allows the database to evolve without impacting applications.

Establish Testing Strategy

Testing must be defined before implementation begins.

Unit Testing

Validate database components:

- Stored procedures
- Views
- Constraints and triggers

Integration Testing

Ensure applications function correctly:

- RPG programs
- Batch processing
- External systems

Performance Testing

Measure improvements against baseline metrics:

- Query performance
- Batch runtimes
- Resource utilization

Testing is not optional. It is the control mechanism that ensures re-engineering does not introduce regressions.

Document and Communicate

Preparation is incomplete without clear communication.

Documentation should include:

- Current and target architecture
- Data mappings
- Migration steps
- Testing plans

Communication ensures that:

- Developers understand new access patterns
- Operations teams are prepared for deployment
- Stakeholders understand impact and timelines

Training may be required, especially when introducing new architectural patterns.

Conclusion

Preparation is the foundation of successful database re-engineering. It transforms a high-risk initiative into a controlled and predictable process.

In IBM i environments, where applications and databases are tightly integrated, this preparation is even more critical. By defining clear objectives, assessing the current system, designing a target architecture, and planning migration and testing in detail, organizations position themselves for success.

Re-engineering is not just about improving the database. It is about enabling the system to evolve—safely, consistently, and with confidence.

How to Deploy Tables and Create Surrogate Files

What This Covers

This guide walks you through deploying new database tables, creating surrogate files to maintain compatibility with existing programs, and altering logical files to reference the new tables. You'll learn how to modernize your database without impacting existing applications.

Before You Start

- ☐ You have generated DDL scripts from your Physical Data Model
- ☐ You have identified a separate schema for new tables
- ☐ You understand which programs reference the files being modernized
- ☐ You have backup procedures in place
- ☐ You have access to run SQL scripts on your database

Steps

Deploying the Tables

1. **Choose your deployment method**
    - Export DDL script to a file system directory for batch execution, or
    - Run the script directly from your SQL script editor
    - **Benefit:** Batch execution can be scheduled during off-peak hours to minimize impact
2. **Configure execution settings**
    - Use "Commit after each statement" to prevent full rollback on errors
    - Enable "Continue on error" to allow partial script execution
    - Run in a new connection to isolate the deployment
    - **Benefit:** These settings allow you to identify and fix individual errors without rerunning the entire script

3. **Execute the DDL script**
    - Run the complete script to create tables, indexes, and constraints
    - Monitor the execution results for any errors or warnings
    - **Benefit:** Creates all database objects in the new schema in a single operation
4. **Handle any errors**
    - Review error messages in the results window
    - Common issues include duplicate object names or constraint violations
    - Select and re-run individual failed statements after resolving issues
    - **Benefit:** Targeted error resolution is faster than rerunning the entire script
5. **Validate the deployment**
    - Use your database explorer to verify all objects exist in the new schema
    - Check that tables contain the expected columns with correct attributes
    - Verify indexes and constraints are properly created
    - Confirm labels and descriptions are present
    - **Benefit:** Thorough validation prevents issues when applications begin using the new tables

Creating and Deploying the Surrogate File

1. **Understand the surrogate file purpose**
    - Surrogate files retain the original record format to prevent program changes
    - They allow new columns to be added to tables without forcing recompilation
    - Existing programs remain unaware of the transformation
    - **Benefit:** Minimizes impact of change and maintains application stability during modernization
2. **Choose the appropriate surrogate file type**
    - For legacy physical files: Use a logical file containing original columns, or use a physical file replica (no data, format reference only)
    - For existing DDL tables: Use a view with the original format, or use an alias pointing to the new table
    - For keyed files being partitioned: Use a non-partitioned index
    - **Benefit:** Different surrogate types support various modernization scenarios
3. **Create the surrogate file**
    - For logical files: Define a file with columns matching the original physical file format
    - For views: Create a view selecting only the original columns from the new table
    - For aliases: Create an alias with the original table name pointing to the new table
    - Ensure the surrogate uses the same name as the original file
    - **Benefit:** Programs continue to reference familiar object names without modification
4. **Verify format compatibility**
    - Check that the Record Format Level ID (RFID) matches the original file
    - Run queries to compare format definitions between old and new objects
    - **Benefit:** Matching RFIDs ensures programs can access data without recompilation
5. **Deploy the surrogate file**
    - Create the surrogate file in the same library as the original file
    - Test access through existing programs to confirm compatibility

- **Benefit:** Existing programs automatically use new tables through the surrogate file

Altering and Deploying DDS Logical Files

1. **Identify logical files to modify**
   - List all logical files built over the physical files being modernized
   - Document which files use format sharing versus unique formats
   - **Benefit:** Understanding dependencies ensures all necessary files are updated
2. **Update the PFILE keyword**
   - Change the PFILE keyword to reference the system name (short name) of the new table
   - Keep all other logical file definitions unchanged
   - **Benefit:** Redirects logical file to new table while maintaining existing functionality
3. **Add FORMAT keyword when needed**
   - If the logical file uses format sharing (implicitly includes physical file fields), add the FORMAT keyword
   - Specify the original physical file format name
   - If the logical file defines its own unique columns, FORMAT keyword may not be needed
   - **Benefit:** FORMAT keyword allows logical files to find the old format even when built over new tables
4. **Handle different logical file scenarios**
   - For unique formats: Update PFILE only
   - For format sharing: Update PFILE and add FORMAT keyword
   - For keyed files over partitioned tables: Consider creating non-partitioned indexes with RCDFMT keyword
   - **Benefit:** Different scenarios require different approaches for successful modernization
5. **Deploy modified logical files**
   - Compile the updated logical file definitions
   - Replace existing logical files in their current libraries
   - **Benefit:** Programs referencing logical files continue working without changes
6. **Verify the modifications**
   - Run queries to display record format information for old and new files
   - Compare format IDs to ensure they match
   - Test program access through the modified logical files
   - **Benefit:** Verification confirms programs can access data without recompilation

Verify It Worked

Confirm success by checking:

- **Tables:** All new tables, indexes, and constraints exist in the new schema with correct attributes
- **Surrogate Files:** Surrogate files are created with matching format IDs to original files
- **Logical Files:** Modified logical files reference new tables and maintain original formats
- **Program Access:** Existing programs can read and write data through surrogate and logical files without errors
- **Data Integrity:** New columns receive default values when programs insert data through surrogate files

Key Takeaways

- Deploy tables to a separate schema to isolate new objects from existing applications

- Use "Commit after each statement" and "Continue on error" for flexible error handling during deployment
- Surrogate files maintain original record formats, allowing table enhancements without program changes
- Different surrogate types (logical files, views, aliases, indexes) support various modernization scenarios
- Updating PFILE keywords redirects logical files to new tables
- FORMAT keyword enables format sharing logical files to work with new tables
- Matching Record Format Level IDs ensures programs work without recompilation
- Thorough validation after each step prevents issues when applications begin using modernized objects

Database Architecture and Management

How do I use SQL views as an abstraction layer?

What It Is

A **virtual data model** is an abstraction layer built on top of physical database tables using SQL Views. Instead of applications directly accessing base tables, they interact with views that present data in a logical, business-oriented structure. This architectural pattern decouples the application layer from the physical data model, allowing each to evolve independently.

SQL Views are defined with an SQL query and act as virtual tables. They contain no data and are performance neutral. They simply project that data corresponding to the SQL query that was used to define the view.

Why It Matters

Business Agility

- **Schema Evolution**: Physical tables can be restructured, renamed, or normalized without breaking existing applications
- **Backward Compatibility**: Legacy applications continue working while new applications use modern structures
- **Reduced Maintenance**: Changes to table structures require updates only to views, not to hundreds of application programs

Security and Governance

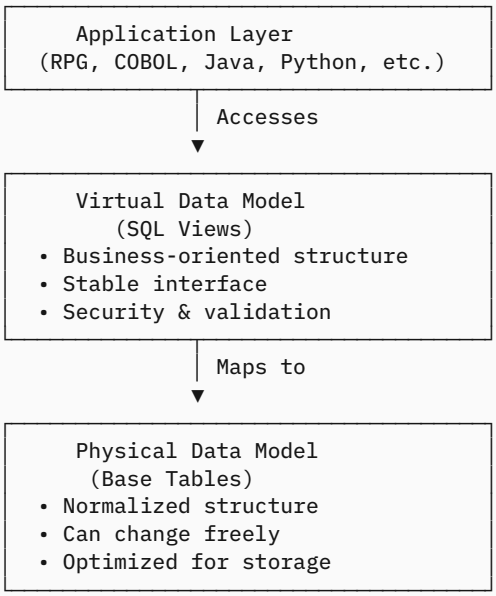
- **Fine-Grained Access Control**: Views can expose only specific columns and rows that users are authorized to see
- **Data Masking**: Sensitive data (SSN, credit cards) can be masked or encrypted at the view layer
- **Audit Trail**: All data access flows through controlled view definitions

Development Efficiency

- **Simplified Application Code**: Applications work with business-oriented views rather than complex normalized structures
- **Reusability**: Common data transformations defined once and shared across applications
- **Testing**: Views can be redirected to test data without application changes

How It Works

Basic Architecture



Practical Example on IBM i

Physical Tables (normalized, subject to change):

```
-- Physical structure optimized for storage
CREATE TABLE CUSTOMER_MASTER (
    CUST_ID INTEGER PRIMARY KEY,
    CUST_NAME VARCHAR(100),
    CUST_TYPE CHAR(1)
);

CREATE TABLE CUSTOMER_ADDRESS (
    ADDR_ID INTEGER PRIMARY KEY,
    CUST_ID INTEGER,
    STREET VARCHAR(100),
    CITY VARCHAR(50),
    STATE CHAR(2),
    ZIP CHAR(10)
);

CREATE TABLE CUSTOMER_CONTACT (
    CONTACT_ID INTEGER PRIMARY KEY,
    CUST_ID INTEGER,
    PHONE VARCHAR(20),
    EMAIL VARCHAR(100)
);
```

Virtual Data Model (stable interface for applications):

```
-- View presents denormalized, business-friendly structure
CREATE OR REPLACE VIEW CUSTOMER_INFO AS
SELECT
    CM.CUST_ID,
    CM.CUST_NAME,
    CASE CM.CUST_TYPE
        WHEN 'R' THEN 'Retail'
        WHEN 'W' THEN 'Wholesale'
        WHEN 'G' THEN 'Government'
        ELSE 'Unknown'
    END AS CUSTOMER_TYPE,
    CA.STREET,
    CA.CITY,
    CA.STATE,
    CA.ZIP,
    CC.PHONE,
    CC.EMAIL,
    -- Computed columns
    TRIM(CA.CITY) CONCAT ', ' CONCAT CA.STATE CONCAT ' ' CONCAT CA.ZIP
        AS FULL_ADDRESS
FROM CUSTOMER_MASTER CM
LEFT JOIN CUSTOMER_ADDRESS CA
    ON CM.CUST_ID = CA.CUST_ID
LEFT JOIN CUSTOMER_CONTACT CC
    ON CM.CUST_ID = CC.CUST_ID;
```

Application Code (unchanged even if tables restructure):

```
-- RPG, COBOL, or any language uses the view
SELECT CUST_NAME, CUSTOMER_TYPE, FULL_ADDRESS, EMAIL
FROM CUSTOMER_INFO
WHERE STATE = 'CA'
ORDER BY CUST_NAME;
```

Advanced Patterns

1. Security Views with Row-Level Filtering

```
CREATE OR REPLACE VIEW MY_CUSTOMERS AS
SELECT * FROM CUSTOMER_INFO
WHERE CUST_ID IN (
    SELECT CUST_ID FROM CUSTOMER_ASSIGNMENTS
    WHERE USER_ID = CURRENT USER
);
```

2. Data Masking Views

```
CREATE OR REPLACE VIEW CUSTOMER_INFO_MASKED AS
SELECT
    CUST_ID,
    CUST_NAME,
    CUSTOMER_TYPE,
    -- Mask sensitive data
    'XXX-XX-' || SUBSTR(SSN, 8, 4) AS SSN_MASKED,
    STREET,
    CITY,
    STATE,
    ZIP
FROM CUSTOMER_INFO;
```

3. Versioned Views for API Stability

```
-- V1 API (legacy applications)
CREATE OR REPLACE VIEW CUSTOMER_V1 AS
SELECT CUST_ID, CUST_NAME, PHONE
FROM CUSTOMER_INFO;

-- V2 API (new applications with additional fields)
CREATE OR REPLACE VIEW CUSTOMER_V2 AS
SELECT CUST_ID, CUST_NAME, PHONE, EMAIL, CUSTOMER_TYPE
FROM CUSTOMER_INFO;
```

Migration Strategy

Phase 1: Create Views Over Existing Tables

```
-- Start with 1:1 views matching current table structure
CREATE OR REPLACE VIEW CUSTOMER AS
SELECT * FROM CUSTOMER_MASTER;
```

Phase 2: Update Applications to Use Views

```
-- Change application code from:
SELECT * FROM CUSTOMER_MASTER WHERE CUST_ID = ?

-- To:
SELECT * FROM CUSTOMER WHERE CUST_ID = ?
```

Phase 3: Evolve Physical Model Independently

```
-- Restructure tables without breaking applications
ALTER TABLE CUSTOMER_MASTER
    ADD COLUMN CUST_SEGMENT CHAR(2);

-- Update view to maintain compatibility
CREATE OR REPLACE VIEW CUSTOMER AS
SELECT
    CUST_ID,
    CUST_NAME,
    CUST_TYPE,
    -- New column with default for existing apps
    COALESCE(CUST_SEGMENT, 'XX') AS CUST_SEGMENT
FROM CUSTOMER_MASTER;
```

Key Takeaways

- SQL views are highly recommended top create a virtual data model
- Every physical file should have at least one SQL view
- Applications should utilize the virtual data model
- Since SQL views do not contain data, their existence isn't a storage concern

How do I manage database objects?

What This Covers

This guide explains how to effectively monitor and manage database objects on IBM i using three key tools: System Limits monitoring, Db2 for i Statistical Catalogs, and the Index Advisor. These tools help database engineers maintain optimal database performance, identify potential issues, and implement recommended improvements.

Why It Matters

Proactive database object management is critical for maintaining system performance and preventing issues before they impact users. Without proper monitoring:

- Tables with high percentages of deleted records waste storage and degrade performance
- Missing indexes cause slow query performance and high CPU usage
- Object limits can be reached unexpectedly, causing application failures
- Statistical insights that could guide optimization decisions are overlooked

By leveraging IBM i's built-in monitoring tools, database engineers can identify problems early, optimize database structures, and ensure the database operates efficiently.

How It Works

1. Monitoring IBM i System Limits

IBM i enforces various limits on database objects that, when approached or exceeded, can cause performance degradation or system failures. The QSYS2.SYSLIMITS_BASIC view provides access to these critical system limits, with database-related limits identified by limit_id values between 15000 and 16806.

Understanding SYSLIMITS_BASIC

The SYSLIMITS_BASIC view exposes system limits and their current usage, allowing proactive monitoring before limits are reached.

```
-- View all database-related system limits, most recent first
SELECT limit_id,
       sizing_name,
       current_value,
       maximum_value,
       DECIMAL(FLOAT(current_value) / NULLIF(FLOAT(maximum_value), 0) * 100, 5, 2)
         AS percent_used,
       CASE
         WHEN DECIMAL(FLOAT(current_value) / NULLIF(FLOAT(maximum_value), 0) * 100, 5, 2) > 90
         THEN 'CRITICAL'
         WHEN DECIMAL(FLOAT(current_value) / NULLIF(FLOAT(maximum_value), 0) * 100, 5, 2) > 75
         THEN 'WARNING'
         ELSE 'OK'
       END AS status,
       sl.*
FROM qsys2.syslimits_basic sl
WHERE limit_id BETWEEN 15000 AND 16806
ORDER BY  last_change_timestamp  DESC;
```

Key Database System Limits to Monitor

Limit 16100 - Maximum Number of Members in a File (CRITICAL):

This is one of the most important limits to monitor as multi-member files are common in IBM i applications.

```
-- Monitor member count approaching file limit
SELECT limit_id,
       sizing_name,
       current_value AS current_members,
       maximum_value AS max_members,
       DECIMAL(FLOAT(current_value) / FLOAT(maximum_value) * 100, 5, 2) AS percent_used,
       CASE
         WHEN DECIMAL(FLOAT(current_value) / FLOAT(maximum_value) * 100, 5, 2) > 90
         THEN 'CRITICAL - Approaching member limit'
         WHEN DECIMAL(FLOAT(current_value) / FLOAT(maximum_value) * 100, 5, 2) > 75
         THEN 'WARNING - Plan to split file or archive members'
         ELSE 'OK'
       END AS status,
       sl.*
FROM qsys2.syslimits_basic sl
WHERE limit_id = 16100
order by percent_used desc;
```

> **Important:** When approaching the member limit, consider:
>
> - Archiving old members to a separate file
> - Splitting the file into multiple files by date range or category
> - Converting to single-member files if appropriate for your application

Limit 15003 - Maximum Row Size:

```
-- Monitor table partition (aka member) size
SELECT limit_id,
       sizing_name,
       current_value AS current_size,
       maximum_value AS max_size,
       DECIMAL(FLOAT(current_value) / FLOAT(maximum_value) * 100, 5, 2) AS percent_used,
       sl.*
FROM qsys2.syslimits_basic sl
WHERE limit_id = 15003
order by percent_used desc;
```

Monitoring Deleted Records and Table Health

While SYSLIMITS_BASIC tracks system-wide limits, you should also monitor table-specific health metrics:

```
-- Monitor deleted record percentages across all tables
SELECT table_schema,
       table_name,
       number_rows,
       number_deleted_rows,
       DECIMAL(FLOAT(number_deleted_rows) /
               NULLIF(FLOAT(number_rows + number_deleted_rows), 0) * 100, 5, 2)
         AS deleted_percentage,
       data_size,
       CASE
         WHEN DECIMAL(FLOAT(number_deleted_rows) /
                      NULLIF(FLOAT(number_rows + number_deleted_rows), 0) * 100, 5, 2) > 50
         THEN 'REORGANIZE CRITICAL'
         WHEN DECIMAL(FLOAT(number_deleted_rows) /
                      NULLIF(FLOAT(number_rows + number_deleted_rows), 0) * 100, 5, 2) > 25
         THEN 'REORGANIZE RECOMMENDED'
         ELSE 'OK'
       END AS status
FROM qsys2.systablestat
WHERE table_schema NOT IN ('QSYS', 'QSYS2')
  AND number_deleted_rows > 1000000
ORDER BY deleted_percentage DESC;
```

2. Using Db2 for i Statistical Catalogs

Db2 for i provides comprehensive statistical catalogs (views in QSYS2) that offer deep insights into database object usage, performance, and characteristics.

Key Statistical Catalog Views

SYSTABLESTAT - Table Statistics:

```
-- Comprehensive table analysis
SELECT table_schema,
       table_name,
       number_rows,
       data_size,
       DECIMAL(FLOAT(data_size) / 1048576, 10, 2) AS size_mb,
       last_used_timestamp,
       days_used_count,
       last_reset_timestamp,
       system_table_name,
       system_table_schema,
       sts.*
FROM qsys2.systablestat sts
WHERE table_schema = 'TOYSTORE3'
ORDER BY data_size DESC;
```

SYSINDEXSTAT - Index Statistics:

```
-- Analyze index usage and effectiveness
SELECT i.table_schema,
       i.table_name,
       i.index_name,
       i.index_size,
       i.last_used_timestamp,
       i.last_query_use,
       i.last_statistics_use,
       i.query_use_count,
       i.query_statistics_count,
       i.days_used_count,
       i.last_reset_timestamp,
       i.*
  FROM qsys2.sysindexstat i
  WHERE i.table_schema = 'TOYSTORE3'
  ORDER BY i.query_use_count + i.query_statistics_count DESC;
```

SYSCOLUMNSTAT - Column Statistics:

```
-- Analyze column statistical detail
SELECT table_schema,
       table_name,
       column_name,
       average_column_length,
       maximum_column_length,
       length_at_90th_percentile,
       overflow_rows,
       number_distinct_values,
       length_at_90th_percentile,
       scs.*
  FROM qsys2.syscolumnstat scs
  WHERE table_schema = 'TOYSTORE3' AND table_name = 'ITEM_FACT'
  ORDER BY column_name;
```

3. Using the Index Advisor

Viewing Index Advisor Recommendations

SYSIXADV - Index Advisor Recommendations:

```
-- View all index recommendations
SELECT table_schema,
       table_name,
       times_advised,
       key_columns_advised,
       mti_used,
       mti_created,
       last_mti_used,
       mti_used_for_stats,
       last_mti_used_for_stats,
       i.*
  FROM qsys2.condidxa i
  WHERE table_schema = 'TOYSTORE3'
  ORDER BY times_advised DESC;
```

Key Takeaways

- **Monitor System Limits regularly**
- **Leverage Statistical Catalogs**
- **Utilize the Index Advisor**
- **Implement proactive monitoring**
- **Regular maintenance is essential**

Should I open SQL objects in F-specs?

What It Is

Opening SQL tables, indexes, or views in F-specs causes the compiler to check the current file structure against the structure compiled into the program.

Why It Matters

If the structures differ, a format level check error occurs, causing your program to abnormally end. Setting level check off doesn't help because you're bringing in a record layout that differs from your program's expectations.

How It Works

Solution: Achieve separation from the underlying data model by avoiding data-related F-specs entirely. All data access should come through the logical layer (stored procedures or SQL SELECT against views).

For impact analysis tools: Use F-specs with USROPN and an impossible condition:

```
Ffile_a if e k disk usropn
IF *INLR AND NOT *INLR;
  OPEN file_a;
ENDIF;
```

Alternative: Use the TEMPLATE keyword for reference-only files:

```
FrcdFile_t IF E K DISK TEMPLATE
F EXTDESC('EMPADDRESS')
```

Should I access base tables directly in my programs?

What It Is

Directly accessing SQL tables in programs creates tight coupling between application code and the physical data model.

Why It Matters

When the underlying table structure changes, programs that directly access the table may break or require recompilation. This creates maintenance overhead and increases the risk of application failures.

How It Works

Solution: Define a view with the columns your program needs. The view should be specific to your program and used as an external data structure.

Benefits:

- New columns added to the table don't affect your program
- Deleted columns can be handled in the view (substitute blanks or get data from another table)
- Increased column sizes can be managed in the view using SUBSTR
- No format-level checks when table structure changes

Where should complex SQL logic reside?

What It Is

Complex SQL statements can be embedded directly in programs or encapsulated in views that programs access.

Why It Matters

Embedding complex SQL in programs reduces readability, duplicates code across applications, and makes maintenance difficult when business logic changes.

How It Works

Solution: Keep SQL statements in programs as simple as possible. Place complex SQL in views and access them from programs.

Benefits:

- Programs easier to read
- Views are reusable component modules
- Complex SQL maintained in one place
- Business logic changes only require view updates
- No program changes needed

Performance consideration: Consider where to place CAST operations:

- **CAST in view:** Runs on every row when view is queried
- **CAST in SELECT:** Runs only on rows returned to program

Database Object Maintenance

How to Change Database Object Definitions

What This Covers

This guide explains the best methods for changing database object definitions in a modernized database environment. You'll learn how to modify column sizes, add new columns, introduce new tables, and deprecate old columns while minimizing impact on existing applications.

Before You Start

- ☐ Your database files are modernized with surrogate logical files or views
- ☐ You understand which programs reference the objects being changed

- ☐
  You have a data modeling tool or SQL access to modify table definitions
- ☐
  You have identified the scope of impact for your proposed changes
- ☐
  You have backup procedures in place

Steps

Increasing Column Size

1. **Modernize the file if not already done**
   - Create the underlying SQL table with enhanced features
   - Create surrogate logical files that maintain the original format
   - **Benefit:** Modernization isolates programs from physical data layout, enabling changes without program modifications
2. **Increase the column length in the SQL table**
   - Use your data modeling tool or ALTER TABLE statement to increase the column size
   - Deploy the modified table to all database instances
   - **Benefit:** The underlying table can accommodate larger values while existing programs continue working
3. **Handle programs that need the longer column**
   - Create a new logical file or SQL view that exposes the longer column
   - Update only the programs that require access to the full length
   - Programs using the surrogate logical file continue processing data that fits the original size
   - **Benefit:** Selective updates minimize the number of programs requiring changes
4. **Understand data truncation behavior**
   - For character columns: Programs using surrogate files read the left portion of longer values
   - For numeric columns: Programs may encounter overflow errors with values exceeding the original size
   - Plan deployment carefully to avoid putting longer values before all programs are ready
   - **Benefit:** Understanding behavior prevents unexpected application failures
5. **Assess impact before deciding approach**
   - Use analysis tools to determine how many programs are affected
   - If only a few programs are impacted, consider revising them directly
   - If many programs are impacted, use surrogate logical files to mask the change
   - **Benefit:** Impact analysis guides the most efficient change strategy

Adding New Columns

1. **Modernize the file if not already done**
   - Follow the same modernization process as for column size changes
   - **Benefit:** Establishes the foundation for flexible schema evolution
2. **Add the column to the SQL table**
   - Use your data modeling tool or ALTER TABLE ADD COLUMN statement
   - Ensure the new column has a default value to maintain updatability of surrogate logical files
   - Deploy the modified table to all database instances

- **Benefit:** Default values ensure existing programs can still insert rows without knowing about the new column

3. **Leave existing programs unchanged**
   - Programs that don't need the new column continue using their surrogate logical files
   - The new column remains invisible to these programs
   - **Benefit:** Zero impact on programs that don't require the new functionality
4. **Provide access for programs needing the new column**
   - Create a new logical file or SQL view that includes the new column
   - Update only the programs that need to access or modify the new column
   - **Benefit:** Targeted updates minimize development and testing effort

Adding New Tables with Foreign Keys

1. **Create the new table**
   - Design and create the new table (e.g., SITE_INFO for site information)
   - Define appropriate primary keys and constraints
   - **Benefit:** New tables support expanded functionality without modifying existing structures
2. **Add foreign key column to existing table**
   - Add a foreign key column (e.g., SITE_ID) to the existing table
   - Use surrogate primary keys added during modernization if available
   - Ensure the column has a default value or is nullable
   - **Benefit:** Establishes relationships while maintaining compatibility with existing programs
3. **Populate the foreign key values**
   - Use SQL UPDATE statements to populate the foreign key column
   - Apply business logic to assign appropriate values
   - **Benefit:** Data relationships are established without program changes
4. **Use the new structure in new applications**
   - New queries and applications can leverage the foreign key relationships
   - Create views that join tables using the new relationships
   - Existing programs remain unaffected through surrogate logical files
   - **Benefit:** New functionality coexists with legacy applications

Consolidating Archive and Current Data

1. **Leverage timestamp columns from modernization**
   - Use the CREATE_ROW_TS column added during modernization
   - This timestamp tracks when each row was created
   - **Benefit:** Eliminates need for separate date tracking columns
2. **Create views for current and archived data**
   - Create a view for current data that filters rows based on age (e.g., less than 3 years old)
   - Create a view for archived data that filters older rows (e.g., 3 years or older)
   - Both views reference the same underlying table
   - **Benefit:** Eliminates long-running batch archival jobs and file unavailability

3. **Replace physical separation with logical separation**
    - Programs access views instead of separate physical files
    - Data remains in a single table, partitioned logically by views
    - **Benefit:** Improves performance and eliminates data movement overhead

Deprecating Old Columns

1. **Identify the column to deprecate**
    - Determine which old column is being replaced (e.g., character date field)
    - Identify the new column or modernization feature that replaces it (e.g., CREATE_ROW_TS)
    - **Benefit:** Clear identification ensures complete transition planning
2. **Create a view that transforms the data**
    - Create a view that returns a default value for the old column
    - Include the new column with proper data transformation (e.g., DATE(CREATE_ROW_TS))
    - Name the view the same as the original table
    - **Benefit:** Programs transparently transition to the new column format
3. **Transition programs to use the view**
    - Update programs to reference the view instead of the base table
    - Test that programs work correctly with the transformed data
    - **Benefit:** Gradual transition reduces risk and allows thorough testing
4. **Drop the old column when transition is complete**
    - Once all programs use the view, drop the deprecated column from the base table
    - Update the view definition if needed
    - **Benefit:** Eliminates redundant data storage and simplifies the schema

Verify It Worked

Confirm your changes are successful by checking:

- **Column Changes:** Modified columns accept new values and existing programs continue functioning
- **New Columns:** Programs that don't need new columns work unchanged; programs needing them can access the data
- **Foreign Keys:** Relationships are established and can be queried; existing programs are unaffected
- **Views:** Archive and current data views return correct data based on filtering criteria
- **Deprecated Columns:** Views return appropriate default values; programs transition successfully

Key Takeaways

- Modernized databases with surrogate logical files enable schema changes without impacting existing programs
- Increasing column size requires careful planning to avoid data truncation or overflow issues
- Impact analysis determines whether to revise programs directly or use surrogate files to mask changes
- Adding columns with default values maintains updatability of surrogate logical files
- New columns remain invisible to programs that don't need them through surrogate files
- Foreign key columns can be added to establish relationships without affecting existing applications
- Views provide logical data separation, eliminating the need for physical archival processes
- Timestamp columns from modernization enable powerful date-based filtering and transformations
- Deprecating columns is accomplished through views that return default values for old columns

- Programs should use SQL and views rather than direct table access for maximum flexibility
- Gradual transitions through views reduce risk and allow thorough testing before finalizing changes

What Are Object Dependencies?

Object dependencies represent relationships between database objects where one object relies on another for its definition or operation. Understanding these dependencies is essential for:

- **Impact Analysis**: Determining which objects will be affected by a planned change
- **Safe Modifications**: Avoiding breaking database relationships
- **Documentation**: Understanding the data model, and defined relationships
- **Troubleshooting**: Identifying issues caused by missing or modified dependencies
- **Migration Planning**: Ensuring all related objects are moved together

Method 1: Using the Schemas feature in IBM i Access Client Solutions (ACS) to identify the related objects for a Db2 for i file.

The most straightforward method is to use the graphical interface provided by IBM i ACS tool. From the main ACS window, select a system and then click the item of Schemas, which is under the Database category to open the browser of Schemas.

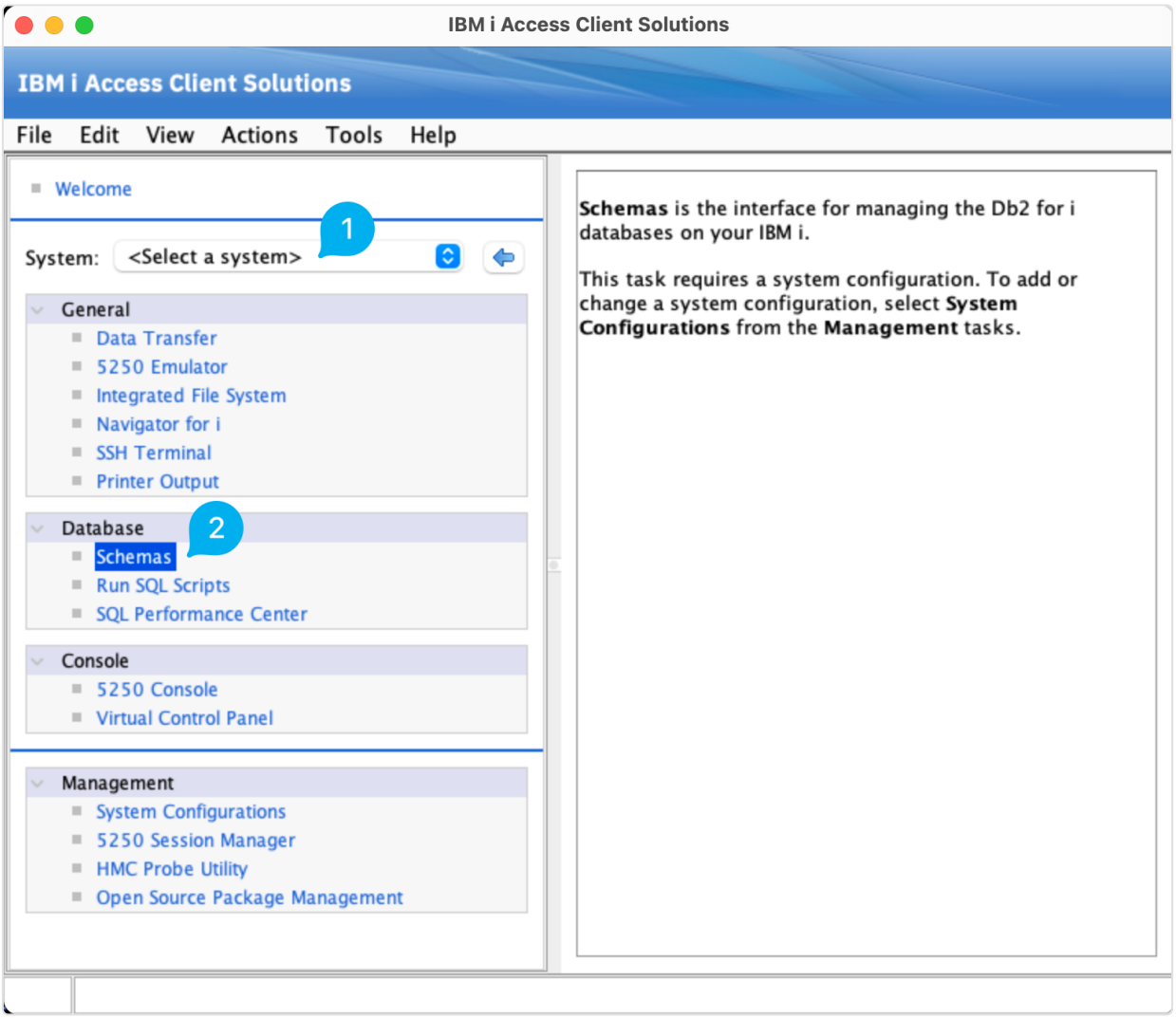


Figure. Find related objects in ACS Schemas 1

Next, add the desired schema to the list of selected schemas.

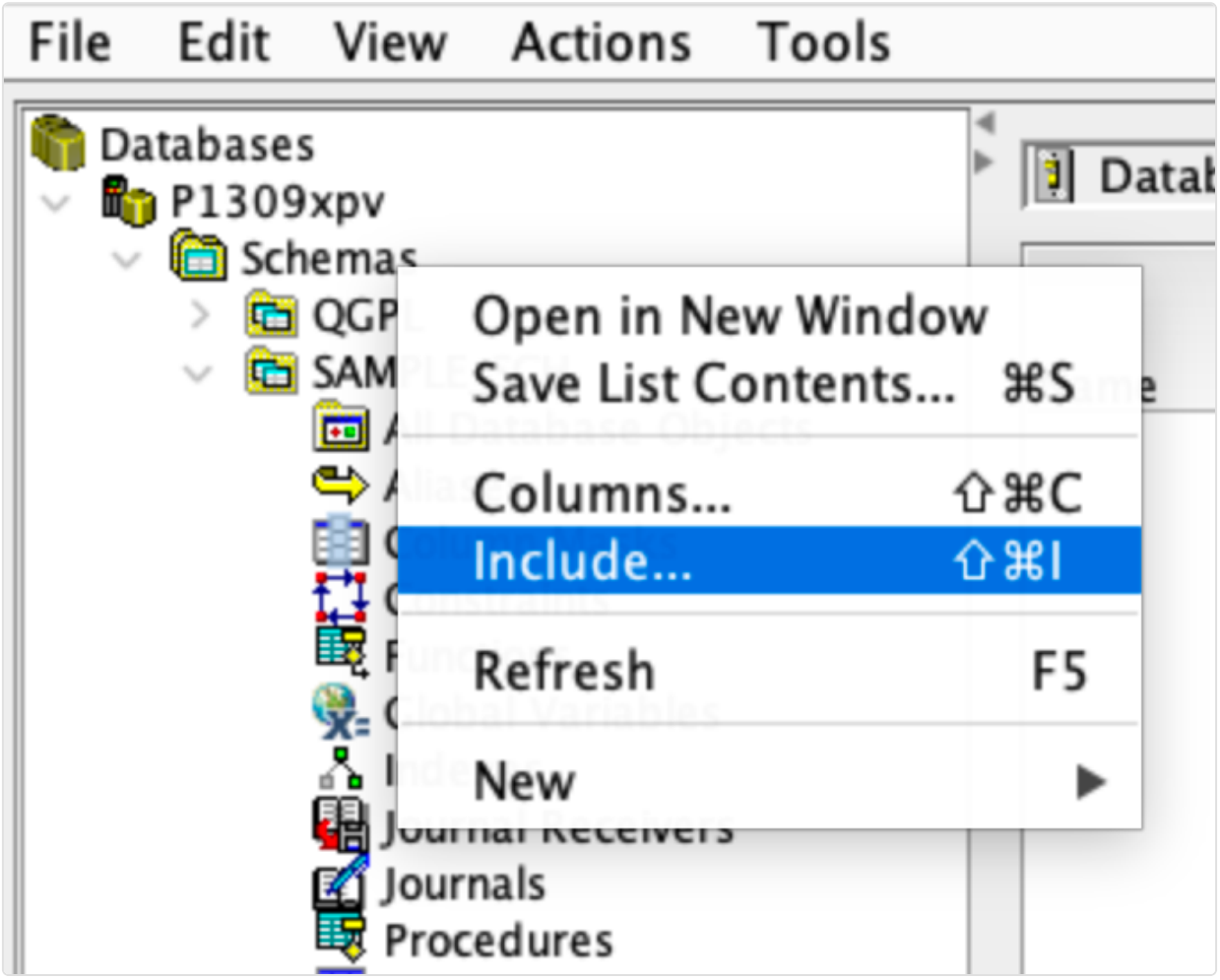


Figure. Find related objects in ACS Schemas 2

After selecting a table or view within the chosen schema, right-click and open the Related Objects tab to display the database objects associated with the selected table or view.

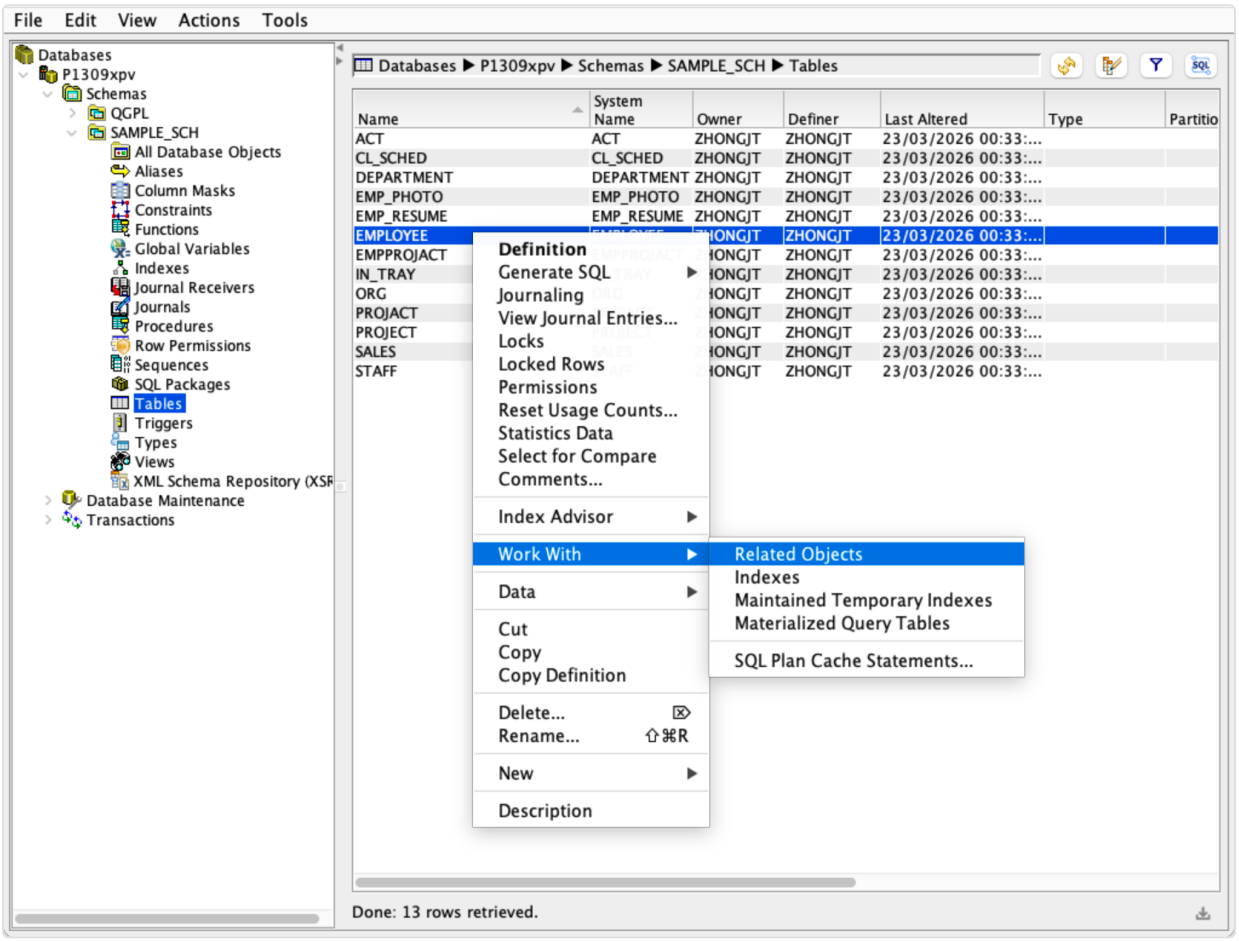


Figure. Find related objects in ACS Schemas 3

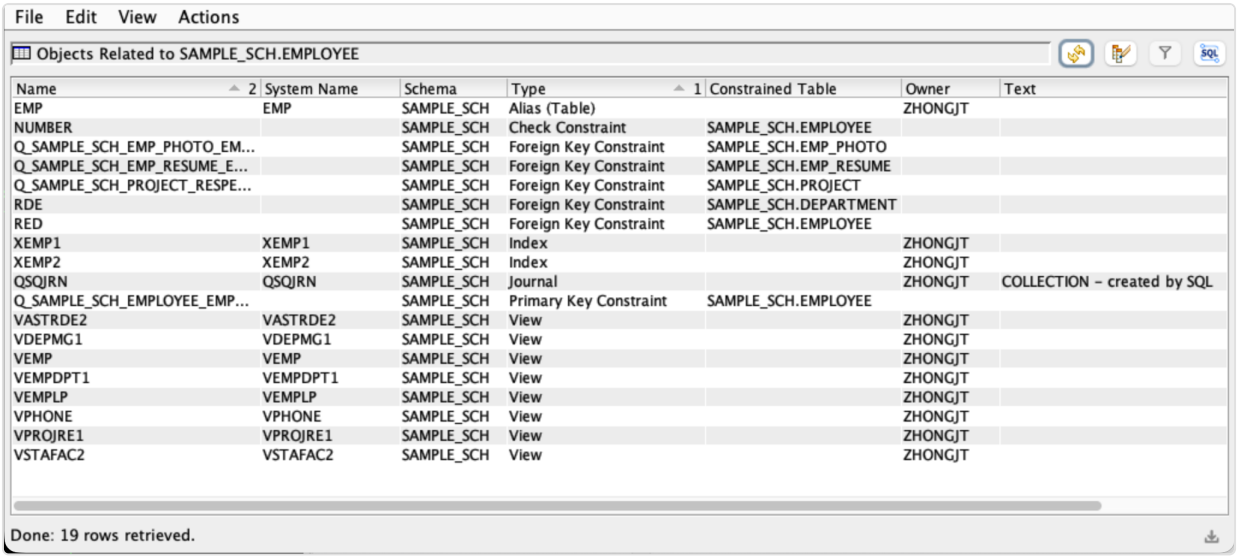


Figure. Find related objects in ACS Schemas 4

Method 2: Using IBM i catalogs to discover Db2 for i object dependencies.

Db2 for i provides several system catalog views in the `QSYS2` schema for discovering dependencies.

Catalog name	Description
QSYS2.SYSCSTDEP	Information about constraint dependencies on tables
QSYS2.SYSCONTROLSDEP	Information about row permissions and column masks
QSYS2.SYSROUTINEDEP	Information about function and procedure dependencies
QSYS2.SYSTABLEDEP	Information about materialized query table dependencies
QSYS2.SYSTRIGDEP	Information about objects used in triggers
QSYS2.SYSVARIABLEDEP	Information about objects used in global variables
QSYS2.SYSVIEWDEP	Information about view dependencies on tables

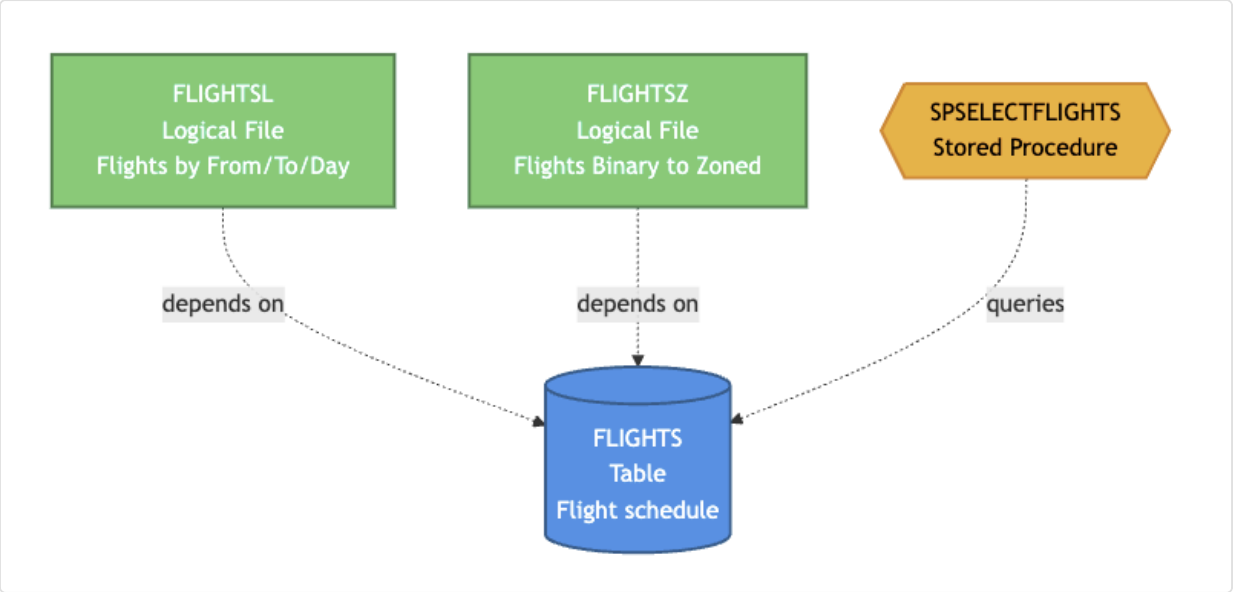


Figure. Find dependent object from catalog view example 1

```
select * from qsys2.SYSROUTINEDEP where OBJECT_SCHEMA = 'FLGHT400';
```

SPECIFIC_SCHEMA	SPECIFIC_NAME	OBJECT_CATALOG	OBJECT_SCHEMA	OBJECT_NAME	OBJECT_TYPE	PARM_SIGNATURE	TABLE_NUMBER	NUMBER_OF_PARMS
FLGHT400	SPSELECTFLIGHTS	QUIN	FLGHT400	FLIGHTS	TABLE	null	0	null

Example 2: The system catalog view QSYS2.SYSVIEWDEP provides information about view dependencies on tables.

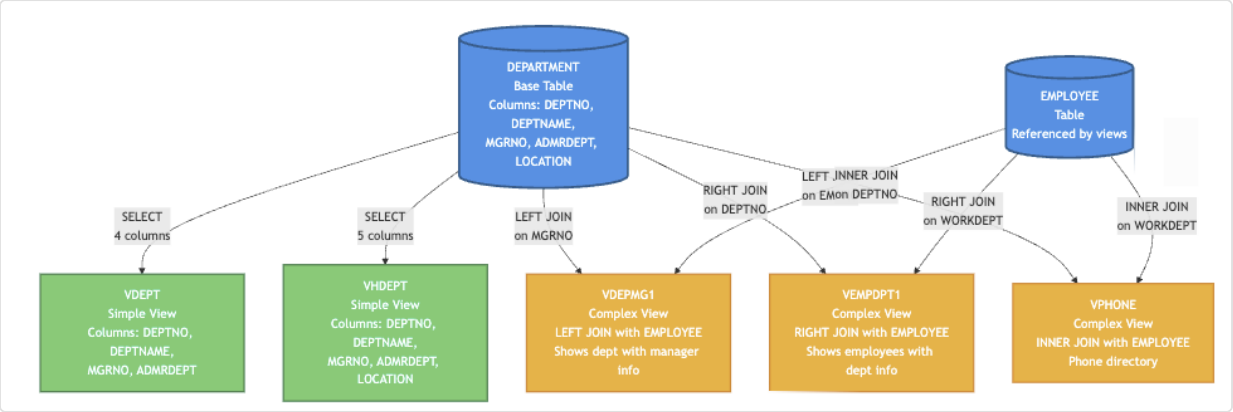


Figure. Find dependent object from catalog view example 2

```
select * from qsys2.sysviewdep where SYSTEM_TABLE_SCHEMA = 'SAMPLE_SCH' and table_name='DEPARTMENT';
```

VIEW_NAME	VIEW_SCHEMA	OBJECT_NAME	OBJECT_SCHEMA	OBJECT_TYPE
VDEPMG1	SAMPLE_SCH	DEPARTMENT	SAMPLE_SCH	TABLE
VDEPT	SAMPLE_SCH	DEPARTMENT	SAMPLE_SCH	TABLE
VHDEPT	SAMPLE_SCH	DEPARTMENT	SAMPLE_SCH	TABLE
VPHONE	SAMPLE_SCH	DEPARTMENT	SAMPLE_SCH	TABLE
VEMPDPT1	SAMPLE_SCH	DEPARTMENT	SAMPLE_SCH	TABLE

Method 3: Using SYSTOOLS.RELATED_OBJECTS

The `RELATED_OBJECTS` table function returns all objects that depend on a database file:

```
SELECT
    SOURCE_SCHEMA_NAME,
    SOURCE_SQL_NAME,
    SQL_OBJECT_TYPE,
    SQL_NAME,
    LIBRARY_NAME,
    OBJECT_TEXT
FROM TABLE(
    SYSTOOLS.RELATED_OBJECTS(
        'FLGHT400',    -- Library
        'FLIGHTS'     -- Database file
    )
);
```

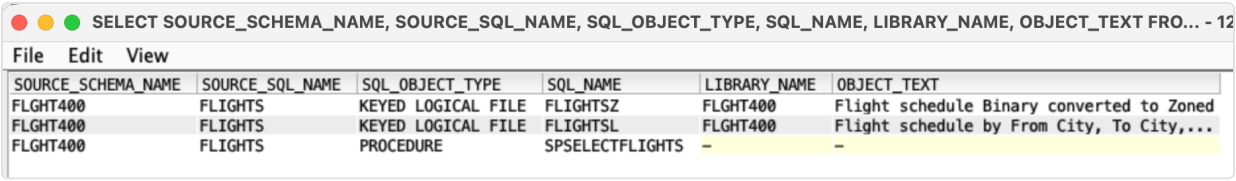


Figure. Find related objects by SQL service 1

Another example to list all the objects that are dependent on the FLIGHTS file in FLGHT400. For each object, return how many recursive steps were required to find the object. This query returns the same rows as the prior example, but includes a column containing the recursion level.

```
SELECT LEVEL, RO.SOURCE_SCHEMA_NAME, RO.SOURCE_SQL_NAME, RO.SQL_NAME, RO.SQL_OBJECT_TYPE, RO.SCHEMA_NAME
  FROM (VALUES('SAMPLE_SCH', 'DEPARTMENT')) I(IN_LIB, IN_FILE),
        TABLE (QSYS2.OBJECT_STATISTICS(IN_LIB, '*FILE', IN_FILE)),
        TABLE(SYSTOOLS.RELATED_OBJECTS(IN_LIB, IN_FILE)) RO
  START WITH SOURCE_SCHEMA_NAME = OBJLONGSCHEMA AND
             SOURCE_SQL_NAME = OBJLONGNAME
  CONNECT BY SOURCE_SQL_NAME = PRIOR SQL_NAME AND
             SOURCE_SCHEMA_NAME = PRIOR SCHEMA_NAME
  ORDER BY 1, 2, 3;
```

LEVEL	SOURCE_SCHEMA_NAME	SOURCE_SQL_NAME	SQL_NAME	SQL_OBJECT_TYPE	SCHEMA_NAME
1	SAMPLE_SCH	DEPARTMENT	VHDEPT	VIEW	SAMPLE_SCH
1	SAMPLE_SCH	DEPARTMENT	XDEPT3	INDEX	SAMPLE_SCH
1	SAMPLE_SCH	DEPARTMENT	VDEPMG1	VIEW	SAMPLE_SCH
1	SAMPLE_SCH	DEPARTMENT	XDEPT1	INDEX	SAMPLE_SCH
1	SAMPLE_SCH	DEPARTMENT	VPHONE	VIEW	SAMPLE_SCH
1	SAMPLE_SCH	DEPARTMENT	XDEPT2	INDEX	SAMPLE_SCH
1	SAMPLE_SCH	DEPARTMENT	VDEPT	VIEW	SAMPLE_SCH
1	SAMPLE_SCH	DEPARTMENT	VEMPDPT1	VIEW	SAMPLE_SCH
1	SAMPLE_SCH	DEPARTMENT	DEPT	ALIAS	SAMPLE_SCH
1	SAMPLE_SCH	DEPARTMENT	EMPLOYEE	FOREIGN KEY	SAMPLE_SCH
1	SAMPLE_SCH	DEPARTMENT	PROJECT	FOREIGN KEY	SAMPLE_SCH
2	SAMPLE_SCH	VDEPMG1	VASTRDE1	VIEW	SAMPLE_SCH
2	SAMPLE_SCH	VDEPMG1	VASTRDE2	VIEW	SAMPLE_SCH

File Edit View

LEVEL	SOURCE_SCHEMA_NAME	SOURCE_SQL_NAME	SQL_NAME	SQL_OBJECT_TYPE	SCHEMA_NAME
1	SAMPLE_SCH	DEPARTMENT	VHDEPT	VIEW	SAMPLE_SCH
1	SAMPLE_SCH	DEPARTMENT	XDEPT3	INDEX	SAMPLE_SCH
1	SAMPLE_SCH	DEPARTMENT	VDEPMG1	VIEW	SAMPLE_SCH
1	SAMPLE_SCH	DEPARTMENT	XDEPT1	INDEX	SAMPLE_SCH
1	SAMPLE_SCH	DEPARTMENT	VPHONE	VIEW	SAMPLE_SCH
1	SAMPLE_SCH	DEPARTMENT	XDEPT2	INDEX	SAMPLE_SCH
1	SAMPLE_SCH	DEPARTMENT	VDEPT	VIEW	SAMPLE_SCH
1	SAMPLE_SCH	DEPARTMENT	VEMPDPT1	VIEW	SAMPLE_SCH
1	SAMPLE_SCH	DEPARTMENT	DEPT	ALIAS	SAMPLE_SCH
1	SAMPLE_SCH	DEPARTMENT	EMPLOYEE	FOREIGN KEY	SAMPLE_SCH
1	SAMPLE_SCH	DEPARTMENT	PROJECT	FOREIGN KEY	SAMPLE_SCH
2	SAMPLE_SCH	VDEPMG1	VASTRDE1	VIEW	SAMPLE_SCH
2	SAMPLE_SCH	VDEPMG1	VASTRDE2	VIEW	SAMPLE_SCH

Figure. Find related objects by SQL service 2

Key Features:

- Returns direct and indirect dependencies
- Includes views, indexes, constraints, triggers, procedures and programs
- Recursive dependency tracking
- Handles complex dependency chains

Method 4: Asking IBM Bob to generate an overview of the database object dependencies.

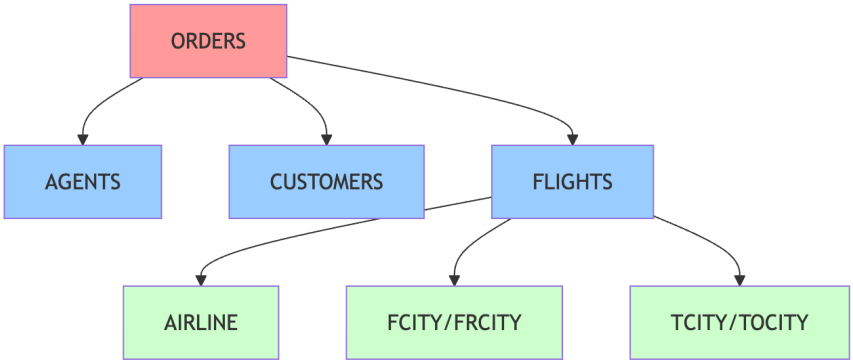
IBM Bob™ is an AI-powered assistant product that helps developers work with IBM i systems by providing intelligent code generation, explanation, modernization, database management, testing, DevOps automation, and comprehensive documentation capabilities across RPG, COBOL, CL, SQL, and DDS languages. In addition to the topics mentioned above, Bob can generate an overview of the database object relationships in Entity Relationship Diagram (ERD) format. By having a visual depiction of the data mode relationships, anyone working with the database can more easily understand the phyical and virtual data model relationships.

Table Relationships

Primary Relationships

ORDERS (Central Transaction Table)

The ORDERS table is the central transaction table that links all core entities:



Relationship Details:

1. **ORDERS → AGENTS**
   - Foreign Key: `AGENT_NO`
   - Relationship: Many-to-One
   - Business Rule: Each order must be booked by an agent
2. **ORDERS → CUSTOMERS**
   - Foreign Key: `CUSTOMER_NO`
   - Relationship: Many-to-One
   - Business Rule: Each order belongs to one customer
3. **ORDERS → FLIGHTS**
   - Foreign Key: `FLIGHT_NUMBER`
   - Relationship: Many-to-One
   - Business Rule: Each order is for one specific flight

Figure. Table Relationships generated by Bob

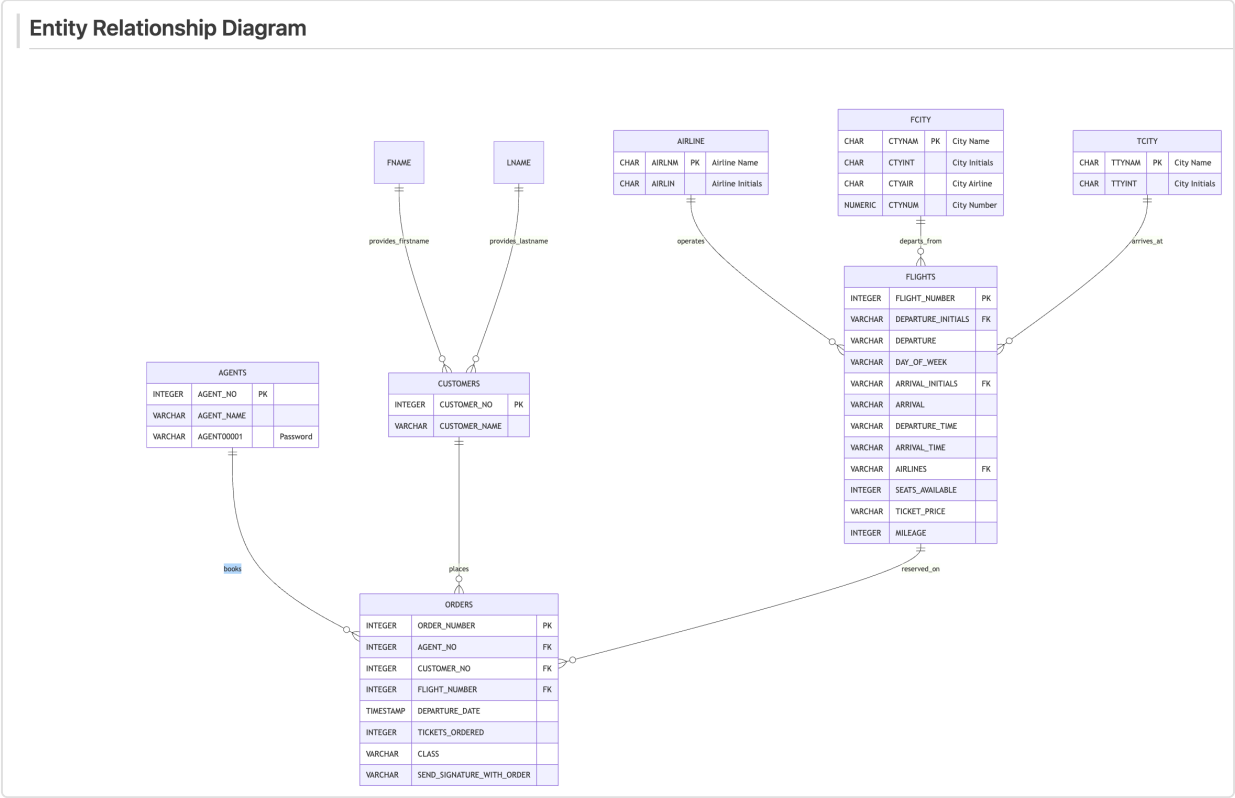


Figure. Entity Relationship Diagram generated by Bob

Method 5: Using CL Commands

IBM i also provides the Display Data Base Relations (DSPDBR) CL command for dependency analysis, which is considered the legacy approach to understanding database relationships. However, the other methods mentioned previously are easier to use and provide a more complete view into this important topic.

```
/* Display Database Relations */
DSPDBR FILE(FLGHT400/FLIGHTS) OUTPUT(*PRINT)
```

What DDS to DDL Conversion Tools Exist?

There are several tools available to assist with wanting to convert Db2 object definitions from DDS to SQL DDL (Data Definition Language). Some of the tools are built-in, and other tools are available from third parties.

Definition Conversion Tools

Db2 for i provides a Generate SQL feature that generates the SQL DDL statements required to recreate a database object. If this function is pointed at a physical file or logical file, it will generate the SQL DDL that is the closest match to the DDS definition.

The SQL DDL can be generated interactively by using the IBM i Access Client Solutions (ACS) interface, specifically the ACS Schemas tool shown in Figure 1. With the Schemas tool, physical files can be found by using the Tables folder in a library, and logical files can be found by using the Views folder.

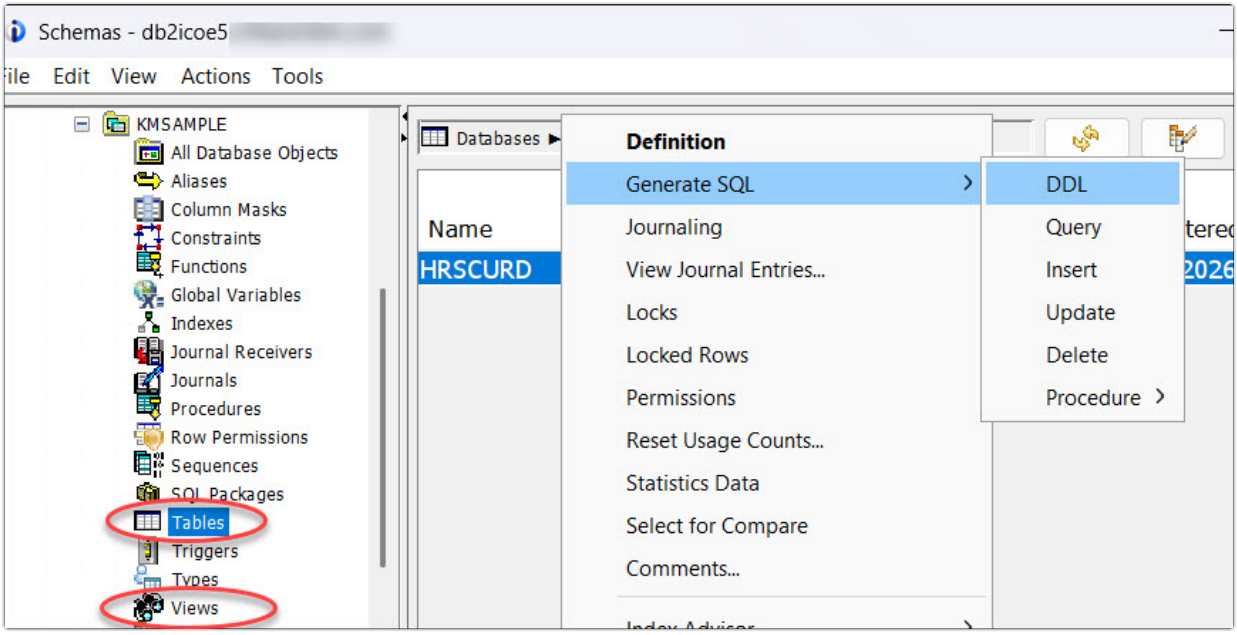


Figure. ACS Schemas

In this example, the HRSCURD physical file has been selected. The Generate SQL function is accessed by right-clicking the file object and selecting **Generate SQL** → **DDL**. This task launches a new Generate SQL window, as shown in the following figure.

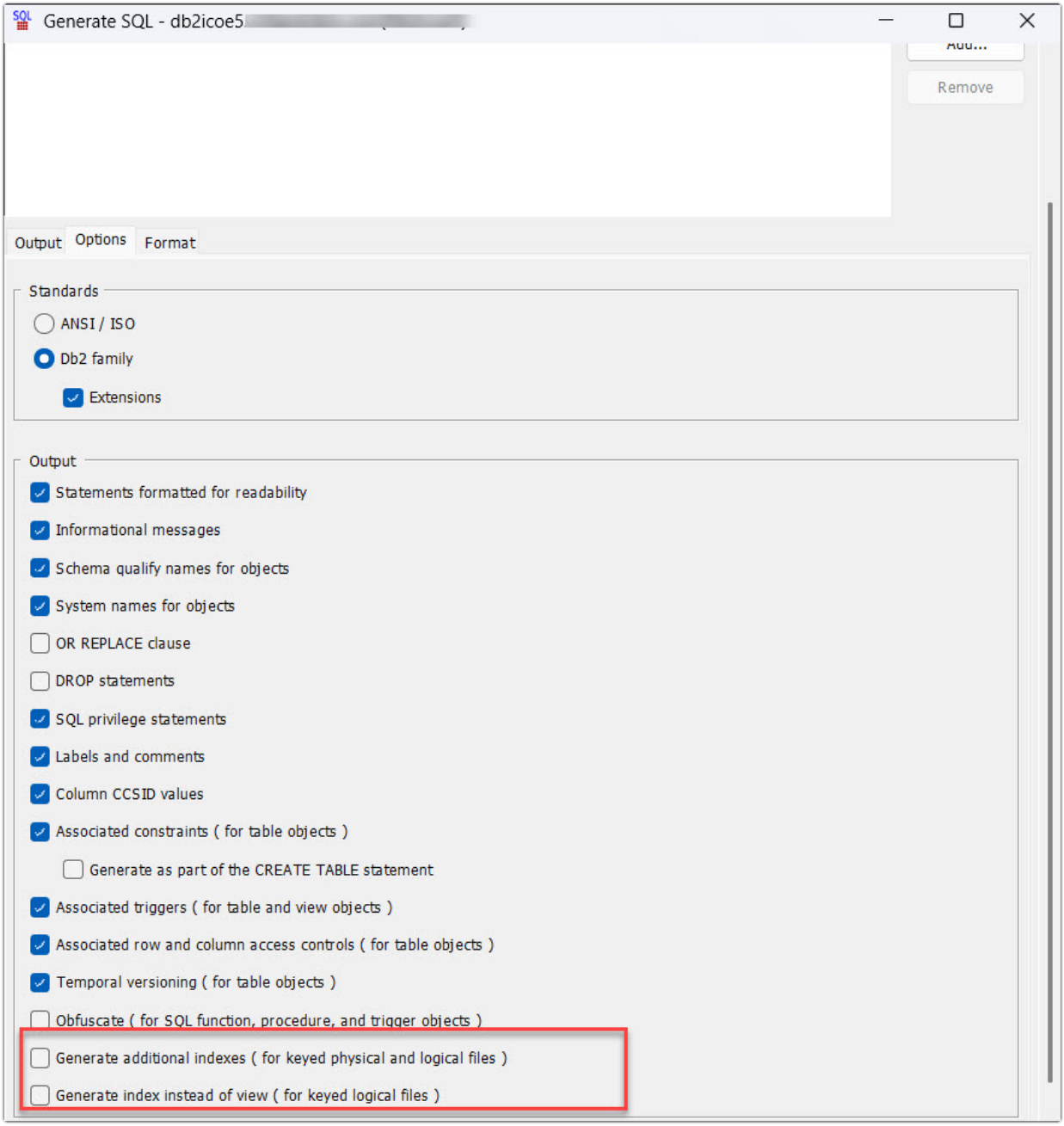


Figure. Generate SQL Options tab

In the Generate SQL window, the Output tab is displayed by default. When using the Generate SQL feature on DDS objects, however, you will want to visit the Options tab. At the bottom of the Options tab, there are two highlighted options that allow you to control specific behavior for physical and logical file objects.

Once the appropriate options have been selected, go back to the Output tab as shown in the following figure. The default output option is to place the generated SQL DDL in an ACS Run SQL Scripts window. Other supported output options are writing the output to a file on your local workstation, a file in the integrated file system (IFS), or a source physical file member.

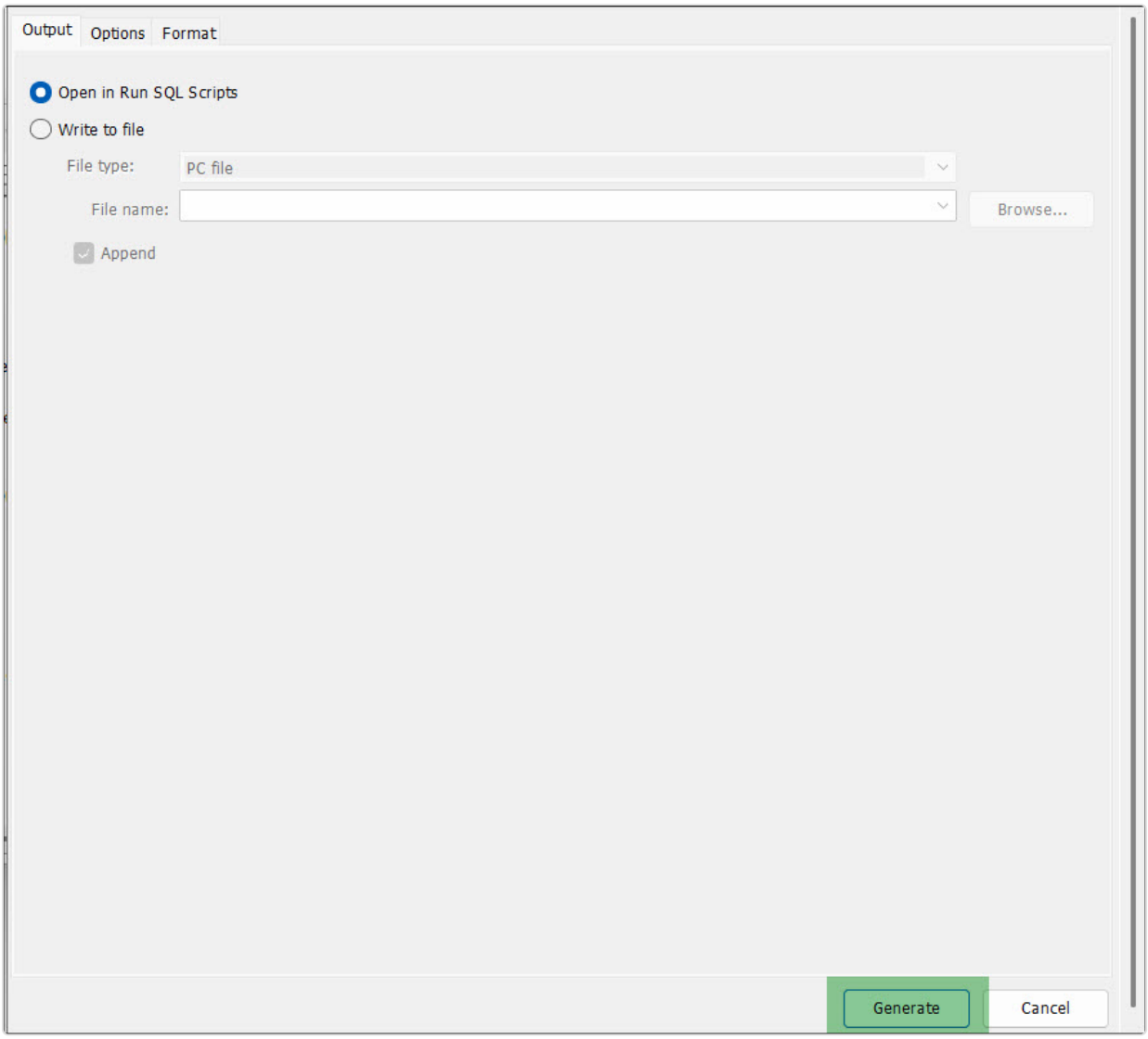


Figure. Generate SQL Output tab

Once the desired output option has been selected, click the highlighted Generate button. Figure 4 contains the Run SQL Scripts output of a Generate SQL request. Notice the comment lines in green that highlight file options such as the reused deleted records setting or the `EDTCDE` keyword that are not supported in the generated SQL DDL.

```
CREATE TABLE KMSAMPLE.HRSCURD (
--   SQL150B   10   REUSEDLT(*NO) in table HRSCURD in KMSAMPLE ignored.
--   SQL1506   30   Key or attribute for HRSCURD in KMSAMPLE ignored.
     EMPNO CHAR(6) CCSID 37 NOT NULL DEFAULT '' ,
     EMPWRKDAY CHAR(6) CCSID 37 NOT NULL DEFAULT '' ,
     EMPHOURS DECIMAL(5, 2) NOT NULL DEFAULT 0 )

     RCDFMT HRSCURR    ;
--   SQL150D   10   EDTCDE in column EMPHOURS ignored.

LABEL ON TABLE KMSAMPLE.HRSCURD
     IS 'Employee Hours By Day (Current Year)' ;

LABEL ON COLUMN KMSAMPLE.HRSCURD
( EMPNO IS 'EMP#' ,
     EMPWRKDAY IS 'Day                  (YYMMDD)' ,
     EMPHOURS IS 'Hours' ) ;
```

Figure. Generate SQL output example

If there are a large number of DDS-created objects that need to be converted, this feature can be accessed programmatically by using the `GENERATE_SQL` service. More details on this service can be found at: https://www.ibm.com/docs/en/i/latest?topic=services-generate-sql-procedure

Data Validation and Migration Tools

Once the SQL table definition is available, the next logical steps are creating the table and populating it with data from the physical file. Copying data from the physical file can be done with the Copy File (`CPYF`) system command or by using an SQL `INSERT ... SELECT` statement.

Before copying the data from the physical file, the data in the physical file should be validated. This validation is necessary because physical files do not provide the same strong data type enforcement as SQL tables. As a result, it is possible for an application using record-level interfaces to write invalid data to a field in a physical file. For example, dirty decimal data can be written to a packed decimal field in a physical file. Invalid data cannot be written to a column in an SQL table because SQL standards prohibit data from being inserted or updated into an SQL table when the data does not adhere to the rules of the target column's data type.

To assist with this data validation, Db2 for i provides tooling that evaluates whether invalidly formed data exists in a physical file. The `VALIDATE_DATA`, `VALIDATE_DATA_FILE`, and `VALIDATE_DATA_LIBRARY` table functions provided in the `SYSTOOLS` library are available for customers to verify whether the data in a physical file member contains invalid data such as dirty decimal data.

The following SQL example shows the `VALIDATE_DATA_FILE` function in action. This function processes all members within the specified input file `STORE42/BADDATA`. For each member, the tool evaluates all of the columns defined with a numeric data type and tries to retrieve those values by using SQL. This process of bringing physical file field values into SQL causes the SQL standard behavior to be applied, which allows the tool to discover instances of invalidly formed data.

```
SELECT *
  FROM TABLE (
    SYSTOOLS.VALIDATE_DATA_FILE(
      library_name => 'STORE42',
      file_name => 'BADDATA'
    )
  );
```

The following figure shows results that could be generated by the `VALIDATE_DATA_FILE` function. Note that there are 102 rows within the file that contain invalid data. The function provides the relative record numbers for the records containing invalid data, as well as the field name that contains the invalid data.

LIBRARY_NAME	FILE_NAME	MEMBER_NAME	RELATIVE_RECORD_NUMBER	COLUMN_NAME
STORE42	BADDATA	FIRSTMBR	189	SRCDAT
STORE42	BADDATA	FIRSTMBR	190	SRCDAT
STORE42	BADDATA	FIRSTMBR	191	SRCDAT
STORE42	BADDATA	FIRSTMBR	192	SRCDAT
STORE42	BADDATA	FIRSTMBR	193	SRCDAT
STORE42	BADDATA	FIRSTMBR	194	SRCDAT
STORE42	BADDATA	FIRSTMBR	195	SRCDAT
STORE42	BADDATA	FIRSTMBR	196	SRCDAT
STORE42	BADDATA	FIRSTMBR	197	SRCDAT
STORE42	BADDATA	FIRSTMBR	198	SRCDAT
STORE42	BADDATA	FIRSTMBR	199	SRCDAT
STORE42	BADDATA	FIRSTMBR	200	SRCDAT
STORE42	BADDATA	FIRSTMBR	201	SRCDAT
STORE42	BADDATA	FIRSTMBR	202	SRCDAT
STORE42	BADDATA	FIRSTMBR	203	SRCDAT
STORE42	BADDATA	FIRSTMBR	204	SRCDAT
STORE42	BADDATA	FIRSTMBR	205	SRCDAT
STORE42	BADDATA	FIRSTMBR	206	SRCDAT

```
Done: 102 rows retrieved.
```

Figure. VALIDATE_DATA_FILE function output

More details on the validate functions can be found at: https://www.ibm.com/docs/en/i/latest?topic=us-validate-data-validate-data-file-validate-data-library-table-functions

Third-party Tools

The following third-party tools can be used to streamline the DDS to DDL conversion process. The first two tools even include the ability to generate surrogate logical files. Surrogate logical files enable DDS objects to be converted to SQL without requiring application programs to be recompiled. None of the tools provided by IBM support the creation of surrogate logical files.

- ARCAD Transformer DB
- XCase for i
- Fresche X-Analysis

Open source languages on IBM i

PHP Development on IBM i

PHP: Hypertext Preprocessor

This section introduces the open source web scripting language known as **PHP**, with the discussion framed specifically for experienced **IBM i developers**. Because many excellent books and resources already exist on learning PHP syntax, this chapter does not focus heavily on language fundamentals. Instead, it highlights aspects of PHP that are most relevant when used on IBM i.

What Is PHP

PHP (Hypertext Preprocessor) is an open source, server-side scripting language that powers more than 70% of known websites worldwide and runs natively on IBM i. Its popularity stems from an easy-to-use syntax combined with a highly scalable and capable runtime. PHP allows developers to accomplish substantial functionality with relatively small amounts of code, a defining characteristic of effective scripting languages.

According to industry tracking data, PHP remains the most widely deployed server-side web language in use today.

Why PHP

Early Popularity

PHP's early and sustained success can be attributed largely to its timing. The early Internet consisted primarily of static HTML pages, which made managing dynamic or repetitive content difficult. As web usage began to grow rapidly, PHP introduced a simple way to combine HTML with database-driven content such as text, reports, or transactional data. This drastically simplified CGI-based development and gave PHP an early and lasting advantage.

Clear Focus

From its inception, PHP was designed specifically for web development. Unlike other popular languages—such as Java, which originated for device programming, or Python, which was built as a general-purpose language—PHP focused almost exclusively on the web. Aside from Node.js, few mainstream languages share this singular heritage. This strong focus allowed PHP engineers to concentrate on solving web-specific problems without the competing priorities seen in more general languages.

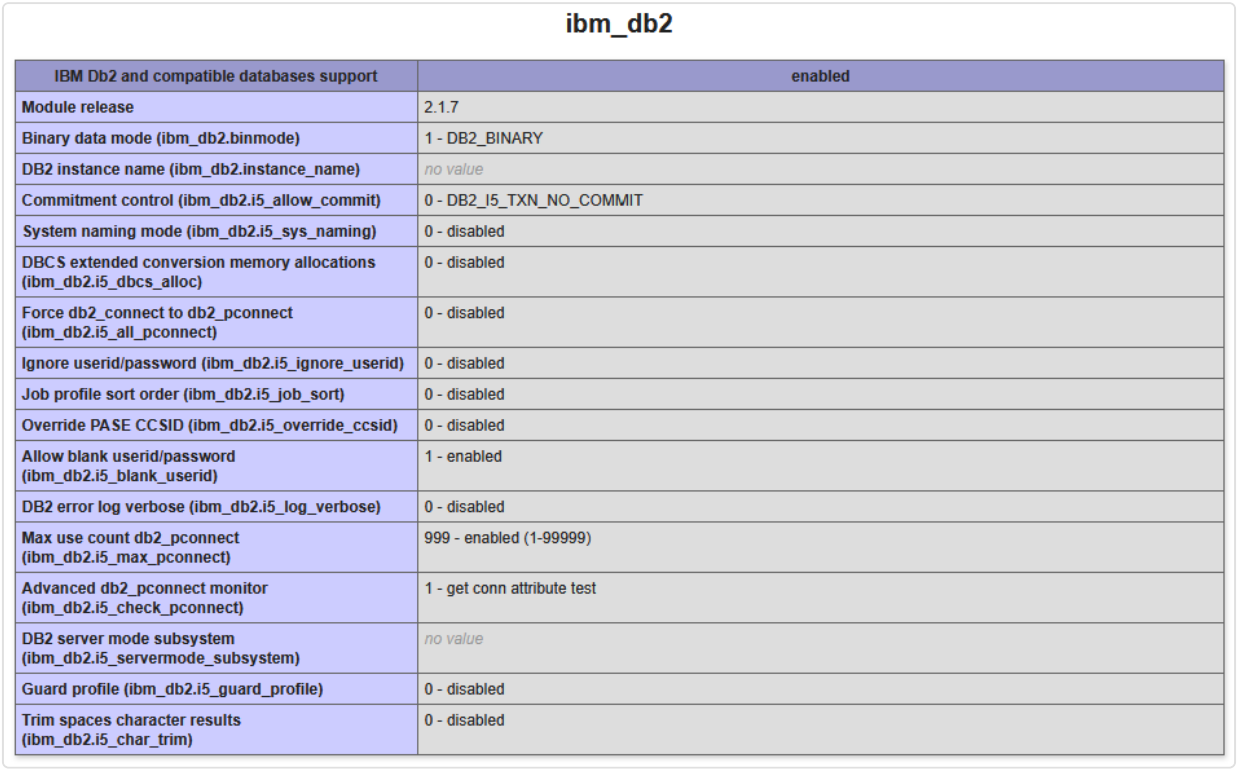
Content Is King

Open source content-management systems such as **WordPress**, **Joomla**, and **Drupal** account for a substantial percentage of PHP-based websites. These platforms gained popularity because they are easy to deploy, inexpensive to host, and feature-rich out of the box. This wave of CMS adoption was one of the earliest and most significant demonstrations of the power of open source software at web scale.

Ease of Use: Blessing and a Curse

PHP is famously easy to learn and use—even today. A simple PHP script can generate dynamic webpage output with only a few lines of code. This low barrier to entry allowed many non-traditional developers to build websites early on, which sometimes resulted in poorly written or insecure applications. As a result, PHP developed an unfair reputation for insecurity.

In recent major releases, PHP has addressed these criticisms through improved language design, stronger typing options, enhanced security features, and major performance gains. The official PHP documentation at https://www.php.net remains one of the most comprehensive and clearly written technical references available.



ibm_db2

IBM Db2 and compatible databases support	enabled
Module release	2.1.7
Binary data mode (ibm_db2.binmode)	1 - DB2_BINARY
DB2 instance name (ibm_db2.instance_name)	*no value*
Commitment control (ibm_db2.i5_allow_commit)	0 - DB2_I5_TXN_NO_COMMIT
System naming mode (ibm_db2.i5_sys_naming)	0 - disabled
DBCS extended conversion memory allocations (ibm_db2.i5_dbcs_alloc)	0 - disabled
Force db2_connect to db2_pconnect (ibm_db2.i5_all_pconnect)	0 - disabled
Ignore userid/password (ibm_db2.i5_ignore_userid)	0 - disabled
Job profile sort order (ibm_db2.i5_job_sort)	0 - disabled
Override PASE CCSID (ibm_db2.i5_override_ccsid)	0 - disabled
Allow blank userid/password (ibm_db2.i5_blank_userid)	1 - enabled
DB2 error log verbose (ibm_db2.i5_log_verbose)	0 - disabled
Max use count db2_pconnect (ibm_db2.i5_max_pconnect)	999 - enabled (1-99999)
Advanced db2_pconnect monitor (ibm_db2.i5_check_pconnect)	1 - get conn attribute test
DB2 server mode subsystem (ibm_db2.i5_servermode_subsystem)	*no value*
Guard profile (ibm_db2.i5_guard_profile)	0 - disabled
Trim spaces character results (ibm_db2.i5_char_trim)	0 - disabled

Figure. PHP Documentation Example

Mature Technology

PHP is now a mature, well-seasoned technology with decades of real-world use behind it. Millions of developers worldwide continue to maintain and extend PHP applications across content, commerce, and enterprise domains. This maturity, combined with ongoing innovation, allows PHP to remain competitive and relevant despite rapid changes in the technology landscape.

How Do I Install and Use PHP on IBM i?

Installing PHP

Although IBM has contributed extensively to PHP and the PHP community, PHP is one of the few open source projects that is not included directly in the standard Open Source Package Management (OSPM) offering. Instead, two organizations—**Perforce** and **Seiden Group**—provide free and supported PHP distributions specifically for the IBM i community. Competition between these vendors has resulted in strong, well-maintained options for running PHP on IBM i.

Perforce

Perforce, formerly Zend, has provided a no-charge PHP distribution for IBM i since the early beta releases in late 2005. With more than 20 years of experience supporting PHP on IBM i, Perforce offers a stable PHP stack with options for long-term support, including extended support for PHP versions that are no longer maintained by the broader PHP community.

Installation instructions are available on the Perforce website:

https://help.zend.com/zendphp/current/content/installation/ibmi_installation.htm

Seiden Group

Alan Seiden and his team at Seiden Group—early contributors to the PHP Toolkit for IBM i—created a competitive PHP distribution focused on core PHP capabilities while carefully addressing IBM i-specific features and nuances. Since 2017, this distribution has provided another strong option for deploying PHP on IBM i.

Installation instructions and documentation for the Seiden Group PHP distribution, including dashboard-based management, are available at:

https://www.seidengroup.com/

Key Points Regarding Installation

Both the Perforce and Seiden Group distributions:

- Install a current, supported version of PHP
- Automatically configure an instance of **IBM HTTP Server powered by Apache**

Simple Script

The following example illustrates a very simple PHP script, demonstrating how easily PHP can generate dynamic output. The script displays a page title, a heading, and two date values formatted using different PHP formatting options. PHP code can exist in standalone files or be embedded directly within HTML files.

The PHP source code executes on the server, generates output (in this case, HTML), and passes the result to Apache, which then delivers the response to the browser.

```
<!DOCTYPE html>
<html>

<head>
    <title>PHP Date example</title>
</head>

<body>
    <h1>Simple PHP Date Example</h1>
    <p>
        <?php

        // Sample date program
        $today = date("r");
        print("Today is $today<br /><br />\n\n");

        $today = date('m/d/y');
        print("Today is $today");

        ?>
    </p>
</body>

</html>
```

The sample output below shows the two formatted dates rendered in the browser.

Figure. PHP Script Output

When viewing the page source in the browser, only HTML is visible. The PHP code itself never reaches the browser; it runs entirely on the server and produces HTML or other content that Apache sends back to the requester.

Figure. Browser View Source

```
<!DOCTYPE html>
<html>

<head>
    <title>PHP Date example</title>
</head>

<body>
    <h1>Simple PHP Date Example</h1>
    <p>
        Today is Mon, 23 Mar 2026 00:20:18 +0000<br /><br />
        Today is 03/23/26
    </p>
</body>

</html>
```

The PHP Support Cycle

New PHP features and deprecations are introduced with each major release (for example, PHP 8.2). Maintenance releases (such as 8.2.5), typically released annually, are intended to provide fixes without introducing changes that could disrupt existing codebases.

Modern PHP has expanded its support lifecycle from three years to four years. During the first two years of a major release, both bug fixes and security updates are provided. During the remaining two years, only security fixes are issued. Full details of the support lifecycle are published on PHP.net.

PHP applications should be reviewed at least every other year to determine whether upgrading to a newer PHP version is appropriate. Many organizations incorporate PHP upgrade planning into their regular development cycles.

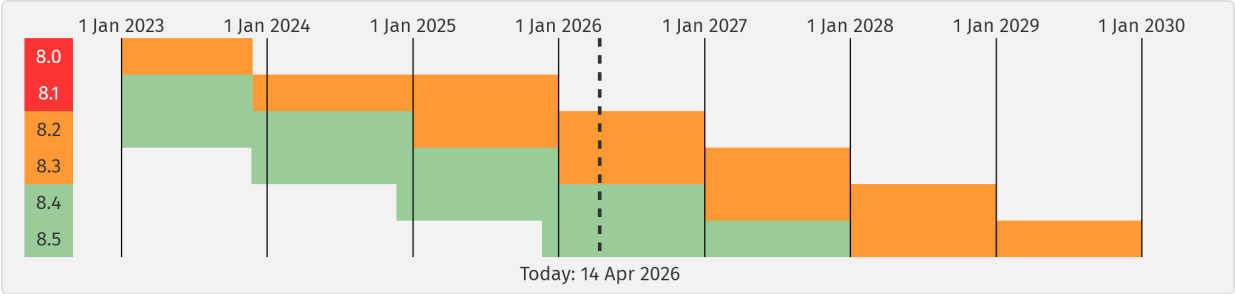


Figure. PHP Support Lifecycle

How Do I Use Composer and PSR Standards?

PHP Package Management

Open source languages and frameworks are extensible in many ways. For PHP there are two major categories: **extensions** and **packages**, though additional options exist beyond these categories.

Extensions are typically managed by the organization responsible for a given PHP distribution, such as Perforce or Seiden Group. These extensions are often visible in the output of the `phpinfo()` function. Because PHP is primarily used for web workloads, performance is a key consideration. As a result, many extensions are written in C, compiled, and built directly into the PHP runtime. The `php.ini` file controls which extensions are loaded into memory for a given installation. Each PHP distribution has its own approach for managing extensions, though default configurations generally meet most requirements.

Packages, on the other hand, are code libraries written primarily in PHP and installed alongside application source code. Although it is possible to manually download and extract these libraries into the IFS, the preferred approach is to use **Composer**, which retrieves packages from the main PHP package repository, Packagist.

Command-line access is typically recommended when working with Composer. In addition to installing requested libraries, Composer also functions as a dependency manager. This means that when a package is requested, all required dependencies are automatically installed. Each PHP distribution may use a slightly different approach to installing Composer, especially if it is not bundled by default, so following the distribution's documentation is advised.

The example below illustrates using Composer to install the phpSpreadsheet library. Many library projects provide explicit Composer commands for installation.

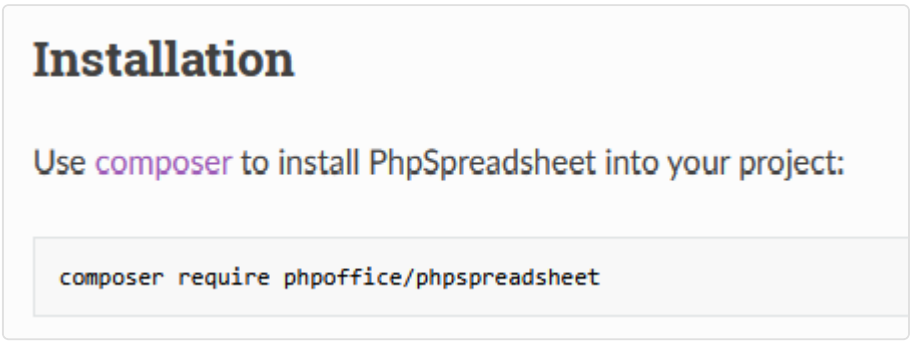


Figure. Composer Install Example

Once installed, PHP scripts can access the package through the `vendor` directory.

```
require 'vendor/autoload.php';

use PhpOffice\PhpSpreadsheet\Spreadsheet;
use PhpOffice\PhpSpreadsheet\Writer\Xlsx;

$spreadsheet = new Spreadsheet();
$sheet = $spreadsheet->getActiveSheet();
$sheet->setCellValue('A1', 'Hello World !');

$writer = new Xlsx($spreadsheet);
$writer->save('/tmp/helloworld.xlsx');
print("Sheet Saved successfully!");
```

Figure. Vendor Folder Example

The primary benefit of using a package manager is that installed libraries are well organized, making maintenance and PHP version upgrades easier to manage.

What Is PSR and the PHP-FIG?

At one time, many PHP frameworks and solutions implemented their own coding standards. While these standards were often similar, they differed enough to create compatibility challenges. This made it difficult to reuse components across frameworks.

The **PHP Standards Recommendation (PSR)** initiative addresses this issue by defining common standards for coding, provisioning, and consuming PHP components. While PSRs do not eliminate all interoperability challenges, they have significantly improved compatibility across frameworks.

The **PHP-FIG (Framework Interop Group)** is responsible for guiding and publishing PSR standards. These standards have enabled tools like Composer to thrive by providing a consistent approach to structuring and consuming PHP libraries. More information is available at https://www.php-fig.org.

PSRs are developed through collaboration among PHP-FIG members and are intended to benefit the broader PHP ecosystem. Although originally targeted at framework developers, many independent developers have also adopted these standards.

Tool vendors such as JetBrains have integrated PSR enforcement into development environments such as PHPStorm. This integration helps ensure adherence to common standards, improving consistency and productivity across development teams.

Most open source languages provide optional standards bodies to encourage consistency in how applications are built. These standards help ensure developers can collaborate and build applications on a common foundation.

How Do I Handle Dates in PHP?

Dates are essential to any transaction-based system. Storing and retrieving date values occurs in nearly every table within an ERP system, as well as in the ancillary systems that support core business processing. Understanding how to work with dates in PHP is therefore fundamental.

To ensure expected behavior when running these examples, the default PHP time zone—which is often set to GMT—may need to be adjusted in the `php.ini` file to match the local system time zone.

The following example demonstrates simple PHP source code that illustrates how PHP date formatting aligns conceptually with edit codes and edit words commonly used in RPG.

```php
<?php

$date = date('m/d/Y G:i:s');
echo "Hello World, today is $date";

echo "<br><br>";

$date = date('l, F j, Y');
$time = date('g:i:sa');
echo "Or, if you prefer a more formal format, then today is $date at $time";
```

Figure. PHP Date Formatting Example

```
Hello World, today is 04/14/2026 4:07:43

Or, if you prefer a more formal format, then today is Tuesday, April 14, 2026 at 4:07:43am
```

Figure. PHP Date Output

Notice how PHP retrieves the current time. The `date()` function accepts two parameters. In the examples shown, only one parameter is used, which specifies the desired output format. When the optional second parameter (the timestamp) is not supplied, PHP automatically defaults to the current timestamp.

All available parameters and formatting options for functions such as `date()` are documented on the PHP website at http://www.php.net. Rather than memorizing the thousands of functions available in PHP, it is more effective to understand how to quickly locate and reference them using the official documentation.

It is also important to understand that while Db2 and RPG provide native date data types, PHP does not. PHP supports only four scalar data types: **string**, **integer**, **float**, and **boolean**. When date or numeric values are retrieved from a Db2 table, PHP receives them as either strings or integers. Boolean values represent true/false fields, while floats are used only when decimal precision is present.

PHP includes powerful **DateTime** classes as part of the standard language runtime. These classes are thoroughly documented on php.net and function identically on PHP environments running on IBM i.

Java Development on IBM i

What is Java and the Java Ecosystem?

Overview of Java

Java is a high-level, object-oriented programming language developed by Sun Microsystems (now owned by Oracle) in 1995. It was designed with the principle of "Write Once, Run Anywhere" (WORA), meaning compiled Java code can run on any platform that supports Java without recompilation.

Key Characteristics of Java

- **Platform Independent**: Java bytecode runs on the Java Virtual Machine (JVM), making it portable across different operating systems
- **Object-Oriented**: Everything in Java is organized around objects and classes
- **Strongly Typed**: Type checking occurs at compile time, reducing runtime errors
- **Automatic Memory Management**: Garbage collection handles memory deallocation
- **Multi-threaded**: Built-in support for concurrent programming
- **Secure**: Built-in security features and sandboxing capabilities
- **Robust**: Strong exception handling and type checking

The Java Ecosystem

The Java ecosystem is a comprehensive collection of technologies, tools, frameworks, and libraries that support Java development:

Core Components

1. **Java Development Kit (JDK)**: Complete development environment including compiler, debugger, and runtime
2. **Java Runtime Environment (JRE)**: Provides libraries and JVM to run Java applications
3. **Java Virtual Machine (JVM)**: Executes Java bytecode and provides platform independence

Popular Frameworks and Libraries

- **Spring Framework**: Enterprise application development
- **Jakarta EE (formerly Java EE)**: Enterprise-grade server-side applications
- **Hibernate**: Object-relational mapping (ORM)
- **Apache Maven/Gradle**: Build automation and dependency management
- **JUnit**: Unit testing framework

Java on IBM i

IBM i (formerly AS/400, iSeries, System i) has robust Java support, making it a powerful platform for Java development and deployment.

Why Java on IBM i?

1. **Native Integration**: Java runs natively on IBM i, not in emulation
2. **Access to IBM i Resources**: Direct access to DB2 for i, data queues, user spaces, and other IBM i objects
3. **Modernization Path**: Allows modernization of legacy RPG/COBOL applications
4. **Web Services**: Easy creation of REST and SOAP web services
5. **Performance**: Optimized JVM implementation for IBM i architecture

IBM i Java Implementation

Java Versions on IBM i

IBM i supports multiple Java versions:

- Java 8 (LTS - Long Term Support)
- Java 11 (LTS)
- Java 17 (LTS)
- Java 21 (LTS)

Each version is available as a Licensed Program Product (LPP) and can be installed via PTFs (Program Temporary Fixes).

Key IBM i Java Features

1. Toolbox for Java (JTOpen)

- Open-source library for accessing IBM i resources
- Provides classes for:
  - Database access (JDBC)
  - Program calls
  - Data queue operations
  - IFS file access
  - Command execution
  - Print services
  - User space manipulation

2. JDBC Driver

- Native JDBC driver for DB2 for i
- Two types:
  - **Toolbox JDBC driver**: Pure Java, can run remotely
  - **Native JDBC driver**: Optimized for local IBM i execution

3. Integration with IBM i Objects

```
// Example: Calling an RPG program from Java
AS400 system = new AS400("systemName", "user", "password");
ProgramCall program = new ProgramCall(system);
program.setProgram("/QSYS.LIB/MYLIB.LIB/MYPGM.PGM");
program.run();
```

4. Web Application Support

- Apache Tomcat
- WebSphere Application Server
- Open Liberty (lightweight Jakarta EE runtime)

Java Development on IBM i

Development Environments

1. **IBM Rational Developer for i (RDi)**
   - Eclipse-based IDE
   - Integrated debugging
   - Remote Systems Explorer (RSE)
   - Source code management
2. **VS Code with IBM i Extensions**
   - Modern, lightweight alternative
   - Code for IBM i extension
   - Remote development capabilities
3. **Command Line Tools**
   - QShell (QSHELL)
   - PASE (Portable Application Solutions Environment)
   - SSH access for remote development

Running Java on IBM i

Methods to Execute Java:

1. **RUNJVA Command**

```
RUNJVA CLASS(com.example.MyClass) CLASSPATH('/myapp/lib/*')
```

2. **QShell/PASE**

```
java -cp /myapp/lib/* com.example.MyClass
```

3. **Batch Jobs**

```
SBMJOB CMD(RUNJVA CLASS(com.example.MyClass)) JOB(MYJAVA)
```

Common Use Cases for Java on IBM i

1. **Web Services Layer**
   - Expose RPG programs as REST APIs
   - Modernize green-screen applications with web interfaces
   - Integration with external systems

2. **Batch Processing**
    - Data transformation and ETL processes
    - Report generation
    - Scheduled tasks using job scheduler
3. **Database Access**
    - Complex SQL operations
    - Data analytics
    - Integration with modern frameworks
4. **Middleware**
    - Message queue processing
    - Event-driven architectures
    - Integration hub for multiple systems

Best Practices for Java on IBM i

1. **Use Connection Pooling**: Reuse database connections for better performance
2. **Leverage Native JDBC**: Use native driver for local applications
3. **Optimize JVM Settings**: Configure heap size and garbage collection appropriately
4. **Use PASE for Modern Tools**: Maven, Gradle, and modern build tools work well in PASE
5. **Implement Proper Error Handling**: Handle IBM i-specific exceptions
6. **Security**: Use IBM i security features (profiles, authorities)
7. **Logging**: Integrate with IBM i job logs and message queues

Integration Patterns

Java ↔ RPG Integration:

1. **Program Calls**: Java calls RPG programs via JTOpen
2. **Data Queues**: Asynchronous communication
3. **Database**: Shared DB2 for i tables
4. **Web Services**: RPG exposed via Java REST services
5. **User Spaces**: Shared memory communication

Performance Considerations

- **JVM Tuning**: Adjust heap size based on application needs
- **Connection Management**: Pool database connections
- **Caching**: Implement caching strategies for frequently accessed data
- **Batch Processing**: Use batch operations for database updates
- **Native Optimization**: IBM i JVM is optimized for the platform

Advantages of Java on IBM i

1. **Modernization Without Migration**: Keep data on IBM i while modernizing applications
2. **Skills Availability**: Large pool of Java developers
3. **Rich Ecosystem**: Access to thousands of open-source libraries
4. **Cloud Integration**: Easy integration with cloud services
5. **Mobile Support**: Backend for mobile applications
6. **Cross-Platform Development**: Same code can run on IBM i and other platforms

Conclusion

Java on IBM i represents a powerful combination of modern programming capabilities with the reliability and security of the IBM i platform. It provides a bridge between legacy systems and modern application architectures, enabling organizations to modernize their applications while leveraging existing investments in IBM i infrastructure and data.

The Java ecosystem on IBM i continues to evolve, with IBM providing regular updates, new tools, and enhanced integration capabilities, making it an excellent choice for both new development and modernization projects.

What is the Java Learning Curve?

What This Covers

This section examines the learning curve for Java development on IBM i, particularly for developers transitioning from traditional IBM i languages like RPG and COBOL. You'll understand the key challenges and advantages, learn strategies for accelerating proficiency, and discover how the learning investment pays dividends in modernization efforts.

What It Is

The Java learning curve for IBM i developers represents the time, effort, and cognitive adjustment required to become productive with Java while leveraging existing IBM i knowledge. Unlike learning an entirely new platform, IBM i developers bring valuable assets to Java development:

Existing Business Logic Understanding: Developers already understand the business domain, data structures, and application workflows-knowledge that often takes years to acquire and is more valuable than syntax familiarity.

Database Expertise: Experience with Db2 for i translates directly to Java database programming. Concepts like SQL optimization, transaction management, and data integrity remain consistent across languages.

System Architecture Knowledge: Understanding IBM i's job model, security architecture, and resource management provides context that helps developers make better architectural decisions in Java applications.

The learning curve itself consists of several distinct phases:

Syntax and Fundamentals (2-4 weeks): Learning Java syntax, object-oriented concepts, and basic language features. For developers experienced in procedural languages like RPG, this involves understanding classes, objects, inheritance, and polymorphism.

Framework Proficiency (1-2 months): Gaining competence with essential frameworks like Spring Boot, JDBC, and JT400. This phase involves understanding dependency injection, configuration management, and framework conventions.

IBM i Integration Mastery (1-3 months): Learning to effectively integrate Java with existing IBM i resources-calling RPG programs, accessing data queues, managing transactions, and leveraging platform-specific features.

Advanced Patterns (6-12 months): Developing expertise in design patterns, microservices architecture, performance optimization, and enterprise-scale application development.

Why It Matters

Understanding the Java learning curve is crucial for organizations planning IBM i modernization initiatives:

Resource Planning and Talent Strategy: Accurate learning curve estimates enable realistic project timelines and resource allocation. Organizations must decide whether to train existing RPG developers in Java, hire Java developers and train them on IBM i, or pursue a hybrid approach-each with different implications and cost structures.

Risk Management: The learning curve directly impacts project risk. Teams with insufficient Java expertise may make architectural mistakes, create performance bottlenecks, or build applications that don't properly integrate with IBM i resources.

Competitive Advantage: Organizations that successfully navigate the Java learning curve can modernize faster, attract broader talent pools, and build applications that leverage both IBM i's reliability and Java's extensive

ecosystem. The learning investment delivers long-term productivity gains, reduced maintenance costs, and improved application capabilities.

How It Works

The Java learning journey for IBM i developers follows a structured progression that builds on existing knowledge:

1. Foundation Building

Initial learning focuses on core Java concepts:

- **Object-oriented programming**: Classes, objects, inheritance, interfaces, and polymorphism
- **Java syntax**: Variables, control structures, methods, and exception handling
- **Development environment**: Setting up IDEs (Eclipse, IntelliJ, VS Code), understanding project structure
- **Build tools**: Maven or Gradle for dependency management and builds

Accelerators: Modern AI-assisted tools like IBM Bob™ can significantly reduce this phase by explaining Java concepts in terms familiar to RPG developers and generating example code with explanations.

2. Database Integration

Leveraging existing database knowledge:

- **JDBC fundamentals**: Connection management, prepared statements, result sets
- **Connection pooling**: Efficient resource management for production applications
- **Transaction management**: Ensuring data consistency across operations

RPG developers' deep understanding of Db2 for i makes this phase relatively straightforward.

3. Framework Adoption

Learning industry-standard frameworks:

- **Spring Boot**: Rapid application development with dependency injection and auto-configuration
- **REST APIs**: Building web services using Spring MVC or JAX-RS
- **Security**: Authentication, authorization, and secure communication
- **Testing**: JUnit for unit testing, integration testing strategies

Framework conventions can initially confuse developers accustomed to explicit, procedural code.

4. IBM i Integration Patterns

Mastering platform-specific integration using the JTOpen library:

- **Program calls**: Invoking RPG, COBOL, or CL programs from Java
- **Data queues**: Asynchronous communication with native applications
- **Service programs**: Calling ILE service program procedures
- **Stored procedures**: Calling SQL procedures and user-defined functions
- **Hybrid architectures**: Designing systems that leverage both Java and native code

Existing IBM i knowledge makes this phase intuitive for designing effective integration patterns.

5. Advanced Concepts and Learning Accelerators

Developing expertise in design patterns, microservices architecture, performance optimization, and DevOps practices is an ongoing process. Several factors significantly reduce the learning curve:

AI-Assisted Development: Tools like IBM Bob provide contextual guidance, explain unfamiliar concepts, generate example code, and help debug issues-effectively providing a 24/7 personal tutor.

Structured Training and Mentorship: Formal courses designed for IBM i developers learning Java, combined with mentorship programs pairing experienced Java developers with IBM i developers, accelerate learning through practical guidance.

Incremental Projects: Starting with small, well-defined projects (like a simple REST API) builds confidence before tackling complex integrations, while IBM i-specific Java communities provide targeted guidance for common challenges.

Key Takeaways

- **The Java learning curve for IBM i developers is manageable and accelerated by existing platform knowledge**, particularly database expertise, business logic understanding, and system architecture familiarity that provide valuable context for Java development
- **The learning progression follows distinct phases**-from syntax fundamentals (2-4 weeks) through framework proficiency (1-3 months) to IBM i integration mastery (2-4 months)-with each phase building on previous knowledge and existing IBM i expertise
- **Modern AI-assisted tools dramatically reduce the learning curve** by providing contextual guidance, explaining concepts in familiar terms, generating example code, and acting as always-available mentors that help developers overcome obstacles quickly
- **Organizations must strategically plan for the learning investment**, considering whether to train existing RPG developers, hire Java developers, or pursue hybrid approaches-with each strategy having different timelines, costs, and risk profiles
- **The learning investment delivers substantial long-term returns** through increased productivity, access to broader talent pools, modern application capabilities, and the ability to leverage Java's extensive ecosystem while preserving IBM i investments

Java 21 LTS Features and Examples

What is Java 21 LTS?

Java 21 LTS was released in 2023 and is the highest Java LTS (Long-Term Support) version supported on IBM i as of this writing.

Why It Matters

LTS stands for Long-Term Support. Any Java version flagged with LTS is guaranteed to be supported and to receive bug fixes and security updates for several years. It will remain an industry standard for a long period of time. It is scheduled to receive support up until 2031.

Using a Java LTS version means more stability and avoid frequent updates that may introduce regressions.

Key features of Java 21

Pattern Matching Enhancements

Java 21 improves pattern matching for `switch` expressions and statements, allowing developers to match patterns directly instead of writing complex conditional logic. `null` values can also be tested directly into switch statements.

Record Patterns

Record patterns makes it easier to deconstruct records and access their components without calling getter methods repeatedly.

Virtual threads

Virtual threads can run task quickly without requiring much resources and memory. A virtual thread doesn’t rely on the operating system thread during its cycle of operation. Virtual threads are decoupled from the hardware, that's for the virtual part.

Key Encapsulation Mechanisms

A new API for secure key exchanges, improving cryptographic security. This new API fills a long-standing gap in the Java Cryptography Architecture (JCA). It provides a standard, provider-neutral way to securely establish shared symmetric keys using public-key cryptography.

Vector API

A new API that enabling high-performance computations. A vector computation is a sequence of operations that work on multiple data elements at the same time, such as complex math and array operations. It can be much faster and more efficient than a scalar computation, which works on one data element at a time. The Vector API is ideal for scientific calculation tasks.

New Garbage Collector: ZGC

The new Z Garbage Collector has been introduced to clean up the memory concurrently without stopping application threads.

It divides memory in two groups:

- "New" objects are considered short-lived and are cleaned up frequently
- "Old" objects, on the contrary, are retained to avoid unnecessary cleanup

This new garbage collection mechanism ensures efficient memory management and keeps applications fast and responsive.

On IBM i

Since September 2025, Java 21 LTS has been released for IBM i 7.6 OS as option 21 of the 5770JV1 product. It is included with the IBM i OS Standard Set Media disc, B_GROUPx_05. It requires SF99965 level 1 or higher.

Key Takeaways

- Java 21 LTS is the highest long-term support version available on IBM i
- Using an LTS version means more stability, security and performance
- Using the highest supported version of Java gives access to the latest language features

How do I use Java 21 features?

What this covers

This topics demonstrate how to use the new language features brought by Java 21.

Before we start

Make sure Java 21 has been installed on your partition. Java 21 is option 21 of the 5770JV1 licensed product. It is included with the IBM i OS Standard Set Media disc, B_GROUPx_05. It requires SF99965 level 1 or higher.

Pattern Matching Enhancements

Java 21 improves pattern matching for `switch` expressions and statements, allowing developers to match patterns directly instead of writing complex conditional logic. `null` values can also be tested directly into switch statements.

In the example below, the `getCommonName` method takes advantage of the pattern matching enhancement.

```
abstract class People {
    String getName(){
       return "John Doe";
    }
}

class Employee extends People {
    String getTitle() {
        return "Engineer " + getName();
    }
}

class Friend extends People {
    String getNickName() {
        return "Seb";
    }
}

static String getCommonName(People someone) {
    //The object itself is used in the switch expression
    switch (someone) {
        case null -> throw new RuntimeException("Oops, someone is null");
        case Employee employee -> return employee.getTitle();
        case Friend friend -> return friend.getNickName();
        default -> return someone.getName();
    };
}
```

Before Java 21, this would have been:

```
if(someone instanceof Employee){
    return ((Emloyee)someone).getTitle();
}
else if(someone instanceof Friend){
    return ((Friend)someone).getNickName();
}
//etc
```

As you can see, the new pattern matching mechanism allows for a clearer and more concize syntax.

Record Patterns

Record patterns makes it easier to deconstruct records and access their components without calling getter methods repeatedly.

```
record Program(String library, String name) {}
...
public static String getPath(Object obj) {
    if(obj instanceof Program(String library, String name)) {
        return library + "/" + name;
    }
    return null;
}
```

Record patterns are flexible. A pattern variable's type only needs to be compatible with the record's components. In the below example `var` is used instead:

```
record Program(String library, String name) {}
Program pgm = new Program("SAMPLE", "TESTPARMS");

...

if(pgm instanceof Program(var lib, var name)) {
   String path = lib + "/" + name;
}
```

You can use record patterns in switch, which now supports pattern matching with Java 21.

```
record Program(String library, String name) {}
record ServiceProgram(String library, String name, String procedure) {}

...

String printedPath = switch(person){
    case Program(var lib, var name) -> lib + "/" + name;
    case ServiceProgram(var lib, var name, var proc) -> String.format("%s/%s(%s)", lib, parm, proc);
};
```

Vector API

Vector computations simultaneously process multiple data elements through operations like mathematical calculations and array manipulations. You can use vector computation to add two arrays of numbers element-by-element, for example.

```
final int[] array1 = {14, 52, 43, 94};
final int[] array2 = {1, 2, 3, 4};
final int[] array3 = new int[4];

// Vector computation
for (int i = 0; i < c.length; i++) {
  array3[i] = a[i] + b[i];
}
```

A vector computation can be much faster and more efficient than a scalar computation, which works on one data element at a time. For example, if you want to do the same array addition with a scalar computation, you would need to do something like this:

```
final int[] array1 = {14, 52, 43, 94};
final int[] array2 = {1, 2, 3, 4};
final int[] array3 = new int[4];

// Scalar computation
array4[0] = array1[0] + array2[0];
array4[1] = array1[1] + array2[1];
array4[2] = array1[2] + array2[2];
array4[3] = array1[3] + array2[3];
```

Scalar computation is slower. The key difference is the vector approach uses a loop to process all elements efficiently in one operation, while the scalar approach requires separate statements for each element. Modern CPUs can execute the vector approach much faster using SIMD (Single Instruction Multiple Data) instructions.

```
final VectorSpecies<Integer> SPECIES = IntVector.SPECIES_PREFERRED;
final int[] array1 = {14, 52, 43, 94};
final int[] array2 = {1, 2, 3, 4};
final int[] array3 = new int[4];

// Vector computation using Java 21 Vector API
for (int i = 0; i < array1.length; i += SPECIES.length()) {
    final IntVector vector1 = IntVector.fromArray(SPECIES, array1, i);
    final IntVector vector2 = IntVector.fromArray(SPECIES, array2, i);
    final IntVector result = vector1.add(vector2);
    result.intoArray(array3, i);
}
```

Considering the above code will run on POWER which supports SIMD, the add operation adds all the lanes of the two vectors in the same CPU cycle, making it more efficient than the first example.

Virtual threads

Virtual threads are lightweight instances of `java.lang.Thread` that are decoupled from OS threads. While they execute on OS threads, when a virtual thread encounters a blocking I/O operation, the Java runtime suspends it and frees the underlying OS thread to handle other virtual threads. This allows a small number of OS threads to support a large number of virtual threads—similar to how virtual memory maps a large address space to limited physical RAM.

Virtual threads typically maintain shallow call stacks, often containing just a single operation like an HTTP request or database query. Although they support thread-local variables, use these cautiously since a single JVM can manage millions of virtual threads, which could lead to significant memory overhead.

Virtual threads excel at handling I/O-bound tasks that spend most of their time waiting for I/O operations to complete. However, they're not designed for long-running, CPU-intensive operations that require continuous processor time.

The following snippet creates and start a Virtual Thread to run a long process:

```
final Thread virtualThread = Thread.ofVirtual().start(Process::aLongProcess);
```

Same example but using a `ThreadBuilder` to start two virtual threads:

```
final Thread.Builder builder = Thread.ofVirtual().name("virtual-thread-", 0);

//"virtual-thread-0"
final Thread t1 = builder.start(Process::aLongProcess);
//virtual-thread-1"
final Thread t2 = builder.start(Process::anotherLongProcess);
t1.join();
t2.join();
```

Executors let you to separate thread management and creation from the rest of your application. The following example creates an `ExecutorService` with the `Executors.newVirtualThreadPerTaskExecutor()` method. Whenever `ExecutorService.submit(Runnable)` is called, a new virtual thread is created and started to run the task. This method returns an instance of `Future`. Note that the method `Future.get()` waits for the thread's task to complete. Consequently, this example prints a message once the virtual thread's task is complete. The `ExecutorService` is `AutoCloseable` so make sure to put it in a `try-with-resource` block:

```
try (final ExecutorService myExecutor = Executors.newVirtualThreadPerTaskExecutor()) {
    final Future<?> future1 = myExecutor.submit(Process::aLongProcess);
    final Future<?> future2 = myExecutor.submit(Process::anotherLongProcess);
    future1.get();
    future2.get();

    //...
}
```

Key Encapsulation Mechanisms

Java 21 introduced a new API for secure key exchanges, improving cryptographic security and filling a long-standing gap in the Java Cryptography Architecture (JCA). It provides a standard, provider-neutral way to securely establish shared symmetric keys using public-key cryptography.

It supports several standardized KEMs:

- RSA-KEM: A KEM built on top of RSA key pairs.
- Elliptic Curve Integrated Encryption Scheme (ECIES): Elliptic-curve-based scheme.
- DHKEM: Diffie-Hellman KEM defined in RFC 9180.

Providers can add new KEM implementations, such as post-quantum KEMs like Kyber, without changing the API logic.

Encapsulating a Shared Secret

Here is an example of a simple method to encapsulate a secret using KEM. The `KEMOUtput` record will hold the key that both parties will use and the opaque byte array sent to the peer so they can derive the same shared secret.

The `encapsulate(...)` method does the sender-side KEM flow. kem.newEncapsulator(publicKey) prepares an encapsulator using the recipient's public key. encapsulator.encapsulate() creates: a fresh SecretKey (the shared secret), and the corresponding encapsulation bytes to transmit. The method returns both pieces as a single record instance.

```
public record KEMOUtput(SecretKey sharedSecret, byte[] encapsulation) {}

public static KEMOUtput encapsulate(String algorithm, PublicKey publicKey) throws Exception {
    //Load a KEM implementation by name (for example, "DHKEM" or "ECIES")
    final KEM kem = KEM.getInstance(algorithm);

    //Prepare an encapsulator using the recipient's public key.
    final KEM.Encapsulator encapsulator = kem.newEncapsulator(publicKey);

    //create a new SecretKey (the shared secret) and the encapsulated bytes to send
    final KEM.Encapsulated result = encapsulator.encapsulate();

    return new KEMOUtput(result.key(), result.encapsulation());
}
```

The sender must then transmits the opaque byte array to the receiver, while keeping the `sharedSecret` private for encryption or key derivation.

Decapsulating the Shared Secret

Let's now look at the method used on the receiver side to decapsulate the received opquae byte array.

```
public static KEMOUtput decapsulate(String algorithm, PrivateKey privateKey, byte[] encapsulation) throws
    //Load a KEM implementation by name (for example, "DHKEM" or "ECIES")
    final KEM kem = KEM.getInstance(algorithm);

    //Reconstruct the shared symmetric key (SecretKey) from the encapsulation blob received from the sende
    final KEM.Decapsulator decapsulator = kem.newDecapsulator(privateKey);
    final SecretKey recoveredSecret = decapsulator.decapsulate(encapsulation);

    //Returns the recovered secret and encapsulated data
    return new KEMOUtput(recoveredSecret, encapsulation);
}
```

Encapsulate/decapsulate flow

Here's is an example that can run locally (even though this mechanism is intended to be used between two remote parties).

```
//Generate a key pair
final KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance("X25519");
final KeyPair keyPair = keyPairGenerator.generateKeyPair();

final KEMOUtput senderResult = KemUtils.encapsulate("DHKEM", keyPair.getPublic());

final KEMOUtput receiverResult = KemUtils.decapsulate("DHKEM", keyPair.getPrivate(), senderResult.encapsul

final SecretKey senderSecret = senderResult.sharedSecret();
final SecretKey receiverSecret = receiverResult.sharedSecret();

//At this point, senderSecret.getEncoded() and receiverSecret.getEncoded() must match
```

What’s Up with Security, Open Source, and IBM i?

For decades, IBM i has earned a reputation as one of the most secure and stable enterprise platforms for business. Its integrated architecture, object-based design, and tightly controlled security model have long differentiated it from other operating systems. At the same time, the platform has evolved significantly. Modern IBM i environments are no longer isolated systems—they participate actively in hybrid architectures that rely heavily on open source software.

Apache, PHP, Python, Node.js, OpenSSL, Git, and many other open source technologies now run natively on IBM i in the PASE environment. While this dramatically expands capability, it also introduces security considerations that may be unfamiliar to teams with deep roots in traditional RPG and CL development. Understanding how open source interacts with the IBM i security model is essential.

Why Open Source Matters on IBM i

Open source on IBM i is not intended to replace RPG or Db2. Instead, it extends the platform. Apache enables modern web delivery. Python facilitates automation, analytics, and integration. PHP and Node.js accelerate web application development. Open source libraries allow teams to reuse proven solutions rather than reinventing common functionality.

From a business perspective, open source enables IBM i organizations to:

- Modernize user interfaces and APIs
- Integrate with cloud and SaaS platforms
- Attract developers with contemporary skills
- Reduce development time through reusable components

PASE, QOpenSys, and the IFS: A Different Security Model

One of the most significant shifts introduced by open source on IBM i involves PASE and the Integrated File System (IFS). Most open source software installs into the `/QOpenSys` directory tree, which behaves more like a UNIX file system than traditional IBM i libraries.

Unlike native IBM i libraries secured via object authorities, the IFS relies heavily on POSIX-style permissions:

- Read (r)
- Write (w)
- Execute (x)

These permissions apply to users, groups, and others and coexist alongside—but are not controlled by—traditional IBM i authority mechanisms.

As a result, an IBM i system may appear secure at the object level while remaining exposed at the IFS level if permissions are not carefully managed. For example, granting public-write access to a web directory can allow unprivileged users to inject malicious code that Apache may later execute.

Apache on IBM i: Power and Responsibility

The IBM HTTP Server powered by Apache is often the first open source component exposed externally, and it is also one of the most security-sensitive.

Apache typically runs under a service user profile such as QTMHHTTP or another configured profile. From a security standpoint, Apache should never run under a powerful user profile. Authorities should be tightly constrained so that compromise of a web application does not imply compromise of the system.

Key Apache security considerations include:

- Assigning minimal required IFS authorities to document roots and CGI directories
- Restricting write access so Apache cannot modify executable code
- Controlling configuration file access under `/www`
- Disabling unnecessary modules and services

Apache operates entirely within the bounds of the user profile and IFS authorities granted to it. When properly configured, it aligns well with the IBM i principle of least privilege.

The Root File System (IFS) Authorities: A Common Risk Area

Misconfigured IFS permissions represent one of the most common security weaknesses in IBM i open source deployments. Administrators unfamiliar with POSIX permissions may resort to broad permissions such as `777` to quickly resolve access issues.

Although this may temporarily restore functionality, it introduces long-term risk. Unrestricted write access can enable:

- Malicious script insertion
- Accidental overwrites
- Privilege escalation via executable files

Recommended practices include:

- Granting execute permissions only where necessary
- Limiting write access to application-specific service profiles
- Using groups rather than public permissions
- Separating source code directories from data directories

IFS authorities should be reviewed with the same discipline applied to native libraries and objects.

User Profiles, Jobs, and Open Source Processes

All open source processes on IBM i execute under a user profile—whether launched by Apache, batch jobs, CL commands, or schedulers. This directly integrates open source security with traditional IBM i concepts.

Best practices include:

- Creating dedicated service profiles for open source workloads
- Avoiding the use of privileged profiles such as QSECOFR or profiles with *ALLOBJ authority
- Ensuring batch jobs that invoke Python or PHP run under controlled contexts
- Auditing adopted authority scenarios when calling open source scripts from RPG or CL

When Python or PHP accesses Db2, database operations occur under the job user's authorities. This is a strength of the IBM i model, provided profiles are configured appropriately.

Package Management and Supply Chain Security

Modern open source introduces new concerns related to software supply chains. Libraries and packages are commonly retrieved from public repositories, accelerating development but increasing risk if uncontrolled.

IBM i teams should:

- Restrict who is authorized to install packages
- Prefer vetted repositories whenever possible
- Track third-party library versions used in production
- Understand update and patch lifecycles

Tools such as Python virtual environments and PHP Composer help isolate dependencies, but they do not eliminate the need for governance.

Auditing and Monitoring in a Hybrid Environment

IBM i auditing capabilities remain just as relevant for open source workloads. Job logs, authority failures, system auditing, and exit points continue to apply.

Recommended actions include:

- Monitoring jobs spawned by Apache and schedulers
- Reviewing authority failures in QSYSOPR and QAUDJRN
- Logging application-level activity
- Including open source workloads in routine security reviews

Security on IBM i should be holistic. Open source components should be audited alongside native applications.

Culture Change: The Hardest Challenge

The most difficult adjustments are often cultural rather than technical. IBM i environments are traditionally secure by default, while open source assumes configurability and flexibility.

Effective teams recognize that:

- Open source requires active security management
- Convenience must be balanced against control

- IBM i security principles must be deliberately extended into PASE

Organizations that succeed treat open source as a first-class IBM i workload, not an external add-on.

The Network Is Your Ally

Open source is now a permanent and valuable component of the IBM i platform. It enables modernization, integration, and innovation, but brings added responsibility. Network architecture plays a critical role, particularly for web-based workloads.

Understanding how components are exposed, segmented, and protected is essential. In addition to firewalls, tools such as reverse proxies and layered network controls should be considered.

Command	Description
ls	List contents of a directory
pwd	Display current directory
cd, mkdir	Change & make directory
head	Show the first lines of a file
tail	Show the last lines of a file
clear	Clear the terminal screen
cp	Copy files
rm, rmdir	Remove links & directories (stream files)
find	Find files with matching expression (name and contents)
mv	Move files (also works as rename...)

Figure. IBM i Network and Open Source Access Model

How Do I Install Node.js on IBM i?

Installing Node.js on IBM i is straightforward using the IBM i Open Source Package Management system. This procedure walks you through the installation process and verification steps to get Node.js running on your system.

Prerequisites

Before installing Node.js, ensure you have:

- Access to IBM i with Appropriate authority to install RPM packages or equivalent permissions
- Access to the IBM i Open Source Package Management tool from IBM i Access Client Solutions (ACS) or from a SSH shell session.

Installation Steps

Step 1: Install Node.js Using yum

Node.js can be installed from a shell session (via SSH and a good bash shell for example) or from IBM i Access Client Solutions (ACS), as with any other open source package on IBM i. IBM i uses the `yum` package manager for open source software.

To install Node.js, run:

```
yum install nodejsXX
```

Replace `XX` with the desired version number (e.g., `nodejs18`, `nodejs24`). Choose an LTS (Long Term Support) version for production use.

The installation process will:

- Download the Node.js runtime
- Install NPM (Node Package Manager) automatically
- Set up the necessary environment paths
- Install required dependencies

Node.js and Package Management

Step 2: Verify the Installation

After installation completes, verify Node.js is installed correctly:

```
node --version
```

This should display the installed Node.js version (e.g., `v18.x.x`).

Verify NPM is also installed:

```
npm --version
```

Step 3: Test Node.js

Create a simple test file to ensure Node.js runs correctly:

```
echo "console.log('Hello from Node.js on IBM i\!');" > test.js
node test.js
```

You should see the output: `Hello from Node.js on IBM i!`

Additional Configuration

Setting Up the Environment

The installation automatically configures the PATH, but you may want to add Node.js to your profile for persistent access:

```
echo 'export PATH=/QOpenSys/pkgs/bin:$PATH' >> ~/.profile
```

Installing Additional Versions

You can have multiple Node.js versions installed simultaneously. Use the version-specific package names where XX is the version number:

```
yum install nodejsXX  # For example: nodejs18, nodejs20, nodejs22, nodejs24
```

Troubleshooting

Issue: Command not found

- Ensure `/QOpenSys/pkgs/bin` is in your PATH
- Log out and log back in to refresh your environment

Issue: Permission denied

- Verify you have sufficient authority to install packages
- Contact your system administrator if needed

Next Steps

With Node.js installed, you can now:

- Install packages using NPM
- Create Node.js applications
- Connect to Db2 for i using the IBM i-specific packages
- Deploy web services and APIs

For detailed information about Node.js on IBM i, refer to the IBM i Open Source Documentation.

What is NPM and How Do I Use It?

NPM (Node Package Manager) is the standard package manager for Node.js and the largest software registry in the world, containing over a million packages. On IBM i, NPM works exactly as it does on other platforms, providing access to the entire ecosystem of JavaScript libraries and tools.

What NPM Does

NPM manages project dependencies, handles versioning, and simplifies the installation of third-party packages. When you install Node.js on IBM i using `yum install nodejsXX` (where XX is the version to install), NPM is automatically included and configured.

Basic NPM Commands

Installing Packages:

```
npm install express          # Install a package locally
npm install -g nodemon       # Install globally
npm install lodash@4.17.21   # Install specific version
```

Managing Dependencies:

```
npm init                     # Create package.json
npm install                  # Install all dependencies from package.json
npm update                   # Update packages
npm uninstall express        # Remove a package
```

Viewing Information:

```
npm list                    # List installed packages
npm outdated                # Check for outdated packages
npm search express          # Search for packages
```

Package.json

The `package.json` file defines your project and its dependencies:

```
{
  "name": "my-ibmi-app",
  "version": "1.0.0",
  "dependencies": {
    "express": "^4.18.0",
    "@ibm/mapepire-js": "^1.0.0"
  }
}
```

Considerations for IBM i

Some npm packages include native code that must be compiled. In these cases, development tools such as GCC may need to be installed in the IBM i open source environment.

NPM on IBM i provides the same powerful package management capabilities you'd find on any other platform, making it easy to leverage the vast Node.js ecosystem for your IBM i applications.

Open Source Ecosystem on IBM i

What Are Some Open Source Utilities on IBM i?

There are many utilities available through Open Source Package Management (OSPM) that can be used to accomplish specific tasks on IBM i. A selection of commonly used open source utilities is described below.

*nix Commands

Many open source solutions on IBM i operate in the PASE environment, which is a partial AIX runtime implemented by IBM to allow additional workloads to run on IBM i without requiring a separate partition. While additional partitions can be beneficial, some organizations prefer a single-system approach, making PASE an effective compromise.

Because of its UNIX heritage, many standard UNIX and Linux commands are available in PASE. Common examples include `ls`, `pwd`, `cd`, and others. These commands are documented in the PASE manuals available on IBM's website.

Command	Description
ls	List contents of a directory
pwd	Display current directory
cd, mkdir	Change & make directory
head	Show the first lines of a file
tail	Show the last lines of a file
clear	Clear the terminal screen
cp	Copy files
rm, rmdir	Remove links & directories (stream files)
find	Find files with matching expression (name and contents)
mv	Move files (also works as rename...)

Figure. Common PASE Commands

cURL

The cURL utility is commonly used to test responses from API endpoints or web pages. It is lightweight and well suited for command-line testing.

To install cURL, open OSPM, select `curl` and `curl-devel`, and choose **Install**. During installation, a console window may appear displaying required dependencies and supporting components.

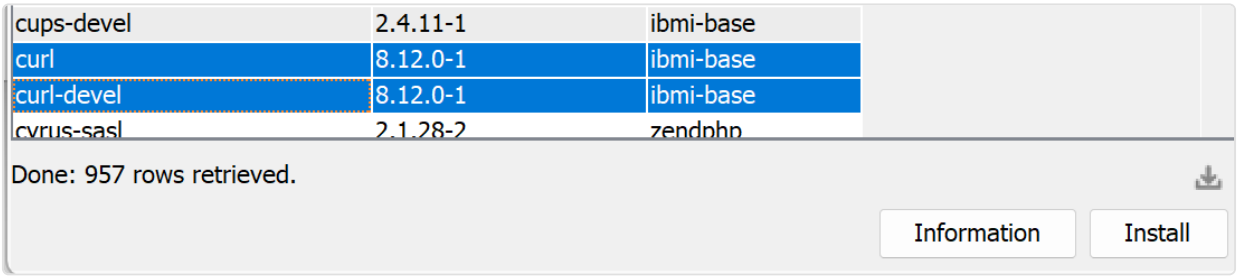


Figure. Install curl package

```
 Package             Arch          Version             Repository          Size
================================================================================
Installing:
 curl-devel          ppc64         8.12.0-1            ibmi-base          725 k
Updating:
 curl                ppc64         8.12.0-1            ibmi-base          177 k
 python2             ppc64         2.7.18-8            ibmi-base           13 M
Updating for dependencies:
 libcurl4            ppc64         8.12.0-1            ibmi-base          789 k
 libssh2-1           ppc64         1.11.1-1            ibmi-base          228 k

Transaction Summary
================================================================================
Install       1 Package
Upgrade       4 Packages

Total download size: 15 M
Is this ok [y/N]: y
Downloading Packages:
(1/5): curl-8.12.0-1.ibmi7.3.ppc64.rpm                     | 177 kB  00:00
(2/5): curl-devel-8.12.0-1.ibmi7.3.ppc64.rpm               | 725 kB  00:00
(3/5): libcurl4-8.12.0-1.ibmi7.3.ppc64.rpm                 | 789 kB  00:00
(4/5): libssh2-1-1.11.1-1.ibmi7.3.ppc64.rpm                | 228 kB  00:00
(5/5): python2-2.7.18-8.ibmi7.3.ppc64.rpm                  |  13 MB  00:08
--------------------------------------------------------------------------------
Total                                          1.5 MB/s |  15 MB     00:10
Running Transaction Check
Running Transaction Test
Transaction Test Succeeded
Running Transaction
  Updating   : libssh2-1-1.11.1-1.ppc64                                    1/9
  Updating   : libcurl4-8.12.0-1.ppc64                                     2/9
  Installing : curl-devel-8.12.0-1.ppc64                                   3/9
  Updating   : curl-8.12.0-1.ppc64                                         4/9
  Updating   : python2-2.7.18-8.ppc64                                      5/9
  Cleanup    : curl-7.76.1-2.ppc64                                         6/9
  Cleanup    : libcurl4-7.76.1-2.ppc64                                     7/9
  Cleanup    : libssh2-1-1.9.0-4.ppc64                                     8/9
  Cleanup    : python2-2.7.18-6.ppc64                                      9/9

Installed:
  curl-devel.ppc64 0:8.12.0-1

Updated:
  curl.ppc64 0:8.12.0-1                python2.ppc64 0:2.7.18-8

Dependency Updated:
  libcurl4.ppc64 0:8.12.0-1             libssh2-1.ppc64 0:1.11.1-1

Complete!
-bash-5.1$
```

Figure. curl installation dependencies

Once installed, cURL can be run from a shell such as PuTTY. The following example checks the installed version:

```
-bash-5.1$ curl --version
curl 8.12.0 (powerpc-ibm-os400) libcurl/8.12.0 OpenSSL/3.0.8 zlib/1.2.13 brotli/1.0.9 libidn2/2.3.0 libssh2/1.11.1
Release-Date: 2025-02-05
Protocols: dict file ftp ftps gopher gophers http https imap imaps ipfs ipns mqtt pop3 pop3s rtsp scp sftp smb smbs smtp smtps telnet tftp ws wss
Features: alt-svc AsynchDNS brotli HSTS HTTPS-proxy IDN IPv6 Largefile libz NTLM SSL threadsafe TLS-SRP UnixSockets
-bash-5.1$
```

Figure. curl version output

Retrieving HTML content from a site that allows crawling can be achieved by issuing the `curl` command followed by the URL:

```
-bash-5.1$ curl https://example.com
<!doctype html><html lang="en"><head><title>Example Domain</title><meta name="vi
ewport" content="width=device-width, initial-scale=1"><style>body{background:#ee
e;width:60vw;margin:15vh auto;font-family:system-ui,sans-serif}h1{font-size:1.5e
m}div{opacity:0.8}a:link,a:visited{color:#348}</style></head><body><div><h1>Exam
ple Domain</h1><p>This domain is for use in documentation examples without needi
ng permission. Avoid use in operations.</p><p><a href="https://iana.org/domains/
example">Learn more</a></p></div></body></html>
```

Figure. curl HTML request

To test API connectivity, cURL can be used against a sample API endpoint such as `jsonplaceholder`, which offers test data and endpoints.

```
-bash-5.1$ curl https://jsonplaceholder.typicode.com/posts/2
{
  "userId": 1,
  "id": 2,
  "title": "qui est esse",
  "body": "est rerum tempore vitae\nsequi sint nihil reprehenderit dolor be
atae ea dolores neque\nfugiat blanditiis voluptate porro vel nihil molestia
e ut reiciendis\nqui aperiam non debitis possimus qui neque nisi nulla"
}-bash-5.1$
```

Figure. curl API request

BASH

In UNIX environments, a shell acts as a command-line interpreter. The default shell in PASE is KSH (Korn Shell). While functional, many developers prefer **BASH (Bourne Again SHell)** for its additional conveniences.

BASH is installed via OSPM like other utilities. One of its most notable features is command history recall using the up-arrow key. Similar to the `<F9>` retrieve function in a 5250 interface, BASH allows users to recall commands from previous sessions, including commands entered days or even months earlier.

system

The `system` utility allows a process running in PASE to invoke a corresponding process in the ILE environment. Output produced by the invoked command is returned to standard output in the PASE session.

The example below shows the `system` utility being used to display a system value.

```
-bash-5.1$ system "DSPSYSVAL QMODEL"
                                                    System Values
                                        Page     1
5770SS1 V7R5M0  220415
                  IDEVPHP   04/13/26  22:32:54 EDT
                Current                          Shipped
 Name           value                            value
Description
 QMODEL          42A                             ' '
System model number
     Note:  > means current value is different from the shipped value
                                       * * * * *   E N D  O F  L I S T I N G
 * * * * *
-bash-5.1$
```

Figure. system command example

This capability is useful when a process in PASE needs to trigger processing on the ILE side. Most CL commands can be run using `system`, although parameter passing is not supported.

Shell Scripting

Just as CL programs group ILE commands into an executable workflow, shell scripts allow multiple shell commands to be combined into a single text file for execution in PASE.

Shell scripts are typically non-interactive and execute commands sequentially, similar to CL programs. However, scripts may also include interactive logic, as shown in the example below, followed by its output.

```
1#!/QOpenSys/pkgs/bin/bash
2
3NAME='Dave'
4
5MYNUM=$(((RANDOM%10+1)))  # Use Built in Function
6
7while true
8do
9    read -p "Hello $NAME, Please enter a value from 1-10: "  NUM1
10
11    if [ $NUM1 -lt $MYNUM ]
12    then
13        echo "My number is bigger"
14    elif [ $NUM1 -gt $MYNUM ]
15    then
16        echo "My number is smaller"
17        continue
18    else
19        echo "They are the same, Good guess"
20        break
21    fi
22
23done
```

Figure. Shell script example

```
-bash-4.4$ ex2IFs.sh
Hello Dave, Please enter a value from 1-10: 6
My number is smaller
Hello Dave, Please enter a value from 1-10: 3
My number is bigger
Hello Dave, Please enter a value from 1-10: 4
They are the same, Good guess
-bash-4.4$
```

Figure. Shell script output

Brief Discussion of Ansible

Introduction to Ansible

Ansible is an open-source automation tool that simplifies IT tasks such as configuration management, application deployment, system updates and task automation. It uses a declarative language (YAML) to describe the desired state of systems, allowing users to automate complex workflows without writing extensive code. Ansible operates over SSH, making it agentless and easy to deploy.

Usefulness with IBM i

- **Automation of Routine Tasks**: Ansible can automate repetitive tasks like software installation, configuration changes, and system updates on IBM i systems.
- **Infrastructure as Code**: Treat IBM i configurations as code, enabling version control, repeatability, and collaboration.
- **Multi-Platform Management**: Manage IBM i alongside other systems (Linux, Windows) in a unified way.
- **Scalability**: Easily scale automation across multiple IBM i partitions or environments.
- **Error Reduction**: Reduce human errors in manual configurations by using playbooks.

Ansible's agentless nature is particularly advantageous for IBM i, as it doesn't require installing additional software on the target systems, minimizing security risks and maintenance overhead.

Prerequisites for Using Ansible with IBM i

In order to use Ansible to manage IBM i systems, the following prerequisites need to be met:

Ansible Control Node

The control node acts as an Ansible Server and manages the automation process.

1. **Ansible Installation**: Ansible should be installed on a control node (Linux, macOS, or Windows with WSL).
2. **Python Environment**: Ansible requires Python 3, hence it needs to be installed on the control node.
3. **Network Connectivity**: The control node should be able to reach IBM i systems over the network, with firewalls configured to allow SSH traffic.
4. **Ansible Collections**: Relevant Ansible collections need to be installed for IBM i integration.

IBM i Managed Node

The IBM i System acts as an endpoint and needs minimal setup in order to start working through Ansible.

1. **Licensed Programs**: There are some specific licensed programs which must be installed.
2. **Open Source Packages**: Open Source software packages like python3, python3-itoolkit and python3-ibm_db needs to be installed.
3. **SSH Access**: SSH should be enabled on IBM i systems.
4. **IBM i User Profile**: A dedicated user profile should be present on IBM i with appropriate authorities (e.g., *ALLOBJ special authority for administrative tasks).

Ansible Collections related to IBM i

Several Ansible collections provide modules and playbooks specifically for IBM i and related Power Systems:

- **ibm.power_ibmi**: This collection offers modules for managing IBM i systems directly, including running CL commands and sql queries, managing PTFs, handling IFS operations, etc.
- **ibm.powervc**: Focused on IBM PowerVC, this collection enables automation of virtualization management on Power Systems and network configuration for virtual environments.

- **ibm.power_hmc**: This collection provides modules for interacting with IBM Power Hardware Management Console (HMC), allowing management of configurations and deployments of Power HMC systems and automating tasks like HMC reboot.

Why Ansible

Ansible's integration with IBM i represents a powerful combination for modernizing and automating, bridging the gap between traditional IBM i management and contemporary DevOps practices. Organizations benefit from reduced manual effort, faster application deployment cycles, and improved consistency across multiple IBM i partitions. Whether managing a single IBM i partition or a complex multi-system environment, Ansible enables enterprises to modernize their systems without abandoning their proven investments in IBM i technology.

What Is Open Source Software

Open Source Software (OSS) is software that is developed by the community, for the community. Depending on the license model, it is usually free to use, enhance, and distribute—primarily with attribution. The maintenance and support, while sometimes available commercially for an extra charge, may lie with the user. Many open source projects benefit from having a project team elected from their respective communities to help steer the project. While no approach is perfect, this creates a much more democratic model for software development and gives all community members a voice in the maintenance and direction of the project.

There are many examples of OSS projects, such as Linux, PHP, Ansible, and others, that have significantly altered the software development landscape.

In each industrial revolution, some inexpensive resource is exploited—such as labor or oil. In the case of the Internet and AI revolutions, the resource clearly leveraged by many disruptors is open source software, as they all share this in common. In the relatively short 20-year window of widespread experience with OSS, more than 50% of IBM i shops are now leveraging it. There are many resources on the web and elsewhere that discuss OSS in depth, and this chapter will explore the incredible value and nuance that open source brings to IBM i.

Why Use Open Source Software

OSS Costs

As indicated, open source software (OSS) can typically be installed and used for no initial cost. A zero *Total Cost of Acquisition (TCA)* can be extremely attractive—and understandably so. However, the realities of OSS are more nuanced, as TCA is not the same as *Total Cost of Ownership (TCO)*. There are real costs associated with evaluating, maintaining, and updating open source software. If these are not considered up front, they can result in significant technical debt over time.

Many organizations such as Zend, Red Hat, and others offer commercial support for open source technologies, and this option should be factored into the overall cost model. Paying for support can significantly reduce risk while preserving the flexibility and innovation benefits of OSS.

One of the clearest ongoing costs is maintaining currency with supported versions and releases. Most open source communities support a given release for a relatively short period—often three to five years. This contrasts sharply with languages like RPG on IBM i, where IBM has maintained backwards compatibility for several decades. As we explore popular OSS languages and projects throughout this series, we will examine their support lifecycles and outline practical strategies for staying current.

While licensing is an important aspect of open source, it has been well documented in many public resources and does not materially change when OSS is deployed on IBM i. For that reason, licensing models will not be a major focus of this discussion.

OSS Opportunities

Open source software is created by communities that can range in size from one or two contributors to thousands of participants worldwide. These communities are often diverse in both background and expertise, and they continuously evolve while the software itself matures. This collaborative development model allows OSS projects to reflect the collective experience and problem-solving ability of many contributors.

For small and medium-sized organizations, this provides enormous value. Common routines, frameworks, and entire subsystems no longer need to be written from scratch. Instead, organizations can build on proven solutions that have already been tested, improved, and refined by others. In many ways, OSS users are standing on the shoulders of those who came before them, leveraging a depth of knowledge that would be difficult or impossible to replicate internally.

While not every open source project is a perfect fit, there are many high-quality projects that continue to thrive and deliver value. When combined with IBM i, open source software enables a powerful blending of proven business logic written in RPG and COBOL with modern capabilities such as web interfaces, APIs, and integration services. OSS allows IBM i developers to focus on solving business problems instead of reinventing foundational code.

How Does Open Source Software and IBM i Work?

Installation & Operation

Since open source software can be acquired in virtually any manner, it can also be installed on IBM i in multiple ways. Several years ago, the IBM team in Rochester addressed the need for standardization by introducing Licensed Program **5733-OPS**. While this approach was initially successful, it was limited in scope and was discontinued in 2019.

It was replaced by the **RPM-based Open Source Package Management (OSPM)** approach, delivered through **IBM i Access Client Solutions (ACS)**. This method aligns IBM i with modern open-source distribution practices while preserving system integrity.

The official IBM documentation should always be consulted for the most up-to-date installation instructions. At a high level, OSPM installation is simplest when the IBM i partition has outbound internet access with proper IP connectivity and DNS resolution. When that is not possible, tunnel-based installation options are available. In restricted environments, some OSS may be deployed using cloned repositories within the local network.

Access via ACS

Using a current version of ACS ensures that the OSPM facility can be installed and accessed correctly. Proper authority is required, and ACS will report any security failures encountered during startup.

OSPM is available from the **front panel of the main ACS window**. Selecting it prompts for IBM i credentials, after which ACS will either grant access or begin the bootstrap (installation) process.

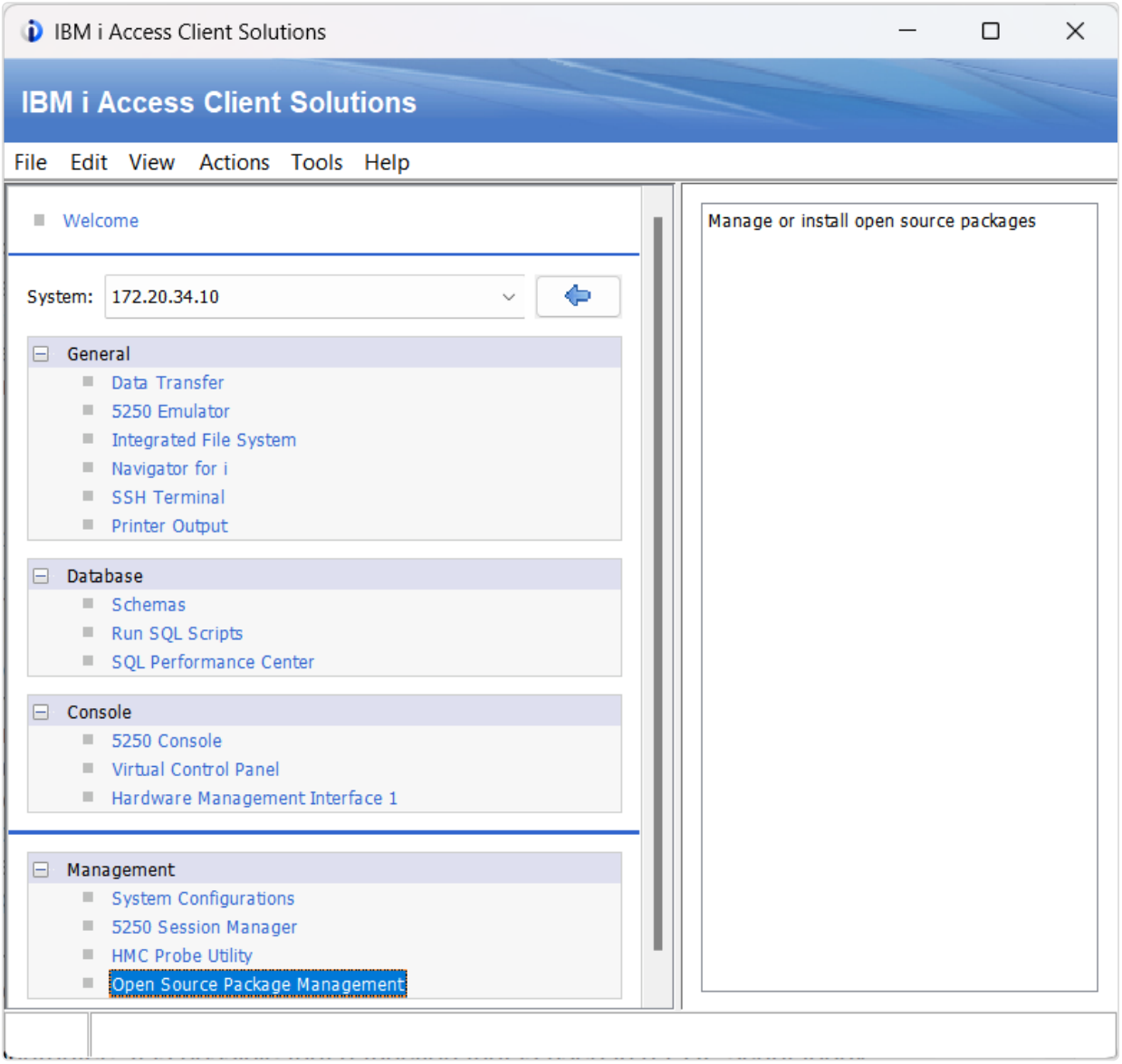


Figure. ACS OSPM Entry

Once opened, OSPM presents three primary tabs:

- **Installed Packages**
- **Updates Available**
- **Available Packages**

The Updates and Available tabs allow users to install and maintain open-source packages. During installation, console windows may appear showing package activity and required dependencies. Dependency management is handled automatically by the package manager.

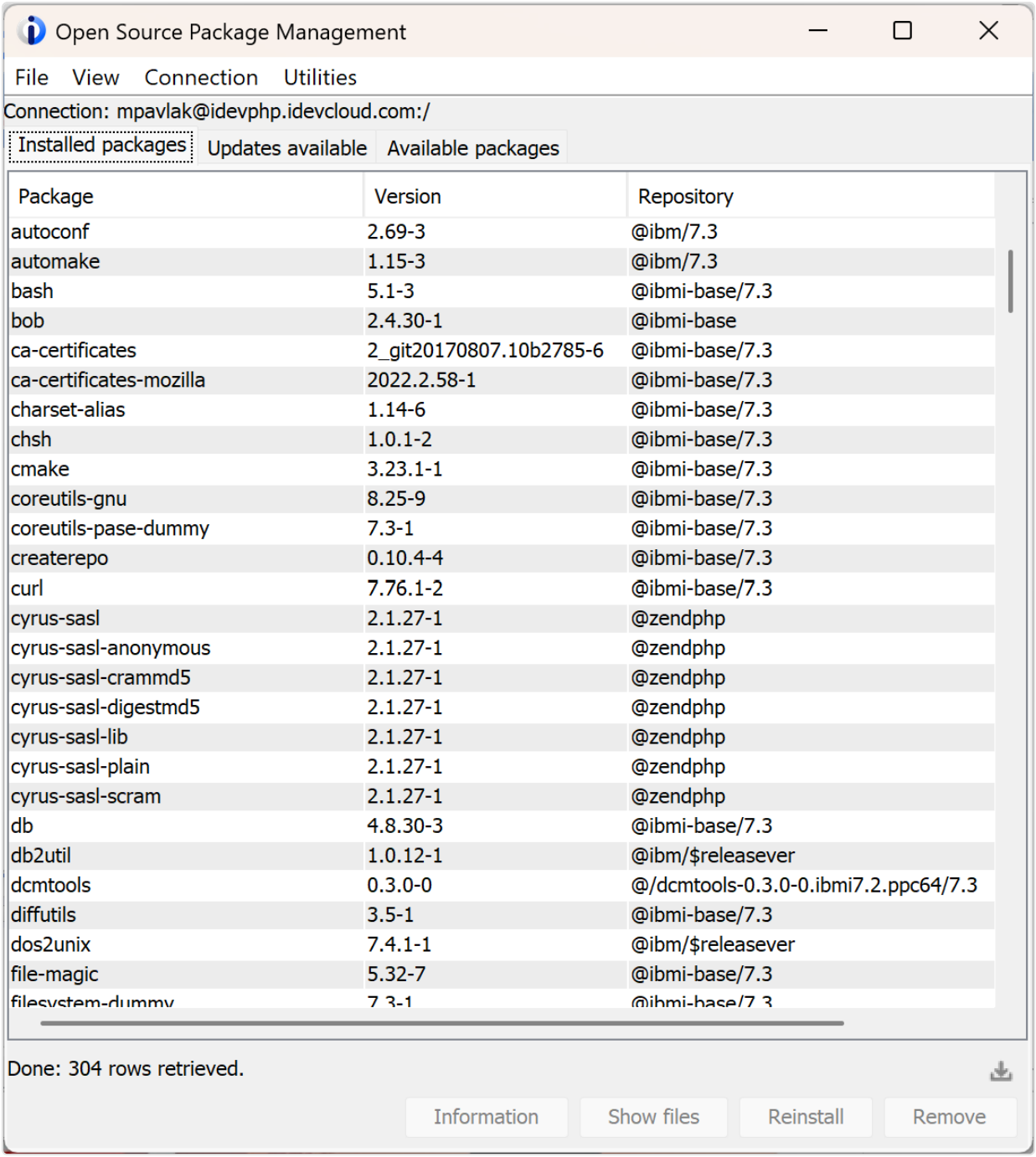


Figure. ACS Package Panels

Access via Shell Client

The same installation and management functions are accessible through the **Portable Application Solutions Environment (PASE)** using a shell client such as SSH.

While some commands can run in a 5250 session, a dedicated shell client such as **PuTTY** provides a more natural experience for OSS management.

By default, the shell may not reference the OSPM binary path. This can be corrected with the following command:

```
PATH=/QOpenSys/pkgs/bin:$PATH
```

OSPM is a port of the **Red Hat Package Manager** ecosystem, and most `yum` commands are supported. To list installed packages:

```
yum list installed
```

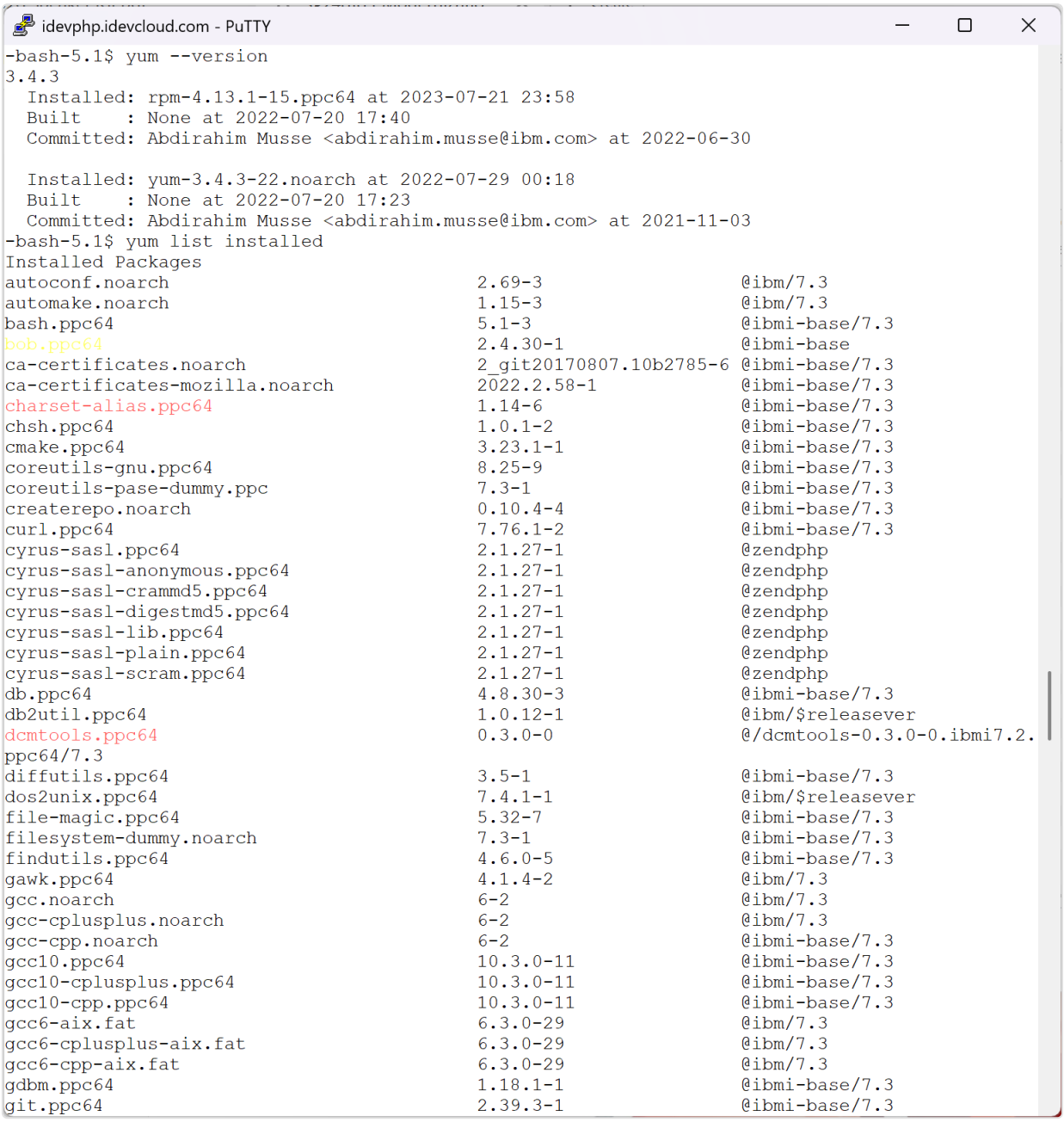


Figure. YUM Command Output

Both **ACS-based OSPM** and **command-line YUM** interfaces can be used interchangeably to manage OSS on IBM i. Where possible, graphical interfaces are preferred, with shell access reserved for advanced scenarios.

OSS on IBM i or Linux: Which One?

While open source software (OSS) predates Linux, much of the modern OSS community views Linux as its natural home. On IBM i, open source software runs primarily in the **Portable Application Solutions Environment (PASE)**, which is not Linux but rather a partial port of AIX. IBM i also runs on **IBM Power Architecture**, not x86, where many OSS binaries are commonly built. As a result, some software available on Linux may not immediately be available or portable to PASE.

This leads to a reasonable and frequently asked question: *Should open source software run on IBM i or on Linux?* As with many architectural decisions, the answer is **"it depends."** The sections below outline the primary advantages and challenges of leveraging OSS directly on IBM i.

Advantages

IBM and the Community Are There

A significant portion of the open source software commonly needed for daily operations and key capabilities is provided by IBM as part of the IBM i operating system at no additional charge. In addition, several vendors offer no-cost community editions of OSS that are specifically supported on IBM Power. This ecosystem creates active collaboration among IBM, vendors, and the IBM i community.

Organizations may also choose to purchase optional commercial support from vendors. Beyond providing technical assistance, this model helps fund ongoing development and long-term viability of open source software on IBM i.

Convenience

Many IBM i environments lack the staff, skill sets, or operational appetite required to manage a separate Linux server. Running OSS directly on IBM i allows teams to work within a **security model they already understand**, using familiar operational practices.

IBM i also provides an internal virtual network where communication between ILE and PASE workloads never leaves the system or traverses physical switches. From an operational standpoint, managing a single system for **backup, high availability, and disaster recovery** can significantly simplify planning and reduce overall complexity.

Challenges

Package Availability

Simply put, there are more developers and contributors supporting Linux than IBM i. As a result, Linux distributions offer a broader selection of open source packages than are currently available for PASE. In some cases, this may lead organizations to deploy a Linux server or partition near IBM i in order to use specific tools or frameworks.

This also means IBM i customers depend on IBM and the supporting open source community for ports and updates, which can sometimes lag behind Linux. However, it is always worth checking within the IBM i community first—many widely used OSS utilities and languages have already been successfully ported.

General Support

Linux skills are commonly taught in colleges and technical programs because Linux is inexpensive and easy to deploy in educational settings. As a result, many graduates entering the workforce already have basic Linux experience.

While the IBM i open source community is exceptionally strong, many external OSS contributors and support resources may be unfamiliar with IBM i or the PASE environment. This can occasionally complicate troubleshooting or third-party support.

Final Thoughts

Each organization must assess its own requirements, skills, and risk tolerance when deciding where open source workloads should run. However, there is little debate that **IBM i is a fantastic platform for getting started with open source software**, particularly when integration with existing business logic, data, and operations is a priority.

Please Read On

This section has introduced open source software as it relates to IBM i. The remainder of this chapter explores several OSS languages and utilities available on the platform and how they can be used effectively.

How Do I Install and Use Python on IBM i?

Installation

Python is likely already present on your system. If Open Source Package Management (OSPM) has been used previously, Python 2 may already be installed. However, the focus should be on a more current version. Select **Python 3** from OSPM and choose install.

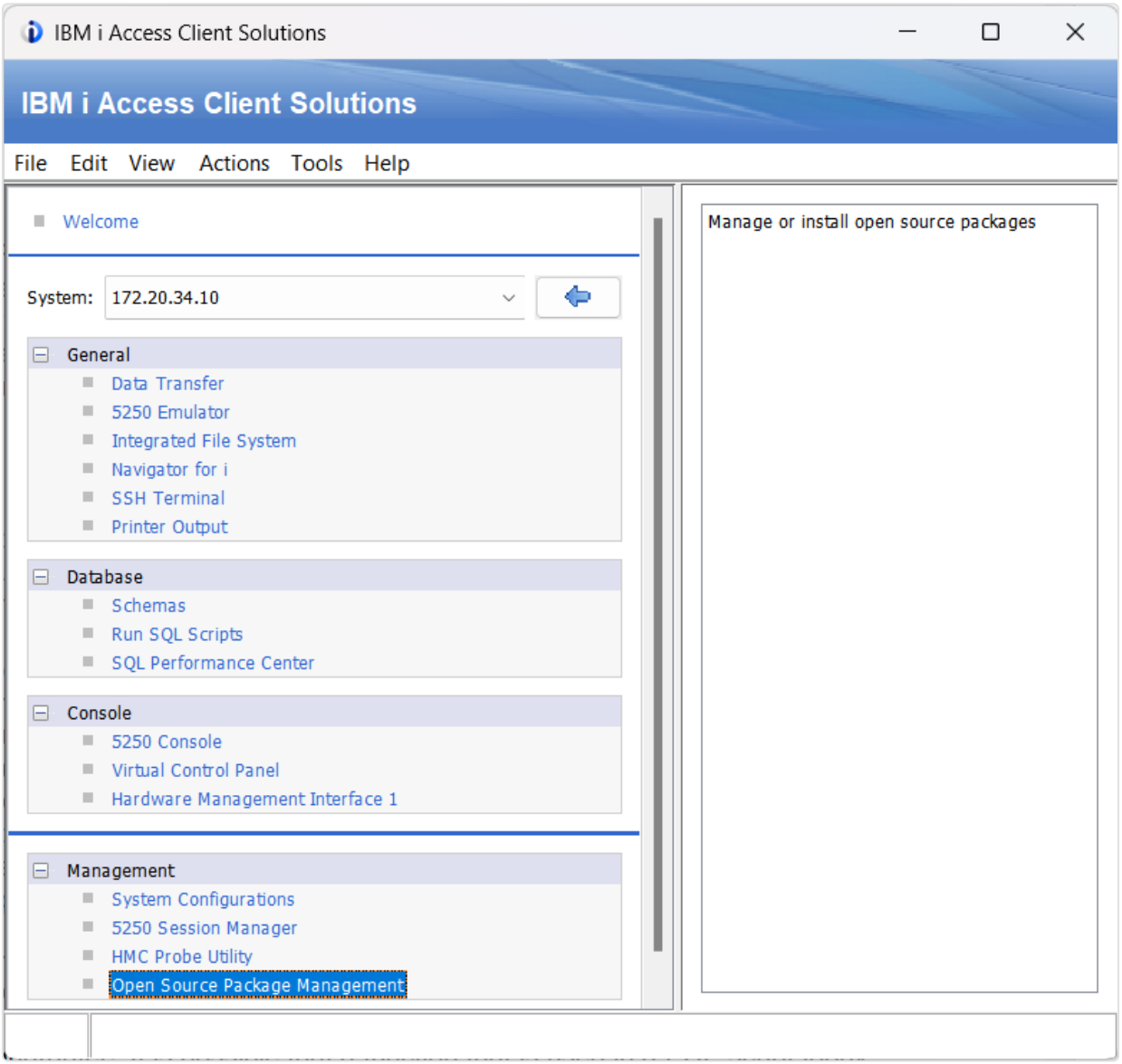


Figure. Python Installation via OSPM

Once installed, using Python is straightforward, assuming the `PATH` environment variable is set to search `/QOpenSys/pkgs/bin` before other directories. With the path configured correctly, the Python interpreter should be found without issue.

To verify the installation, check the version using the `python --version` command.

```
-bash-5.1$ python
Python 3.13.2 (main, Mar 28 2025, 17:15:29) [GCC 6.3.0] on os400
Type "help", "copyright", "credits" or "license" for more information.
>>> 2+2
4
>>> name = "IBM i"
>>> print("This system is called " + name)
This system is called IBM i
>>> import math
>>> print(math.pi)
3.141592653589793
>>>
```

Figure. Python Version Check

Using Python

There are two primary ways to use Python, each suited to different purposes.

The first is the **Python shell**, which can be accessed from any PASE command-line environment such as PuTTY, QP2Term, or QSH. The Python shell is designed for experimentation rather than production use. Common activities include basic mathematics, running Python statements one line at a time, and testing or inspecting Python modules.

The shell is launched by typing `python` at the command prompt. The `>>>` prompt indicates that Python is ready for input. The session can be ended by typing `quit()` or, in some shell clients, by pressing `CTRL-D`.

```
-bash-5.1$ python
Python 3.13.2 (main, Mar 28 2025, 17:15:29) [GCC 6.3.0] on os400
Type "help", "copyright", "credits" or "license" for more information.
>>> 2+2
4
>>> name = "IBM i"
>>> print("This system is called " + name)
This system is called IBM i
>>> import math
>>> print(math.pi)
3.141592653589793
>>>
```

Figure. Python Interactive Shell

The second approach is invoking Python through a **script file**. Python code is saved in a text file with a `.py` extension, such as `hello.py`, and stored in the Integrated File System (IFS).

Within the script, Python code is written and, when executed, is interpreted into bytecode and run by the Python interpreter. In the example shown, a simple script contains three comments and two lines of code. Comments begin with a `#`, and blank lines are ignored by the interpreter. The variable `place` is created dynamically at runtime, and the `print` statement concatenates strings before sending output to standard output, commonly referred to as the console.

```
#!/QOpenSys/pkgs/bin/bash

NAME='Dave'

MYNUM=$(((RANDOM%10+1)))  # Use Built in Function

while true
do
    read -p "Hello $NAME, Please enter a value from 1-10: "  NUM1

    if [ $NUM1 -lt $MYNUM ]
    then
        echo "My number is bigger"
    elif [ $NUM1 -gt $MYNUM ]
    then
        echo "My number is smaller"
        continue
    else
        echo "They are the same, Good guess"
        break
    fi

done
```

Figure. Simple Python Script Example

```
-bash-4.4$ ex2IFs.sh
Hello Dave, Please enter a value from 1-10: 6
My number is smaller
Hello Dave, Please enter a value from 1-10: 3
My number is bigger
Hello Dave, Please enter a value from 1-10: 4
They are the same, Good guess
-bash-4.4$
```

Figure. Python Script Output

What and Why about Python Modules & Virtual Environments

Modules such as the math module and pyodbc are examples of how Python code is extended. The math module is part of the Python Standard Library (PSL). There are over 200 modules that make up the PSL and they are documented at python.org. The pyodbc module is compiled from an open source project and adapted for IBM i by IBM in Rochester. As previously shown, this module is installed via Open Source Package Management (OSPM).

Other third-party modules such as yfinance, NumPy, or xlsxwriter are built by the community and can be installed on IBM i using the `pip` command from the central repository at pypi.org, which contains hundreds of thousands of projects available at no charge.

Modules installed via OSPM should run well on IBM i. However, some third-party modules may include components compiled for x86 architectures. Due to IBM i running on Power architecture, these modules may not run correctly. The recommended approach is to test such modules within a virtual environment.

Virtual Environments

A virtual environment is a container used to isolate Python modules for individual projects. The purpose of a virtual environment is to provide a safe space where multiple developers or projects can coexist on a single partition without dependency conflicts. By installing modules and dependencies inside a virtual environment, projects avoid collisions caused by differing dependency versions.

Using the `pip` command in the global environment may result in warnings indicating that the global environment is externally managed. Virtual environments are the preferred way to avoid this issue.

```
mpavlak@dc3-i74sales ~ $ python -m pip install xlsxwriter
error: externally-managed-environment

× This environment is externally managed
╰─> To install Python packages system-wide, try yum install
    python313-xyz, where xyz is the package you are trying to
    install.

    If you wish to install a Python package not available via the package manager,
    create a virtual environment using python3.13 -m venv path/to/venv.
    Then use path/to/venv/bin/python and path/to/venv/bin/pip.

    If you wish to install a Python application not available via the package
    manager, it may be easiest to use pipx install xyz, which will manage a virtual
    environment for you. Make sure you have pipx installed.

note: If you believe this is a mistake, please contact your Python installation or OS distributi
on provider. You can override this, at the risk of breaking your Python installation or OS, by p
assing --break-system-packages.
hint: See PEP 668 for the detailed specification.
mpavlak@dc3-i74sales ~ $
```

Figure. Global pip warning

The high-level steps to use a Python virtual environment are:

- Create
- Activate
- Load
- Use
- Deactivate
- Delete

Create

To create a virtual environment, first create a directory for the project and change into that directory. Then issue the Python command:

```
python -m venv env
```

This command invokes the `venv` module and creates a virtual environment named `env`. Naming conventions may vary, but consistent naming simplifies management. The command creates a folder named `env`, which contains the project's modules and supporting files.

```
>>> quit()
(env) mpavlak@dc3-i74sales ~/python/projectA $ deactivate
mpavlak@dc3-i74sales ~/python/projectA $
```

Figure. Create virtual environment

Activate

Activating the virtual environment differs by platform. Since PASE behaves similarly to Linux, the Linux-style command is used:

```
source env/bin/activate
```

Once activated, the shell prompt changes to include the environment name, indicating that Python commands reference the virtual environment. To allow the virtual environment to search the global environment after the local one, include the `--system-site-packages` option when creating it:

```
python -m venv --system-site-packages env
```

```
>>> quit()
(env) mpavlak@dc3-i74sales ~/python/projectA $ deactivate
mpavlak@dc3-i74sales ~/python/projectA $
```

Figure. Activate virtual environment

Load

Third-party modules are installed using the `pip` command. Pip searches for modules, downloads them, installs dependencies, and reports success or failure. In the example below, the Wikipedia module is installed:

```
pip install wikipedia
```

Dependency modules such as beautifulsoup and urllib3 are installed automatically. The most common installation issues arise when a module includes components compiled for x86.

```
(env) mpavlak@dc3-i74sales ~/python/projectA $ pip  install wikipedia
Collecting wikipedia
  Downloading wikipedia-1.4.0.tar.gz (27 kB)
  Installing build dependencies ... done
  Getting requirements to build wheel ... done
  Preparing metadata (pyproject.toml) ... done
Collecting beautifulsoup4 (from wikipedia)
  Downloading beautifulsoup4-4.13.4-py3-none-any.whl.metadata (3.8 kB)
Collecting requests<3.0.0,>=2.0.0 (from wikipedia)
  Downloading requests-2.32.3-py3-none-any.whl.metadata (4.6 kB)
Collecting charset-normalizer<4,>=2 (from requests<3.0.0,>=2.0.0->wikipedia)
  Downloading charset_normalizer-3.4.2-py3-none-any.whl.metadata (35 kB)
Collecting idna<4,>=2.5 (from requests<3.0.0,>=2.0.0->wikipedia)
  Downloading idna-3.10-py3-none-any.whl.metadata (10 kB)
Collecting urllib3<3,>=1.21.1 (from requests<3.0.0,>=2.0.0->wikipedia)
  Downloading urllib3-2.4.0-py3-none-any.whl.metadata (6.5 kB)
Collecting certifi>=2017.4.17 (from requests<3.0.0,>=2.0.0->wikipedia)
  Downloading certifi-2025.4.26-py3-none-any.whl.metadata (2.5 kB)
Collecting soupsieve>1.2 (from beautifulsoup4->wikipedia)
  Downloading soupsieve-2.7-py3-none-any.whl.metadata (4.6 kB)
Collecting typing-extensions>=4.0.0 (from beautifulsoup4->wikipedia)
  Downloading typing_extensions-4.13.2-py3-none-any.whl.metadata (3.0 kB)
Downloading requests-2.32.3-py3-none-any.whl (64 kB)
Downloading beautifulsoup4-4.13.4-py3-none-any.whl (187 kB)
Downloading certifi-2025.4.26-py3-none-any.whl (159 kB)
Downloading charset_normalizer-3.4.2-py3-none-any.whl (52 kB)
Downloading idna-3.10-py3-none-any.whl (70 kB)
Downloading soupsieve-2.7-py3-none-any.whl (36 kB)
Downloading typing_extensions-4.13.2-py3-none-any.whl (45 kB)
Downloading urllib3-2.4.0-py3-none-any.whl (128 kB)
Building wheels for collected packages: wikipedia
  Building wheel for wikipedia (pyproject.toml) ... done
  Created wheel for wikipedia: filename=wikipedia-1.4.0-py3-none-any.whl size=11757 sha25
6=f876ef8272b5f95e5b016afc344757cf2c4adaf18c3015e1e08b69c355ddc9a1
  Stored in directory: /home/MPAVLAK/.cache/pip/wheels/79/1d/c8/b64e19423cc5a2a339450ea5d
145e7c8eb3d4aa2b150cde33b
Successfully built wikipedia
Installing collected packages: urllib3, typing-extensions, soupsieve, idna, charset-norma
lizer, certifi, requests, beautifulsoup4, wikipedia
Successfully installed beautifulsoup4-4.13.4 certifi-2025.4.26 charset-normalizer-3.4.2 i
dna-3.10 requests-2.32.3 soupsieve-2.7 typing-extensions-4.13.2 urllib3-2.4.0 wikipedia-1
.4.0

[notice] A new release of pip is available: 24.3.1 -> 25.1.1
[notice] To update, run: pip install --upgrade pip
(env) mpavlak@dc3-i74sales ~/python/projectA $
```

Figure. Install module with pip

Use

After installation, the module is available for use within the virtual environment. Documentation and examples are typically available on pypi.org or other online resources. In the example shown, the Python shell is used to test a module interactively.

```
(env) mpavlak@dc3-i74sales ~/python/projectA $ python
Python 3.13.2 (main, Mar 28 2025, 17:15:29) [GCC 6.3.0] on os400
Type "help", "copyright", "credits" or "license" for more information.
>>> import wikipedia
>>> result = wikipedia.page("AS/400")
>>> print(result.summary)
The IBM AS/400 (Application System/400) is a family of midrange computers from IBM announ
ced in June 1988 and released in August 1988. It was the successor to the System/36 and S
ystem/38 platforms, and ran the OS/400 operating system. Lower-cost but more powerful tha
n its predecessors, an estimated 111,000 installations existed by the end of 1990 and ann
ual revenue reaching $14 billion that year, increasing to 250,000 systems by 1994, and ab
out 500,000 shipped by 1997.
A key concept in the AS/400 platform is Technology Independent Machine Interface (TIMI),
a platform-independent instruction set architecture (ISA) that is translated to native ma
chine language instructions. The platform has used this capability to change the underlyi
ng processor architecture without breaking application compatibility. Early systems were
based on a 48-bit CISC instruction set architecture known as the Internal Microprogrammed
 Interface (IMPI), originally developed for the System/38. In 1991, the company introduce
d a new version of the system running on a series of 64-bit PowerPC-derived CPUs, the IBM
 RS64 family. Due to the use of TIMI, applications for the original CISC-based programs c
ontinued to run on the new systems without modification, as the TIMI code can be re-trans
lated to the new systems' PowerPC Power ISA native machine code. The RS64 was replaced wi
th POWER4 processors in 2001, which was followed by POWER5 and POWER6 in later upgrades.
The AS/400 went through multiple re-branding exercises, finally becoming the System i in
2006. In 2008, IBM consolidated the separate System i and System p product lines (which h
ad mostly identical hardware by that point) into a single product line named IBM Power Sy
stems. The name "AS/400" is sometimes used informally to refer to the IBM i operating sys
tem running on modern Power Systems hardware.
```

Figure. Use module in Python shell

Deactivate

The virtual environment remains active for the duration of the shell session. To exit the environment, issue the `deactivate` command.

The shell prompt returns to normal, indicating that the environment has closed. The environment remains available and can be reactivated later.

```
>>> quit()
(env) mpavlak@dc3-i74sales ~/python/projectA $ deactivate
mpavlak@dc3-i74sales ~/python/projectA $
```

Figure. Deactivate virtual environment

Delete

A virtual environment is simply a directory in the IFS. To remove it, delete the directory (for example, `env`) and all associated files will be removed.

Tips

When working with virtual environments:

- Do not store application code inside the virtual environment directory.
- Place application code in a directory adjacent to the virtual environment.
- Use the `--system-site-packages` option when access to globally installed modules such as pyodbc is required.

What Is the Python Community Support Model?

Guido van Rossum invented Python in the late 1990s and served as its chief steward on and off for many years. Python is licensed and governed by the Python Software Foundation and will remain under its stewardship for the foreseeable future. In more recent years, Guido was employed by Microsoft to lead a team of developers focused on

advancing Python features and improving performance. Given Python's importance in data analytics and artificial intelligence, this investment aligns with broader industry needs.

The Python Software Foundation includes contributors from organizations around the world, including Google, Meta, and Anthropic, who contribute time, resources, and expertise to the ongoing improvement of Python and its ecosystem.

Like all open source projects, Python has a finite lifecycle with respect to community support. Although Python code will continue to run beyond its official support window, community-provided security patches and bug fixes typically end approximately five years after a version is released.

Python follows a regular release cadence, with new versions generally released annually, usually in October. Because of this predictable cycle, it is strongly recommended that organizations adopting Python invest in comprehensive unit testing early. A robust test suite enables easier validation, migration, and upgrade to newer Python versions on an annual or biennial basis.

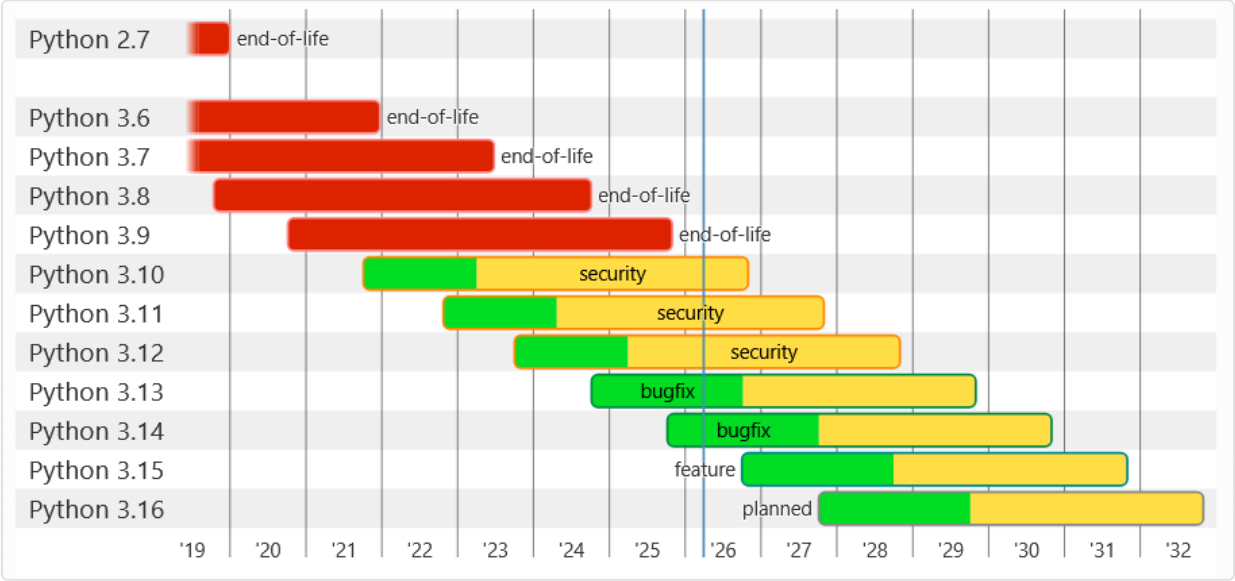


Figure. Python Release and Support Lifecycle

Web serving on IBM i

Web Serving Fundamentals

Web Serving: What and Why

Web-based access forms the foundation of modern application architecture where users expect to interact with business applications through browsers and mobile devices, not (only) terminal emulators. IBM i offers multiple approaches to web serving, from native web servers to containerized architectures, each with distinct characteristics and use cases. In today's hybrid cloud world, web serving must address challenges including security, scalability, integration with cloud-native services, and seamless connectivity across on-premises and cloud environments. This chapter examines these options and provides guidance on selecting the right approach for your modernization goals.

What Is a Web Server vs. an Application Server?

These two terms are often used interchangeably, leading to confusion. Understanding the distinct role of each is important when designing or modernizing web-based applications on IBM i.

The Web Server

A web server is the entry point for any browser-based request. When a user navigates to a URL, that request travels across the internet and arrives at a web server. The web server is primarily responsible for receiving that request and deciding what to do with it.

The web server handles many different facets of that request:

- **Security**: The web server acts as a gatekeeper. It ensures that incoming requests can only access designated or approved content, protecting the rest of the system from unauthorized access.
- **Encryption**: The web server can be configured to encrypt traffic between the browser and the server using TLS (formerly SSL). The system scrambles data in transit, and the browser decrypts it, preventing interception.
- **Load balancing**: When request volume exceeds the capacity of a single server, the web server can distribute incoming requests across multiple back-end servers, ensuring consistent response times.
- **Static content serving**: The web server efficiently returns HTML pages, images, stylesheets, JavaScript files, and other static assets. Caching strategies are commonly used to keep frequently accessed content close at hand for fast delivery.
- **Front door**: The web server often sits at the perimeter—outside or at the edge of the firewall—validating and routing requests before passing them through to back-end systems. It verifies that requests are for known resources and that the requesting user is authorized. Once validated, the request is forwarded to the appropriate back-end service, whether that is a traditional program via CGI or a modern application server.

Apache HTTP Server on IBM i

On IBM i, the primary web server is the **IBM HTTP Server Powered by Apache** (product 5770-DG1). It is built on the open source Apache HTTP Server project—a collaborative, community-driven effort to produce a robust, commercial-grade, freely available HTTP server implementation.

IBM takes the Apache source code, ports it to IBM i, recompiles it into native program objects, and integrates IBM i-specific capabilities where appropriate. The result is a web server that benefits from the breadth of the Apache ecosystem while taking full advantage of IBM i's reliability and security characteristics.

Apache on IBM i is the foundation for most web-serving scenarios on the platform, whether serving static content, acting as a reverse proxy, or fronting an application server.

The following figure shows common deployment of the different components when using web-based applications. The HTTP server is often the front end to the application, and is often situated outside the firewall to help ensure a safe environment.

```
flowchart LR
    subgraph clients["Client Tier"]
        B1["Browser A"]
        B2["Browser B"]
    end

    subgraph ibmi_front["IBM i — Web Tier"]
        APACHE1["Apache IBM HTTP Server
        5770-DG1
        TLS · Reverse Proxy
        Load Balancing
        IBM http plugin "]
    end

    subgraph ext_front["Any Platform — Web Tier"]
        APACHE2["Apache HTTP Server
        TLS · Reverse Proxy
         Load Balancing
        IBM http plugin"]
    end

    subgraph ibmi_app["IBM i — Application Tier"]
        APPSVR["Application Server
        WebSphere Liberty / IAS / Tomcat /
        Node / Python / PHP ...
        REST APIs · Jakarta EE"]

        DB["Db2 for i"]
        PGM["Native Programs
        RPG · COBOL · CL"]
        IFS["IFS Resources
        HTML · Images · Files"]

        APPSVR --> DB
        APPSVR --> PGM
        APPSVR --> IFS
    end

    B1 -- "HTTPS" --> APACHE1
    B2 -- "HTTPS" --> APACHE2

    APACHE1 -- "Reverse Proxy / AJP" --> APPSVR
    APACHE2 -- "Reverse Proxy / AJP" --> APPSVR

    style clients fill:#f0f4ff,stroke:#4a6fa5,stroke-width:1px
    style ibmi_front fill:#e8f4e8,stroke:#2d7a2d,stroke-width:2px
    style ext_front fill:#fff4e8,stroke:#b36b00,stroke-width:2px
    style ibmi_app fill:#e8f4e8,stroke:#2d7a2d,stroke-width:2px
    style APACHE1 fill:#4a90d9,color:#fff,stroke:#2c5f8a
    style APACHE2 fill:#e8a020,color:#fff,stroke:#b36b00
    style APPSVR fill:#2d9e6b,color:#fff,stroke:#1a6b47
    style DB fill:#5b9bd5,color:#fff,stroke:#2c5f8a
    style PGM fill:#7b68ee,color:#fff,stroke:#4a3fa5
    style IFS fill:#70b870,color:#fff,stroke:#2d7a2d
```

Figure. Application server configuration—IBM i or external HTTP server proxying to the IBM i application tier

Beyond IBM i, the web server landscape is rich and diverse. Understanding the most widely deployed web servers helps contextualize where Apache DG1 fits and what alternatives exist when designing hybrid or cloud-connected architectures around IBM i. **Apache HTTP Server**, **nginx**, **Microsoft IIS**, **HAProxy**, **Traefik**, **Envoy**, and **LiteSpeed** are all popular web servers that can serve IBM i applications.

Web Servers on IBM i

While Apache DG1 (5770-DG1) remains the gold standard on IBM i, several other web servers can run on IBM i through the **PASE environment**.

nginx on IBM i For a detailed comparison of Apache DG1 and nginx on IBM i—including security, JWT/OAuth integration, and IWS proxying—see Chapter 12, *DG1 versus nginx*.

Node.js-based web servers Several lightweight HTTP servers are built on Node.js and run in PASE on IBM i:

- **Express.js**: The most popular Node.js web framework. While primarily an application framework, Express can serve static files and act as a lightweight HTTP server for simple use cases.
- **Fastify**: A high-performance Node.js web framework with a focus on low overhead. Like Express, it blurs the line between web server and application framework.

Python-based HTTP servers Python's standard library includes a built-in HTTP server (`python3 -m http.server`) suitable for development and testing. For production use, Python web frameworks, such as **Uvicorn** (an ASGI server for async frameworks like FastAPI), can run in PASE and serve Python web applications. These are application-layer servers rather than general-purpose web servers.

These Node.js-based or Python-based options are best understood as **application frameworks with built-in HTTP serving**, rather than dedicated web servers. For production static content serving or reverse proxying, for production IBM i web serving, Apache DG1 or nginx remains the appropriate choice. Other options are best suited for development, testing, or specialized use cases.

The Application Server

Where the web server handles the front door, the application server handles the business. An application server provides the infrastructure for creating dynamic, data-driven applications. It can be thought of as the operating system for your web applications—insulating application code from the underlying hardware, operating system, and network.

The application server is responsible for:

- Building dynamic page content by gathering data from databases or back-end programs
- Executing business logic
- Providing a runtime environment for web applications, web services, and APIs
- Acting as middleware between back-end systems and client devices

Application servers support a range of client types—browsers, mobile applications, and service consumers—making them central to modern integration architectures.

There are many different types of application servers that can run on IBM i. Servers are based on a wide array of languages, including Java, PHP, Ruby, Node.js (JavaScript), Python, etc.

```
flowchart LR
    subgraph ibmi["IBM i Platform"]
        direction LR

        subgraph websvr["Web Server — Apache 5770-DG1"]
            SEC["Security
            TLS · Auth · Firewall"]
            STATIC["Static Content
            HTML · CSS · Images"]
        end

        subgraph appsvr["Application Server — Liberty / Tomcat / Node.js / Python"]
            direction TB
            UI["Presentation Layer
            REST APIs · Jakarta EE
            JSP · Servlets · React / Vue"]
            RUNTIME["Runtime
            Java · Node.js · Python"]
            UI --> RUNTIME
        end

        subgraph core["Core Business — IBM i Native"]
            direction TB
            PGM["Programs
            RPG · COBOL · CL"]
            SRVPGM["Service Programs
            Business Logic"]
            DB2["Db2 for i
            SQL · Stored Procedures"]
            PGM --- SRVPGM
            SRVPGM --> DB2
        end

        SEC --> UI
        STATIC --> UI
        RUNTIME -- "JTOpen / JDBC / REST" --> PGM
        RUNTIME -- "JDBC / SQL" --> DB2
    end

    BROWSER["Browser
    Mobile App
    API Client"] -- "HTTPS" --> SEC

    style ibmi fill:#e8f4e8,stroke:#2d7a2d,stroke-width:2px
    style websvr fill:#f5deb3,stroke:#b36b00,stroke-width:1px
    style appsvr fill:#b2e0f7,stroke:#2c5f8a,stroke-width:1px
    style core fill:#f5e6ff,stroke:#6a3fa5,stroke-width:1px
    style BROWSER fill:#f0f4ff,stroke:#4a6fa5,stroke-width:1px
    style SEC fill:#e05050,color:#fff,stroke:#a02020
    style STATIC fill:#d4a020,color:#fff,stroke:#8a6010
    style UI fill:#2d9e6b,color:#fff,stroke:#1a6b47
    style RUNTIME fill:#4a90d9,color:#fff,stroke:#2c5f8a
    style PGM fill:#1a1a2e,color:#7fff7f,stroke:#444
    style SRVPGM fill:#1a1a2e,color:#7fff7f,stroke:#444
    style DB2 fill:#2d9e2d,color:#fff,stroke:#1a6b1a
```

Figure. Application server internal architecture on IBM i—from browser to Db2 through the web and application tiers

Container and Hybrid Cloud Architecture

Web Serving in a container architecture

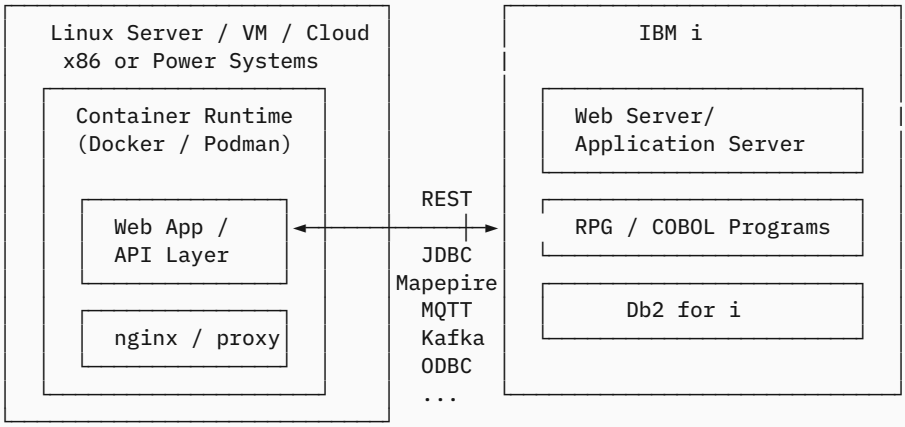
Web serving on IBM i today requires understanding hybrid cloud architectures and microservice-based design patterns. IBM i applications are increasingly integrated with containerized front-end components running on Linux—whether on-premises, on Power Systems with Linux on Power, or in public cloud environments—creating a distributed architecture where IBM i provides the authoritative data and business logic layer while containers handle modern web interfaces, API gateways, and integration middleware. This is a well-established and practical architecture. Many IBM i shops run containerized web applications, API layers, and middleware on Linux, while keeping their business logic and data on IBM i.

Docker and Podman are the two main container runtime environments used in modern application deployments at the time of writing this edition, often accompanied by orchestrators such as Compose or Kubernetes (K8s) to name just a few.

Docker and Podman are the two primary container runtimes used in modern application deployments. An important clarification upfront: **Docker and Podman do not run on IBM i**—not natively, and not in the PASE environment. IBM i's architecture is fundamentally different from Linux, and OCI container runtimes require a Linux kernel.

Containers are deployed on **separate Linux servers or virtual machines** that run alongside IBM i. The containerized applications then integrate with IBM i over the network—accessing Db2 for i via JDBC, ODBC or Mapepire, calling IBM i services via REST APIs, or invoking programs through a toolkit like JTOpen.

Typical Deployment Architecture



Prerequisites

Before deploying containers in an IBM i environment, ensure the following are in place on the **Linux host**:

- A supported Linux distribution (RHEL, Ubuntu, Debian, or similar)
- Docker or Podman installed on the Linux host
- Network connectivity between the Linux host and IBM i
- IBM i configured to accept remote connections : Db2 for i TCP/IP port 446 open, or REST APIs exposed via Apache/Liberty/IWS, etc.
- Appropriate IBM i user profiles and object authorities for the connecting application

Key Operational Considerations

- **Network security**: Ensure that only the Linux container host can reach IBM i on the required ports. Do not expose Db2 for i (port 446) or program call ports directly to the internet.
- **Credentials management**: Use environment-specific secrets injection, a secrets manager (HashiCorp Vault, AWS Secrets Manager, etc.), or Kubernetes secrets rather than hardcoding IBM i credentials in images or compose files.
- **Image storage**: Container images are stored on the Linux host filesystem. Monitor available disk space, especially when maintaining multiple image versions.
- **Logging**: Container logs are accessible via `podman logs <container-name>` or `docker logs <container-name>`. Forward logs to a centralized system (ELK stack, IBM Log Analysis, IBM Instana...) for production environments.
- **Updates**: To update a containerized application, build a new image, push it to a registry, pull it on the host, and recreate the container. This is faster and more reliable than in-place application updates. In this context, having a CI/CD pipeline that builds and deploys container images is highly recommended, and it is also recommended to have a clear DevOps strategy and a CI/CD pipeline to deploy the changes on the IBM i side.
- **High availability**: For production workloads, consider running containers under an orchestrator such as Kubernetes or OpenShift, which provides automatic restarts, health checks, and rolling updates.

AI-assisted tools like IBM Bob™ can help generate Dockerfiles, compose configurations, and integration code for connecting containerized applications to IBM i back-end programs and Db2 for i.

Conclusion: IBM i is built for Hybrid Cloud

IBM i's architecture is inherently well-suited for hybrid cloud and microservice integration. The platform provides standard, open protocols at every layer—application, middleware, and database—enabling seamless integration with containerized and cloud-native workloads, both synchronously and asynchronously.

Application layer: IBM i Web Services (IWS) exposes RPG and COBOL programs as REST or SOAP services. Liberty and Apache DG1 provide modern API gateways with OAuth 2.0 and JWT support. Containers consume these APIs over HTTPS, treating IBM i as a standard microservice endpoint. Use case example: React or Angular applications running in containers call IBM i REST APIs to retrieve customer data, submit orders, or check inventory.

Middleware layer: IBM i supports industry-standard messaging protocols including MQTT (for IoT and event-driven architectures), Kafka (for event streaming and real-time data pipelines), and MQ (for reliable asynchronous messaging). Containers publish events to IBM i or subscribe to IBM i-generated events, enabling event-driven integration patterns. Use case example: Event-driven architecture where containerized microservices publish events to IBM i via Kafka, and IBM i programs process these events to update inventory or generate reports.

Database layer: Db2 for i is accessible via JDBC, ODBC, and modern SQL-over-HTTP tools like Mapepire. Containers query and update IBM i data using standard database drivers, connection pooling. Change Data Capture (CDC) and replication tools enable near-real-time data synchronization between IBM i and cloud data warehouses or analytics platforms.

IBM i's integration capabilities are not an afterthought. They are core platform strengths that make it a natural fit for modern hybrid cloud architectures and you'll find more technical details in other chapters of the book, like Chapter 11 - Open Source.

WebSphere Application Server Options

What Is WebSphere Application Server on IBM i?

WebSphere Application Server (WAS) has been IBM's flagship Java application server for decades. On IBM i, it has been available since Version 3.5 and has evolved significantly since then. WebSphere Application Server is the exact same run time on IBM i that is run on other platforms. Today, the WebSphere family on IBM i encompasses both the

modern Liberty runtime and the traditional full-profile server, giving organizations a range of options to match their needs.

WebSphere Liberty

WebSphere Liberty is the current strategic direction for Java application serving on IBM i. It is a lightweight, modular, and developer-friendly runtime that starts in seconds and supports hot deployment—drop updated application files into the application directory and they take effect immediately, without a server restart.

Liberty's modular architecture means you enable only the features your application requires. This keeps the runtime footprint small and startup times fast, whether you are running a simple REST API or a full Jakarta EE application.

On IBM i, Liberty is included in the license and serves as the foundation for the Integrated Web Application Server (IAS), Integrated Web Service Server (IWS), and other IBM i embedded services. It supports:

- Jakarta EE (formerly Java EE) web profile and full platform
- MicroProfile for cloud-native microservices
- REST APIs via JAX-RS
- WebSockets
- OpenAPI documentation generation
- OAuth 2.0 and OpenID Connect for modern authentication

WebSphere Application Server (full profile)

The traditional WAS profile (sometimes called WAS Base or WAS Full Profile or Traditional WAS) provides the complete Java EE runtime with the full set of enterprise features. It is appropriate for organizations with existing WAS-based applications that have not yet migrated to Liberty, or for workloads that depend on features not yet available in Liberty.

On IBM i, **WebSphere Application Server Express and Base** are available at no additional charge through the IBM Web Enablement packages (5722-WE2 for WAS 8.0 and 5733-WE3 for WAS 8.5+/9.X). Unlike distributed platforms, the bundled entitlement on IBM i is not licensed by processor cores (PVUs), allowing multiple instances to run within an IBM i partition, limited by system resources rather than license metrics. Again, it is good to notice that these alternatives are compatible with Java 8 by default, and the future of Java-serving solution on IBM i is WebSphere Liberty that are compatible with more recent java versions. For more information about what's included with IBM i, see IBM WebSphere Application Server editions on IBM i.

WebSphere Application Server Network Deployment (full profile or Liberty)

For enterprise deployments requiring high availability and horizontal scalability, WebSphere Application Server Network Deployment (WAS ND) extends the base server with:

- **Clustering**: Distribute application workloads across multiple server instances for load balancing and failover
- **Dynamic scalability**: Automatically adjust capacity in response to demand
- **High availability**: Automatic failover ensures service continuity if a server instance fails
- **Advanced centralized management**: A single administrative console manages all servers in a distributed topology

WAS ND is appropriate for mission-critical applications where downtime is not acceptable and where workloads may vary significantly over time.

Java Version Dependency (5770-JV1)

WebSphere Liberty and traditional WAS on IBM i require the **IBM Java (5770-JV1)** licensed program to be installed. The specific Java version required depends on the WAS release—Liberty supports IBM Semeru Runtimes (Java 11, 17, and later). Ensure the correct JV1 option is installed and matches the Java version configured for your application server instance before installation.

Installation and Management

On IBM i, the installation process of WebSphere Liberty and traditional WAS instances is done using the command line and response file method, or through the IBM i Web Administration GUI. Once installed, these instances are managed through the **IBM i Web Administration GUI** (Web Admin), accessible at `http://hostname:2001/HTTPAdmin`.

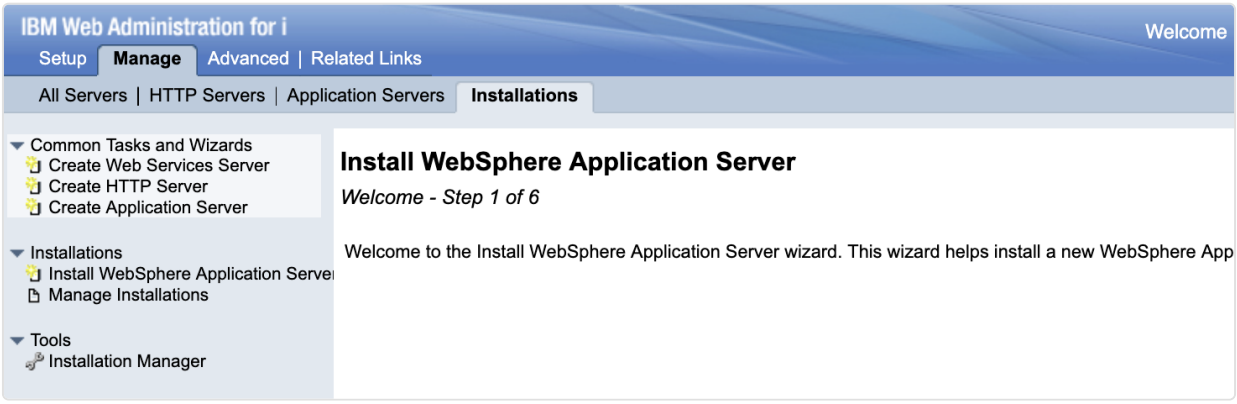


Figure. IBM i Web Administration web page

From this console you can create, start, stop, and manage application server instances, install fix packs, and deploy applications. The Web Admin GUI simplifies what would otherwise be complex command-line operations, making it accessible to administrators who are not Java specialists.

Choosing Between Liberty and Traditional WAS

For new development, Liberty is the clear choice. It is faster, lighter, more portable, and better aligned with modern development practices. IBM's strategic investment is in Liberty, and it continues to receive new features and capabilities with each release.

Connecting to IBM i: JTOpen and JDBC

Liberty, WebSphere, Tomcat, any Java-based applications interact with IBM i native resources through **JTOpen** (the open source IBM Toolbox for Java, `net.sf.jt400:jt400`), which provides JDBC access to Db2 for i and direct program call capabilities. Full coverage of JTOpen and IBM Toolbox for Java is provided in Chapter 11.

In Liberty, the recommended approach is to configure a **managed data source** in `server.xml` rather than obtaining JDBC connections directly in application code. This enables connection pooling, transaction management, and JNDI lookup:

```
<!-- server.xml: declare the JTOpen JDBC driver -->
<library id="jt400Lib">
    <fileset dir="${server.config.dir}/lib" includes="jt400.jar"/>
</library>

<!-- Define a managed data source for Db2 for i -->
<dataSource id="db2i" jndiName="jdbc/db2i">
    <jdbcDriver libraryRef="jt400Lib"
                className="com.ibm.as400.access.AS400JDBCConnectionPoolDataSource"/>
    <properties.db2.i serverName="ibmi-host"
                      databaseName="*SYSBAS"
                      user="myuser"
                      password="mypassword"/>
</dataSource>
```

Applications then look up the data source via JNDI (`java:comp/env/jdbc/db2i`) or inject it with `@Resource`, keeping connection management out of application code and under Liberty's control.

Additional Web Technologies

What Is PHP?

Definition

PHP (Hypertext Preprocessor) is an open source, interpreted web scripting language that runs more than 70% of known websites. It integrates tightly with web servers such as Apache or NGINX to provide a dynamic back end for web applications, including websites, APIs, and other web-based services.

PHP and IBM i

PHP has been available on IBM i since 2005. According to the Fortra survey, PHP operates in approximately 20% of IBM i installations. PHP provides numerous integrations with IBM i that are discussed in more detail in other sections of the Open Source chapter.

Security and Server Comparison

TLS Security for Web Applications on IBM i

Transport Layer Security (TLS) encrypts data in transit between clients and servers, preventing eavesdropping and tampering. For web applications handling credentials, business data, or personal information, TLS is mandatory.

TLS Fundamentals

TLS establishes secure connections through:

1. Server presents a **digital certificate** containing its public key
2. Client verifies the certificate against a trusted Certificate Authority (CA)
3. A secure session key is negotiated for encrypted communication

Protocol versions: Use TLS 1.2 and TLS 1.3 only. Disable TLS 1.0, TLS 1.1, SSL 3.0, and SSL 2.0—these have known vulnerabilities.

Web Serving Methods with TLS on IBM i

IBM i supports TLS across multiple web serving options:

- **Apache HTTP Server (5770-DG1)**: Native IBM i web server with DCM integration
- **WebSphere Liberty / IAS**: Java application server with built-in TLS support
- **Open source web servers**: nginx and others running in PASE or containers

Recommended Architecture

For production deployments, use a dedicated web server (Apache DG1 or nginx) to handle HTTPS and terminate TLS. This approach:

- Centralizes certificate management
- Offloads TLS processing from application servers
- Simplifies security configuration and monitoring
- Enables consistent security policies across multiple backend services

The web server acts as a reverse proxy, forwarding decrypted traffic to backend application servers (Liberty, IAS, or containerized applications) over the internal network.

Certificate Management with DCM

IBM i's **Digital Certificate Manager (DCM)** is the native tool for managing certificates. DCM integrates directly with Apache HTTP Server (5770-DG1) and WebSphere Liberty, allowing you to:

- Create and manage certificate stores
- Generate certificate signing requests (CSRs)
- Import certificates from external CAs
- Assign certificates to IBM i applications

Access DCM through the IBM i web administration interface. DCM is the recommended approach for Apache DG1 and Liberty certificate management.

Configuring HTTPS on Apache HTTP Server (5770-DG1)

The IBM i HTTP Server includes a configuration wizard that simplifies HTTPS setup:

1. Access the wizard through IBM Navigator for i
2. Select your HTTP server instance
3. Choose "Enable SSL/TLS"
4. Select or create a certificate in DCM
5. The wizard generates the necessary `httpd.conf` directives

This is the recommended method for initial HTTPS configuration.

Configuring HTTPS on Liberty and IAS

Liberty manages TLS in `server.xml`. Certificates can reference DCM keystores or Liberty-managed keystores. WebSphere Full Profile, Liberty or IAS normally use Java keystores and commonly, administrators export certificates from DCM into a Java keystore.

Here is an example of a basic Liberty TLS Configuration:

```
<server>
    <featureManager>
        <feature>ssl-1.0</feature>
        <feature>transportSecurity-1.0</feature>
    </featureManager>

    <keyStore id="defaultKeyStore"
              location="/QIBM/UserData/ICSS/Cert/Server/DEFAULT.KDB"
              type="CMS"
              password="{xor}encoded-password" />

    <ssl id="defaultSSLConfig"
         keyStoreRef="defaultKeyStore"
         sslProtocol="TLSv1.2" />

    <httpEndpoint id="defaultHttpEndpoint"
                  host="*"
                  httpPort="-1"
                  httpsPort="9443" />
</server>
```

Setting `httpPort="-1"` disables plain HTTP.

Configuring HTTPS with Open Source Web Servers

Open source web servers like nginx run in PASE or containers and manage certificates independently of DCM.

nginx TLS Configuration

```
server {
    listen 443 ssl;
    server_name myapp.example.com;

    ssl_certificate     /etc/nginx/certs/myapp.crt;
    ssl_certificate_key /etc/nginx/certs/myapp.key;
    ssl_protocols TLSv1.2 TLSv1.3;

    location / {
        proxy_pass http://localhost:9080;
        proxy_set_header Host $host;
        proxy_set_header X-Forwarded-Proto $scheme;
    }
}

server {
    listen 80;
    return 301 https://$host$request_uri;
}
```

Certificate Renewal

Let's Encrypt

For internet-facing applications, **Let's Encrypt** provides free, automatically renewable certificates. The `certbot` tool is available in the IBM i open source ecosystem or can run in containers. On IBM i, the procedure is detailed in Certbot/ Let's Encrypt on IBM i.

DCM-Managed Certificates

For internal applications or commercial CAs, use DCM to manage certificate lifecycle. Monitor expiration dates and renew certificates before they expire to avoid service disruption.

Which Web Server: NGINX or Apache on IBM i

With two web servers available, IT teams have a choice of which to deploy to support web workloads on IBM i. This section reviews both options and discusses their respective opportunities and challenges.

IBM HTTP Server Powered by Apache

The Apache server for IBM i has been available for many years, following the retirement of the Domino Go server. While Apache itself is an open source project, IBM's implementation is tightly integrated with the IBM i operating system, including its security model, jobs, and objects.

One of Apache's primary strengths on IBM i is its integration with ILE. Apache is commonly used as a front end for RPG, COBOL, and CL applications through technologies such as CGI, PHP, XMLSERVICE, and Node.js. Requests processed by Apache run within standard IBM i jobs, honoring user profiles, adopted authority, CCSIDs, and auditing. This model allows Apache to function as a natural extension of traditional IBM i workloads rather than as an external component.

Apache is also highly mature and configurable, with a long history in regulated industries. Capabilities such as FastCGI, authentication modules, URL rewriting, logging, and virtual hosting are well understood and extensively documented. IBM supports Apache as part of the IBM i operating system lifecycle, which is important for organizations that value stability, compliance, and predictable support.

This deep integration also introduces tradeoffs. Apache's process and threading model is heavier than that of newer web servers, and configurations can become complex over time. For workloads that involve high-volume static

content or very large numbers of concurrent connections, Apache may require additional tuning and system resources compared to event-driven alternatives.

NGINX

NGINX runs on IBM i within the PASE environment, alongside other open source tools such as Python, Node.js, and Git. Unlike Apache, NGINX is not integrated with ILE or native IBM i program execution. Instead, it follows a Linux/UNIX-style deployment model and focuses on performance and scalability.

NGINX uses an event-driven, asynchronous architecture that allows it to handle very high numbers of concurrent connections with relatively low memory usage. This makes it suitable for serving static content, operating as a reverse proxy or load balancer, acting as an API gateway, and fronting application servers built with Node.js, Python, or containerized workloads.

NGINX configuration is generally more centralized and simpler than Apache's module-based configuration model. While NGINX can be easier to adopt from a configuration standpoint, documentation and support resources specific to IBM i environments are more limited.

A key limitation of NGINX on IBM i is the lack of direct integration with native ILE programs. Interactions with traditional IBM i application logic typically occur through APIs or intermediary services rather than direct invocation. In addition, NGINX follows an open source RPM-based lifecycle rather than being supported as part of the IBM i operating system, which may be a consideration for environments that prefer a traditional IBM i support model.

Which One?

Both Apache and NGINX are capable, enterprise-grade solutions for supporting web workloads on IBM i. Either can satisfy the majority of use cases. As a result, the deciding factors often come down to available skills and workload characteristics.

Apache is a natural fit for organizations with existing Apache expertise and workloads closely integrated with RPG business logic. NGINX may be preferable for scenarios that require a lightweight, high-performance solution capable of extreme scalability, or when team skills more closely align with modern Linux-style web server administration.

Application Server Options on IBM i

An application server provides the runtime infrastructure required to build and run enterprise applications. It offers middleware services such as web application execution, security, transaction management, messaging, and database connectivity. In many ways it can be viewed as a runtime platform for business applications, providing standardized APIs and services that simplify development. Typical capabilities include support for dynamic web applications, web services, business components, transaction processing, and messaging systems.

IBM i supports a range of application server options, from lightweight integrated runtimes to full enterprise-grade containers. Choosing the right one depends on your application's complexity, your team's skills, and your operational requirements.

Choosing the right Application Server

Not every application needs the same infrastructure. IBM i offers a spectrum of options that scale from simple, low-overhead deployments to highly available, clustered enterprise environments. The following questions help guide the decision:

- **How complex is the application?** Simple web applications with servlets and JSPs have different needs than large-scale transactional applications.
- **What are the availability and scalability requirements?** High-traffic or mission-critical applications may require clustering and failover capabilities.
- **What is the team's experience?** Liberty is developer-friendly and portable. Full WebSphere Network Deployment requires more operational expertise. Node.js uses the same JavaScript technology on the client and the server side.
- **Is the application already running elsewhere?** Liberty's portability makes it easy to develop on a workstation and deploy to IBM i without changes. Same of most of the Open Source runtimes.

Available Application Server Options on IBM i

WebSphere Application Server Liberty and Full Profile

WebSphere Liberty is the strategic, modern, lightweight Java application server from IBM. It is the recommended starting point for new web application development on IBM i and for migrating existing applications away from heavier containers. Liberty is the foundation of the Integrated Web Application Server (IAS). It supports modern Java EE and Jakarta EE standards, MicroProfile, REST APIs, and WebSockets. The traditional WebSphere Application Server (WAS) profile provides a full Java EE runtime with a rich set of enterprise features. It is appropriate for organizations with existing WAS-based applications or those requiring the full Java EE specification.

See Chapter 12, *What Is WebSphere Application Server?*, for more details.

Integrated Web Application Server (IAS)

The Integrated Web Application Server is built into IBM i and requires no separate installation. It is based on WebSphere Liberty and is the right choice for running IBM-provided applications (such as IBM Navigator for i, and the Web Administration GUI) as well as simple custom web applications. IAS is an excellent starting point for teams new to Java web development on IBM i.

See Chapter 12, *What Is the Integrated Web Application Server (IAS)?*, for more details.

Apache Tomcat

Tomcat is a popular open source servlet container from the Apache Software Foundation. It is a pure Java application and can be downloaded and run on IBM i with minimal effort. Tomcat is well-suited for teams already familiar with it from other platforms and for applications that do not require the full Java EE (Jakarta) stack. Tomcat is one of the open source Java application servers that can run on IBM i like WildFly, OpenLiberty, and Eclipse Jetty.

Open Source Runtimes (Node.js, Python, Ruby)

IBM i's open source ecosystem has expanded significantly in recent years. Node.js, Python, and Ruby are all available through the IBM i open source package manager (yum/dnf). These runtimes open IBM i to a broad range of modern web frameworks—Express.js, Node-RED, Django, Flask, Ruby on Rails—and are increasingly used for building REST APIs and microservices that integrate with IBM i back-end programs. These runtimes run in the IBM i PASE environment and can call IBM i native programs and access Db2 for i directly.

PHP on IBM i

PHP has a long history on IBM i and remains in active use at many sites. It is a scripting language well-suited for web application development, and its relatively low barrier to entry made it an attractive option for IBM i shops looking to build modern web interfaces without a full Java stack.

PHP on IBM i is available in distinct flavors.

For more information, see IBM i PHP documentation.

Conclusion: Application Server Options on IBM i

The IBM i platform's flexibility means you are not locked into a single choice. Many organizations run multiple application servers simultaneously, each serving a different workload or team.

Security on IBM i

What is 'Deny by Default' and How is it Implemented on IBM i?

What It Is

Deny by default is the assumption that access to an application and its data is denied. For a user to access (run) the application, the access controls must have been configured to allow access.

Why It Matters

Let's face it, until recently, security was not a major concern for most organizations unless they were in a regulated industry or had to comply with a regulation such as the Payment Card Industry's Data Security Standard (PCI DSS). As such, many applications simply took the system defaults, leaving most library-based objects with a default setting allowing data to be read and worse, modified, by any user on the system. Objects in directories are often left with the authority providing all users the ability to not just read or modify, but delete the entire object. Couple this with the plethora of ways to access the system today - ODBC, SSH, DRDA, HTTP, etc as compared to the easily locked down 5250 emulation environment of the past, and organizations are finding their data highly exposed.

Because of the rise in malware (including ransomware), threats from social engineering, vulnerabilities discovered in popular applications and the insider threat, many organizations have made security a priority. Taking the approach that everyone is denied, except if they're specifically allowed, significantly reduces the risk to their data.

How It Works

Deny by default on IBM i requires that the default *PUBLIC authority of application objects (especially objects containing data) be set to deny (*EXCLUDE). For objects in the IFS, this means that *PUBLIC will be set to DTAAUT(*EXCLUDE) OBJAUT(*NONE).

Key Takeaways

- Deny by default is now the expected security strategy.
- Due to the rise in attacks, data is vulnerable when not secured.
- Deny by default means that *PUBLIC authority is *EXCLUDE for library-based objects and DTAAUT(*EXCLUDE) OBJAUT(*NONE) for IFS objects.

What is Least Privilege Access and How is it Implemented on IBM i?

What It Is

Least privilege access requires users be assigned only the access and capabilities required to perform their jobs ... and no more.

Why It Matters

Granting users more access or capabilities than they need to perform their tasks risks users performing tasks for which they are not qualified (with the risk of the task being done incorrectly - by accident or on purpose), accessing data for which they have no business requirement (potentially putting the business out of compliance with laws and regulations), or modifying or deleting data (by accident or on purpose) causing business disruption.

How It Works

Capabilities on IBM i are called 'special authorities.' Special authorities give users the ability to perform certain tasks on the system. The table below describes the capabilities provided by each special authority. One special authority - *ALLOBJ - provides all access to every object on the system. In the past, some application providers required

application users to be assigned *ALLOBJ special authority. This ensured there would be no security issues with the application! However, this approach is no longer acceptable.

> **Note:** Application programs can be owned by a profile with the necessary special authorities, and be configured to adopt authority, eliminating the need for application users to have the special authority themselves.

Some organizations demand justification for requiring users have special authorities. Therefore, applications requiring users have a special authority should be prepared to describe (justify) why the special authority is required. Applications adopting a profile that has special authorities should also be prepared to describe why the special authorities are required. In other words, be prepared to describe the operating system functions in use that require the special authority.

Special Authority	Description
*ALLOBJ	All access to every object on the system
*AUDIT	Management of audit configuration
*IOSYSCFG	Communications configuration and management including the TCP/IP configuration and the creation of file shares
*JOBCTL	Management of any job on the system
*SAVSYS	Ability to save / restore any object on the system - including the entire system. With or without authority to the objects
*SECADM	Create/Change/Delete user profiles
*SERVICE	Ability to use Service Tools
*SPLCTL	Management of any spooled file on the system regardless of authority to the outq

In addition, when granting access to users or groups to application objects via an explicit (private) authority, they should only be granted sufficient authority to perform the task. For example, when a user or group needs to download information from a database file, they should only be granted *USE authority not *CHANGE or *ALL.

Finally, users and groups should only be given access to tasks that correspond to their job responsibilities. This pertains to menu options whether green screen or graphical, functions defined via function usage, etc.

Key Takeaways

- Only require special authorities when the underlying operating system function requires them.
- Grant only the authority required for users or groups to successfully perform an operation.
- Allow access for users and groups only to the tasks that correspond to their job function

---

What is Role-based Access Control (RBAC) and How is it Implemented on IBM i?

What It Is

RBAC is the concept of allowing access to application features based on a user's job function. For example, in a banking application, tellers may be able to see an account but only teller managers can create or remove one. Access controls are then defined to support this separation of duties.

Why It Matters

Ease of managing access to application features requires an RBAC architecture. In addition, many regulations require separation of duties. If the application is not a single feature, it's best to architect access to individual features to allow management of access via a user's role.

How It Works

RBAC is implemented via group profiles on IBM i. Users are assigned to a group profile that has been created to represent the role (job function) that they perform. If required to successfully perform their job function, the group profile is assigned the special authorities required for the role. (Many roles require no special authorities for the users to successfully perform their tasks.) For example, a group profile representing system operators is typically assigned *SAVSYS and *JOBCTL special authorities while application users require no special authorities. Often, users are a member of more than one group (role). For example, if some application users also need access to application data via an ODBC connection, they may be assigned to a group that has been granted *USE authority to the authorization list securing the database files to which they need access. RBAC helps administrators implement the concept of 'least privilege access' by granting access only to the roles that require it for their job function. RBAC makes application access administration easier. Once all the roles (group profiles) have been defined, when a new user joins the organization, the security administrator simply has to assign them to the group(s) corresponding to their role(s) to configure their access.

Key Takeaways

- RBAC makes managing access controls easier
- RBAC is required by many regulations
- RBAC helps implement the concept of least privilege access

What is Identification and Authentication and How Does it Occur on IBM i?

What It Is

In security parlance, the core of access control is composed of Identification, Authentication and Authorization (IAA). Authorization will be discussed later. Identity is the process of assigning a name to an entity so you can know the identity associated with a task - for example, Carol. Authentication requires the Identity to provide something (or multiple things in the case of Multi-factor Authentication (MFA)) that provides proof they are who they say they are. This is something they know (such as a password or pin), they have (such as a security token or smart phone with an authentication app) or they are (such as a retinal scan or fingerprint.) Location (such as an IP address) may also be used as a fourth type of authentication.

How It Works

On IBM i, an Identity is a User Profile (*USRPRF) object. To establish a connection or sign on to IBM i, the Identity (Carol) must provide authentication information because you can't blindly trust that Carol is Carol. The Authentication aspect of access control requires that Carol prove that she is who she says she is. Authentication information on IBM i most often comes in the form of a password. Provide a valid user profile / password combination and you've

successfully authenticated to IBM i. Another option is for the Identity to present a Kerberos ticket for authentication if the IBM i is participating in an organization's Single Sign-on (SSO) configuration. Finally, if connecting via SSH, you have the option to authenticate via public/private keypair.

As of IBM i 7.6, the operating system can be configured to require multiple authentication factors (MFA). Information on configuring IBM's MFA can be found at Security Multi-factor authentication (MFA). Note that prior to IBM i 7.6 some IBM i security vendors provided their own MFA solutions and continue to offer them, using the feature IBM provided to extend MFA beyond the operating system features.

Web applications that aren't using IBM i user profiles (*USRPRF) as the Identity have other options. In the past, they may have used a validation list to store an identity and its authentication information then used this information when a user attempted to log into the web application. This method was an implementation of Basic authentication. However, many security issues exist with basic authentication including the constant exchange of user/password combinations, credentials being cached in the browser and lack of support for Multi-factor authentication (MFA) to name a few. As such, this method of Authentication has been deprecated and is no longer supported by most major applications and browsers. Instead, many have turned to well-known industry standards such as OAuth that allow you to authenticate to Active Directory or use Apple, Google or other well-known user repositories. This technology uses tokens for the authentication mechanism and is significantly more secure than basic authentication. While this technology is not available to perform authentication to sign in to an IBM i session, vendor solutions exist that allow you to use this technology for authentication when developing APIs on IBM i.

Information on implementing Single Sign-on (SSO) using Kerberos on IBM i can be found at PDF file for Single sign-on.

Information on configuring public/private key pairs for SSH authentication can be found at How to Configure and Use SSH on IBM i.

Key Takeaways

- To establish a connection on IBM i, one must present an Identity (a *USRPRF) and valid Authentication information - typically a password or Kerberos ticket.
- As of IBM i 7.6, the operating system can be configured to require MFA.
- Modern authentication techniques can (should) be used for Authentication when writing APIs for IBM i.

What is Data Classification?

What It Is

Data Classification is the process of analyzing data to determine its value and importance to an organization and assigning it to a category (classification) so it can be handled properly. Data classification isn't just assigning a clearance (for example, Top Secret, Confidential, Internal Use or Public) to determine which roles can access the data, the classification also defines how the data is secured (deny by default, read-only, etc), whether the data needs to be encrypted both in motion and at rest, how long the data is retained, data disposal requirements as well as appropriate use of the data (including whether it's allowed to be used to train AI models.)

Why It Matters

Classifying data shows that an organization recognizes the value of the data and wants to ensure the data is managed appropriately.

How It Works

Classifying data is typically performed at the organization level - not by a team representing a particular operating system. This is to ensure data is handled consistently across all technologies. A key player in the classification of data is the data owner. They, rather than system administrators or developers, should be best informed about the value of the data to the organization and whether there are legal or compliance requirements for their data and should be the one to set the data's requirements. Developers are responsible for creating an application with a security schema that

supports the data's classification requirements. Administrators are the data custodians and responsible for managing the data to the data owner's data classification requirements.

Some organizations - when they hear 'data classification' assume that it means that every column in every table needs to be classified. While that is one approach, that's not the approach we recommend primarily because that's a huge task that would never be completed - especially when classifying an existing application for the first time. Instead, most organizations know whether an application stores PII or cardholder data or comes under some type of regulatory requirements. Organizations can also value or put a level of importance on the application's data, regardless of whether it comes under laws or regulations. So initially, an organization can define a classification to an entire application. Then, if they choose to classify more granularly, they can find the tables that contain the data they value the most or is under a law or regulation.

When designing a new application, developers should document the tables containing PII and other regulated information as well data deemed highly sensitive to the organization. This allows access and access approval to this information to be much more granular should an organization desire this approach (rather than to apply a classification to the entire application.)

> **Note:** that data classification is not a one and done exercise. Data classification should be reviewed annually to determine if all requirements are still valid. It's possible that some data that hadn't been considered PII (Personally Identifiable Information) now is or new laws or regulations may have been passed governing additional types of data or technology may have advanced requiring new considerations (such as the use of data with AI.)

Key Takeaways

- Data classification indicates the value of the data to the organization and ensures that the data is handled consistently and appropriately throughout the organization.
- The data owner should be the one setting the data classification requirements.
- Data classification should be reviewed annually.

How Do You Determine if Additional Audit and/or Compliance Requirements are Necessary?

What It Is

Audit and compliance requirements are typically driven by the organization's industry (such as healthcare or finance), the type of data being stored or processed (such as PII - Personally Identifiable Information) or the laws of the specific country, or in the case of the United States, the State in which the organization resides or does business. Finally, the organization's classification of the data may require additional logging - perhaps when someone accesses the data or perhaps only when they modify or delete it. The data owner and/or application architect should know the requirements of the data your application works with and should be able to articulate the requirements. If they don't, search the Internet for the compliance requirements for your location and organization's industry (such as finance) as well as the current definition of PII data to understand the compliance requirements you must code to.

Why It Matters

Showing how your application adheres to the laws and regulations pertaining to the application's data is an important requirement if you are developing a commercially available application. When developing an internal application, understanding all compliance and audit requirements and designing them into the initial architecture is far easier than attempting to retrofit them into a 'final' product.

How It Works

Ask the data owner/application architect for the exact law(s) and regulation(s) and read them for yourself. The requirements often sound obscure and unreasonable. But read in the context of the entire writing, the requirements are usually more clear and understanding the context often brings about a better implementation of the requirement.

Key Takeaways

- Ask the data owner or application architect for the compliance and audit requirements associated with the application's data.
- If additional requirements are driven by a law or regulation, read it for yourself.
- Architect the requirements from the beginning of the project - don't leave them until the project is nearly finished.

What are the Options for Providing Access to IBM i Application Objects?

What It Is

The third aspect to access control in the security world is Authorization. (The other two are Identification and Authentication discussed earlier.) To run a program or update a table or do any other task on IBM i the Identity running the job (that is, the user profile (*USRPRF object)) requires sufficient authority.

Why It Matters

A variety of methods exist that will provide sufficient authority to run applications; however, some methods - while providing access - actually cause application data to be vulnerable. In the age of 'secure by design' the focus must be on providing access without creating vulnerabilities.

How It Works

When a command is called or a program is run, IBM i checks to ensure the process has sufficient authority. The options available for a user to have authority are:

- *ALLOBJ special authority assigned to the user's profile or one of its group profiles
- an explicit or 'private' authority granted to the user or one of its groups
- a private authority granted to an authorization list securing the object (such as a database file)
- primary group authority granted to one of its groups
- *PUBLIC authority which is the default authority for the object. It's the authority one has to an object if the authority doesn't come from another source.

The exact order in which IBM checks authority is discussed in IBMI-SEC-PROCEDURE-194.

To implement the concept of 'deny by default', you'll need to:

- Set the default access (*PUBLIC authority) to the application container(s) to be 'denied'. For a library (*LIB), the setting is *PUBLIC(*EXCLUDE). For a directory (*DIR), the setting is *PUBLIC DTAAUT(*EXCLUDE) OBJAUT(*NONE). If the application profile doesn't own the container, they'll (obviously!) need to be authorized to it. If your application has multiple users, the users will also have to be authorized to the container. In this case, users are typically placed in a group profile and the group profile is authorized to the container.
- With the container restricted, it will depend on your organization's policy as to whether the rest of the application's objects are also set to deny by default.
    - We highly recommend that database files be set to deny by default so it is a conscious choice who is granted authority to the application data, thus providing authority that allows them to access the data regardless of the interface being used.

- If you're relying on *PUBLIC authority, we recommend that the rest of the application objects be set to *PUBLIC *USE (or *RX for objects in a directory).
- Note that some object types require a user have *CHANGE authority - *MSGQs, *DEVD for example

Requiring users to have *ALLOBJ to gain sufficient authority to run an application (such as an AI Agent or an API) is no longer an acceptable practice because it violates the premise of 'least privilege access.'

This is true whether you're writing a web application, an application that makes an ODBC or DDM/DRDA connection, an AI Agent or a traditional application that runs in a 5250 emulator. Also not an option is requiring the application owning profile to be one of the application users' group profiles. Any objects that a group owns, the members also own. This means that application users have *ALL authority to application objects. This, again, violates the premise of least privilege access. So how do you configure your application's security schema and how (which profile) should own your application? The answer to these questions depends on the type of application.

If your application is making a connection to IBM i and running (establishing a job) as the profile that's being authenticated:

- Create a profile to represent your application or task that's being performed on IBM i
- Authorize that profile to the data it needs to access, granting only sufficient authority no more. (Again, it is not acceptable to simply grant this profile *ALLOBJ. See the topic How do I verify that the security schema I have in place is appropriate? for tips on determining how much authority is required.
- Because a job is going to run as the profile being authenticated, this typically this means a hardcoded password must be stored so that it can be passed on the connection. If you must use a hardcoded password, assign the strongest password allowed by the IBM i being connected to and secure the source code or script that stores the password. Use of this profile/password should not be available to the entire development community!
- If you have more than one application or a different task that makes these types of connections, create a separate profile for each application. For example, each AI agent should have its own profile. This makes debugging easier and audit entries more meaningful. It also ensures that you can authorize the profile only to the objects (especially data) that is required for the task. Also, if a special authority is required for one task (think AI Agent performing different roles - security tasks vs system operator tasks) that special authority can be assigned only to the Agent performing the task and isn't available to other Agents.
- While these profiles are typically created with a non-expiring password, compliance standards may require you to change it periodically. Make sure documentation exists listing where this profile connects and what application its used for so you'll know what will be affected when the password needs to be changed.
- Ensure all connections (even internal) are encrypted. That is, communications should be made using the TLS 1.3 encryption protocol so that the user id and password - as well as the data - does not flow in cleartext.
- If your application requires the user to be authenticated but that user is not going to be the user profile (*USRPRF) that runs the task on IBM i (Think web application (defaulting to run as QTMHHTTP and QTMHHTP1 profiles) or APIs using the Integrated Web Services server defaulting to run as QWSERVICE), create a profile that will run the application programs and access the application data rather than using the default IBM i profile defined for the server. This will make debugging easier and auditing of the access more clear. Using a profile representing each individual application or type of connection (e.g., one profile to access sales data and another accessing inventory) ensures access to data is only granted to the appropriate profile. Using one profile for all applications means one application may have authority to data for which it should have no access.
- Grant this profile only sufficient authority to only the objects needed to perform the actions of the web application or API.
- Do not use Basic authentication. Rather, use OAuth or some other form of modern authentication where possible.
- Configure the instance to make an encrypted connection.

If you are making a connection to a remote database using DDM/DRDA, you have the option to use Server authentication entries for sending encrypted authentication information to the target system. You can even use Kerberos authentication if that's been configured for your network. For more information, see:

- Achieve Improved Database Interoperability with SQL and RDB Aliases
- Add RDB Directory Entry (ADDRDBDIRE)
- Add QDDMDRDASERVER server authentication entry special value
- If you are modernizing an existing, traditional 'green screen' application where some users will continue to use a 5250 interface, the best way to secure the application is to: -- Ensure all application objects are owned by the application profile -- Do NOT make the application owning profile a group and require all users to be a member. Rather, configure the application programs to adopt authority (see What is Adopted Authority below). The adopted authority will provide sufficient authority for the users operating within the application. -- Secure all database files and any other object that contains data with an authorization list created specifically to secure the application's data. -- For users accessing application data from an interface outside of the application menus (such as Run SQL Scripts, downloading to a spreadsheet, etc.) grant users (or their group) *USE authority to the authorization list securing the data and set the authorization list to *PUBLIC *EXCLUDE. (While not typical, some users may be allowed to modify the application data from outside of application menus. If that situation exists, grant those users *CHANGE to the authorization list.) -- For other considerations and complete, step by step instructions for securing a traditional green screen application see the book IBM i Security and Compliance, 3rd Edition, available from Amazon.com

Which Profile Should Own the Application Objects?

You should create your own user profile to own your application objects. Resist the temptation to have IBM-supplied profiles such as QPGMR own your application. Doing so creates confusion between what's an operating system object and what's been created by a development team. It also causes exposures when a vendor application is owned by QPGMR, for example, and the organization makes all their developers a member of QPGMR. Now all developers own the vendor application's objects! (Anything a group owns, the members also own.) Finally, some organizations modify IBM-supplied profiles such as QTMHHTT1, QSYSOPR and even QUSER granting them *ALLOBJ special authority. Or remove special authorities from profiles such as QPGMR. As such, you cannot be assured of the special authorities the owning profile will have.

> **Note:** IBM i 7.6 introduced a set of profiles that cannot be changed - QPGMR_NC for example. Even though these profiles cannot be changed, we still DO NOT recommend that these profiles be used as application owner profiles for the other reasons listed.

What are the Appropriate User Profile Attributes for Application Profiles?

When creating the profile to own your application, it should be created with only the capabilities required for the application to run without authority failures - no more. The user profile attributes recommended below reduce the risk of the profile being exploited and should also keep it off of audit/compliance reports that will require an administrator to explain/justify the setting.

For profiles that own the application and profiles that are used in place of IBM-supplied profiles (e.g, QWSERVICE and QTMHHTP1):

```
CRTUSRPRF USRPRF(APPOWNER)
  PASSWORD(*NONE)
  PWDEXPITV(*SYSVAL)
  STATUS(*DISABLED)
  SPCAUT(*NONE)
  GRPPRF(*NONE)
  INLPGM(*NONE)
  INLMNU(*SIGNOFF)
  ATNPGM(*NONE)
  LMTCPB(*YES)
  TEXT('Meaningful description')
  AUT(*EXCLUDE)
```

For profiles that have to make a connection and require a password:

```
CRTUSRPRF USRPRF(AGENT1)
  PASSWORD('Make this a passphrase if possible!')
  PWDEXPITV(*NOMAX)
  STATUS(*DISABLED)
  SPCAUT(*NONE)
  GRPPRF(*NONE)
  INLPGM(*NONE)
  INLMNU(*SIGNOFF)
  ATNPGM(*NONE)
  LMTCPB(*YES)
  TEXT('Meaningful description')
  AUT(*EXCLUDE)
```

Key Takeaways

- Create your own profile to own your application and to run tasks - do not use an IBM-supplied profile
- Whether creating a profile to own your application or to connect to IBM i and perform a task, create the profile with the least amount of privilege possible
- Granting *ALLOBJ to an application profile or profile running a task just to ensure there are no authorization errors is not acceptable!

What is Adopted Authority?

Adopted authority is a method that provides additional authority to the user as long as the object adopting is in the call stack. While active, the process can use the authority of the owner of the object to perform tasks or access objects. It's used to provide the caller with temporary authority to an object or temporary use of a special authority without having to assign it to the profile or grant access permanently.

Why It Matters

Adopted authority is a popular method to temporary give users additional authority; however, it's critical one understands the details of how it works to both take fully advantage of this feature as well as use it without causing exposures.

How It Works

Adopted authority may be used as a way to provide users sufficient authority to work with application data while in the context of an application but when application database files (for example) are accessed using a non-application

interface such as Run SQL Scripts, the users have no authority. Adopted authority may also be used to provide temporary elevated authority. For example, the help desk may need to reset user passwords. This would require that they have *ALLOBJ and *SECADM special authorities to successfully work with all user profiles but they do not need all of the access these special authorities provide! In this situation, a program that is owned by a profile that has *ALLOBJ and *SECADM can be created to perform the password reset that can be called by members of the HELPDESK group. Adopted authority provides the authority (*ALLOBJ and *SECADM) that's needed but only when the program is called and only for as long as the program is active - that is, in the program call stack.

The types of objects that can be configured to adopt its owner are: *PGM, *SRVPGM and *SQLPKG. An object is configured to adopt when the User profile attribute is set to *OWNER when compiling a program. The default setting for *PGM and *SRVPGM objects is *USER which means it does not adopt. The default for an *SQLPKG is *OWNER.

When creating an SQL ILE object (by using the CRTSQLRPGI command for example) you must be aware of a different default for the User profile attribute - that is, *NAMING. Left at this default, if the object is created with naming convention *SQL, the User profile attribute will be set to *OWNER when the object is created. If the naming convention is *SYS, then User profile will be set to *USER. Or... you can change the default and specify the value - either *OWNER or *USER. In addition, you must also consider setting another attribute - Dynamic user profile (DYNUSRPRF). The Dynamic User Profile attribute must be set to *OWNER if you wish the dynamic SQL statements contained in the object being created to adopt the owner.

The User profile attribute is set when the object is created. If you decide later that a program or service program should (or shouldn't) adopt, you can change the User profile attribute using the appropriate change command. However, this is not an option for the Dynamic user profile attribute. Prior to adding the following Db2 for i Utility service, the only way to change the DYNUSRPRF setting was to re-compile. This was prohibitive for many reasons. Calling this utility flips the attribute to the opposite setting. If the setting was *OWNER, calling this utility changes it to *USER likewise, if it was *USER, calling the utility changes it to *OWNER.

```
CALL QSYS2.SWAP_DYNUSRPRF('APP_LIB', 'PGM_NAME', '*PGM')
```

Documentation for SWAP_DYNUSRPRF can be found at SWAP_DYNUSRPRF procedure.

You can view the User profile and Dynamic user profile attributes using the QSYS2.SYSPROGRAMSTAT view, which contains one row for each program, service program, and module that contains SQL statements. For more information, see SYSPROGRAMSTAT.

Restricting adopted authority

As stated previously, adopted authority is based on the object adopting being in the program call stack. As long as the object is in the stack, adopted authority is available. In fact, if multiple programs in the stack adopt and are owned by different profiles, all of the owners' authorities are available to be used. But, on occasion, you may want the user's access and capabilities to be evaluated based only on the user - not including adopted authority. For example, if a program presents a command line, you'd only want the user's authority in effect, not additional adopted authority. To accommodate this, change the Use adopted authority (USEADPAUT) attribute of the program presenting the command line. By default, adopting objects are created with USEADPAUT set to *YES - meaning that adopted authority is available to the currently running program from other programs already in the call stack. Changing this value to *NO means that adopted authority from previously called programs is not available. In other words, it breaks the flow of the adopted authority. Alternatively, you can use the MODINVAU MI instruction within your program to stop propagating adopted authority to subsequently called programs.

Viewing adopted authority attributes

You can view objects' current settings for User profile, Use adopted authority and Dynamic user profile by using one of these SQL services:

- qsys2.program_info for program (*PGM) and service program (*SRVPGM) objects
- qsys2.sysprogramstat for information for each program, service program and module containing SQL

- qsys2.bound_module_info for module (*MODULE) objects

Care needs to be taken when using this powerful feature:

- The program owner should have only the capabilities required to perform the intended task. If a program needs to perform a task that requires a special authority - especially *ALLOBJ, the task should be moved into its own program and owned by a powerful profile. The program is called, the task is performed and then returns so that the adopted authority of this powerful profile does not remain in the call stack with the ability to be used and potentially exploited.
- Programs that adopt should always make library qualified calls whether calling another program or when accessing a database or running embedded SQL to make sure the correct object is being accessed and not an object by the same name that's in a library higher in the library list.
- If a command line is being presented, separate the call into another program and change it to Use adopted authority *NO (from the default of *YES) to ensure only the user's capabilities are in effect at the command line.
- If you decide to adopt a profile with *ALLOBJ, be prepared to justify the need for this power with your compliance team or to your customers if you are a vendor.

Key Takeaways

- Adopted authority is a method of temporarily giving access to a user or elevating their privileges while a program is active
- Adopted authority flows down the call stack and by default is used by every program. To stop its use within a program, change the program to USEADPAUT *NO
- When a program is configured to adopt (USRPRF *OWNER) make sure every object access is library qualified.

In What Order is Authority Checked on IBM i?

What It Is

Part of the architecture of a Machine Instruction (MI) is the authority required to successfully execute the instruction. Before an MI instruction is executed, this authority checking algorithm is run to determine whether the process has sufficient authority. The algorithm executes in SLIC (System Licensed Internal Code) and is therefore very difficult, if not impossible, to be by-passed or modified. This is one of the strengths of IBM i.

How It Works

To successfully implement a deny by default security scheme, one must understand the order in which authority is checked on IBM i. The following figure shows the flow of the algorithm.

IBM i Authority Checking Algorithm

Figure. IBM i Authority Checking Algorithm

Step 1 - User level: The key takeaway when examining this algorithm is that, when sufficient authority is found, the algorithm stops and does not go on to the next step. This is why, if you grant a user a private authority of *EXCLUDE to an object but they have *ALLOBJ special authority assigned to their user profile, the user can access the object. The algorithm recognizes the user's *ALLOBJ and allows access. It never sees that the user has a private authority that would prevent access.

Step 2 - Group level: If no authority is found at the user level, the algorithm proceeds on to the group(s). The premise is the same as the User level. When authority is found, the algorithm stops. The strange thing about this portion of the algorithm is if the user has multiple group profiles, the algorithm will search all groups until sufficient authority is found or you run out of groups. Even stranger, groups' authority are cumulative. For example, if GROUP1 has a private authority of *OBJOPR to a *FILE object, GROUP2 has *READ, GROUP3 has *EXCLUDE and GROUP4 has *ADD,

*UPD, *DLT, *EXECUTE, the net result after processing the group list is that the user has *CHANGE authority to the file.

Step 3 - *PUBLIC authority: If no authority is found at either the User or Group levels, the algorithm proceeds to the *PUBLIC level. *PUBLIC authority can come from either the object itself or, if it's secured with an authorization list and the object's *PUBLIC authority is *AUTL, then the *PUBLIC authority comes from the authorization list.

Step 4 - Adopted authority: At this point the algorithm stops and the user receives a Not Authorized message unless adopted authority is in effect. If the program currently executing or a program in the call stack has been configured as USRPRF(*OWNER), then adopted authority is available to the process. The User portion of the algorithm is evaluated for the owner of the current program (Note: only the User portion of the algorithm is examined - it never proceeds to the owner's group(s)). If the owner's authority isn't sufficient and other programs in the call stack adopt a different owner, those owner's User authorities are evaluated for sufficient authority. If adopted authority is not in effect or it's insufficient, the user receives a Not Authorized message.

Step 5 - RCAC: If Row and Column Access Controls (RCAC) have been defined, the user must have sufficient authority to the database file before RCAC are evaluated.

Finally, while this is the logical flow of the algorithm, various performance enhancements have been implemented over the years that allow parts of the algorithm to be skipped in certain conditions. (The logic wasn't changed - just some aspects of how it's checked to streamline performance.) These are discussed in the Performance section.

Authority Checks and the Audit Journal

Some CL commands, APIs, and SQL interfaces have authorization requirements that require a user have a specific special authority (such as *IOSYSCFG) or be allowed access via a function usage. For interfaces that support both special authority and function usage, special authority is checked first and function usage is checked last. You might wonder what to expect to see in the audit journal for this situation.

First, when a user receives a typical 'Not Authorized' message - as in a user needs *USE authority to an object but is excluded, the audit journal will reflect an AF audit journal entry with a violation type A - Not authorized.

When the operation falls under the situation where either a special authority or function usage will suffice and the user lacks both of the authorization options, only the Generic Record (GR) audit journal entry, with Function registration operations (F) entry type will appear in the audit journal. You might expect to also see an Authorization Failure (AF), with violation type Special authority violation (K) produced, because after all, the user did fail the check for special authority.

However, the operating system avoids producing the AF-K record in this situation for two reasons:

1. If the user has the required function usage, they successfully pass the authorization requirements and an AF-K record would be inaccurate.
2. If the user lacks the required function usage, the GR-F record reflects the authorization failure and avoids logging two audit journal entries for a single user action.

Key Takeaways

- Authority checking is done in a clear order
- Authority checking stops as soon as the check is successful
- For performance critical processing, you can leverage this topic's detail to tune security for performance

How Does a Deny by Default Security Scheme Affect Performance?

What It Is

The simple answer is that a deny by default security scheme should not affect performance. Because of the speed of the processors and the performance enhancements added to the authority checking algorithm over the years, it takes a very messy security configuration to affect performance. For example, if access is controlled by private authorities

and users have different authorities to different objects and every application object has private authorities, this could cause a performance issue. In addition to possibly causing performance issues, this very messy approach to securing an application is very difficult to maintain and is not recommended.

Why It Matters

Some architects and security administrators who have been around from the beginning of the operating system remember that authorization lists caused performance issues when they were first introduced (for example). But that issue has LONG since been resolved. Back in the day, adopted authority was also a slow check. But again, that issue was addressed MANY years ago. The rule is that, if you can easily maintain a security scheme, it should not cause a performance issue. You should feel free to use the IBM i security features available to you without fear of causing a performance problem.

How It Works

Here are some of the enhancements that have been made to the authority checking algorithm throughout the life of the operating system:

- Authority caches: -- For each user, an authority cache containing 32 private authorities and another containing 32 private authorities to authorization lists exist in memory. If a user is consistently accessing the same object or objects secured with the same authorization list, this is a very fast check. -- The authority caches are internal objects - not viewable or modifiable by a user program. -- If a user's authority is changed, their authority cache is also updated.
- Adopted authority check: -- If the program running adopts authority and the program owner is the same as the object being accessed, the check occurs at the beginning of the algorithm and is a very fast check
- Authority Performance indicators stored in every object's header: -- One performance indicator was added to indicate that no private authorities exist for the object and no authorization list has been assigned. This allows the authority checking algorithm to jump straight to the *PUBLIC authority value which is also stored in the header of the object, making this a very fast check done at the beginning of the algorithm. --Another performance indicator was added to indicate that, while there are private authorities granted to the object, none are more restrictive than the object's *PUBLIC authority setting. (For example, users may have been granted *CHANGE authority to the object but *PUBLIC is set to *USE. In this case the indicator would be on because *CHANGE is not more restrictive than *USE.) If *PUBLIC authority is sufficient for the access being attempted, this is again a fast check done at the beginning of the algorithm.
- Primary group authority: -- If the object has primary group authority, the check for sufficiency is made at the beginning of the algorithm and is a fast check.
- Algorithm optimization: --Throughout the years, the algorithm itself has been performance optimized.

Details of how the system checks authority are documented in Chapter 5 of the IBM i Security Reference manual.

Key Takeaways

- An easily maintained security scheme should not cause a performance issue.
- Use the IBM i security features available (authorization lists, adopted authority, group profiles, etc) without the fear of causing a performance issue.

Which is Better - Group Profiles or Authorization Lists?

What It Is

The simple answer to this question is that one is not better than the other. Both are techniques provided by the operating system to aide in the management of IBM i Security.

How It Works

Group profiles are meant to help Administrators manage users that require the same authorities. Special authorities granted to the group filter down to its members. Likewise, private authorities granted to the group filter down to the individual user (assuming that the user doesn't also have a private authority to the object.) Group profiles help Administrators implement Role-Based Access Control (RBAC).

> **Note:** Anything a group owns, all members also own, which means group members have *ALL authority to anything owned by the group. This means that an application owner should NOT be a group profile. Doing so violates the principle of least privilege access because application users should not have *ALL authority to application objects. Today's IBM i functionality makes exploitation of this configuration incredibly easy.

There are a number of SQL services that can be used to review group profile and authorization list detail. For more information, see Security Services.

The following example shows how the **QSYS2.user_info** and **QSYS2.group_profile_entries** services can be combined to yield comprehensive answers to important security questions.

```
--
-- Which users have *ALLOBJ authority either directly
-- or via a Group or Supplemental profile?
--
SELECT authorization_name, status, no_password_indicator, previous_signon, text_description
  FROM qsys2.user_info
  WHERE special_authorities LIKE '%*ALLOBJ%' OR authorization_name IN (SELECT user_profile_name
            FROM qsys2.group_profile_entries
            WHERE group_profile_name IN (SELECT authorization_name
                  FROM qsys2.user_info
                  WHERE special_authorities LIKE '%*ALLOBJ%'))
  ORDER BY authorization_name;
```

Authorization lists are available to help Administrators manage a set of objects to which users (or groups) need the same level of access. For example, an authorization list may be assigned to some of an application's database files. These files contain data that is updated by a service account and read by a group of analysts. The authorization list would contain a private authority *CHANGE granted to the service account and *OBJOPR & *READ granted to the analyst group.

Which method you use - group profiles or authorization lists - depends on what's being managed - users or object authorities. As you can see from the authorization list example, groups and authorization lists are often used together. That is, groups are often granted authority to an authorization list.

Key Takeaways

- Neither a group nor an authorization list is 'better'
- Groups are provided to help administer users requiring the same access and capabilities.
- Authorization lists are provided to help administer objects where users (or groups) need the same access to the set of objects.
- Groups are often granted authority to an authorization list.

What are Additional Access Considerations for IFS Objects?

What It Is

The IFS - specifically objects in the root ('/') and '/QOpenSys' file systems - are based on UNIX. In fact, the IFS was added to more easily enable the porting of UNIX applications to what was then iSeries. As such, some differences exist in how one needs to consider security for objects in the IFS from the traditional library-based objects. Note that everything described applies to both objects in the '/' and the 'QOpenSys' file systems.

Why It Matters

It's just as important to implement a 'deny by default' approach to application objects stored in the IFS as it is for application objects in libraries. Important pieces of applications (for example keystores, configuration files and images) are stored in directories in the IFS and steps need to be taken to secure them appropriately. This is especially true because the default access is wide open. In addition, objects available via a file share are at risk of being infected by malware - especially if not secured properly.

How It Works

Authorities

The first thing to understand is that the authorities on IFS objects use UNIX terminology - *R (Read), *W (Write) and *X (Execute). These map into what we typically think of as Data authorities for objects in libraries. But all objects - regardless of whether they reside in a library or a directory - also have object authorities. These are the authorities that allow us to delete the object, grant others authority etc. As a result, when configuring authority on an IFS object, two sets of authority must be specified - Data authorities (DTAAUT) and Object authorities (OBJAUT). Setting the DTAAUT to *EXCLUDE but forgetting to set the object authorities may leave it so users cannot read the object but they can delete it! The following figure shows how IFS authorities map into the traditional IBM i object authorities.

Figure. How IFS authorities map to traditional IBM i object authorities

Primary Group Authority

Primary group authority (PGP) was introduced at the same time as the IFS to accommodate the UNIX concept of 'group authority' - as in the triplet User/Group/Other. Primary group authority - as the name implies - can only be granted to a group profile. But contrary to the name, it does NOT have to be granted to the user's first group (sometimes referred to as a user's 'primary' group) for the user to obtain access from this authority. The group to whom this authority has been assigned simply has to be one of the user's groups - their first group or one of their supplemental groups. An object can have no groups with primary group authority or one - no more. This authority is stored in the header of an object so it's a slightly faster check than a group having a private authority to the object. Since it's stored in the object's header, it's save/restored with the objects unlike private authorities. Primary group authority is not limited to IFS objects. Library-based objects can also have a primary group authority granted to it.

Determining Application Object Authorities

In general, IFS objects inherit the authorities from the directory they're being created into. One exception is when performing a Copy to Stream file or Copy to Import File. By default, these actions will set the *PUBLIC and primary group authority to *EXCLUDE, set the owner's authority to DTAAUT(*RWX) OBJAUT(*NONE) and ignore (not copy) any authorization list or private authorities granted to the directory. When performing a copy to stream file or import file, be sure to specify that the authorities are to come from the directory the file is being created into.

As the system ships, both '/' and '/QOpenSys' *PUBLIC authority is DTAAUT(*RWX) OBJAUT(*ALL). Therefore, if no action is taken, objects will inherit this wide-open access. Obviously, this will not meet the requirement of 'deny by default'! The easiest way to determine what authority is required is to use Authority Collection. If the application's objects are only going to be used by a service account, start the authority collection for the user, specifying to log

access to IFS objects. The same applies when creating a profile to act as an AI agent. For example: STRAUTCOL USRPRF(SERVICE1) INCFSOBJ(*DIR *STMF)

After examining the results of the authority collection, you may determine that it's appropriate for the service account or AI agent to own the application's IFS objects. This is typically true if the task is to delete and recreate or replace a file. But if the profile is simply using or updating ,the objects, it would be better (more secure) if the application objects were owned by a different profile and the service account or AI agent were authorized with only the authority required for the task.

If the application objects are going to be used by multiple users, configure authority collection on the IFS objects being accessed to see what authority is required then start the collection For example: CHGAUTCOL OBJ('/your-application-directory') AUTCOLVAL(*OBJINF) SUBTREE(*ALL) STRAUTCOL TYPE(*OBJAUTCOL) Then, have the objects owned by the application owner profile, secure the directory with an authorization list and authorize the application users (or a group profile) to the list with sufficient authority to perform their tasks as indicated through the results of the authority collection. Alternatively, the user's group could be granted primary group authority to the directory.

Once you've authorized the service account, AI agent or application users, don't forget to set *PUBLIC authority to DTAAUT(*EXCLUDE) OBJAUT(*NONE).

It's important to note that adopted authority is not available as a source of authority. Users must have sufficient authority on their own, through their group(s) or via *PUBLIC authority. The same authority checking algorithm that's run for objects in a library is run for objects in the IFS, but it will never check for adopted authority in the case of an IFS object.

File Shares

A file share that's defined on IBM i allows users to map a drive, making what is shared, another directory on their device. While handy, this presents a huge vulnerability should the user's device become infected with malware (including ransomware.) Applications requiring users to map to a file share need to define the file share as close as possible to the data needing to be shared to limit the damage should the user's device become infected. In addition, if users only need to read or download the data, the file share should be defined as read-only, not read-write.

Key Takeaways

- The IFS uses different names for data authorities
- Adopted authority is not available as a source of authority
- File shares should be created on the directory closest to the data to limit what will be affected should someone connect via a mapped drive get infected with malware.

What are Validation Lists and How Can They Be Used?

What It Is

A **validation list (*VLDL)** is an IBM i object that allows for storage of encrypted authentication data. Their development came when IBM i (then iSeries) was first being used as a web server. Web applications needed to authenticate users but didn't want to require the application users to have an actual iSeries user profile (*USRPRF) object. A method to securely store the authentication information was required ... thus validation lists were born. You may still see the term 'internet users' when reading documentation about implementing basic authentication for web applications. While basic authentication has largely been deprecated, validation lists are increasingly being used to store a password that is needed to make a connection to another server where you don't want to store the password in cleartext or attempt to store then encrypt/decrypt the password yourself. In practice, you can store any piece of information you'd prefer to store encrypted (such as API keys) in a validation list entry.

Why It Matters

Authentication information is often stored by an application in an unsecure manner. Using a validation list removes this burden from application developers.

How It Works

Validation lists are created using the Create Validation List (CRTVLDL) command. They are populated with user authentication information using the Validation Lists APIs or the systools.add_validation_list_entry() scalar function. When adding a validation list entry, you specify the user identifier (up to 100 characters), the information to be encrypted (up to 600 characters) and optionally, up to 1000 characters of freeform (cleartext) information. When you add an entry using an API, you can also specify whether the information to be encrypted can be decrypted when the validation list entry is added. (Using the SQL scalar function defaults to adding a password which is not decryptable.)

When the password is not decryptable, the intention is to use the entry for authentication. Meaning, you pass in the user information and password, the password is encrypted using same method used when the entry was added and the results compared. If they are the same, the results will return an indication as to whether they match - in other words, authentication is successful or not. Validation list entries are verified using QsyVerifyValidationLstEntry().

One consideration when using a validation list for user authentication - you must manage the entries yourself. For example, you must manage the enforcement of taking the appropriate action for invalid password attempts, enforcing password changes and defining and enforcing password composition rules, etc. You must also manage the clean up of entries. For example, the removal of entries due to inactivity. Attributes regarding the entry such as Creation, Last used and Encrypted data change dates as well as the number of invalid password attempts are kept with each entry and are returned using the Find Validation List Entry (QSYFDVLE) API. Entries can be removed using the systools.remove_validation_list_entry scalar function. And a reminder - Basic authentication for web applications has been deprecated as a valid means of authentication.

When the stored password is decryptable, you can retrieve the cleartext password just before using it on a connection. Use the Find Validation List Entry (QSYFDVLE) API to retrieve the information (which will include the cleartext version of the encrypted value).

> **Note:** What is stored in the password field is not limited to an actual password. For example, if an API key is shorter than 600 characters, you can add an identifier for the key in the user identification field and the API key in the field to be encrypted, then retrieve it prior to using. Bottom line is that if you have something that you want stored encrypted that you later need to have decrypted and presented in clear text and it's less than 600 characters, you can use a validation list.

Documentation for the systools.add_validation_list_entry scalar function can be found at ADD_VALIDATION_LIST_ENTRY scalar function.

For more information, see REMOVE_VALIDATION_LIST_ENTRY scalar function.

The validation list APIs can be found at Validation List APIs.

Key Takeaways

- Validation lists can be used to store authentication information when an IBM i user profile is not required.
- Validation lists can be used to store other information that needs to be encrypted at rest but decrypted prior to use such as a password for a service account used on a connection to another server.
- Basic authentication (implemented for authenticating web application users) has been deprecated.

What is Function Usage and How Might it be Used by an Application Running on IBM i?

What It Is

Function Usage provides the ability to control access when there's no object to secure. It's really easy to stop someone from accessing a database file, for example, but how do you prevent someone from taking an action? Enter Function Usage! The operating system has made extensive use of function usage to provide access controls for IBM Navigator for i, IBM i Access Client Solutions (ACS) features as well as functions performed within the operating system itself. For example, IBM has defined functions to control access to Navigator for i and additional functions to control features within Navigator for i. This way, if an administrator needs to grant access to Navigator for i for operators to manage jobs, they can then restrict access to the Navigator for i features which fall outside of their job function - Security, for example.

Why It Matters

A basic tenet of security is to avoid granting excess authority. With function usage, the security administators have a granular security control to leverage. Additionally, function usage is not only easy to configure, but its also auditable. Figure 1 shows how the audit journal Generic Record (GR) entry type contains evidence of when function usage configurations were changed, when a user passed a function usage check, and when a user was denied access to a specific function usage.

```
--
-- Audit todays function usage activity
--
SELECT entry_type_detail, user_profile_name, function_name, function_registration_operation
  FROM TABLE (
      systools.audit_journal_gr(starting_timestamp => CURRENT DATE)
    ) a
  WHERE entry_type = 'F'
  ORDER BY entry_timestamp DESC;
```

ENTRY_TYPE_DETAIL	USER_PROFILE_NAME	FUNCTION_NAME	FUNCTION_REGISTRATION_OPERATION
Function registration operations	JOEUSER	QIBM_DB_SQLADM	USAGE FAILURE
Function registration operations	JOEUSER	QIBM_DB_SQLADM	USAGE FAILURE
Function registration operations	JOEUSER	QIBM_DB_SQLADM	USAGE FAILURE
Function registration operations	SCOTTF	QIBM_DB_SECADM	CHECK USAGE
Function registration operations	SCOTTF	QIBM_DB_SECADM	CHECK USAGE
Function registration operations	SCOTTF	QIBM_DB_SECADM	CHECK USAGE
Function registration operations	SCOTTF	QIBM_DB_ZDA	CHECK USAGE
Function registration operations	SCOTTF	QIBM_DB_SECADM	CHECK USAGE
Function registration operations	SCOTTF	QIBM_DB_SECADM	CHECK USAGE
Function registration operations	DAWNM	QIBM_DB_SECADM	CHECK USAGE
Function registration operations	DAWNM	QIBM_DB_SECADM	CHECK USAGE
Function registration operations	DAWNM	QIBM_DB_SECADM	CHECK USAGE
Function registration operations	JOEUSER	QIBM_DB_SQLADM	CHANGE USAGE

Figure. Function usage and the systools.audit_journal_gr() table function

How It Works

Function Usage is not limited to use by the operating system. The Register/De-register APIs (QSYRGFN/ QSYDRGFN) allow you to add/remove your own functions to/from the system's registration facility. For example, perhaps you want to control who can add new accounts to your application. You can define a function and then check access prior to allowing access to that part of your application.

When defining your application's function(s), two of the settings are security-related - first, the Default usage setting (think *PUBLIC authority). Unfortunately, it defaults to allow access to anyone attempting to use the function. Obviously, for a 'deny by default' model, you'll want to change this to deny access by default. Then, the individuals and/or group profiles needing to perform the function can be granted access. Another setting is the *ALLOBJ indicator which determines whether having *ALLOBJ allows access. Again the default is that users with *ALLOBJ are allowed

to use the function. If this is a highly sensitive function, you may not want to allow all profiles with *ALLOBJ to perform it. If that's the case, set the default so that *ALLOBJ does not provide automatic access. In other words, users will have to be specifically allowed access to the function.

Once registered, your function will be displayed in and can be managed using the Security→Function usage section of Navigator for i as well as the Work with Registration Information (WRKREGINF) command. The SQL Service qsys2.function_info lists all functions along with their attributes (including the default usage setting and *ALLOBJ indicator.), while qsys2.function_usage lists each function along with the users and/or group profiles who've been configured to be allowed or denied to use that function.

Within your application, prior to performing the function, check to see if the user is allowed to run it using the QSYCKUFU API or the SQL scalar function QSYS2.SQL_CHECK_FUNCTION_USAGE. For example: `VALUES QSYS2.SQL_CHECK_FUNCTION_USAGE('YOUR_APP_NEW_ACCOUNTS');`.

For more information, see SQL_CHECK_FUNCTION_USAGE scalar function.

Key Takeaways

- Function usage provides the ability to control actions (vs access to an object).
- Applications can define their own function usage functions.
- Functions can be defined to deny access by default as well as deny use of *ALLOBJ to gain access.

When Should an Application Add Multi-factor Authentication (MFA) ?

What It Is

MFA - as the name implies - requires more than one authentication method prior to allowing the user to proceed with the action they're trying to take.

Why It Matters

Requiring an additional factor when a user signs on has become commonplace. But in addition to log in, some organizations have started to implement 'zero trust' and may require an application to re-authenticate the user before allowing access to the application. Or, some feature or function of an application may be so sensitive that an additional authentication is required (for example, the de-cryption of sensitive PII (Personally Identifiable Information.))

How It Works

IBM i Security vendors have MFA offerings for the IBM i. Additionally, with IBM i 7.6, the IBM i includes integrated MFA capabilities and a new Universal Authentication Exit Point (QIBM_QSY_AUTH).

For more information on implementing MFA on IBM i, see to the IBM i manual Security Multi-factor Authentication (MFA).

What are the Options for Encrypting Data at Rest in a Table?

What It Is

Field Procedures (FIELDPROC) is a Db2 for i data-centric technology that allows you to encrypt data at the column level without having to rearchitect the application. Prior to this feature being added in IBM i 7.1, encrypting data in a column involved significant application rework because encrypted values are most often larger than the data in cleartext. Files were usually modified to hold the larger field. Therefore, any program using the file had to be re-compiled. With FIELDPROC, the operating system takes care of storing the encrypted value internally. Field sizes aren't increased so no program has to be recompiled. However, IBM does not provide the program that does the actual encryption / decryption. That's the job of the field procedure exit routine - a user-written program.

Why It Matters

Various laws and regulations require that data at rest (vs data being transmitted) be encrypted. For example, the Payment Card Industry's Data Security Standard (PCI DSS) requires that cardholder data be encrypted. Most IBM i organizations implement FIELDPROC to accomplish this requirement. When encryption is required for regulatory compliance (vs encrypting the data for privacy) it's important that an approved algorithm be implemented. In the past, developers attempting to be 'clever' came up with their own encryption algorithms. Home grown algorithms will NOT meet regulatory requirements. Approved algorithms have gone through rigorous testing and review by cryptographers. It is not worth your time attempting to create your own algorithm. It's important that you use the strongest approved algorithm available at the time of implementation. As computers become faster, older, weaker algorithms are being deprecated because they're too easily broken.

In addition, some organizations attempt to pass compliance requirements for encrypting data at rest by utilizing full disk encryption and data masking. The Payment Card Industry's Data Security Standard (PCI DSS) has clarified that this approach does not meet compliance requirements. See PCI DSS Section 3.5.1.2 for further details.

How It Works

Field procedures (FIELDPROC) is implemented as a user-written program. The program is assigned to the FIELDPROC attribute of the column whose data is to be encrypted using the CREATE TABLE or ALTER TABLE (SQL) statements. This program - known as a field procedure exit routine - must be an ILE program, cannot contain SQL and cannot be a service program (*SRVPGM). At a high level, the program is called when the field is being written allowing for the encryption of the data prior to writing it to disk and called when the field is read allowing for the decryption of the data. The following figure shows the flow.

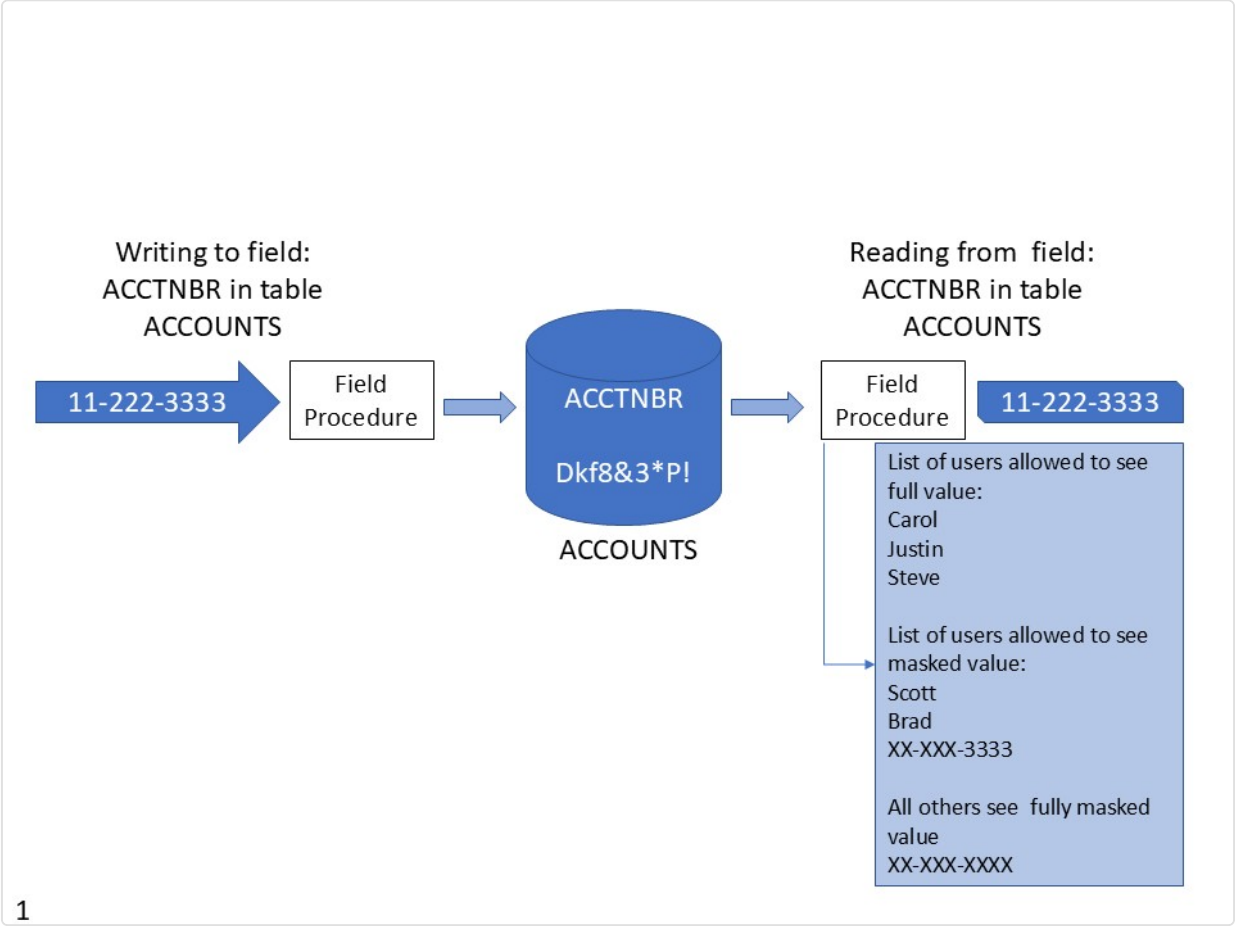


Figure. Field Procedure Example - showing encryption on write and decryption on read

For detailed information about the operations that cause the field procedure exit routine to be invoked, see Using field procedures to provide column level encryption.

Several IBM i vendors have written products that implement field level encryption using FIELDPROC. Rather than blindly decrypting the data when their exit program is called, they provide logic and a user interface that allows the definition of which users can see the fully decrypted value, the value in a masked format or a fully masked value. The user interface also provides for the management of encryption keys. You may want to evaluate the vendor solutions rather than developing your own FIELDPROC solution.

Key Takeaways

- Disk level encryption is not sufficient to satisfy the legal requirement for encrypting data at rest.
- Field procedures are the system-supplied method provided to satisfy encryption-at-rest requirements.
- You must provide your own encryption / decryption routines.

How Do I Determine if the Application's Current Security Schema is Appropriate?

What It Is

In striving to achieve 'secure by design' and a 'deny by default' application security scheme, the assumption is that all objects will be set to have ***PUBLIC - *EXCLUDE**. How then, are you supposed to know whether an end user has sufficient authority to perform their tasks. Or, how do you know how much additional authority is required for administrative tasks. The answer? A feature in the operating system called Authority Collection.

Introduced in IBM i 7.3, you can start authority collection on a user and determine what authority the operating system requires to every action the user takes along with its current authority to the object and the source of that authority. An IBM i 7.4 enhancement to authority collection allows you to collect information about who is accessing an object, what authority is required by the operating system, the user's current authority to the object and the source.

Why It Matters

Authority collection takes the guesswork out of what authority is required for an application to run successfully (that is, without security errors). It eliminates the tendency to over-authorize (that is, grant everyone *ALLOBJ special authority) to ensure security is never an issue.

How It Works

Access Authority Collection under the Security section of Navigator for i to configure Authority Collection on either an object or a user. Or, if you prefer to use commands, to start the collection on a user run the Start Authority Collection (STRAUTCOL) command. To start the collection on an object, run the Change Authority Collection (CHGAUTCOL) command to configure the collection options, then run STRAUTCOL to ensure the collection has been started.

After letting the collection run for a bit (length of time depends on the situation for which you're collecting information) you'll view the information via an SQL service. The information collected is kept in an internal object and the only way to analyze it is via an SQL view. Use one of the following views:

- QSYS2.authority_collection to view user-based authority collection information

- QSYS2.authority_collection_object to view information about the use of objects in libraries

- QSYS2.authority_collection_IFS to view information about IFS object usage

Examples of using each of these can be found in ACS - Run SQL Scripts - Insert from Examples. Search Security - Authority Collection for the list, as shown in figure 1. While you can also view the collection using Navigator for i, using the SQL Service provides much more opportunity to customize the result making it easier to determine what authorities are required.

ACS Authority Collection SQL Examples

Figure. ACS's Authority Collection SQL examples

> **Note:** the one downside to Authority Collection is that, it does not indicate when a special authority is required. For example, authority collection will tell you that you need *USE and *OBJEXIST authority to delete a user profile but it doesn't tell you that you also need *SECADM special authority.

Details about Authority Collection can be found in Chapter 10 of the IBM i Security Reference Manual. For practical examples, see Carol Woodbury's book, Mastering IBM i Security.

Key Takeaways

- Configure Authority Collection for a user to determine what authority they need to access objects
- Configure Authority Collection on an object to determine what authority users need to access the object (either a library-based object or an object in the IFS)
- Authority Collection does not indicate when a special authority is required

How do I Ensure I Avoid Creating Unintended Exposures?

What It Is

You may not realize the exposure some security settings or development practices create. Here are some of the exposures that are prevalent. This is not meant to be a comprehensive list. But, hopefully, thinking through these will help you identify and prevent others you may have previously (inadvertently) created.

How It Works

Not checking or having a way to quickly resolve vulnerabilities in Open Source

- If you use Open Source in your application, you must have a process in place to keep it current to avoid your application being exploited by bad actors looking for that vulnerability.

Not making library qualified calls

- Not explicitly specifying the object you're accessing or program or command you're calling exposes applications to executing trojan horse programs (a program with the same name as an application program but placed in a library higher in the library list) or calling a different version of a command than what was intended. These programs often take advantage of an application's adopted authority or an application's requirement that a powerful profile run the application to gain power and perform nefarious acts. Running a different command can cause an application to perform unintended tasks or run a command in a way not expected by the application.

Setting the *PUBLIC authority of libraries to *CHANGE or *ALL

- Libraries with *PUBLIC authority *CHANGE or *ALL allow the addition of objects into that library. Application libraries are often added to the library list - often in the system portion (QSYSLIBL). Allowing anyone to add objects (especially *PGM objects) opens organizations to the threat of having a Trojan horse program or command added to the library. Couple this with applications not making library qualified calls and the threat of bad actors - including insiders - taking advantage is very real.

Adding QGPL to the user or system portion of the library list. IBM ships this library with *PUBLIC(*CHANGE) allowing it to be used as part of an attack which exploits non-library qualified object references.

Setting the *PUBLIC authority of database files to *CHANGE or *ALL

- The IBM i default setting for *PUBLIC authority is *CHANGE. That's why many objects - including tables - have *CHANGE for their *PUBLIC authority. This unfortunate default and worse, setting *PUBLIC authority to *ALL, causes huge vulnerabilities for organizations' data. *PUBLIC authority of *CHANGE or *ALL provides end users with the authority to do significant damage to application data including modifying it for nefarious reasons (such as AI dataset poisoning.) Use the search word 'Inject' in Access Client Solutions' Run SQL Scripts' Edit→Examples→Insert from Examples for the Security-Db2 for i SQL examples to run to list your vulnerable application tables.

Unnecessarily adopting a powerful profile

- A misconception exists that when a program (or other object that adopts) is configured to adopt its owner's authority, that the owner must have *ALLOBJ special authority. No such requirement exists! It's best practice to have the application owner have no special authorities unless it's required to complete the underlying operating system task. Even then, the application can be owned by a non-powerful profile and adopt its authority. Then, if it runs an IBM i function requiring power, the task can be isolated into its own program that is owned by a powerful profile, performs the function and returns. Adopting a powerful profile throughout the application opens up the possibility of adopted authority being exploited - either via a command line or inserting a trojan horse program that takes advantage of an application making non-library qualified calls and/or a library in the library list that is *PUBLIC *CHANGE or *ALL.

Propagating adopted authority to a command line

- Adopted authority is available to a process for as long as the program adopting is in the call stack. When calling a program that should only use the caller's authority - such as when providing a command line interface - separate that call or feature into a program of its own and create it with the attribute of Use adopted authority *NO. This causes the program to ignore any adopted authority that is currently in the call stack. The default is to use all adopted authority available. Creating (or changing) the program to ignore adopted authority of programs currently in the call stack will help you avoid inadvertently propagating adopted authority to features where it shouldn't be available (e.g., a command line.)
    - For example, CHGPGM APP_LIB/CMD_PGM USEADPAUT(*NO)
    - For more information on adopted authority considerations see IBMI-SEC-PROCEDURE-193a.

Making an application owner profile a group profile for application users

- Object owners - by default - have *ALL authority to their owned objects. When a group owns an object, all members also have *ALL authority to the group's owned objects. When an application user is a member of the profile that owns the application, they have *ALL authority to the entire application - including tables (database files). Given all of the ways data can be accessed without using application interfaces, this authority schema creates huge vulnerabilities for organizations. While this method was fine when users were easily confined to a 5250 emulation session, it is no longer acceptable given today's technology.
    - For more considerations on application profiles see IBMI-SEC-PRDOCEDURE-193.

Requiring a read/write share to root ('/')

- Requiring an organization to create a read/write share to root is literally exposing the entire system (including / QSYS.lib) to exploitation from malware including ransomware. Shares should be defined directly at the point of the object being shared. For example, a share should be defined at /application/company1/sales/receipts to share the contents of that path. It should not be defined at /application. In addition, if users are only downloading information,

the share should be defined as a read-only share - not read/write (adhering to the concept of least privilege access.)

Using /tmp or QGPL

- Using the /tmp directory or the QGPL library as a means of temporarily storing application information may be tempting because they are both shipped with every system and have a *PUBLIC authority setting allowing objects to be added. The problem is, many objects don't get cleaned up... they're written and never deleted. Often, the data left behind is sensitive or confidential leaving it vulnerable to being read (at best) and misused (at worst.) We highly recommend that you create your own container - be it a directory or library - which is owned by your application profile and secure it - for libraries *PUBLIC *EXCLUDE and directories *PUBLIC DTAAUT(*EXCLUDE) OBJAUT(*NONE) rather than using these publicly available containers.

Qualification of Unqualified objects within SQL statements

When an SQL statement is executed, and one or more objects within the SQL statement are unqualified, the database has a figure out WHERE to look to find the object. If you don't strictly control your choices, you might end up using rules different from what was intended, and inadvertently open the door to an attack vector.

Make no mistake about it, this is a complex topic. **The simple path forward is to always schema qualify references to SQL objects.**

For those who decide to use unqualified object references, here is where to look to understand the rules for object qualification. To correctly discern the expected behavior, you will need to understand static vs. dynamic statements, run-time authorization ID, default schema, and more. Further, the rules are different depending upon the type of SQL object that has the unqualified reference.

- Tables, Views, and other objects. For more information, see Unqualified alias, constraint, external program, index, mask, nodegroup, package, permission, sequence, table, trigger, view, and XSR object names.
- Functions, Procedures, and other objects. For more information, see Unqualified function, procedure, specific name, type, and variables.

Key Takeaways

- Certain coding techniques such as not specifying the library when accessing objects or calling programs or commands can inadvertently cause exposures.
- Leaving *PUBLIC authority set to the default of *CHANGE can cause exposures to data as well as providing others to insert a Trojan horse.
- This is not a comprehensive list!
- Schema qualifying your SQL objects is the simpler and safer option

Character Encoding and Globalization

CCSID Fundamentals

What is CCSID and why does it matter?

A **Coded Character Set Identifier (CCSID)** is a numeric identifier that describes how character data is encoded and interpreted in a system. A CCSID uniquely defines:

- The **character set** (which characters exist)
- The **encoding scheme** (how characters are represented in bytes)
- The **code page** (the numeric mapping of characters)

In practice, a CCSID tells the system **how to interpret the binary value stored in memory or a database field**.

For example, the hexadecimal value `x'0B'` may represent different characters depending on the CCSID. Without knowing the CCSID, the system cannot determine the intended character.

CCSIDs are used throughout IBM i:

- Database fields
- Program variables
- Source files
- Job environments
- Communication interfaces

When data moves between environments with different CCSIDs, **automatic character conversion** occurs. The system converts characters from the source CCSID to the target CCSID.

However, if a character exists in one CCSID but **not in the destination character set**, the conversion can result in a **substitution character**, causing permanent data loss.

However, there is a special CCSID value that **disables automatic character conversion**: **65535**. Using this value is generally discouraged. Instead, it is recommended to configure the system value **QCCSID** with a valid default CCSID.

The reason is straightforward. When character conversion is required, the system must know both the **source** and **target** character encoding (character set and code page) in order to perform the conversion correctly. If the CCSID is set to **65535**, this information is not available.

In such cases, when data must be transferred or interpreted using another encoding scheme, the system may attempt to **infer the CCSID**. Typically, it relies on the **language identifier (language ID)** to determine a likely default CCSID. Because this is only an approximation, it can lead to incorrect character interpretation.

Important: Maintaining **data integrity** depends on properly identifying the character encoding. Assigning an appropriate **CCSID value** ensures that characters are interpreted and converted accurately across environments.

Job and Job Default CCSID

A job attribute, job default CCSID (DFTCCSID), is created for jobs with a CCSID of `65535`. The DFTCCSID value is used by a system code when a CCSID other than `65535` is needed. The value of DFTCCSID is determined as follows: If the job CCSID is not `65535`, the DFTCCSID equals the job CCSID. If the job CCSID is `65535`, the DFTCCSID value is based on an appropriate value derived from the job language identifier (LANGID).

And at its level, the job CCSID can be based on cascade on the *USRPRF that can be based on *SYSVAL.

To avoid

Don't leave your system QCCSID with the value `65535`. Always set it to an explicit value according to your primarily language setting. Otherwise, any time a conversion is required and the source or target string CCSID is not explicitly defined, the deducted from or to CCSID can differ from one job or user to an other.

IBM Documentation

- Job default coded character set identifier
- Working with Unicode

What are character sets and code pages?

A **character set** is the collection of characters available for representing written text. A character set might include:

- Letters
- Digits
- Punctuation
- Accented characters

- Symbols

Examples include:

- Latin character sets for Western languages
- Cyrillic for Russian
- Japanese character sets
- Chinese character sets

However, defining a set of characters is not enough. Computers must also know **where each character is located in the encoding table**.

A **code page** defines the mapping between characters and their numeric representation. For example:

Character	Code Page Value
A	x'41'
B	x'42'

Different code pages may contain the **same characters but assign different numeric values** to them.

The CCSID identifies **both the character set and its specific code page mapping**, allowing systems to interpret stored character data correctly.

Invariant character set

Within any character set, there is a subset of characters that remains consistent across most character sets and code pages. This subset is known as the **invariant character set**, meaning these characters retain the same hexadecimal value and representation across different encodings.

The invariant set represents characters that are commonly supported regardless of the specific CCSID or code page in use.

In the **EBCDIC encoding scheme**, the invariant characters include:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
0123456789
+<=>%&*"'(),_-./:;?
```

To avoid

Avoid using non-invariant character like $, @, # or any characters non contained in the above list ** Invariant character set ** for any identifier or names of any kind because the code points used to represent them vary depending on the CCSID of the string in which they are contained. If they are used, unpredictable results may occur, especially when moving from one system's CCSID to another. This is to avoid for any system objects naming or variables naming in RPG or SQL.

IBM Documentation

- Identifiers

Unicode Concepts

What is Unicode?

Unicode is a universal character encoding standard designed to represent characters from all writing systems in a single, consistent framework.

Older encoding systems were designed for specific languages and could represent only a limited number of characters. As systems became global, these limitations caused significant interoperability problems.

Unicode solves this by providing a **single global character set capable of representing characters from all languages**.

Common Unicode encoding formats include:

Encoding	Characteristics
UTF-8	Variable length (1–4 bytes), widely used on the web
UTF-16	Uses 2 or 4 bytes per character
UCS-2	Fixed 2-byte encoding used historically

On IBM i, common Unicode CCSIDs include:

- **1208** → UTF-8
- **1200** → UTF-16
- **13488** → UCS-2

Unicode is important when working with **multilingual applications**, because it avoids the data loss that can occur when converting between incompatible character sets. Consequently Unicode is particularly important as it is the **"language of the internet"**.

Takeaway

Next to multilingual applications, technologies like REST, JSON and XML are typically found in UNICODE form. Therefore it is advisable to set them in Unicode all way long within your application processes. Any time a Unicode JSON or XML string is moved to a character string (and vice-versa) your application will degrade performance and this could be avoided.

IBM Documentation

- Working with Unicode

What are SBCS, DBCS, and MBCS?

Character encodings define how textual data is represented in bytes. The number of bytes used per character can vary based on the encoding scheme, which leads to different types of character sets.

SBCS – Single-Byte Character Set

SBCS encodings use exactly **one byte per character**, allowing up to 256 unique characters.

They are typically used for languages with relatively small character sets such as:

- English
- French
- Spanish

- German

Examples include many **EBCDIC and ASCII code pages**.

DBCS – Double-Byte Character Set

DBCS encodings use **two bytes per character**, allowing thousands of characters.

They are required for languages with very large character sets, including:

- Japanese
- Chinese
- Korean

These encodings allow representation of complex ideographic writing systems.

MBCS – Multi-Byte Character Set

MBCS encodings allow **characters of varying byte lengths**. This flexibility enables support for a wide range of languages and symbols within a single encoding system.

A data stream can contain:

- Single-byte characters
- Double-byte characters

Special control codes, like **shift-out (SO)** and **shift-in (SI)** indicate when the encoding switches between character types.

Unicode encodings such as **UTF-8** are modern examples of multi-byte encoding systems.

IBM Documentation

- DBCS field data types
- SQL and DBCS

Data Integrity and Programming

How do I ensure data integrity across environments?

Maintaining character data integrity requires careful management of CCSIDs and character conversions.

When data moves between environments with different CCSIDs, the system performs a conversion. This conversion typically follows two steps:

1. Identify the **character represented in the source CCSID**
2. Locate the **equivalent character in the destination CCSID**

Problems occur when the destination character set **does not contain the character**.

In such cases, the system inserts a **substitution character**, which permanently replaces the original character.

For example:

```
Original character: ½
Target character set: does not contain ½
Result: replacement character
```

Once this happens, the original character cannot be recovered.

To reduce the risk of data loss:

- Use **Unicode** whenever possible
- Avoid unnecessary conversions
- Use invariant characters for identifiers
- Maintain consistent CCSIDs across applications
- Detect conversions that may cause substitutions

Unicode is often the safest solution because it contains characters from many languages.

How do I handle CCSID in programs?

Programs must be aware of the CCSIDs associated with character variables and data sources.

In RPG, all string data has a CCSID, including:

- Alphanumeric fields
- Graphic fields
- Unicode fields

Default behavior in RPG:

Data Type	Default CCSID
Alphanumeric	Job CCSID
Graphic	ignored unless enabled
UCS-2	CCSID 13488

Developers can override defaults using compiler directives:

```
ctl-opt CCSID(*CHAR:37);
```

Variables may also specify explicit CCSIDs:

```
dcl-s description char(100) ccsid(*UTF8);
```

The CCSID of database fields can be inspected using:

```
DSPFFD
```

Proper CCSID management ensures correct character interpretation during program execution.

Tips regarding identifiers in you programs

When developing applications, it is considered best practice to **use only invariant characters** for object names, field names, and variable identifiers. Doing so helps ensure that the application behaves consistently across systems that may use different CCSIDs. For more information, see "What are character sets and code pages?".

Using characters outside the invariant set can lead to compatibility problems when code or objects are moved between IBM i systems with different character encodings. For this reason, it is recommended to avoid characters

such as **$, @, or #** in object and variable names, since they are **not part of the invariant character set** and may be interpreted differently depending on the CCSID in use.

Takeaway

- Code CCSID(*EXACT) so that RPG will handle CCSIDs correctly
- Code CCSIDCVT(*EXCP) so that conversion failures will cause an exception
- Code CCSIDCVT(*LIST), possibly temporarily, and study the CCSID conversions that are happening in your module
- Be sure to define UTF-8 fields long enough
- Use only invariant characters for variables naming

IBM Documentation

- RPG and Unicode

Unicode Implementation

How do I implement Unicode?

Unicode can be implemented in:

- Database tables
- Application variables
- Interfaces and APIs
- User interfaces

In RPG, Unicode variables are defined using specific data types or CCSIDs. There are three types of Unicode data:

- UCS-2: data type C with CCSID(13488)
- UTF-16: data type C with CCSID(1200)
- UTF-8: data type A with CCSID(*UTF8)

Examples:

```
dcl-s name varucs2(50);     // UCS-2
dcl-s text char(200) ccsid(*UTF8);
```

Conversions between traditional encodings and Unicode may occur automatically during assignment.

Example:

```
dcl-s name varchar(20) ccsid(*JOBRUN);
dcl-s unicodeName char(20) ccsid(*UTF8);

unicodeName = name;
```

RPG automatically converts the value to Unicode during the assignment.

In DDS or SQL, Unicode variables are defined using specific data types and CCSIDs. There are three types of Unicode data:

- Graphic data type with CCSID(13488)
- Graphic data type with CCSID(1200)
- Character data type with CCSID(1208)

Example:

```
CREATE TABLE QTEMP.MYFILE1 (
UNICODE1 GRAPHIC(20) CCSID 1200,
CHAR1 CHAR(20) CCSID 37);

CREATE TABLE QTEMP.MYFILE2 (
CHAR2 CHAR(20) CCSID 37,
UNICODE2 GRAPHIC(20) CCSID 1200);

CREATE TABLE QTEMP.MYFILE3 (
UNICODE3 CHAR(20) CCSID 1208,
CHAR3 CHAR(20) CCSID 37);

INSERT INTO QTEMP.MYFILE1 (UNICODE1, CHAR1)
VALUES ('hello', 'world');

INSERT INTO QTEMP.MYFILE2 (CHAR2, UNICODE2)
SELECT UNICODE1, CHAR1 FROM QTEMP.MYFILE1;

INSERT INTO QTEMP.MYFILE3 (UNICODE3, CHAR3)
SELECT CHAR2, UNICODE2 FROM QTEMP.MYFILE2;

SELECT * FROM QTEMP.MYFILE1;
SELECT * FROM QTEMP.MYFILE2;
SELECT * FROM QTEMP.MYFILE3;
```

SQL automatically converts the value to Unicode during the assignment.

Unicode implementation helps ensure that applications support **multiple languages without data loss**.

IBM Documentation

- Unicode on IBM i
- Db2 for i
- SQL tables
- CCSID (Coded Character Set Identifier) keyword for physical and logical files

How do I move to Unicode?

Migrating to Unicode involves both **database and application changes**.

Typical migration steps include:

1. Identifying fields that store textual data subject to be in Unicode
2. Converting existing data
3. Updating programs to handle Unicode variables
4. Testing

To change your existing database to Unicode:

1. Create a new version of the old-table to a new-table that used Unicode encoding.
2. Copy the old-table records to the new-table: With SQL, insert rows from old-table to the new-table (the conversion to Unicode will then occur).
3. Delete/rename the old-table and rename the new-table.

To change your existing RPG to Unicode:

1. Change variables to Unicode type as described in chapter ** How do I implement Unicode? **.

2. Check propagation:
    a. Check any potential issue for any Unicode fields or variables propagation into non-Unicode fields or variables, as they might result in data loss.
    b. Change any impacted non-Unicode fields or variables to Unicode and reiterate from point 2.

RPG provides tools for identifying potential problems, at compilation time:

```
CCSIDCVT(*LIST)
```

This option produces a report of CCSID conversions in a module. It instructs the compiler to add a **CCSID Conversion Summary** section to the compile listing. This section details all implicit and explicit CCSID conversions and indicates which conversions might result in data loss.

Another option, at runtime:

```
CCSIDCVT(*EXCP)
```

This causes an RPG exception (status code 452) at runtime if a CCSID conversion results in data loss, such as a substitution character being inserted.

Using these tools helps prevent silent data corruption during Unicode migration.

The best practice is to use the full options:

```
CCSIDCVT(*LIST:*EXCP)
```

Testing:

1. It is recommended to test the before and after behaviour for your tables or RPG changes.
    a. You can test manually or with an automated tool. For more information, see Testing & TDD.
    b. Run the new version against the old and check all I/O data are the same.
2. During and after migration, developers should detect conversions that may introduce substitution characters at runtime.
    a. In RPG: With the control specification **CCSIDCVT(*LIST:EXCP)* you will get an RPG exception (status code 452) at any data loss.
    b. In SQL: You will get an SQL exception (SQL State: 01517 or SQL Code: 335) at any data loss.

Configuration and Localization

How do I configure emulators for Unicode?

Terminal emulators must be properly configured to display Unicode characters correctly.

In a full stack of an application handling Unicode with an emulator we would have:

- Database Unicode-capable (cf. CCSID field attribute)
- Programs Unicode-capable (cf. CCSID variable attribute)
- Display-files Unicode-capable (cf. CCSID field attribute)
- Terminal emulator Unicode-capable (cf. example configuration bellow with IBM ACS)

Modern emulators often support UTF-8 sessions, allowing display of characters from multiple languages simultaneously.

How to configure Unicode with IBM ACS

Enable Unicode support within the emulator by going to: Main Menu → Communication → Configuration → Connection. You will then be able to select Unicode Options.

IBM Documentation

- IBM i Access Client Solutions

What are locale considerations?

A **locale** defines regional settings that influence how information is presented and interpreted.

A locale typically specifies:

- Language
- Country or region
- Character encoding
- Sorting rules
- Currency format
- Date and time format
- Decimal separators

Locale settings affect:

- Sorting and collation
- Numeric formatting
- Date formatting
- Language-specific behaviour

Correct locale configuration ensures that applications behave correctly for users in different regions.

IBM Documentation

- Globalization and localization

Source Control and Git for IBM i

Fundamentals of Source Control on IBM i

Why use Source Control on IBM i?

Introduction

Using source control on IBM i has shifted from a "nice-to-have" to an essential discipline for any organisation that develops or maintains applications on the Power platform.

As IBM i shops modernise, integrate with distributed systems and adopt DevOps practices, the need for structured version management becomes even more critical.

Source control provides a single, authoritative record of your application's evolution. On IBM i, where many organisations still store source in physical files inside libraries, the absence of versioning creates blind spots. Developers often overwrite members, lose historical context, or rely on informal backups.

A modern source control system replaces these fragile practices with a structured, auditable and collaborative workflow.

The most immediate benefit is traceability. Every change is recorded with a timestamp, author and explanation. This is invaluable when diagnosing production issues or understanding why a particular piece of logic exists. In traditional IBM i environments, developers may find a member that has been modified dozens of times with no record of who changed it or why.

With any source control tool, the entire history is preserved, allowing teams to step back through time, compare versions and recover previous states instantly.

Enabling Modern DevOps on IBM i

DevOps is no longer exclusive to distributed systems. IBM i can participate fully in automated build pipelines, continuous delivery and test automation, but only if source control is in place.

Tools such as Git, Jenkins, Azure DevOps and GitHub Actions integrate seamlessly with IBM i when the source is stored in a repository rather than in QSYS.

Once code is in source control, automated processes can compile objects, run unit tests, deploy to test environments and promote to production. This eliminates manual steps that are prone to error and ensures consistent, repeatable builds.

It also supports audit and compliance requirements by providing a clear record of what was deployed, when and by whom.

Protecting the Business from Knowledge Loss

Many IBM i applications have been running for decades. They contain critical business logic that is often poorly documented. Without source control, the only record of changes may be in the minds of long-serving developers. As these individuals retire, organisations risk losing essential knowledge.

Source control acts as a living documentation system. Commit messages and historical context provide insight into how the application has evolved.

This helps new developers understand the codebase more quickly and reduces the learning curve. It also supports cross-training and reduces the risk associated with single points of failure.

Supporting Modern Development Tools

The rise of VS Code, RDi and open-source tooling has made IBM i development more accessible and modern. These tools integrate naturally with Git and other SCM systems, enabling workflows that feel familiar to developers from other platforms.

This is particularly important for attracting new talent. Younger developers expect version control, code review and modern IDE features. By adopting source control, IBM i shops create an environment that aligns with contemporary development practices.

Enhancing Quality Through Code Review and Automation

Source control enables structured code review processes. Instead of informal peer checks or relying on senior developers to spot issues manually, teams can use pull requests to review changes before they are merged. This improves code quality, encourages knowledge sharing, and reduces defects.

Automation also becomes easier. Linters, syntax checkers, and static analysis tools can run automatically on each commit. This ensures consistent coding standards and catches issues early in the development cycle. For IBM i teams transitioning from traditional workflows, this represents a significant improvement in quality assurance.

Strengthening Security and Compliance

Many industries require strict audit trails for software changes. Without source control, proving compliance can be difficult.

Source control provides a history of changes, complete with author identification and timestamps. Security also improves because source control reduces the need for developers to have broad access to production libraries.

Instead, they work with local or sandboxed copies of the code and only approved changes are promoted through controlled processes.

Preparing IBM i for the Future

IBM i remains a powerful and reliable platform, but its long-term success depends on adopting modern development practices.

Source control is the foundation for everything that follows: DevOps, automation, cloud integration, hybrid workflows and modern tooling.

By moving source out of QSYS and into a repository, organisations unlock new possibilities and ensure their applications remain maintainable for decades to come.

Understanding Git Concepts and Workflow

What is Git & How Does it Work?

Introduction

Git is a distributed version control system created by Linus Torvalds in 2005.

It was originally built to manage the development of the Linux kernel.

At its core, Git tracks changes to files over time, allowing developers to record a history of their work in what are called commits, or snapshots.

Each commit acts like a snapshot of your project at a specific point in time, making it easy to see what changed, when it changed and who changed it.

This history is stored in a repository, which can live locally on your machine or remotely on a server.

One of Git's most powerful features is branching. A branch is essentially an independent line of development that lets you work on a new feature, bug fix or experiment without affecting the main codebase.

Once your work is ready, you can merge that branch back into the main branch, integrating your changes cleanly.

For more information, see the section 'What are Git Branching Strategies?'

This workflow makes it incredibly easy for teams to work on multiple changes simultaneously without stepping on each other's toes.

Git is also distributed, which means every developer gets a full copy of the entire project on their own PC/laptop.

This is a big deal because it means you can work completely offline, commit changes, browse history and create branches without needing a network connection or a central server.

It also adds resilience. If a central server goes down, every developer's local copy serves as a full backup.

Platforms like GitHub, GitLab and Bitbucket have built on top of Git to add collaborative features like pull requests, code reviews and issue tracking.

This makes Git the backbone of modern software development. Whether you're an individual developer or part of a large enterprise team, Git gives you the safety net to experiment freely, collaborate efficiently and maintain a clean, auditable history of your entire project.

Terminology

No introduction to Git is worth its weight if it doesn't list the terminology that Git uses.

Term	Description
Repository	A directory tracked by Git that contains your project files and the entire history of changes
Commit	A saved snapshot of your project at a specific point in time, identified by a unique SHA-1 hash
Branch	An independent line of development that diverges from the main codebase
Main/Master	The default primary branch of a repository where production-ready code typically lives
Clone	A full local copy of a remote repository, including its entire history
Fork	A personal copy of someone else's repository, typically used to propose changes or experiment independently
Staging Area	A preparation area where you place changes before committing them
Working Directory	A local folder on your PC where you edit files
Pull	Getting changes from a remote repository and merging them into your local branch
Fetch	Download changes from a remote repository without merging
Merge	Combining changes from one branch into another
Rebase	Moving or replaying commits from one branch onto another to create a cleaner history
Merge Conflict	A situation where two branches have conflicting changes to the same part of a file, requiring manual resolution
Head	A pointer that refers to the current commit or branch you are working on
Remote	A version of your repository hosted on a server, such as GitHub or GitLab
Stash	A temporary storage area for uncommitted changes so you can switch context without losing work
Diff	A comparison showing the differences between two commits, branches, or files
.gitignore	A file that tells Git which files or directories to leave untracked
Pull Request (PR)	A request to merge changes from one branch into another, typically reviewed by others before approval (This is a GitHub term).

How Does Git Work

When you start using Git, the first thing you do is initialise a repository with the command `git init` or clone (copy) an existing one with the `git clone` command. This creates a hidden `.git` folder in your project directory that stores

everything Git needs, your complete history, configuration, and metadata of your project. From this point on, Git watches the files in that directory and is aware of any changes you make, though it doesn't automatically record them. You have full control over what gets tracked and when.

Git uses a three-stage workflow to record changes: the working directory, the staging area, and the repository.

The figure below shows the different areas of a typical Git workflow.

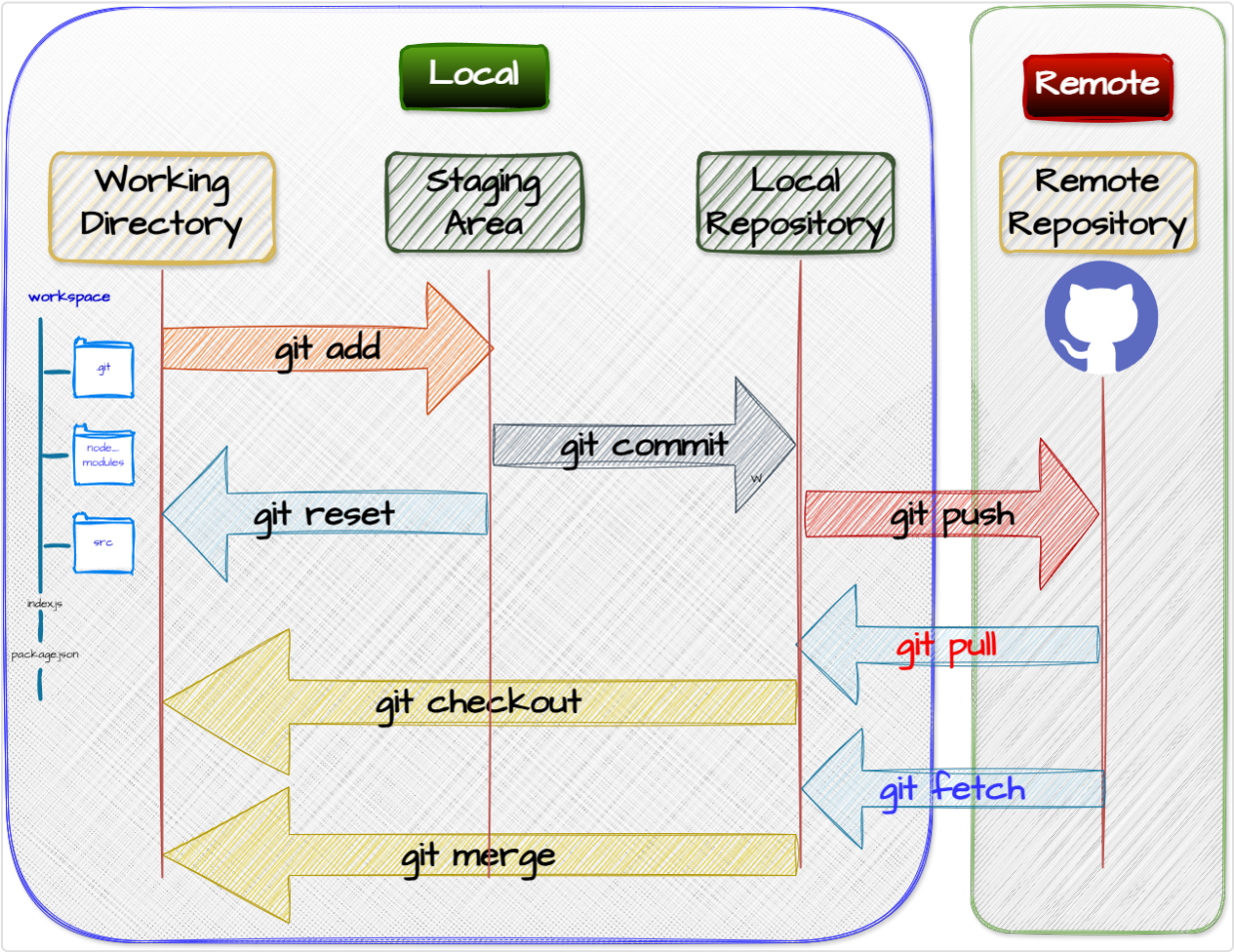


Figure. Git workflow showing working directory, staging area, and repository

Your working directory is simply where you edit files. When you're ready to record a change, you use `git add` to move those changes into the staging area, which acts like a preparation zone where you decide exactly what will be included in your next snapshot. Once you're satisfied with staged changes, run git commit. This permanently saves that snapshot to the repository, along with a message describing what you did. This two-step process gives you control over what gets recorded in each commit.

Under the covers, Git stores data in a simple way. Rather than saving the differences between files like some older version control systems, Git stores complete snapshots of your project at each commit.

To keep things efficient, if a file hasn't changed between commits, Git simply references the previous version rather than storing a duplicate.

Each commit is identified by a unique hash: a long string of characters generated from the contents of that commit. This is how Git keeps track of everything and ensures the integrity of your history. The `git log` command is used to show your history and commits.

The figure below shows an example of the `git log` command

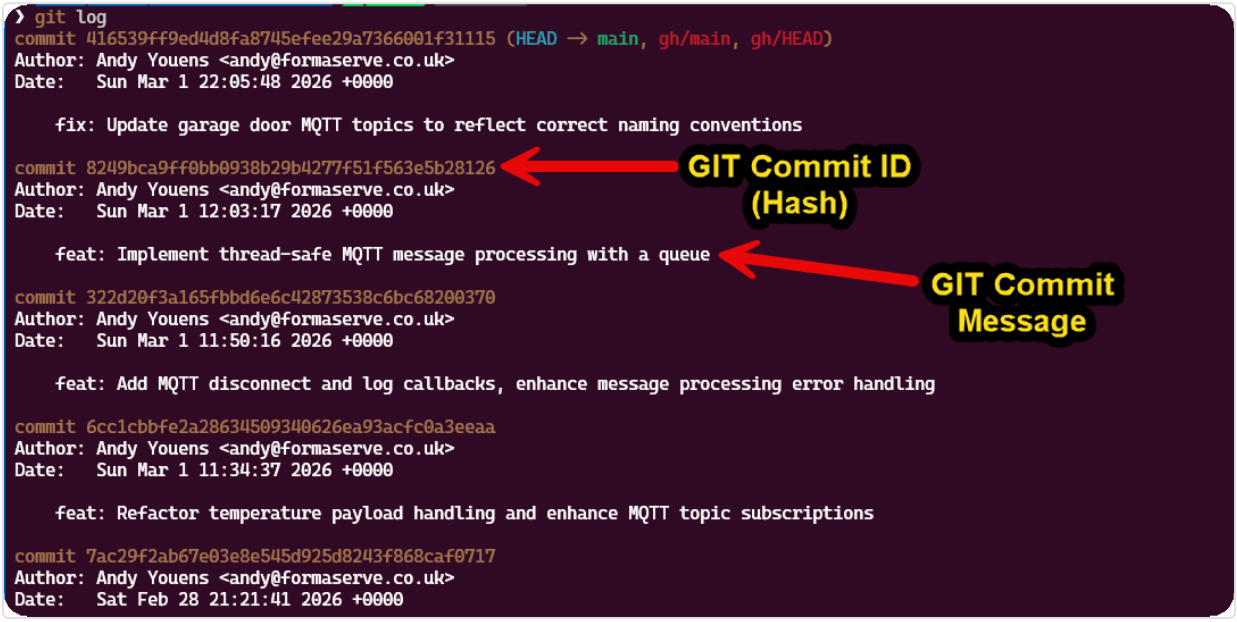


Figure. Example of git log command output

Collaboration in Git happens through remotes, which are versions of your repository hosted on a server.

When you run `git push`, you send your local commits up to the remote repository so others can access them.

When you run `git pull`, you retrieve changes others have made and merge them into your local copy.

If two people have edited the same part of a file in conflicting ways, Git will flag a merge conflict and ask you to resolve it manually before proceeding.

This combination of local control and remote synchronisation is what makes Git both flexible and powerful for teams of any size.

Managing Source Code in the IFS

How do I manage IFS based source?

Introduction

For decades, IBM i developers stored their source code in Source Physical Files (SPFs), members inside objects like QRPGLESRC or QCLLESRC sitting neatly in a library. That model served the platform well, but the modern development landscape has changed. Teams now want Git-based version control, IDE integration, open-source tooling, and continuous integration pipelines.

All of those things point to one place, the IFS!

Moving your source to the IFS does not mean abandoning traditional IBM i development. It means meeting it where the rest of the industry already is. This topic walks you through the key concepts, tools and day-to-day practices you need to manage IFS-based source effectively.

What Is the IFS and Why Use It for Source?

The IFS supports long file names, directory trees, stream files and standard Unix-style permissions. Unlike source members, which are fixed at 92 or 112 bytes per line, the IFS stream files are free-format and handle modern character encodings including UTF-8.

Storing source in the IFS unlocks several capabilities that are simply not available with source physical files e.g. native Git repositories, direct editing from VS Code via the Code for IBM i extension, integration with npm and other open-source package ecosystems and automated builds using tools like GNU Make or vendors own tools.

Organizing Your IFS Project Structure

A consistent directory layout makes projects easier to navigate and simplifies your build scripts.

A common convention is to create a project root under your application name and then organise source by type:

```
/projects/app/
  qrpglesrc/   - RPG source files (.rpgle, .sqlrpgle)
  qcllesrc/    - CL programs (.clle)
  qsqlsrc/     - SQL scripts (.sql)
  qcpysrc/     - Copybooks and /COPY members (.rpgleinc)
  .gitignore, README.md, Makefile or iproj.json
```

File extensions matter! The IBM i extension for VS Code and vendor tools use the extension to determine the object type and compiler to invoke.

Always use .rpgle for free-format RPG, .sqlrpgle for embedded SQL and .clle for CL and be consistent across your team.

Editing IFS Source

The most widely adopted editor for IFS-based IBM i development is Visual Studio Code (VS Code) with the Code for IBM i extension with the IBM i extension pack.

It connects to your IBM i over SSH, browses the IFS and lets you open, edit and save stream files directly.

Syntax highlighting, code completion via RPGLE language server support and integrated compile-and-view error messages are all available.

From the 5250 green screen, you can edit IFS files using the EDTF command (Edit File), though it lacks the features of a modern IDE. For quick one-off edits or scripted changes, QShell (QSH) and PASE give you access to vi, nano and sed directly from the command line.

Version Control with Git

Git is available natively on IBM i through the open-source package management feature of ACS (Access Client Solutions).

Once installed, you initialise a repository just as you would on any Linux system:

```
cd /projects/myapp
git init
git add .
git commit -m 'initial commit'
```

You can then push to any remote, GitHub, GitLab, Azure DevOps or an on-premises GitHub instance.

This gives your team branching, pull requests, code review and full history.

It also means you can trigger builds automatically when code is merged, using GitHub Actions or a self-hosted pipeline that SSH-calls your IBM i server.

Compiling from the IFS

IBM i compilers have supported IFS source as input for many years. In CL, you simply reference the stream file path with the SRCSTMF parameter rather than SRCFILE/SRCMBR:

```
CRTRPGMOD
MODULE(MYLIB/MYPGM) SRCSTMF('/projects/myapp/qrpglesrc/mypgm.rpgle')
```

Alternatively, a well-structured Makefile gives you fine-grained control and integrates naturally with CI pipelines.

Managing Encodings and Line Endings

One practical consideration when working with IFS source is character encoding.

IBM i EBCDIC and the ASCII/UTF-8 world of modern editors do not mix well without care.

Set your IFS stream files to CCSID 1208 (UTF-8) and configure VS Code and Git to use LF line endings on the server.

A .gitattributes file with text eol=lf prevents Windows-style CRLF endings from creeping in and causing compile errors.

Getting Started

Managing IFS-based source on IBM i is no longer niche.

It is the direction the platform is firmly heading and IBM tooling reflects that.

The practical steps to begin are straightforward:

> Install open-source tools via ACS
>
> Connect VS Code to your server

> Move a small project to the IFS
>
> Initialize a Git repository

From there, the broader ecosystem of modern DevOps tooling opens up.

The IFS simply gives your source code a better home.

Git Branching and Workflow Strategies

What are Git Branching Strategies?

What is a Branch?

In many older version control systems, branching involved making a physical copy of your entire project directory. It was slow, heavy and expensive.

In Git, a **branch** is incredibly lightweight.

To understand a branch, you first have to understand a **commit**. A commit in Git is a snapshot of your files at a specific moment in time. Each commit points to its parent, creating a chain.

A branch is simply a **movable pointer** to one of these commits in this chain. When you create a branch, Git doesn't copy your files, it just creates a new label (like a sticky note) and sticks it on a specific commit. When you make new commits while that branch is active, the pointer automatically moves forward to the latest commit, but you can move backwards if necessary.

Why Branching Matters

Without branches, development is a balancing act. If you’re working on a new feature and a critical bug appears in production, you would have to manually undo your half-finished feature code just to fix the issue.

Branching solves this by providing:

- **Isolation:** You can work on "Feature A" without "Feature B" interfering.
- **Safety:** You can experiment with new ideas. If it fails, you simply delete the branch. Your "Main" code remains untouched.
- **Current Development:** Multiple team members can work on different tasks simultaneously on the same code.
- Branching creates a **multi-track workflow**, so the 'bug fix' track and the 'new feature' track never collide.

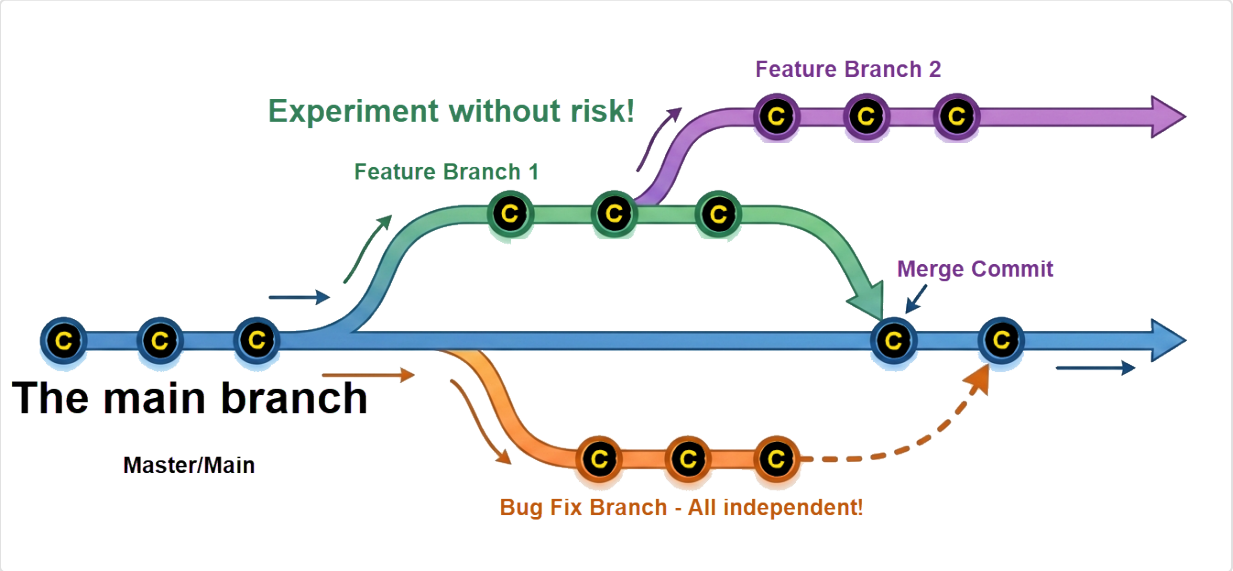


Figure. Git branching multi-track workflow diagram

Merging: Bringing it All Together

When your work on a branch has completed and you need to bring those changes back into the "main" line. This is called **Merging**.

There are two ways Git handles this:

Fast-Forward Merge

If the `main` branch hasn't moved since you created your feature branch, Git performs a "Fast-Forward." It simply slides the `main` pointer forward to match your feature branch. No new "merge commit" is created because the history is a straight line.

Three-Way Merge

If `main` has moved, perhaps a teammate merged a different fix while you were working, Git performs a three-way merge. It looks at the common point of both branches and the two latest snapshots. It creates a new **Merge Commit** that has two parents, effectively tying the histories back together.

Common Branching Workflows

Professional teams don't just branch randomly; they follow "Workflows" to keep the history clean.

Workflow	Description	Use Case
Git Flow	Uses a main branch for production and a develop branch for integration, with specific branches for features, releases and hotfixes.	Complex, scheduled releases.
GitHub Flow	A simpler model where everything in main is deployable. You create a branch for a feature, open a Pull Request and merge when ready.	Continuous Deployment (SaaS).
Trunk-Based	Developers merge small, frequent updates to a single "trunk" (main) branch, avoiding long-lived branches.	High-velocity, senior teams.

Best Practices for Clean Branching

To keep your project from becoming a spaghetti of lines, it is advisable to follow these rules:

1. **Keep Branches Short-Lived:** The longer a branch exists, the harder it will be to merge. Try to merge back to main every few days if this is possible.
2. **Descriptive Naming:** Use `feature/login-page` or `bugfix/header-alignment` instead of `my-edits`.
3. **One Task per Branch:** Don't fix a typo in one part, while you're building a new feature on the same branch.
4. **Delete After Merging:** Once a branch is merged, delete the pointer to keep your repository tidy. After merging, the branch has no use.

How to Version Control Database Objects

IBM i database version control, IFS-based management, and hash-based release validation

Introduction

Database objects on IBM i—tables, views, indexes, constraints, triggers, functions, and stored procedures—are often treated differently from RPG or CL source code. In many environments, schema changes are still applied through one-off SQL scripts, clumsily deploying a generated object, or ad hoc updates in development and production. That approach introduces risk, weakens traceability, and makes it harder to reproduce environments consistently.

Version controlling database objects brings structure and discipline to schema management. Just like RPG source, database definitions should be stored in source control, reviewed, versioned, and deployed through a controlled process. This ensures that every change is traceable, reproducible, and aligned with the application lifecycle.

A well-designed database version control strategy on IBM i supports DevOps practices, enables automation, reduces production risk, and improves audit readiness. The goal is to treat the database schema as a first-class asset, managed with the same rigor as application code.

This procedure focuses on the areas that matter most in practice: implementing reliable versioning via Git, applying practical tips for day-to-day database control, introducing API-based hash validation, and integrating all of it into consistent release promotion from Development (DVP) through Integration (ITG) and Production (PRD).

Git for DB2 Management

Storing DDL and migration scripts in Git allows teams to manage database changes the same way they manage application source. A single member for each catalog entry is required to provide a coherent audit trail and any existing separate Alter script should not be loaded into Git.

The first step is to make sure you have current source for your production database model since it may have been altered adhoc. IBM has provided tools and an API to retrieve current DDL source. Be sure that Create or Replace is consistently used through out the set of source to allow efficient deployment of changes. A best practice is to make

sure that the source is viable and that the test environments are up to date by deploying the newly generated source, also a great test of the deployment process AND your test data.

This ensures a repeatable operating model. Release packages can be reviewed using plain text diffs. Audit teams can trace what changed, when it changed, and why. Optional is setting the source attribute to identify the resulting object as is common in make based build systems. If the build and deployment methodology requires the specific attribute now is the time to do it before any source is loaded into Git with generic SQL attributes.

The location of the source once you have a good baseline can be IFS or library based as required by your Build and deployment process.

Versioning

There are three common approaches to versioning database objects on IBM i: model-first, migration-first, and hybrid.

Model-First maintains a canonical schema definition and rebuilds environments from that source. It provides clarity, but becomes difficult when large production schemas and live data must be preserved. This is how enterprise CM packages worked before Alter and then Create or Replace were common.

Migration-First applies sequential change scripts over time. It creates a strong audit trail, but long migration chains can become hard to reason about.

Hybrid (Recommended) combines both:

- Start with a current baseline schema model
- Apply incremental changes to that source leveraging Create or Replace
- Refer to Git commits to identify specific changes for a release

Versioning also has an implicit and explicit dimension:

- **Implicit versioning** uses catalog state (QSYS2 tables) for validation (Format ID plus additional attributes)
- **Explicit versioning** uses versioning information from Git based on the release branch.

Example:

```
CREATE OR REPLACE TABLE APP_SCHEMA_VERSION (
  VERSION_ID   VARCHAR(50)  NOT NULL PRIMARY KEY,
  APPLIED_AT   TIMESTAMP    NOT NULL,
  APPLIED_BY   VARCHAR(50)  NOT NULL,
  DESCRIPTION  VARCHAR(256) NOT NULL
);
```

Tips for DB Version Control

The mechanics of version control matter, but daily discipline matters just as much.

Core practices:

- Treat all DDL as source code and start with a complete and current set in Git
- Test migrations of each change in a production like test environment
- Include and test rollback using previous member, especially for complex changes
- Never use manual DDL in production
- Use consistent naming conventions
- Validate schema before and after deployment
- Maintain release notes tied to commits
- Review both technical and operational impact

Additional guidance:

- Prefer additive changes first, new DB2 objects then the objects that use them
- Backfill data in controlled steps in post deployment scripts
- Remove deprecated structures in later releases
- Explicitly define core objects
- Control and document implicit objects

While leveraging Create or Replace has a very high success rate it doesn't always work. Production data may not support application of a constraint or refactoring of a field. It's important to have a test environment that has a very production like set of data to verify that the deployment will work and what downtime window is required for a complex deployment.

These practices keep database version control sustainable and predictable.

DB2 validation

Since using a hash is not practical then the single source member for each DB2 can be leveraged to validate that environments, either test or multiple production environments are in sync. Commonly the file format ID is compared to validate that file structure is consistent, but there are other attributes that contribute to data integrity that are not reflected in the file format ID. Since the first thing that was done was to retrieve the current source for the DB2 structure in production if we script this process we can use that resulting source to verify that the different environments are in sync.

The QSYS2 catalog files by schema provide the control for executing the IBM API to retrieve source which can then be compared to one of two controls. If the environment should match production it can be compared to the trunk or extracted source from production itself. Because schema can be referenced in the source when generated it can be required to replace that string with a consistent value. Once it's been genericized of expected differences a text compare can be used.

For test environments generally they should match a branch in Git. Fortunately we can easily create a temporary branch and load it with the extracted source in an automated way to allow easy comparison between branches. Automation is important as the generation and subsequent upload of the DB2 specific source to Git can take some time, but the detailed result is worth it. Unlike a hash which only tells us there is a difference, this source comparison tells us what specific attribute is different and how.

This reduces operational risk and improves sustainability.

Conclusion

IBM i database version control works best as a complete operating model.

- The IFS provides the management layer
- Hybrid versioning provides control and clarity
- Governance and discipline ensure sustainability
- Periodic DB2 comparison raises confidence in the process and reduces problems

The practical takeaway is clear: Store schema definitions and migrations in a structured repository, use explicit version tracking with valid commit messages, and modernize release comparison with API-driven source comparison. This approach enables stronger governance, better automation, and more reliable releases—while fully leveraging the strengths of the IBM i platform.

How to Version Control Database Objects (IFS for Management)

product: "Source Control" audience: "Intermediate" time: "20 minutes"

How do I implement Gitflow for IBM i?

IBM i development has always operated at two speeds. Even organizations striving to be agile typically run a mix of slow, complex projects alongside fast, tactical ones — and that's before you factor in hotfixes. The platform's deep interdependencies mean that a single file or module change can have ripple effects that would be trivial on other stacks. This reality shapes everything about how teams should approach Git branching on IBM i, and it explains why the goal is almost always the same: **build and deploy just the changes, not the tip of the branch**.

Understanding GitFlow — and Its Limits on IBM i

GitFlow is a widely adopted branching model built around feature branches and multiple hierarchal primary branches. Developers work in isolation on a feature or sandbox branch and delay merging back to the release or higher branch until the feature is complete. Primary branches can be environment-based or release-based, and the model works with multiple levels of branches as needed.

But IBM i introduces friction that makes standard GitFlow problematic. Few IBM i shops have fully broken down their large programs into microservices. File changes carry significant downstream impact, and ILE module changes can be even more far-reaching. Large-impact projects simply take longer — not just to develop, but to adequately test. When you layer environment-based branches on top of that reality, you quickly hit a familiar pain point: slow projects "color" fast ones. A sluggish feature can block or contaminate testing an urgent fix. Concurrent changes stack up, reverting becomes complicated, and IBM i sourcefile properties can add yet another layer of friction.

A Better Flow: Feature → Release

The recommended alternative is a **Feature → Release** workflow. Rather than merging directly into environment-based branches, developers merge feature branches into dedicated date based release branches. This single shift has outsized benefits.

Most importantly, the release branch becomes the designated place to surface and resolve conflicts. This is "shift left" thinking in practice — identifying problems early, testing conflict resolutions, and verifying the integration of multiple features before anything touches production. It also offers more flexibility in how environments are managed: teams can use environment-based release branches, packaged feature/release deployments, or a direct feature/release-to-environment model depending on their maturity and tooling.

The Stale Branch Problem

One of the most underappreciated hazards in IBM i Git workflows is the **stale branch**. In Git, a branch is simply a snapshot of its parent at the moment it was created. As time passes and other changes land in the main branch, the feature branch drifts further behind — it becomes stale.

On most platforms, refreshing a stale branch is routine. On IBM i, it's a minefield. Using Git's `fetch` to pull changes from master into a stale branch will indeed bring in those changes — but the next build can compile *all* of them as if they belong to the current version, even if they're already in production. Cherry-picking commits can seem more surgical, but commits often bundle multiple changed members together, meaning you still pick up more than you intended.

Rebasing is the most reliable way to refresh a branch without triggering unwanted builds. However rebasing a common branch like a Release can have the desired effect on changes from one feature and have negative impact on another feature. Even better: **branches are cheap — just create a new one**. Creating a fresh target branch from the current state of the trunk and merging only your specific changes across is cleaner, more predictable, and avoids the build side effects entirely. The Feature → Release pattern supports this naturally: a new release branch can be created for each integration cycle, eliminating the stale problem by design.

this *fresh branch* approach can be formalized into an environment based Release branch, that is to have the requirement for each environment a new Release branch is created. This gives the opportunity for an environment specific Build with the correct library list for each progressive test environment, as the set of changes heading to production together becomes more and more stable.

Merge Conflicts and IBM i Specifics

Merge conflicts on IBM i are more common than developers expect. Git will flag a conflict not only when the same line is changed in two branches, but also when *adjacent lines* are modified — dramatically raising the frequency. IBM i adds another twist: `.properties` and other metadata files (how attributes of the resulting object are managed) travel alongside source members and can generate their own conflicts. Fortunately these are being deprecated by some tools as they get incorporated into the source as comments similar to H-specs.

The tooling options for resolving conflicts vary significantly by Git platform. Azure DevOps and Bitbucket offer no conflict resolution in their default web UI — teams must drop into Visual Studio, VS Code, or Sourcetree. GitHub and GitLab allow in-browser resolution, though branch protection rules may block direct commits to release branches.

The best practice is consistent: **resolve conflicts inside the feature branch, not in the release branch**. Pull the release branch into the feature, let the conflict surface there, resolve it using a side-by-side editor, then build and test the merged result. Only once the conflict is proven clean should it be merged back into the release via pull request. This ensures the tested resolution — not just an assumed one — is what lands in integration.

The Bottom Line

Git is not magic. It gives you visibility and tools; it does not fix a broken workflow. Every IBM i team should be operating on a Feature → Release flow, avoiding concurrent changes until their process is solid, and choosing tooling that supports conflict resolution *before* a crisis forces the issue. The goal stays constant: build and ship just the changes — nothing more.

Common Git Issues and Solutions

This section covers common issues that developers may encounter when using Git in IBM i development and provides solutions to resolve them effectively. Some of the common issues include:

1. Merge Conflicts Merge conflicts occur when Git cannot automatically reconcile differences between two branches. This usually happens when multiple developers edit the same lines of a file or when changes overlap.

How to Resolve:

- Identify the file(s) in conflicts. Git marks conflicting sections in the file with `<<<<<<<`, `=======`, and `>>>>>>>`.
- Discuss and decide which changes to keep and edit the file(s).
- After editing, run:

```
git add <filename>
git commit
```

2. Uncommitted Changes on Branch Switch If there are uncommitted changes in your working directory and you try to switch branches, Git will prevent the switch to avoid losing those changes. If you want to switch branches, you need to either commit or stash your changes.

How to Resolve:

- Stash changes:

```
git stash
git checkout <branch-name>
```

- Commit changes: If ready, commit before switching:

```
git add .
git commit -m "Save work before switching branches"
```

3. Files Not Added for Tracking Sometimes files are created but missed to be added to Git's tracking. This can impact the final deployment to production if important files are not included in commits.

How to Resolve:

- Always verify the current status of the files in the repository before doing commit by running:

```
git status
```

- If all files in the working directory are to be added, use:

```
git add .
```

- If specific files are to be added, use:

```
git add <filename>
```

- Check status again to confirm the files are staged for commit.

4. Push Rejected Due to Conflicts Push rejections occur when your local branch is behind the remote branch, meaning there are commits on the remote that you don't have locally. This can happen when multiple developers are working on the same branch.

How to Resolve:

- First, pull the latest changes from the remote branch to update your local branch before starting to work on it:

```
git pull origin <branch>
```

- After making your changes and committing, push again:

```
git push
```

5. Committing on wrong branch While working on multiple branches, it's easy to accidentally commit changes to the wrong branch. It can lead to confusion and require additional steps to fix.

How to Resolve:

- First, identify the commit hash of the incorrect commit by running:

```
git log
```

- Reset the current branch to the previous commit to remove the incorrect commit:

```
git reset
```

- Switch to the correct branch and cherry-pick the commit to move it to the correct branch:

```
git checkout correct-branch
git cherry-pick <commit-hash>
```

6. Large Repositories and Files Sometimes, Git can struggle with large repositories or files, leading to performance issues or push failures. This is especially common when the repository gets very big or when large binary files are added.

How to Resolve:

- Use Git Large File Storage (LFS) for handling large files:

```
git lfs install
git lfs track "*.psd"
```

- Use multiple smaller commits instead of one large commit to improve performance
- Use .gitignore to exclude large files that don't need to be tracked

7. Git not found in PATH When trying to run Git commands, you may encounter an error indicating that Git is not found in the system PATH. It's very common that you may have Git installed, but if it's not added to the PATH environment variable, the system won't recognize it.

How to Resolve:

- Add Git to PATH by setting the path in the environment variable
- Add Path variable in .profile or .bashrc file and add it to the home directory

```
export PATH=$PATH:/path/to/git/bin
```

8. Cannot run less When running Git commands such as 'git log' and 'git status' that produce a lot of output, Git uses the 'less' pager to display the output. If 'less' is not installed or configured properly, you may encounter an error.

How to Resolve:

- Install 'less' on your system:

```
yum install less
```

- Configure Git to use 'less' as the default pager:

```
git config --global core.pager less
```

9. Multi-Threaded Cloning Issues When cloning large repositories, you may encounter issues with multi-threaded cloning, which can lead to errors or incomplete clones.

How to Resolve:

- Use the `--depth` option to create a shallow clone, which can reduce the amount of data being cloned:

```
git clone --depth 1 <repository-url>
```

- Use environment variable 'QIBM_MULTI_THREADED' to control multi-threaded cloning behavior. This variable can be added by command 'addenvvar' set to 'ON' or 'OFF'.

10. UTF-8 Encoding Issues Since Git uses UTF-8 CCSID(1208) by default, you may encounter issues related to UTF-8 encoding. It can lead to problems with migrated repositories not displayed correctly when viewed from VS Code or RDi.

How to Resolve:

- Custom scripts or tools used in the development process should use UTF-8 ccsid(1208) to ensure compatibility with Git.
- Use tools that has inbuilt support to handle UTF-8 encoding issues, such as Visual Studio Code.

Conclusion

Git is a powerful tool for version control, it can come with its own set of challenges. By understanding common issues and their solutions, developers can effectively navigate and resolve problems that may arise during Git usage in IBM i development.

How do I create Git ready projects?

audience: "Intermediate" time: "16 minutes"

How do I integrate Git workflows?

Transitioning from traditional library-based development to Git-based version control is the "heart transplant" of IBM i modernization. It moves your source of truth from the source physical file (SPF) to a distributed repository, allowing for modern DevOps automation.

What This Covers

You will learn how to implement three distinct branching strategies on the IBM i, shifting from a "lock-and-edit" mindset to a collaborative, parallel development model.

Before You Start

- ☐ Install git on your workstation
- ☐ Install Open Source Tools on IBM i: RPM-based git and ssh packages are installed.
- ☐ Select a Build Tool: You'll need a compiler engine (like TOBi) that can orchestrate the compilation of streamfiles into IBM i objects.

Steps

"Main-only" workflow (The Continuous Deployment Model)

Overview: All developers clone the repository and push directly to the main branch. There are no long-lived feature branches. This is best paired with an automated build that pushes to a development library immediately upon commit.

```
graph TD
    A[Clone Repository] --> B[Make Changes to Source Files]
    B --> C[Stage Changes<br/>git add .]
    C --> D[Commit Changes<br/>git commit -m 'message']
    D --> E[Push to Main<br/>git push origin main]
    E --> F{Automated Build}
    F -->|Success| G[Deploy to DEV Library]
    F -->|Failure| H[Notify Team<br/>Fix Immediately]
    H --> B
    G --> I[Continue Development]
    I --> B

    style F fill:#f9f,stroke:#333,stroke-width:2px
    style H fill:#f66,stroke:#333,stroke-width:2px
    style G fill:#6f6,stroke:#333,stroke-width:2px
```

Implementation Steps:

1. **Developer Workflow**

```
## Clone the repository

git clone <repository-url>
cd my-project

## Make changes to source files

## Edit RPGLE, SQLRPGLE, CL, etc.

## Stage and commit changes

git add .
git commit -m "Fix: Corrected customer lookup logic"

## Push directly to main

git push origin main
```

2. **Establish Team Conventions**

- Require descriptive commit messages (e.g., "Fix:", "Feature:", "Refactor:")
- Implement pre-commit hooks to validate syntax or formatting
- Set up notifications (Slack, email) for build failures

What to Expect: High velocity but high risk. This is ideal for small, senior teams or "hotfixes" where speed is the priority. It requires a robust automated testing suite to prevent breaking the build. Expect immediate feedback—if your commit breaks the build, everyone knows within minutes.

IBM i Considerations:

- Ensure your build tool can handle concurrent commits
- Consider using library lists to isolate developer testing from the shared development library
- Monitor object locks during compilation to avoid conflicts

"Gitflow" workflow (The Managed Release Model)

Overview: Use a structured hierarchy: main (production), develop (integration), feature/ (new work), and release/ (stabilization). Developers work in feature branches, merge to develop for QA, and only merge to main during a scheduled production release.

```
graph TD
    A[Initialize Repository] --> B[Create Main Branch]
    B --> C[Create Develop Branch]
    C --> D[Feature Development]

    D --> E[Checkout Develop<br/>git checkout develop]
    E --> F[Create Feature Branch<br/>git checkout -b feature/name]
    F --> G[Work on Feature]
    G --> H[Commit Changes]
    H --> I{Feature Complete?}
    I -->|No| G
    I -->|Yes| J[Merge to Develop<br/>git merge feature/name]
    J --> K[Delete Feature Branch]
    K --> L{Build & Test in QA}
    L -->|Issues Found| M[Create Bug Fix Branch]
    M --> G
    L -->|QA Approved| N[Create Release Branch<br/>git checkout -b release/vX.X.X]

    N --> O[Stabilization Phase<br/>Bug Fixes Only]
    O --> P{Critical Bug?}
    P -->|Yes| Q[Fix in Release Branch]
    Q --> O
    P -->|No| R[Merge to Main<br/>git merge release/vX.X.X]

    R --> S[Tag Release<br/>git tag -a vX.X.X]
    S --> T[Merge Back to Develop]
    T --> U[Delete Release Branch]
    U --> V[Deploy to Production]
    V --> W{Emergency Fix Needed?}
    W -->|Yes| X[Hotfix Workflow]
    W -->|No| D

    X --> Y[Create Hotfix Branch<br/>from Main]
    Y --> Z[Fix Critical Issue]
    Z --> AA[Merge to Main & Tag]
    AA --> AB[Merge to Develop]
    AB --> AC[Delete Hotfix Branch]
    AC --> V

    style L fill:#ff9,stroke:#333,stroke-width:2px
    style N fill:#9cf,stroke:#333,stroke-width:2px
    style V fill:#6f6,stroke:#333,stroke-width:2px
    style X fill:#f66,stroke:#333,stroke-width:2px
```

Implementation Steps:

1. **Initialize Gitflow Structure**

```
## Create the repository with two long-lived branches

git init my-project
cd my-project

## Create and push main branch

git commit --allow-empty -m "Initial commit"
git push origin main

## Create and push develop branch

git checkout -b develop
git push origin develop
```

2. **Feature Development Workflow**

```
## Start a new feature from develop

git checkout develop
git pull origin develop
git checkout -b feature/invoice-redesign

## Work on the feature, committing regularly

git add src/INVOICE.rpgle src/INVDSP.dspf
git commit -m "Redesign invoice layout for new branding"

## Push feature branch for backup/collaboration

git push origin feature/invoice-redesign
```

3. **Merge Feature to Develop**

```
## When feature is complete, merge to develop

git checkout develop
git pull origin develop
git merge feature/invoice-redesign
git push origin develop

## Delete the feature branch

git branch -d feature/invoice-redesign
git push origin --delete feature/invoice-redesign
```

4. **Build and Test in QA**

- Your CI/CD pipeline detects the merge to develop
- Builds all objects into QALIB
- QA team tests the integrated changes
- If bugs are found, create new feature branches, fix, and merge back to develop

The Business Case for Source Control

Why use Source Control in IBM i Development?

Imagine stepping into a time machine and landing in the early days of IBM i development. Picture a world where code was etched onto green-screen terminals, where every change was a gamble, and where the phrase "version control" meant nothing more than keeping multiple copies of the same source file with cryptic descriptions like "PROGRAM_V1", "PROGRAM_V1_FINAL", and "PROGRAM_V1.2". This wasn't just inefficient, it was a recipe for disaster. In today's fast-paced development landscape, source control management (SCM) has emerged as the unsung hero, transforming how we build, maintain, and evolve software. But why is it so crucial, especially in the realm of IBM i development? Let's dive deep into the why, the how, and the transformative impact of embracing source control.

Why Source Control Matters: A Foundation for Modern Development

At its core, source control is about managing changes to code over time. It's the digital equivalent of a meticulous librarian who tracks every book borrowed, returned, and modified. But it's more than just record-keeping. It's about enabling collaboration, ensuring reliability, and fostering innovation. Without source control, developers are flying blind, making changes without a safety net, unable to revert mistakes, and struggling to coordinate with teams.

Consider the basic benefits: version tracking allows you to see exactly what changed, when, and by whom. This creates an audit trail that's invaluable for debugging, compliance, and learning. Branching lets you experiment with new features without disrupting the main codebase, while merging brings those innovations back seamlessly. And let's not forget the peace of mind that comes from backups, which means that your code is safe, even if a server crashes.

But source control isn't just a tool for large teams or complex projects. Even solo developers benefit from it. It encourages better habits, like committing small, frequent changes, which makes debugging easier. It also serves as a personal time machine, letting you revisit past states of your code to understand how things evolved.

The Unique Challenges of IBM i Development Without Source Control

Now, zoom in on IBM i development, where the stakes are often higher due to the system's critical role in enterprise environments. IBM i, with its RPGLE, CL, and COBOL languages, has long been the backbone of business applications, think inventory systems, financial processing, and manufacturing controls. Yet, traditional IBM i workflows were built for a different era, one where development was siloed and changes were manual.

Without source control, developers faced an avalanche of challenges. Imagine working on a complex RPGLE program that spans thousands of lines. You make a change to fix a bug, but it introduces a new one. Without versioning, you're stuck: either live with the regression or painstakingly recreate the previous state from memory or scattered backups. Collaboration was a nightmare. Multiple developers couldn't work on the same program simultaneously without risking overwrites. Code reviews? They were possible but painfully manual, with no easy way to compare versions or track who changed what.

Deployment was equally fraught. Pushing changes to production often involved manual file transfers, with no easy way to roll back if something went wrong. This led to "big bang" releases, where months of changes were deployed at once, increasing the risk of catastrophic failures. And perhaps most critically, the lack of traceability meant that understanding the history of a program was nearly impossible. Why did that subroutine change last year? What was the rationale behind that data structure? Without source control, this knowledge was lost, making maintenance a guessing game. In an ecosystem where applications can run for decades, this lack of traceability was a ticking time bomb.

What Source Control Unlocks

Enter source control, and suddenly, the impossible becomes routine. In IBM i development, tools like Git paired with VS Code's seamless integration, open doors that were once firmly shut.

First, collaboration explodes. Teams can now work in parallel, with each developer branching off the main codebase to tackle features or fixes. Pull requests enable code reviews, where peers can comment on changes before they're

merged. This not only improves code quality but also builds a culture of shared knowledge. No more stepping on each other's toes. Instead, it's a symphony of coordinated effort.

Reliability is another game-changer. With every commit, you have a snapshot of your code. If a deployment goes south, rollback is as simple as checking out a previous commit. This minimizes downtime and boosts confidence in releases. For IBM i's mission-critical apps, this is gold. Imagine confidently updating a payroll system without fearing data loss.

Innovation flourishes too. Branching allows for safe experimentation. Want to try a new API integration or refactor legacy code? Create a branch, test it thoroughly, and merge only when it's ready. This encourages bold ideas without risking the stability of production code. In IBM i, where modernization efforts often involve migrating from fixed-format RPG to free-form or integrating with modern APIs, source control provides the safety net needed for such transformations.

Finally, source control democratizes development. Junior developers can learn from history, mentors can guide through diffs, and remote teams can collaborate as if in the same room. It bridges the gap between traditional IBM i practices and modern DevOps, making the platform more attractive to new talent.

Embracing the Future

In the grand tapestry of software development, source control is the thread that weaves reliability, collaboration, and innovation into a cohesive whole. For IBM i developers, it's not just a tool; it's a lifeline. The challenges of the past—lost code, risky deployments, and siloed ways of working—are replaced by a world of controlled progress and shared success.

DevOps Practices for IBM i

DevOps Fundamentals

What is DevOps for IBM i?

What This Covers

This section introduces DevOps practices for IBM i development. You'll learn how DevOps principles apply to IBM i environments, understand the unique challenges and opportunities, and discover how modern DevOps practices can transform IBM i application delivery while respecting the platform's traditional strengths.

What It Is

DevOps for IBM i represents the application of modern software development and operations practices to IBM i application delivery. It combines cultural philosophies, practices, and tools that increase an organization's ability to deliver applications and services at high velocity while maintaining the reliability and security that IBM i is known for.

Core DevOps Principles on IBM i:

Continuous Integration (CI): Automatically builds and tests code changes as developers commit them to version control. On IBM i, this means compiling RPG, COBOL, and CL programs, running unit tests, and validating SQL objects whenever code changes are pushed to repositories like Git.

Continuous Delivery/Deployment (CD): Automates the process of moving code through development, test, and production environments. For IBM i, this includes deploying compiled objects, database changes, and configuration updates through automated pipelines rather than manual processes.

Infrastructure as Code: Manages IBM i system configurations, library structures, and environment settings through version-controlled scripts rather than manual commands. This ensures consistency across environments and enables rapid environment provisioning.

Automated Testing: Implements comprehensive test suites that run automatically, including unit tests for individual programs, integration tests for multi-program workflows, and regression tests to ensure changes don't break existing functionality.

Monitoring and Feedback: Continuously monitors application performance, system health, and user experience, then feeds insights back into the development process to drive improvements.

Unlike traditional IBM i development, where changes move slowly through manual processes with long release cycles, DevOps enables rapid, reliable delivery while maintaining quality and stability.

Why It Matters

Adopting DevOps practices on IBM i addresses critical business and technical challenges:

Accelerating Time to Market: Traditional IBM i development often involves lengthy manual processes for building, testing, and deploying changes. DevOps automation reduces deployment cycles from weeks or months to days or hours, enabling organizations to respond quickly to business needs and competitive pressures.

Improving Quality and Reliability: Automated testing catches defects early when they're cheaper to fix. Consistent, repeatable deployment processes reduce human error. Version control provides complete change history and rollback capabilities. These practices result in fewer production incidents and faster recovery when issues occur.

Enabling Modernization: DevOps practices create the foundation for modernization efforts. Version control enables collaboration between RPG developers and Java developers. CI/CD pipelines support hybrid applications mixing traditional and modern technologies. Automated testing provides confidence when refactoring legacy code.

Reducing Risk: Smaller, more frequent deployments are less risky than large, infrequent releases. Automated rollback capabilities minimize downtime. Comprehensive testing reduces the likelihood of defects reaching production. These factors make change safer and more predictable.

Bridging Skills Gaps: DevOps tools and practices are familiar to modern developers, making IBM i development more accessible to new team members. Automation reduces dependency on specialized knowledge for routine tasks, while version control preserves institutional knowledge.

How It Works

Implementing DevOps on IBM i involves several integrated components and practices:

Version Control Integration

Modern IBM i DevOps starts with version control:

- **Git Integration**: Source code (RPG, COBOL, CL, SQL, DDS) stored in Git repositories
- **Branching Strategies**: Feature branches, pull requests, and code reviews
- **Change Tracking**: Complete history of who changed what and why
- **Collaboration**: Multiple developers working on the same codebase safely

Tools like Git for IBM i, VS Code extensions, and RDi Git integration enable seamless version control for IBM i source code.

Build Automation

Automated compilation and object creation:

- **Build Scripts**: Automated compilation of programs, modules, and service programs
- **Dependency Management**: Tracking and building dependencies in correct order
- **Build Servers**: Dedicated systems or partitions for automated builds
- **Artifact Management**: Storing compiled objects and tracking versions

Tools like The Object Builder for i (TOBi), GNU Make, or custom CL scripts automate the build process.

Automated Testing

Comprehensive test automation:

- **Unit Testing**: Testing individual programs and procedures using frameworks like RPGUnit or iRPGUnit
- **Integration Testing**: Testing multi-program workflows and database interactions
- **Regression Testing**: Ensuring changes don't break existing functionality
- **Performance Testing**: Validating response times and resource usage

CI/CD Pipelines

Automated deployment workflows:

- **Pipeline Tools**: Jenkins, GitLab CI/CD, Azure DevOps, or GitHub Actions
- **Environment Promotion**: Automated movement through DEV → TEST → PROD
- **Deployment Automation**: Scripts to deploy objects, apply database changes, update configurations
- **Approval Gates**: Manual approvals for production deployments when required

Monitoring and Feedback

Continuous observation and improvement:

- **Application Monitoring**: Tracking application performance and errors
- **System Monitoring**: IBM i system health, job queues, resource usage
- **Log Aggregation**: Centralized logging for troubleshooting
- **Metrics and Dashboards**: Visualizing deployment frequency, lead time, and failure rates

Tools like IBM Navigator for i, Splunk, or custom monitoring solutions provide visibility into system and application health.

Key Takeaways

- **DevOps for IBM i applies modern software delivery practices to the platform**, including continuous integration, automated testing, and deployment pipelines that accelerate delivery while maintaining IBM i's legendary reliability and security
- **Adopting DevOps addresses critical challenges** including slow time-to-market, quality issues from manual processes, and skills gaps by introducing automation, version control, and practices familiar to modern developers
- **Implementation requires integrated components** including Git version control, build automation, comprehensive testing frameworks, CI/CD pipelines, and monitoring systems—all adapted to IBM i's unique characteristics
- **The benefits are substantial**: organizations achieve faster deployments, higher quality, reduced risk, and better collaboration while creating the foundation for modernization efforts that blend traditional and modern technologies
- **DevOps is not all-or-nothing**: organizations can adopt practices incrementally, starting with version control and automated builds, then progressively adding testing, pipelines, and monitoring as teams gain experience and confidence

CI/CD Pipeline Implementation

Step 1: Move Source Code into Git

The foundation of any CI/CD pipeline is version control, and Git is the standard. IBM i source members must be exported into the IFS as stream files before Git can manage them.

Open-source tools and IBM's own **IBM i Open Source** ecosystem (available via the yum RPM) provide utilities to help. The community-maintained **RPGLE-to-Git** tooling and the **Tobi** (The Object Builder for i) build framework are particularly popular for this migration.

Steps to migrate source:

1. Install Git on the IBM i IFS via package manager yum.
2. Convert source members to stream files using the `CPYTOSTMF` command or tools like `cvtsrcpf` from open-source community scripts.
3. Initialize a Git repository in the IFS, push to a remote such as GitHub, GitLab, or Azure DevOps.
4. Establish a branching strategy (GitFlow or trunk-based development both work well).

Once source lives in Git, every change is auditable, reviewable, and automatable.

Step 3: Set Up a CI Server

You have many choices for running CI jobs:

Remote CI (GitHub Actions, GitLab CI, Jenkins): The CI server connects to the IBM i system over SSH, triggers a build via the shell that runs locally on the IBM i, and captures results. This is the most common approach. You configure an SSH key pair, add the IBM i host to your CI environment's secrets, and write a pipeline script that SSHs in and runs `makei build`.

A minimal GitHub Actions workflow might look like this:

```
name: IBM i Build

on:
  push:
    branches: [main, develop]

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Build on IBM i
        uses: IBM/ibmi-Tobi@main
        with:
          host: ${{ secrets.IBMI_HOST }}
          username: ${{ secrets.IBMI_USER }}
          private-key: ${{ secrets.IBMI_SSH_KEY }}
```

IBM has released an official GitHub Action (`IBM/ibmi-Tobi`) that wraps SSH connectivity and Tobi invocation into a reusable step, dramatically simplifying configuration.

Step 5: Automate Deployment

Deployment on IBM i means moving compiled objects from a build library into production libraries. Tools like **Ansible** and custom CL scripts can automate this promotion. At its simplest, a deployment script:

1. Archives objects from the target library to a backup library.
2. Promotes objects from the Build library or SAVF to the target library using `MOVOBJ` or `RSTOBJ`.
3. Manages data in files if needed and manages object authority.
4. Sends a notification on success or failure.

For regulated environments, pairing automated deployment with an approval gate — a manual sign-off step in your CI/CD platform — satisfies change-management requirements without sacrificing automation. Pipeline tools commonly support an approval step.

GitLab CI/CD Integration

How do I use GitLab CI/CD and IBM i?

Originally, Jenkins was the primary automation tool. As the methodology matured, all Git packages now provide some level of automation tools. As a complete DevSecOps platform, GitLab provides source control, CI/CD pipelines, and artifact management in a single interface — and with the right configuration, it can drive automated builds directly on IBM i. Here is how it works and why it matters.

GitLab CI/CD in a Nutshell

GitLab CI/CD is built around a single file: `.gitlab-ci.yml`. Placed at the root of your repository, this YAML file defines the stages, jobs, and scripts that execute every time a pipeline is triggered. Pipelines can fire on any Git event — a push to a branch, a merge request, a tag, or a scheduled timer. Each pipeline moves through stages in order (typically build, test, and deploy), with jobs within a stage running in parallel when runners are available.

The agent that actually executes those jobs is called a **GitLab Runner**. Runners can live anywhere — on a Linux server, a Windows machine, in a container, or on IBM i itself via PASE. When a pipeline triggers, GitLab hands a job to an available runner, which clones the repository, executes the defined scripts, and reports results back to GitLab.

This methodology is now common among all the Git packages, with the Git server functioning as the trigger manager and these automation agents, hosted or remote, doing the work. Of course for IBM native source, there is a build engine that runs on the IBM i itself to do the compiles, no matter what starts it.

Connecting GitLab to IBM i

There are three primary ways to connect GitLab CI/CD to an IBM i build environment.

The first is the **SSH executor**. In this model, GitLab Runner is installed on any host — a Linux server, a CI/CD appliance, or even a cloud VM — and configured to execute jobs over SSH on a remote IBM i system. The runner connects to the IBM i and runs commands directly starting a Build process, which allows capture of the log into the Build log as part of the SSH session. SSH key-based authentication keeps credentials out of the pipeline definition and off the command line.

The second approach is the **shell executor**, where GitLab Runner is installed directly in IBM i's PASE environment. PASE provides the bash shell, GNU tools, and the open-source tools that GitLab Runner expects. Once registered, the runner polls the GitLab server and executes builds natively on the IBM i partition. This eliminates the SSH hop and gives the pipeline direct access to IBM i libraries, the IFS, and build tools.

The third option — and arguably the most powerful for CI/CD — is **Ansible**. Rather than having GitLab Runner talk directly to IBM i, a GitLab pipeline job invokes an Ansible playbook that orchestrates the build. Ansible is agentless: it connects to IBM i over SSH and executes tasks using IBM's supported collection of IBM i modules, which are written in Python and handle IBM i-specific operations such as compiling objects, managing library lists, and controlling object authority. IBM provides a dedicated Ansible collection for IBM i (`ibm.power_ibmi`), which includes modules like `ibmi_cl_command` for running CL commands and `ibmi_object_authority` for managing object permissions.

All three approaches are in active use. The SSH executor is often the simplest to set up; the shell executor running natively in PASE offers the tightest integration (but the most challenging to setup); and Ansible is the right choice when the build pipeline needs to help out a less mature build process. It is also helpful in the CD phase if you aren't using a CM package already.

Triggering a Build with Tobi

Once the runner is connected to IBM i, the real work of the build pipeline can begin. One build engine for IBM i open-source workflows is **The Object Builder for i (TOBi)**, which uses GNU Make and Makefile syntax to compile ILE objects with full dependency tracking. Tobi is available as an RPM package and installs with a single `yum install tobi` command.

Most Change Management vendors also have a build process now that is integrated with Git that can be started with a command.

A typical `.gitlab-ci.yml` for an IBM i build project might define a `build` stage that executes `makei build -d {directoryname}` against the project folder on the IFS. Tobi reads the `Rules.mk` file in each project directory, determines which objects need to be compiled based on what has changed, resolves dependencies, and compiles in the correct order. Because `Rules.mk` is committed to the Git repository alongside the source, build rules are versioned, auditable, and consistent across every pipeline run.

The pipeline is scoped to branches so that pushes to branches trigger builds into isolated branch libraries, and when successful then deploys into a shared integration environment. GitLab's `rules` keyword provides fine-grained control over when builds fire.

Variables, Artifacts, and Feedback

GitLab CI/CD variables — defined at the project, group, or instance level — handle the configuration that changes between environments: target library names, compile flags, and credentials. These values are injected into pipeline jobs at runtime.

Build logs are captured automatically by the runner and uploaded to GitLab, where they are browsable job by job. Depending on the location of the runner, logs from the IFS can be retrieved.

Why It Matters

Automating IBM i builds through GitLab means every code change triggers a consistent, repeatable compile process with full dependency resolution — and every developer on the team can see the result within minutes of pushing code. GitLab brings IBM i into the same DevOps workflow used by the rest of the organization.

How do I automate deployments?

Deployment requirements on IBM i can be very different from other platforms, and it's common to have a high level of audit requirements and complex authority models. The IBM i world has been living DevSecOps before it was a common term, so how can we automate the deployment beyond just starting a process?

First: Build Is Not Deployment

One of the most important distinctions in any IBM i DevSecOps workflow is the difference between a build and a deployment — and why these should never be the same step.

A build compiles changed source and dependencies into objects and places them into a staging library. A deployment takes those objects and moves them into target environments. Treating a build as a deployment skips the safety net: there is no rollback, no clean separation between what was tested and what goes to production, and limited opportunity to manage object authority or locking in a controlled way.

The recommended approach is to build into a dedicated library — ideally one library per branch — and then deploy from that library using a separate, purpose-built deployment process. This pattern allows failed builds to be isolated and discarded without touching the target environment, and it gives the deployment tool a clean, known starting point every time.

Security and deployment are deeply intertwined in this model. Object authority, authority adoption, and the question of who is permitted to trigger a deployment all need to be addressed explicitly. Separation of duties — ensuring that the person who writes code is not the same person who promotes it to production — is a common requirement that the tooling must support, not undermine.

The Role of Automation

The common use of automation is to start a process, that is to take a request from one tool and transfer it to IBM i to start another process and provide feedback about the result. In this context we have an automation tool (perhaps a pipeline) that has determined the previous step (the Build) was successful so then we start the Deploy, CI/CD. The Deploy process must also provide feedback. If there is a failure in the Deploy process, the pipeline reacts, indicates failure, does not run the next step (testing), and possibly initiates rollback.

Designing the Automation Stack

Building a reliable deployment automation stack means starting from the lowest level and working upward. The goal is to avoid custom one-off scripts for every deployment scenario and instead build reusable, manifest-driven processes while maintaining the isolation the Git branch provides.

At the foundation is the branch representing the change. It should be built into a branch specific library for packaging. On IBM i, the standard packaging artifact is a Save File (SAVF). Naming conventions matter: a consistently named SAVF that identifies the branch, stored in an artifact repository like JFrog Artifactory, becomes a traceable, promotable unit that moves through staging, test, and production environments in a predictable way. The IFS serves as a natural staging area for artifacts in transit.

A manifest accompanies the SAVF, identifying what objects are included, driving the deployment process, and serving as a feedback artifact once deployment completes. IBM i system views like `QSYS2/OBJECT_STATISTICS` and `SYSMEMBERSTAT` can inform manifest generation from the build library, giving the process precise knowledge of what changed and what needs to move.

Triggering deployment can happen in several ways: automatically as the result of a successful build (for test environments), from a pipeline without a build step (for pre-packaged releases), or via an existing enterprise change management tool (for production). The right trigger depends on environment, compliance requirements, and organizational process — and all three patterns have a legitimate place in a mature application lifecycle workflow.

Open Source Tools That Bridge the Gap

Four categories of tools fill out the deployment pipeline: artifact management, triggering tools, communication tools, and feedback tools.

Ansible deserves particular attention as a communication and orchestration layer. It is agentless, meaning IBM i does not need to run a persistent agent. Ansible modules written in Python — such as `ibmi_object_authority.py` — handle IBM i-specific tasks. Playbooks written in YAML orchestrate sequences of these modules, with logic to handle conditions like skipping a PTF installation if it is already applied. IBM provides a collection of Ansible modules for IBM i, and the getting-started documentation is readily available on GitHub.

For feedback, the options include status trapping via log grep, integration with project management systems, email notifications using existing infrastructure, and even spoolfile capture by copying output to the IFS for downstream processing.

The overarching advice is straightforward: use the tools already in use at your company. Leverage existing expertise. The goal is synchronization across platforms, not the creation of a parallel IBM i-only toolchain that nobody else in the organization understands or supports.

An option is enterprise deployment tools that handle IBM i, DevOps Deploy from IBM as well Change Management packages that have open APIs.

Getting Started

Begin with a workflow draft. Review the tools currently in use across your organization, identify the gaps, and determine which open source tools can fill them. Build a sandbox, test the workflow end to end, revise, and test again. Plan for at least three iterations before committing to going live.

IBM i can participate fully in a modern DevSecOps pipeline. The platform's strengths — reliability, security, and object-level control — are assets in this model, not obstacles. The key is building the integration thoughtfully, one layer at a time.

Infrastructure as Code & DevSecOps

How do I implement IaC for IBM i?

What is Infrastructure as Code?

Infrastructure as Code (IaC) is the practice of managing and provisioning computing infrastructure through code rather than through manual processes. For a more in-depth look at the definition of IaC, see Infrastructure as Code.

Things to consider when implementing DevOps IaC for IBM i

There is a fundamental difference between deploying IBM i native languages versus deployments for other languages. On most other platforms, when one line in the source code is changed, the whole application is rebuild and redeployed. After the build, the application is typically packaged in some sort of standard format; jar, war, ear, vsix, docker image, etc..

Deployment usually means taking the application offline for a couple of seconds and then redeploying the whole application again. The database is always separated from program code and other resources. This means that programs can be replaced without replacing the data. The data sits in some other application, like MySQL, Mongo, or any of the other well known databases.

Now imagine that changing one line in a CLLE program will trigger the complete recompile of all your programs in your financial or logistic application and after the recompile is complete, all these programs will be deployed again. Bonkers!

Aspects of an automated deployment pipeline

- Typical IBM i deployment mechanisms deploy a small subset of components (programs, etc..).
- When deploying database tables, care must be taken to convert the data from the old tables.
- When an object is level-dependent on other objects, also these objects must be recompiled and redeployed. When one program is not included the deployment must be blocked to prevent file and service program level checks.
- Various runtime attributes can be set on objects, e.g. the use of adopted authority, the user profile to run under, file attributes, etc... These attributes must be either compiled in, or re-applied after deployment.
- Authority settings are very granular on the IBM i. A blessing, sure, but also a deployment curse.
- Deploy to multiple productions machines
- Enable rollback when there is something not quite working.

DIY or commercial tooling?

There are a number of DevOps vendors for IBM i that have extensive experience with all the mentioned aspects. When you plan on creating your own deployment mechanisms, be prepared to hire a DevOps team that takes care of the deployments and programmer support. Building a CI/CD Pipeline is not easy and requires ongoing maintenance.

How to compile

You should try to keep a default compile command for a specific object type. Ideally, all RPGLE programs are compiled exactly the same. Unfortunately, some objects require special way of compiling. Some compile specific attributes can be defined inside the source code. e.g., by using the RPGLE control options. In cases where this is not possible, e.g., display and printer files, you must 'remember' the compile instruction for that specific object. You can store that in the compile script, put it in the source code as comment lines, or stick it in a database.

Data conversion

When deploying a new version of table, the conservative way is to move the old table from the target library to a temporary rollback location, create the new table in the target library and the convert the data either by using `CPYF *MAP *DROP` or by calling a conversion program.

With SQL you can do an in-place update `ALTER table` command. The drawback is that the old file no longer exists which makes a rollback harder. You have to also create a reverse `ALTER table` to be applied when the business decides to do an emergency revert.

When deploying changed tables, all indexes and other database dependencies must be re-applied on the new object.

Atomic deployments and immediate rollback

A deployment must be atomic, which means indivisable. When deploying 20 objects and the last object fails to deploy for some reason (locks, compile failures, etc..) the 19 previously deployed objects must be automatically rolled back.

Delayed rollback

A deployment can be technically successful but sometimes still rollback is required because of bugs or other issues in the new programs. This either requires some sort of staging area where a backup of the old deployment is kept or you ask your programmers to revert their changes, rebuild, and redeploy.

Authority

To minimize an attack surface, all objects must be properly authorized. Preferably `*PUBLIC *EXCLUDE` but all is dependent on the specifics of an application. Many legacy applications were designed and build in a time where cyber-attacks did simply not exist.

Redesigning authority is also an important modernization aspect.

Components of IBM i IaC

1. Source-Controlled CL Programs

CL (Control Language) programs can serve as primary IaC mechanism on IBM i. An example script in our environment can be:

```
/* Spin up isolated test environment */
pgm parm(&prefix)
  dcl &prefix *char 2
  dcl &testlib *char 10
  dcl &testlibd *char 50

  chgvar &testlib  ('ISOTEST' *cat &prefix)
  chgvar &testlibd ('Isolated test environment' *bcat &testlib)

  /* Create test library and copy data */
  crtlib lib(&testlib) text(&testLibd)
  cpylib fromlib(myappdata) tolib(&testlib)

  /* Point config to this environment */
  chgdtaara dtaara(&testlib/config) value(&testlib)
endpgm
```

2. REXX

Although not very popular, IBM i comes with its own native scripting language: REXX Database structures should be defined in SQL DDL scripts stored in version control:

2. SQL DDL Scripts

Database structures should be defined in SQL DDL scripts stored in version control:

```
-- Clone test schema with fresh data
create schema isotest01;

-- Copy table structure and data from common test
create table isotest01.customers as (
  select * from myapptest.customers
) with data;

-- Update config to point to this environment
update isotest01.config
  set schema_name = 'ISOTEST01'
  where config_key = 'TARGET_SCHEMA';
```

3. IFS-Based Configuration Files

Various scripting languages exist on the IBM i that can be used to execute IaC components.

A build script is also an early example of IaC. Although strictly spoken they are Pipeline as Code, there are often various pure IaC components available in a build script. Especially on the IBM i where a modern 'makefile' also creates the infrastructure that the code needs to comfortably run.

Recipes, Scripts, Playbooks, Manifests and Dockerfiles

Ansible for IBM i

Ansible is an automation system that runs on the IBM i. If we add a 'Playbook' for our IBM i test automation script:

```
## ansible playbook example

- name: Configure IBM i environment
  hosts: ibmi_servers
  tasks:
    - name: Refresh Test Environments
      ibmi_cl_command:
        cmd: "PIPELINE/CRTTSTENV ACTION(*REFRESH) ENV(*ALL)"
      ignore_errors: yes

    - name: Restore objects
      ibmi_object_restore:
        savefile: /tmp/myapp.savf
        library: MYAPP
```

Jenkins

One pure IBM i example is the Jenkin `IBM i Steps plugin` that was created especially for IBM i automation. Jenkins is a Job Automation workhorse that runs natively on the IBM i. It is primarily used for build jobs but it can do literally anything.

```
pipeline {
  agent any
  triggers {
    cron('0 0 * * *')  // Midnight daily
  }
  stages {
    stage('Refresh Test Environment') {
      steps {
        onIBMi('DEV') {
          ibmiCommand 'PIPELINE/CRTTSTENV ACTION(*REFRESH) ENV(01)'
        }
      }
    }
  }
}
```

Build Automation

Pipeline as Code

Jenkins

100% free and open source and IBM i ready here is Jenkins for its second appearance. A example snippet of a pipeline job for build automation and deployment is below. Warning, creating a good PaC component can become complicated:

```
pipeline {
  agent any
  stages {
  ...
    stage('Build all changed sources') {
      steps {
        onIBMi('DEV') {
          changedFiles.eachWithIndex { file, index ->
            try {
              // Upload file preserving directory structure
              ibmiPutIFS(from: file, to: targetDir)

              // Build the source in some way
              def buildResult = ibmiCommand(
                command: "BLDIFSOMS ACTC(*PUSH) BRANCH('${currentBranch}') STMF('${targetFile}')",
                failOnError: false)
            } catch (Exception e) {
              failedFiles.add([file: file, error: e.message])
              // Always show exceptions - these are unexpected errors
              echo "ERROR: ${file}\n  ${e.message}"
            }
  ...
}
```

GNU Make on IBM i

Traditional make files adapted for IBM i:

```
## Makefile for IBM i

LIBRARY = MYLIB
TGTRLS = *CURRENT

%.pgm: %.rpgle
    system "CRTSQLRPGI OBJ($(LIBRARY)/$*) SRCSTMF('$<') COMMIT(*NONE)"

all: mypgm.pgm mysrvpgm.srvpgm
```

The Object Builder for i (TOBI)

Available as an open source solution on top of GNU Make TOBI is a build system for creating IBM i-native objects.

Conclusion

Infrastructure and Pipeline as Code on IBM i can deliver full or partial automation of labour intensive processes. They are repeatable and precise and will save many hours of work.

Getting to a fully automated pipeline is a serious effort and takes time and ongoing maintenance by specialized DevOps engineers. Take a look at all the available options, open source and commercial and determine what is the best in your situation.

How do I Implement DevSecOps

This chapter describes integration of security practices into the software development pipeline.

What is DevSecOps

The distinction between **DevOps** and **DevSecOps** is primarily about **where and how security is integrated** into the software delivery lifecycle.

The IBM i platform has consistently maintained integrated security as a fundamental architectural component, but application and network level access require security beyond the responsibility of the OS.

OS Level

- **Objects and Data** - Access Authority is defined on the object level, including the use of data for objects containing data.
- **User Authentication & Authorization** - The OS enforces user based access control using user profiles and role-based access control using group profiles

Application Level

When the user is allowed to access a specific application, the application is responsible for functional access to the data of the application. E.g., who is allowed to read, change add and delete customer data.

Network Level

Modern programs rely heavily on the network by opening network ports for communication. The network engineers are responsible for port access through VPN and firewalls. The security behind the port is again Application Level Security and is in control of the program that opened the port and waits for communication.

DevOps (Development + Operations)

As discussed in previous chapter, DevOps is a practice that unifies development and operations to **deliver software faster and more reliably**.

DevOps Workflow

Below a global list of actions that fit in the DevOps pipeline

Pipeline step	DevOps
Plan	Define requirements and goals
Code	Write application and infrastructure code, use version control.
Build	Compile, package artifacts and dependencies
Test	Run unit, integration, and system tests
Deploy	Roll out to environments using automation and repeatable procedures
Operate	Run and support the system, manage performance, availability, and incidents
Monitor	Track logs, usage and other metrics
Improve	Lessons learned, refine tooling and process

The focus is on **speed, stability, and collaboration**

Security in DevOps

Security is often handled **after development**, typically done by separate security teams using late stage testing (e.g., penetration testing in Test or User Acceptance Testing stages). This makes it reactive rather than proactive.

DevSecOps Workflow (Development + Security + Operations)

When we look at the pipeline steps from a **DevSecOps** point of view, we add security measures to the pipeline.

Pipeline step	Security
Plan	Identify risks, security requirements, compliance needs
Code	Use secure coding practices, manage secrets*, peer review
Build	Scan dependencies, check for vulnerable components*, prepare SBOM*
Test	Run security tests such as SAST*, DAST*, Pen Tests* and configuration checks
Deploy	Automatic setting of object and software security settings
Operate	Manage access, rotate secrets*
Monitor	Detect suspicious activity, audit events, vulnerabilities, and violations
Improve	Remediate findings, strengthen controls, update policies

Secrets

Secrets are confidential values that systems use to authenticate or encrypt things.

Examples:

- API keys
- Database passwords
- OAuth tokens

If a secret leaks, an attacker may get direct access to systems or data. Programmers often accidentally check-in configuration files with secrets.

Good practice is to never hardcode them in source code or config files and rotate them regularly. Use peer review and SAST scanning rules to search for typical patterns like *password=12345*. API Keys and other secrets should be rotated regularly.

SAST

SAST means **Static Application Security Testing**.

It analyzes source code, bytecode, or binaries **without running the application**.

What it looks for:

- SQL injection risks
- Unsafe input handling
- Hardcoded secrets
- Weak cryptography usage
- Insecure coding patterns

Usually it runs early in the pipeline, such as on pull requests or during build, and it helps catch security problems early, before the software is deployed. However, it may report findings that are theoretical or not actually exploitable in practice. SAST tooling like can often be configured to recognize false postitives.

DAST

DAST means **Dynamic Application Security Testing**.

It tests the application **while it is running**, usually by sending requests to it like an external attacker would.

What it looks for:

- SQL injection
- Cross-site scripting
- Authentication weaknesses
- Broken access control
- Security header issues

The scans usually run against a test, staging, or deployed environment and it is looking for issues that only appear when the app is actually running.

However, it cannot see the source code, so it may miss deeper internal issues.

SBOM

SBOM means **Software Bill of Materials**.

It is a structured inventory of all software components used in an application.

Typical contents:

- Libraries
- Packages
- Versions
- Suppliers

- Dependency relationships

Once the SBOM is known, it can easily be checked against a list of know vulnerabilities.

When a major issues appears like the one we had in Log4j, the SBOM helps you quickly answer:

- Do we use it?
- Where do we use it?
- Which versions are affected?

It improves transparency and speeds up incident response and compliance work.

Vulnerability scan

A **vulnerability scan** is an automated check for known security weaknesses.

It can scan:

- Dependencies
- Containers
- Operating systems
- Servers
- Cloud resources
- Network services

It detects:

- Known CVEs
- Outdated packages
- Insecure configurations
- Exposed services

It gives broad coverage and helps prioritize remediation but it only finds **known** problems, not brand-new or custom logic flaws.

Penetration Testing (Pentests)

Penetration testing is a controlled security assessment performed by specialists who simulate real-world attacks against a running system. It helps find weaknesses that automated tools may miss, especially issues involving business logic, access control, and combinations of smaller flaws.

In DevSecOps, pentesting is typically used periodically or before major releases, rather than on every build.

The penetration testers typically have access to all the information about the application so that they can try to break the application from every angle.

How they fit together

A simple way to see it:

- **SAST** : inspect the blueprint
- **DAST** : test the finished house
- **SBOM** : list every material used in the house
- **Vulnerability scan** : check whether any known faulty materials or weak spots are present
- **Penetration testing** : try to get into the house without the keys
- **Secrets** : the keys and alarm codes that must be protected

Use of Artificial Intelligence

AI can be used in many DevSecOps areas. For example, it can design tests that target the entry points in the application. AI Code review must be instructed to also look for security vulnerabilities.

More information about AI can be found in other chapters of this redbook.

Key takeaways

- **DevSecOps** means embedding security throughout the full software lifecycle, not adding it at the end.
- On **IBM i**, we have to use **OS**, **application** and **network** security to protect the data and application
- **DevOps** focuses on speed, stability, automation, and collaboration.
- **DevSecOps** adds security checks to every stage
- Key practices include:
  - protecting and rotating **secrets**
  - using **SAST** to scan code early
  - using **DAST** to test running applications
  - creating an **SBOM** to track components
  - Use **Penetration testing** to test the application like a real attacker
  - running **vulnerability scans** for known issues
- Main idea: **build fast, but securely at every step**.

AI and IBM Bob for IBM i Development

AI Fundamentals for IBM i

What is AI for IBM i Modernization?

What This Covers

This section introduces how Artificial Intelligence (AI) is transforming IBM i application modernization. You'll learn what AI means in the context of IBM i, why it's becoming essential for modernization efforts, and how AI-powered tools like IBM Bob™ are changing the way developers work with legacy systems.

What It Is

AI for IBM i modernization refers to the application of artificial intelligence technologies-including large language models (LLMs), natural language processing (NLP), and machine learning-to assist developers in understanding, maintaining, and transforming IBM i applications. These AI-powered tools act as intelligent assistants that can:

- **Understand legacy code** written in RPG, COBOL, CL, and DDS
- **Explain complex business logic** embedded in decades-old programs
- **Generate modern code** following current best practices
- **Analyze database schemas** and suggest optimizations
- **Automate documentation** creation and maintenance
- **Assist with code refactoring** from fixed-form to free-form
- **Provide contextual guidance** based on IBM i-specific knowledge

Unlike traditional development tools that require exact syntax and commands, AI tools understand natural language requests and can interpret intent, making them accessible to both experienced IBM i developers and those new to the platform.

Why It Matters

IBM i systems power critical business operations across industries, with many applications running code written 20-40 years ago. Organizations face several modernization challenges:

1. **Skills Gap**: Experienced RPG and COBOL developers are retiring, and fewer new developers are learning these languages
2. **Knowledge Transfer**: Business logic is often undocumented and exists only in the code itself
3. **Technical Debt**: Legacy code patterns make maintenance costly and risky
4. **Modernization Pressure**: Businesses need to integrate with modern technologies, APIs, and cloud services
5. **Time Constraints**: Manual code analysis and refactoring is time-consuming and error-prone

AI addresses these challenges by:

- **Accelerating understanding** of unfamiliar codebases
- **Reducing learning curves** for new developers
- **Automating repetitive tasks** like code conversion and documentation
- **Preserving institutional knowledge** by extracting and documenting business logic
- **Enabling incremental modernization** with AI-assisted refactoring
- **Improving code quality** through AI-powered analysis and suggestions

How It Works

AI for IBM i modernization typically operates through several key mechanisms:

1. Context-Aware Analysis

AI models are trained on vast amounts of code and documentation, including IBM i-specific languages and patterns. When you provide code or ask a question, the AI:

- Analyzes the syntax and structure
- Identifies patterns and relationships
- Accesses relevant IBM i documentation and best practices
- Generates contextually appropriate responses

2. Natural Language Interface

Instead of memorizing complex commands, developers can:

- Ask questions in plain English: "Explain what this RPG program does"
- Request specific actions: "Convert this fixed-form RPG to free-form"
- Get guidance: "How do I optimize this SQL query for Db2 for i?"

3. Code Generation and Transformation

AI can generate or transform code by:

- Understanding the intent behind legacy code
- Applying modern coding patterns and best practices
- Maintaining business logic while improving structure
- Generating test cases and documentation

Important: Developers remain fully responsible for all code in their applications. AI-generated code is a suggestion that must be reviewed, tested, and approved by the developer before use. The developer owns the final code, not the AI.

4. Continuous Learning

Modern AI tools can:

- Learn from your codebase and coding patterns
- Adapt suggestions to your organization's standards
- Improve recommendations based on feedback
- Stay updated with latest IBM i features and best practices

5. Integration with Development Workflow

AI tools integrate directly into:

- IBM Bob, VS Code and other IDEs
- Command-line interfaces
- CI/CD pipelines
- Code review processes

Key Takeaways

- **AI for IBM i modernization uses advanced language models to understand, explain, and transform legacy IBM i code**, making modernization faster and more accessible
- **It addresses critical challenges** including the skills gap, knowledge transfer, and technical debt by acting as an intelligent assistant for developers at all experience levels
- **AI tools work through natural language interfaces**, eliminating the need to memorize complex syntax while providing context-aware analysis, code generation, and continuous learning capabilities
- **This technology enables incremental modernization** by automating repetitive tasks, preserving business logic, and helping organizations maintain their IBM i investments while adopting modern development practices
- **IBM Bob is a leading example** of AI-powered assistance specifically designed for IBM i developers, offering specialized knowledge in RPG, COBOL, CL, Db2 for i, and IBM i system services and applications

What is AI-Powered Testing?

What This Covers

This section explores how artificial intelligence is transforming software testing on IBM i. You'll learn what AI-powered testing means in practice, understand how it differs from traditional testing approaches, and discover how AI tools like IBM Bob can accelerate test creation, improve test coverage, and enhance quality assurance for IBM i applications.

What It Is

AI-powered testing refers to the application of artificial intelligence-particularly large language models and machine learning-to automate and enhance various aspects of software testing. Unlike traditional testing tools that rely on predefined scripts and manual test case creation, AI-powered testing tools can:

Intelligent Test Generation: Automatically create comprehensive test cases by analyzing code structure, business logic, and data flows. For IBM i applications, this means generating tests for RPG programs, SQL procedures, and CL scripts based on understanding the code's intent and potential failure points.

Test Data Creation: Generate realistic test data that covers edge cases, boundary conditions, and typical usage patterns. AI can analyze existing data structures in Db2 for i and create appropriate test datasets that exercise all code paths.

Automated Test Maintenance: Update test cases automatically when code changes, reducing the maintenance burden that often causes test suites to become outdated. AI can identify which tests need modification based on code changes and suggest or implement updates.

Defect Prediction: Analyze code patterns to identify areas most likely to contain defects, allowing teams to focus testing efforts where they'll have the greatest impact. This is particularly valuable for legacy IBM i code where certain patterns are known to be error-prone.

Natural Language Test Creation: Allow testers and developers to describe test scenarios in plain language, which the AI then converts into executable test code. For example: "Test the order processing program with an invalid customer number" becomes a complete test case with setup, execution, and validation.

These capabilities integrate with existing IBM i testing frameworks like RPGUnit and iRPGUnit, enhancing rather than replacing traditional testing approaches.

Why It Matters

AI-powered testing addresses critical challenges facing IBM i development teams:

Accelerating Test Creation: Manual test creation is time-consuming, often taking as long as writing the code itself. AI can generate initial test suites in minutes rather than hours or days, allowing developers to focus on refining tests rather than creating them from scratch. This acceleration is crucial for teams adopting DevOps practices where rapid, frequent deployments require comprehensive automated testing.

Improving Test Coverage: Traditional testing often misses edge cases and unusual code paths because humans naturally focus on expected scenarios. AI analyzes all possible execution paths and generates tests for scenarios developers might not consider, including error conditions, boundary values, and unusual data combinations. This comprehensive coverage reduces production defects and improves application reliability.

Reducing Maintenance Burden: Test maintenance is a major pain point-as applications evolve, test suites become outdated and eventually abandoned. AI-powered tools can automatically identify which tests are affected by code changes and suggest or implement updates, keeping test suites current and valuable. This is particularly important for IBM i applications where code may have been maintained for decades.

Bridging Skills Gaps: Writing effective tests requires deep understanding of both the application and testing best practices. AI tools act as testing experts, helping less experienced developers create high-quality tests by suggesting appropriate assertions, identifying missing test cases, and recommending testing patterns. This democratizes testing expertise across the team.

Enabling Legacy Code Testing: Many IBM i applications lack automated tests because they were written before testing frameworks existed. Manually creating tests for decades-old code is daunting. AI can analyze legacy RPG, COBOL, and CL programs to generate initial test suites, making it feasible to add test coverage to legacy applications during modernization efforts.

How It Works

AI-powered testing for IBM i operates through several integrated capabilities:

1. Code Analysis and Understanding

AI models analyze source code to understand:

- Program structure and control flow
- Data dependencies and transformations
- Business logic and validation rules
- Integration points with other programs and databases
- Potential failure scenarios and edge cases

2. Test Case Generation

Based on code analysis, AI generates:

- **Unit tests** for individual procedures and programs using RPGUnit or iRPGUnit
- **Integration tests** that verify interactions between programs and database operations
- **Regression tests** that ensure existing functionality remains intact after changes
- **Edge case tests** covering boundary conditions, null values, and error scenarios

3. Test Data Synthesis

AI creates appropriate test data by:

- Analyzing database schemas and constraints
- Generating realistic data that exercises all code paths
- Creating boundary and edge case values
- Ensuring referential integrity across related tables

4. Natural Language Interface

Developers and testers interact conversationally:

- "Generate unit tests for this RPG program"
- "Create test data for the customer order scenario"
- "Add tests for error handling in this procedure"
- "Update tests affected by this code change"

Important: Developers remain fully responsible for reviewing and validating all AI-generated tests. AI-generated tests are suggestions that must be examined, potentially modified, and approved before use in production test suites.

5. Continuous Learning and Adaptation

Advanced AI testing tools:

- Learn from existing test patterns in your codebase
- Adapt to your organization's testing standards
- Improve test quality based on feedback
- Identify patterns in defects to generate better tests

Key Takeaways

- **AI-powered testing uses machine learning to automate test creation, maintenance, and optimization**, dramatically reducing the time and expertise required to build comprehensive test suites for IBM i applications
- **It addresses critical testing challenges** including slow test creation, poor coverage, high maintenance costs, and skills gaps by acting as an intelligent testing assistant that understands code and generates appropriate tests
- **AI testing tools integrate with existing frameworks** like RPGUnit and iRPGUnit, enhancing rather than replacing traditional testing approaches while supporting natural language interactions for ease of use
- **The technology is particularly valuable for legacy IBM i code** where automated tests often don't exist, enabling teams to add test coverage during modernization efforts without massive manual effort
- **IBM Bob provides AI-powered testing capabilities** specifically designed for IBM i, understanding RPG, COBOL, CL, and SQL to generate contextually appropriate tests that follow platform best practices

Working with IBM Bob

How to Use Bob for Code Analysis

What is AI Code Analysis

AI code analysis is the use of artificial intelligence to evaluate source code for quality, performance, security, and maintainability. Unlike traditional static analysis tools that rely on predefined rules, AI-driven analysis understands patterns, context, and developer intent. It identifies not only syntax errors, but also deeper issues such as inefficient logic, duplication, and structural weaknesses.

In IBM i environments, AI code analysis applies to RPG, CL, and SQL programs, where complex business logic and long-lived applications are common. AI models are trained on large codebases and best practices, enabling recommendations that reflect real-world development patterns. This allows developers to receive guidance similar to an experienced reviewer, delivered consistently and in real time.

IBM Bob brings AI code analysis directly into the development workflow, shifting teams from reactive debugging to proactive quality improvement.

Why it Matters

The quality of enterprise code directly impacts system reliability, maintainability, and business continuity. Manual code reviews are valuable but often inconsistent and time-consuming. In IBM i environments, where RPG and CL applications frequently support critical operations, even small inefficiencies or defects can create significant downstream impact.

IBM Bob provides a consistent and scalable approach to code analysis. It identifies defects early, highlights performance issues, and enforces coding standards across the team. This reduces rework, shortens feedback cycles, and improves overall delivery quality.

Bob also strengthens long-term maintainability. By analyzing complexity, dependencies, and duplication, it surfaces areas that require refactoring. This is especially important in legacy systems where knowledge may be limited or fragmented.

Another important factor is how developers interact with Bob. Because Bob is AI-driven, the quality of the prompt directly influences the quality of the analysis. Clear and specific prompts produce targeted and actionable recommendations, while vague prompts result in general feedback. Effective prompting allows teams to guide Bob toward performance tuning, refactoring, or impact analysis.

For organizations adopting modern practices such as DevOps and CI/CD, Bob ensures that IBM i applications are included in automated quality gates. This allows legacy and modern systems to evolve together without sacrificing standards or stability.

How it Works

Using Bob for code analysis combines full-repository awareness with AI-driven analysis using a unified mode, skills, and workflows. This approach enables developers to move from simple code inspection to guided, intelligent problem-solving.

1. Setup and Configuration

Load your entire repository into the Bob workspace to maximize the effectiveness of AI-driven analysis. Bob delivers the most accurate and valuable insights when it has full visibility into your codebase, including cross-program dependencies, shared structures, and database interactions.

During code analysis, Bob examines the entire repository—not just individual programs—to ensure it has all necessary context. This includes relationships between programs, files, fields, and database objects. Without full repository visibility, analysis results may be incomplete or lack accuracy.

After loading the repository:

- Connect IBM Bob to IBM i source libraries or version control systems
- Define analysis scope
- Configure coding standards and performance rules
- Apply organizational security and maintainability policies

This configuration ensures IBM Bob aligns with enterprise development standards.

2. Select Analysis Approach (Mode, Skills, and Workflows)

IBM Bob operates using a single unified mode that combines analysis, development, and modernization capabilities into one experience. Instead of switching between multiple modes, developers interact within one environment and use skills and workflows to perform specific tasks.

Single Developer Mode

The unified Developer mode supports:

- Code analysis and understanding
- Code modification and refactoring
- Impact analysis and dependency tracing
- Modernization and transformation activities

This simplifies the user experience and removes the need to switch contexts.

Skills (Task Execution)

Skills are the primary way to perform actions in Bob. They replace traditional commands and represent specific, reusable capabilities.

Examples of skills include:

- Refactor code for readability
- Analyze variable usage
- Identify performance issues
- Trace field-level dependencies

Skills may be:

- **Visible** – selectable by developers
- **Hidden** – automatically used by IBM Bob during analysis

Workflows (Guided Execution)

Workflows provide structured, multi-step interactions to solve complex problems. They guide developers through decision points and apply the appropriate skills at each step.

Workflows are especially useful for:

- Code modernization
- RPG to SQL transformations
- Handling OPM vs ILE program differences
- Managing conditional logic and design decisions

Example workflow behavior:

- Detect a cycle-based RPG program
- Prompt for transformation decisions
- Guide file and structure creation
- Execute skills to complete the process

How They Work Together

- **Mode** provides the unified working environment
- **Skills** execute individual tasks
- **Workflows** coordinate multi-step processes

For effective code analysis:

- Use **skills** for targeted analysis tasks
- Use **workflows** for complex or multi-step scenarios
- Use clear prompts to guide IBM Bob toward desired outcomes

This model replaces earlier multi-mode and command-based approaches.

3. Code Scanning

Once the analysis approach is defined, initiate a scan. IBM Bob parses source code and evaluates it within the unified Developer mode.

- Analysis and understanding occur within the single mode
- Skills and workflows determine how analysis is executed

Because IBM Bob evaluates the full repository context during scanning, it can correlate changes across programs, detect hidden dependencies, and provide more accurate recommendations than isolated file analysis.

Prompting plays a critical role at this stage. Clearly define what you want IBM Bob to analyze.

Examples:

- Identify performance issues in this RPG program
- Analyze downstream impact of changing this field
- Recommend modernization improvements for this module

The better the prompt, the better the results.

4. Analysis and Recommendations

IBM Bob generates detailed recommendations aligned with the selected skills and workflows.

Typical recommendations include:

- Simplifying complex or nested logic
- Refactoring duplicated or tightly coupled code
- Improving readability and maintainability
- Identifying potential runtime risks
- Highlighting modernization opportunities

Each recommendation includes context and explanation so developers understand both the issue and the suggested improvement.

5. Reporting and Visualization

IBM Bob produces structured reports that summarize findings and highlight areas of concern.

Common metrics include:

- Code complexity
- Duplication levels
- Dependency relationships
- Modernization opportunities

These reports provide visibility to development teams and leadership, helping prioritize improvements.

6. Integration with Development Workflow

IBM Bob integrates with development pipelines and version control systems.

Code analysis can occur during:

- Code commits
- Pull requests
- Build processes
- Release validation

This ensures continuous quality enforcement throughout the development lifecycle.

7. Iterative Improvement

IBM Bob improves over time through continuous use and feedback.

Teams can:

- Refine analysis rules
- Adjust thresholds
- Update coding standards
- Align recommendations with internal policies

This ensures IBM Bob evolves alongside the organization.

Bob Capability Levels (Base and Premium)

IBM Bob provides different capability levels depending on deployment and licensing.

Base Capabilities

- Core AI-driven code analysis
- Unified Developer mode
- Skills for task execution
- Basic workflow support
- Code quality and maintainability recommendations

Premium Capabilities

- Native connectors for direct IBM i integration
- Advanced workflows for modernization and transformation
- Deeper repository-wide analysis

- Enhanced scalability for large codebases
- Expanded automation capabilities

Premium capabilities are designed for enterprise-scale modernization and complex system analysis.

Key Takeaways

- AI code analysis provides deeper insight than traditional rule-based tools
- IBM Bob examines the entire repository for accurate and complete analysis
- Full repository visibility significantly improves results
- A single unified mode simplifies the developer experience
- Skills replace commands and execute specific tasks
- Workflows guide complex, multi-step processes
- The quality of prompts directly impacts the effectiveness of analysis
- Integration into development workflows enables continuous quality enforcement
- IBM Bob supports both modernization and long-term sustainability of IBM i applications

How do I use IBM Bob for documentation?

What It Is

IBM Bob is an analysis-driven documentation capability for IBM i environments. It generates technical documentation directly from source code by first building an accurate understanding of how a system works. Instead of relying on manually written descriptions (often incomplete or outdated), IBM Bob reads programs, service programs, and dependencies to construct a model of actual system behavior.

From this model, IBM Bob produces multiple forms of documentation:

- Program headers describing purpose, parameters, and dependencies
- Procedure-level documentation explaining inputs, outputs, and logic
- Architecture documentation showing how components interact

Outputs can be formatted for inline source comments, markdown, or formal technical documents. The key principle is simple: documentation is generated from verified understanding, not written from assumption.

Why It Matters

Documentation gaps are common in IBM i systems. Over time, complexity increases while documentation becomes outdated or disappears entirely. This creates a gap between what the system does and what teams understand.

That gap leads to:

- Slow onboarding for new developers
- Increased time spent reverse-engineering code
- Higher risk when making changes
- Dependency on a small number of experienced developers

IBM Bob reduces this risk by making documentation faster, more accurate, and repeatable:

- Accuracy improves because documentation is derived from source code
- Speed increases by replacing manual analysis with automated understanding
- Consistency is enforced through structured prompts and workflows
- Modernization is enabled with clear visibility into system behavior

The result is a practical shift: documentation becomes part of the development process instead of an afterthought.

How It Works

IBM Bob follows a structured, iterative workflow:

1. Establish Context

Load relevant source members. This may include a single RPG program, a service program with prototypes, or multiple components for broader analysis.

2. Analyze First

Prompt IBM Bob to analyze the code before requesting documentation. Focus on:

- Program purpose
- Key procedures and logic
- File and database dependencies
- Data flow and error handling

This creates the foundation for accurate output.

3. Validate the Analysis

Review the results. Correct any missing or incorrect interpretations. This step ensures the model is reliable.

4. Generate Documentation in Sections

Produce documentation in logical parts:

- Program header
- Procedure documentation
- Usage or interface details
- Architecture summary

This improves clarity and review quality.

5. Review and Format

Refine each section and specify the output format (comments, markdown, or prose). First-pass output should always be treated as a draft.

Hands-On: Generating Documentation with IBM Bob

The following example shows how to document an RPG program.

Scenario: Generate a program header and procedure documentation for an existing RPG program.

Step 1: Load the Source

Provide the RPG source member to IBM Bob. If applicable, include related prototypes or copybooks.

Step 2: Run Analysis Prompt

Start with analysis, not documentation.

Example prompt: "Analyze this RPG program. Summarize its purpose, key procedures, file dependencies, and error handling."

Step 3: Validate Output

Review the response:

- Does the purpose match expectations?
- Are all files and dependencies identified?

- Are key procedures correctly described?

Refine with follow-up prompts if needed.

Step 4: Generate Program Header

Once validated, request the header.

Example prompt: "Generate a standard IBM i program header including purpose, parameters, and dependencies based on the validated analysis."

Step 5: Generate Procedure Documentation

Next, document procedures.

Example prompt: "Document the procedures in this program, including inputs, outputs, and processing logic."

Step 6: Review and Insert

Review the generated documentation. Adjust wording or structure as needed, then insert into the source or publish to your documentation platform.

Common Mistakes to Avoid

- Skipping the analysis step
- Requesting all documentation in a single prompt
- Not validating dependencies or logic
- Using vague prompts that produce generic output

Key Takeaways

- **Start with analysis, not documentation** - Understanding the code is the foundation of accurate output
- **Use structured, repeatable steps** - Load, analyze, validate, generate, and refine
- **Generate documentation in sections** - Smaller outputs improve clarity and review quality
- **Validate everything** - IBM Bob accelerates the process but does not replace engineering judgment
- **Standardize prompts and formats** - Consistency across teams improves long-term maintainability
- **Integrate documentation into development** - The best time to document is during active development or modernization

IBM Bob transforms documentation from a manual, inconsistent task into a structured engineering process. When used correctly, it closes the gap between code and understanding, enabling teams to maintain and evolve IBM i systems with greater speed and confidence.

Implementing AI Tools

How to Implement AI Testing Tools for IBM i Applications

What This Covers

You'll learn how to implement AI-powered testing tools for IBM i applications, covering setup, test generation, and framework integration. For integration testing and end-to-end testing, see the **Test Driven Development with Automation**.

Before You Start

- ☐ Access to AI-powered testing tools (e.g., IBM Bob with testing capabilities)
- ☐ Existing testing framework installed (RPGUnit or iRPGUnit) or willingness to set one up
- ☐

Sample programs or modules to create initial tests for

- ☐ Understanding of your application's critical business logic and test requirements
- ☐ Team agreement on code review processes for AI-generated tests

Steps

1. Set Up Your Testing Framework and AI Tool Integration

Establish the foundation for AI-powered testing:

- **Install Testing Framework**: Set up RPGUnit or iRPGUnit. Create a dedicated test library for storing test programs
- **Configure AI Tool**: Install your AI testing tool (IBM Bob or VS Code with AI extensions). Set up authentication and workspace access
- **Organize Tests**: Create a directory structure mirroring source code with `/tests` suffix
- **Define Standards**: Document test naming conventions (e.g., `TEST_programname`) and assertion patterns
- **Set Up Test Environment**: Create test database schemas separate from production

What to expect: Setup takes 2-4 hours. Once configured, AI generates framework-specific test code. Your isolated test environment prevents production data corruption.

2. Generate Initial Unit Test Cases Using AI

Use AI to create your first automated unit tests:

- **Start Simple**: Select a straightforward program. Ask: "Generate unit tests for this RPG program using RPGUnit"
- **Review Structure**: Examine generated setup, test cases, assertions, and teardown logic
- **Customize Scenarios**: Request specific tests: "Add tests for boundary conditions" or "Generate error handling tests"
- **Generate Test Data**: Ask: "Generate test data for the customer table that exercises all validation rules"
- **Compile and Run**: Verify functionality and address any errors

What to expect: AI generates comprehensive test suites in minutes versus hours manually. Expect 20-30% of assertions to need adjustment. Start with 2-3 simple programs before tackling complex modules.

> **Important**: AI-generated tests must always be double-checked as AI is non-deterministic and can be biased. Auto-generated tests always carry operational risk and require human validation.

3. Integrate AI Testing into Your Development Workflow

Embed AI-powered testing into daily development. For comprehensive testing strategies including integration and end-to-end testing, refer to the **Test Driven Development with Automation** chapter.

- **Test-Driven Development**: Generate unit tests first: "Create tests for a procedure that validates customer credit limits"
- **Legacy Code Coverage**: Generate test suites for existing programs without tests
- **Regression Tests**: Create tests that reproduce bugs when fixing issues

What to expect: Teams see 40-60% unit test coverage improvement within 2-3 months. AI catches edge cases developers might miss. Remember that AI-generated tests require thorough review as AI can produce biased or incorrect assertions.

Verify It Worked

Confirm successful AI testing tool implementation:

- **Unit Test Coverage**: Achieved 70-80% unit test coverage for critical business logic, up from previous baseline
- **Team Adoption**: At least 70% of developers regularly use AI tools for test generation
- **Test Quality**: AI-generated tests pass code review after thorough validation and necessary corrections

Key Takeaways

- **This chapter focuses on AI-powered unit testing only**-for integration testing and end-to-end testing strategies, refer to the **Test Driven Development with Automation** chapter
- **Proper framework setup is essential**-install RPGUnit/iRPGUnit correctly before AI can generate framework-specific unit tests, ensuring consistency across the team
- **AI generates tests quickly but requires thorough validation**-AI is non-deterministic and can be biased, so developers must carefully review all generated tests for correctness. Auto-generated tests always carry operational risk

How to Decide When to Use AI Tools for IBM i Development

What This Covers

You'll learn when to use AI tools for IBM i development tasks, covering decision criteria for maximizing AI value.

Before You Start

- ☐ Understanding of your current development task or challenge
- ☐ Access to AI-powered development tools (e.g., IBM Bob, VS Code with AI extensions)
- ☐ Familiarity with your organization's AI usage policies and code review requirements
- ☐ Clear understanding of task complexity, time constraints, and quality requirements
- ☐ Knowledge of available traditional tools and approaches for comparison

Steps

1. Assess Task Characteristics and Complexity

AI tools excel at specific types of work. Use AI when explaining unfamiliar legacy code (RPG, COBOL, CL), performing repetitive tasks like converting fixed-form to free-form RPG, or tackling knowledge-intensive activities where you lack deep platform expertise. AI also accelerates exploratory work, brainstorming modernization approaches, and time-sensitive deliverables like documentation or prototyping.

What to expect: Tasks with multiple favorable characteristics are prime candidates. Avoid AI for simple changes or well-understood business logic where manual work is faster.

2. Evaluate Risk and Quality Requirements

Risk tolerance determines AI appropriateness. Use AI cautiously for production-critical logic, preferring it for supporting code rather than core algorithms. Avoid AI for security-sensitive operations like authentication or encryption unless you can validate thoroughly. For performance-critical paths, use AI for initial suggestions but validate with profiling. Exercise caution with compliance-regulated code subject to audit requirements.

What to expect: Categorize tasks as low-risk (documentation, learning), medium-risk (features, refactoring), or high-risk (security, compliance) to determine validation requirements.

Important: Developers remain fully responsible for all code in their applications. AI-generated code is a suggestion that must be reviewed, tested, and approved by the developer before use. The developer owns the final code, not the AI.

3. Compare AI Efficiency Against Traditional Approaches

Use AI when it saves at least 30% of total time including prompting and review. AI often generates more comprehensive documentation and consistent code patterns than manual approaches. It accelerates learning new technologies by providing explanations and examples. However, skip AI if you have proven templates to adapt or if validating AI output is more complex than creating code manually.

What to expect: AI excels when tasks are time-consuming but validation is straightforward.

4. Consider Team and Organizational Context

Team capabilities and organizational policies influence AI adoption. Use AI extensively if your team crafts effective prompts and validates output well, but ensure adequate review capacity since AI code needs rigorous examination. AI helps preserve knowledge and accelerate onboarding, though avoid it when building deep expertise is the goal. Respect code sharing policies, using on-premises tools for restricted code, and consider stakeholder expectations regarding AI disclosure.

What to expect: Teams with strong governance use AI broadly. Developing teams start with low-risk cases and expand gradually.

5. Make the Decision and Plan Validation

Use AI when task characteristics align (repetitive, knowledge-intensive), risk is acceptable, efficiency gains exceed 30%, and your team supports it. Avoid AI when tasks are simple manually, risk is unacceptable, validation is complex, or policies prohibit use. Consider hybrid approaches where AI generates initial code and you manually refine critical sections. Always define your validation approach-code review, tests, or benchmarks-and have a fallback strategy if AI output proves unusable.

What to expect: Document your approach for significant tasks and track outcomes to refine criteria over time.

Verify It Worked

Confirm your decision-making process is effective by measuring time efficiency (30-70% savings for appropriate use cases), quality metrics (AI code meeting standards with comparable defect rates), and team confidence in validating AI output. Team members should gain knowledge from AI interactions rather than accepting output blindly. Similar tasks should receive consistent decisions across team members, with minimal instances of AI-generated code requiring complete rewrites.

Key Takeaways

Use AI whenever possible for code understanding, repetitive work, and knowledge-intensive activities, but apply strict validation for production-critical, security-sensitive, or compliance-regulated code. AI should save at least 30% of total time (including review) to justify use. Team proficiency, review capacity, and organizational policies shape when AI is appropriate. Hybrid approaches-using AI for initial generation then manually refining critical sections-often deliver the best results by combining efficiency with expertise.

Security and Best Practices

What are the Security Best Practices when Using AI?

What It Is

Security considerations must be woven throughout the development and deployment of any type of AI project. From the simple AI-enabled chatbot to Agentic AI, it is imperative that a project goal be to ensure the integrity of your organization's data throughout the entire project.

Why It Matters

The saying 'garbage in, garbage out' could not be more true when describing the effect of data being fed into an AI project when the data shouldn't be a part of it or is inaccurate. When training a model, data cannot be 'unlearned.' Confidential or inaccurate data fed into a model prevents successful project outcomes. When training a model or updating a model, bias can be introduced, inaccurate data introduced due to accidental or purposeful inaccurate changes by the trainers or bad actors. When developing Agentic AI projects, Agents with too much access can go rogue causing business disruption. Unsecured git repositories can allow bad actors to inject bad code to purposefully cause damage or to leak data. Good security practices throughout an AI project can help prevent all of these issues.

AI projects are exciting and everyone wants to jump right in. Unfortunately, this zeal to use and be a part of the newest technology will often cause the security aspects of the project to be completely ignored. We encourage you to ask the security questions PRIOR to the project taking off and be an advocate for implementing sound security practices from start to finish.

How It Works

Familiarize yourself with your organization's security policy and regulatory requirements for the data

If you aren't aware of the regulatory requirements for the data being used for your AI project, it's very easy to violate them. And unfortunately, ignorance of a law or regulation doesn't excuse a violation or the fines that follow. It's also easy to violate the use of Personally Identifiable Information (PII) and/or expose it inappropriately within your organization or even to the world if you don't know what constitutes PII or its appropriate use. Finally, your Organization has confidential data that makes it unique. Understand whether this data is allowed to be used and what you must do to protect it.

Only input your Organization's Data in the LLM being used internally by your Organization

Well known data leaks have occurred when developers, eager to explore AI, used company confidential data in a free LLM, making it available to anyone in the world using that LLM. Don't be that person! You cannot retrieve your data. Once presented to an LLM, it can't 'unlearn' it!

Secure your Organization's data being used in your AI project

Proper access control settings are key to the security of your AI project. If the access control settings of your data used for initial training or the data used to update your model can be updated, it can be subject to data poisoning or insertion of bias. An open gist containing your Agentic AI tool definitions can be subjected to code injection. Or an open MCP server configuration file may allow for account takeover. These are just a few of the vulnerabilities created when the default access is not restricted. Every piece of data fed into a model, source code and configuration files must be protected.

At every step, use every security tool at your disposal

Security features are often integrated into your toolset. While security is usually not a feature developers enjoy, we're simply asking you to use what's at your disposal and not turn it off or ignore it. For example, GitHub comes with security features. Likewise, your IDE may come with code scanning technology such as the technology that comes

with the Bob plug-in. Like all code issues, the earlier you find them in the development process, the easier they are to address. The same applies to security vulnerabilities.

Don't use or encourage Shadow AI

The challenge for IT used to be Shadow IT where parts of the business develop or purchase their own applications outside of the regular purchasing process. The problem with this practice is that they don't have the regular business support in debugging problems, backing up data, upgrading the application, etc. The problem is now compounded with advent of Shadow AI, where groups within an Organization have started to use easily obtained LLMs to perform their own business processes. What's the harm with this? It is far more likely that these groups are going to leak confidential data via a commercially available LLM as well as leave unsecured source scattered around the Organization or in their own gists. Please do not encourage the use of Shadow AI and don't be one of those individuals that go around the practices and policies the organization has put in place defining how AI is to be used.

When developing Agents, consider when - or if - a Human needs to be involved

Agents are being developed to replace tasks Humans perform but does that mean that humans are totally removed from the process? It depends on the task the Agent is performing. HITL - Human-in-the-Loop is when a task is performed by an Agent and then is reviewed by a human. An example of this may be a highly regulated industry that requires review and approval at each step of a process (for example, the pharmaceutical industry.) Slightly less involvement is a form of HOTL - Human-on-the-Loop. The best example of this we've observed is supervised self-driving cars. Finally, the total lack of human involvement is called HOOTL - Human-out-of-the-Loop. We suggest that this approach only be used once the process is fully tested and you can prove that the Agents will stick to their tasks and not go rogue.

IBM Bob's Security Recommendations

IBM Bob comes with some security recommendations described at Security guidelines. Here are some highlights:

- Review and limit auto-approve settings. While the constant back and forth of the approval process may tempt you to auto-approve your workflow, you can open up your Organization to data loss. Auto-approving command line access is particularly risky. Know your risk before you enable auto-approve.
- IBM Bob, and all other LLMs, can hallucinate. Output must be verified.
- LLMs cannot keep a secret! They are not like your best friend to whom you can tell your secrets and know they will never be passed along. Even a temporary inclusion can cause that information to escape and create a risk for your Organization.
- IBM Bob recommends tracking all actions for accountability. And we definitely agree with that premise but you need to understand what is and what is not logged. Do not expect an immutable audit log. The log is more of a tracking mechanism for the actions that have been taken. Great for debugging issues but may not be the audit log that will meet your compliance requirements.

Key Takeaways

- Consider security at every step in the development process
- Secure your data and environment to help the reduce the risk of it being compromised
- Use every security tool available to you

What are Security Best Practices when Developing AI Agents?

What It Is

Agents - whether running tasks on IBM i or any other platform - must run with a user account created on that platform. Given the future of Agents running tasks on their own and deciding on its own to run others, it's imperative that security guardrails be put in place as Agents are developed.

Why It Matters

Agentic AI has the potential to make significant improvement in workflows - taking away mundane tasks and automating a lot a work currently done manually. As a human, we have roles that we play within an organization and are authorized (allowed to do things) based those roles. AI Agent profiles - whether on IBM i or any other platform - that are created to represent a human or to perform human-like tasks, should be authorized similarly to their human counterparts. Some argue that even more - not fewer - restrictions should be placed on these profiles to ensure the Agents stay within the role for which they're intended and don't overstep their intended place in the organization - in other words ... don't go rogue.

How It Works

First, ensure the IBM i profile representing an AI Agent is created with least privilege access - the complete opposite of what you might be tempted to do. One might be tempted to create a superuser profile with all privileges (that is, all special authorities including *ALLOBJ) to ensure that the profile has sufficient authority to perform its task. The problem with this approach is that there's no guarantee with an AI Agent that it will stay within it's intended task. In fact, that's part of the point of an agent - that it will learn and start to take on more tasks. Creating the profile as a superuser allows it to not just learn new tasks and perform but to also walk throughout the system doing whatever it wants to ... including cancelling scheduled jobs, ending others' jobs, downloading data, modifying data and worse, deleting data, deleting backups, changing users' passwords and/or disabling or deleting user profiles, ending auditing, etc. As a superuser one can do anything. Hopefully you can understand why this is a bad idea when it comes to creating a profile for an AI Agent!

To create a profile with least privilege access, we recommend the following settings. This is an example of creating an Agent profile for running Inventory processes:

```
CRTUSRPRF USRPRF(AGT\_INV)
PASSWORD(\*NONE)
PWDEXPITV(\*SYSVAL)
STATUS(\*DISABLED)
SPCAUT(\*NONE)
GRPPRF(\*NONE)
INLPGM(\*NONE)
INLMNU(\*SIGNOFF)
ATNPGM(\*NONE)
LMTCPB(\*YES)
TEXT('AI Agent running inventory processes')
AUT(\*EXCLUDE)
```

This is an example of a profile that may be only running processes or accessing the system via APIs and is authenticating via a Modern authentication technique such as OAuth. If the profile requires a password, make the password as long as the system it's connecting to allows. If the password is provided on a connection, ensure the connection is made using an encrypted session (using TLS 1.3) and the source storing the password is secured.

What Do You Do if the Operating System Function Requires *ALLOBJ?

We're recommending least privilege access but what happens if a special authority is required for the Agent to perform its tasks? For example, perhaps the Agent is tasked with identifying old objects no longer in use. If the Agent runs the QSYS2.OBJECT_STATISTICS or QSYS2.IFS_OBJECT_STATISTICS SQL services, the result will only list the objects for which the Agent has some authority (not *EXCLUDE.) You could think about wrappering this service in a CL program that adopts but adopted authority is ignored when accessing IFS objects. Enter Function Usage - a system feature that can mitigate the need for *ALLOBJ and other special authorities in some cases. In this specific case, the Agent could be granted access to the QIBM_LIST_ALL_OBJS and QIBM_LIST_ALL_OBJS_SQL function usage identifiers which will allow objects to be listed even if the profile has no authority or is explicitly excluded. See this IBM doc for the specific details of this function. See Appendix H of the IBM Security Reference manual for a list of all functions that may allow you to mitigate the need for *ALLOBJ and other special authorities such as *IOSYSCFG. For a full description of Function Usage see section IBMI-SEC-CONCEPT-197.

Create one Agent per Task

To ensure the ability to create profiles with only the authority they need for their task, create one profile per task. Or at least one profile per role. Overloading the same profile with multiple roles will cause some profiles to run with more authority than required, potentially allowing them to perform tasks outside of their role.

Using separate profiles also makes debugging failed jobs easier and makes audit journal entries more meaningful.

Should Agents be a Member of a Group Profile?

Since an Agent is taking on the tasks of a Human, you have to determine how to authorize the Agent's profile to the same tasks as the Human's profile. One obvious choice is to make the Agent's profile a member of the Human's group. Or is it? Doing so would certainly give the Agent the same access. But before doing that, you'll need to think through whether you want the Agent to have the full access and capabilities of the Human. Considerations include:

- All special authorities granted to a group filter down to the members
- Any objects owned by a group are also owned by the members

If neither of these two things are of concern, then you may want to make the Agent a member of the Human's group. However, if these items are giving away too much access or you're just not comfortable doing this (like us), we recommend creating a group profile for the Agent assigning only the special authorities required and granting authority only to the objects needed for the Agent's task. By creating a group profile for the Agent, you are following the premise of Role-based Access (RBAC) where each role is represented by a group. If another Agent is created with the same responsibilities (that is, is in the same role) you can simply assign the New Agent to this group and their authorities and capabilities are automatically assigned (via the group).

Agents Should Not Talk to All Other Agents

Also, the popular thought is that Agents should talk to all other Agents to find efficiencies. We encourage you to reconsider that, taking a 'deny by default approach' allowing Agents to talk to other Agents only when it makes sense. For example, should Agents running processes associated with Inventory have a need to talk to an Agent running HR processes? Likely not. Given the sensitivity of HR data and processes, only Agents with a specific requirement should be allowed to talk to the HR Agent. Access to Agents should follow the same access determinations as access to data - and allow access only for those whose role requires it.

Put as Many Guardrails in Place as Possible

Since we really don't know what an Agent might eventually try to do (that's the allure of Agentic AI after all), we encourage you to put as many guardrails in place as possible to confine an Agent to its intended role. In addition to creating the Agents' user profile (*USRPRF) with the attributes listed earlier, now's the time to implement multiple layers of defense - that is, if the Agent gets around one restriction, what's the next restriction that will catch and prevent the Agent from going rogue? Suggestions include, specifically granting Agent profiles private authority *EXCLUDE when data is *PUBLIC *USE to prevent access to data for which the Agent has no business reason to access it. Same goes for directories containing data. If the directory remains *PUBLIC DTAAUT(*RX), grant the Agent a private authority DTAAUT(*EXCLUDE) OBJAUT(*NONE) to prevent access. Ensure the Agent profile has no access to any functions in Function Usage (including the ability to use Navigator for i and the parts of Access Client Solutions (ACS) it shouldn't be using.) And if your organization has implemented exit point software from one of the security vendors, now's the time to use it. Put rules in place to specifically disallow Agents' access to operations such as FTP, ODBC, DDM, etc.

> **Note:** If you choose to create a profile with *ALLOBJ special authority, very few guardrails are available to effectively control the Agent. Resist the urge to create a profile with *ALLOBJ special authority!

Add Additional Logging (auditing) to the Agent Profile

You can attempt to restrict access for an Agent but another security aspect to consider is to add additional auditing beyond what is enabled at the IBM i system level.

Assuming that auditing is configured at the system level (QAUDCTL is set to at least *AUDLVL and QAUDLVL contains at least *AUTFAIL, *CREATE, *DELETE, *PTFOPR, *SAVRST, *SECURITY (or *SECRUN and *SECCFG) and *SERVICE (it's ok if it contains more values) we recommend that the following be added at the user level.

> **Note:** No need to specify any of the previously listed values unless they aren't contained in the QAUDLVL system value. If that's the case, add them to the AUDLVL parameter on this command:

CHGUSRAUD AGT_INV AUDLVL(*CMD *JOBBAS *NETCMN *NETSECURE *NETUDP *OBJMGT)

The values *NETSMBSVR and *NETTELSVR will also help you track the actions of the Agent profile; however, they can only be specified at the system value level (QAUDLVL) and most organizations do not configure them because they generate large numbers of audit journal entries.

Security administrators can then use the audit journal helper functions to monitor the Agent profile's actions. For example, the following looks for authority failure (AF) audit journal entries. Regular examination of these specific entries could be an indication of a couple of things ... First, as an Agent learns and expands its own tasks, it may attempt to do things or access obtains for which it doesn't have authority. Review of these audit journal entries will let the security administrator will observe failures encountered by agents, and a review can be held to determine if access should be granted. On a more negative side, examination of these entries could be an indication that the Agent has gone rogue or has been taken over/infiltrated by a bad actor. While AF entries aren't often part of a regular review, we encourage it in this case.

```
SELECT entry_timestamp,
       user_name,
       qualified_job_name,
       remote_address,
       violation_type,
       violation_type_detail,
       object_library,
       object_name,
       object_type,
       path_name
FROM TABLE (
systools.audit_journal_af(starting_timestamp => CURRENT TIMESTAMP - 7 DAYS)
)
WHERE user_name = 'AGT_INV'
ORDER BY entry_timestamp;
```

This is just one example. We encourage you to work with your organization's security administrators to determine the other audit journal entries that they can monitor for that will be an indication of an Agent going outside of its intended role.

MCP Servers

Exact security considerations for MCP servers will depend on where they are implemented – on the client side or the server side. Pay attention to the implementation’s security recommendations, paying specific attention to authentication methods to ensure no credentials are compromised.

Here are some recommendations relating to IBM Bob and MCP servers:

https://bob.ibm.com/docs/ide/configuration/mcp/understanding-mcp

Resources

IBM partnered with Anthropic to develop this white paper: Architecting Secure Enterprise AI Agents

IBM has also developed several YouTube videos on this topic. Search for the IBM Think Technology channel.

Key Takeaways

- Create the Agent profile with only the capabilities required for the intended task - no more
- Create one Agent per task
- Only allow Agents to talk to one another if it makes sense
- Add as many guardrails as possible to restrict what an Agent can do

Advanced Integration

Why Use MCP to Interface AI with IBM i

Model Context Protocol (MCP) is an open standard, created by Anthropic and implemented by most modern business operating systems, delivers a standardized way for Artificial Intelligence (AI) assistants to securely connect with IBM i systems, query databases, execute commands, and interact with applications. MCP is designed to allow for many different types of AI to interface to any business server (including IBM i) using a single protocol, that is scalable, securable, auditable, whilst allowing for business rules (often called AI Guardrails) to be implemented.

The following diagram shows three different classes of AI on the left, connecting via the same MCP interface in the middle, to different server-side resources on the right.

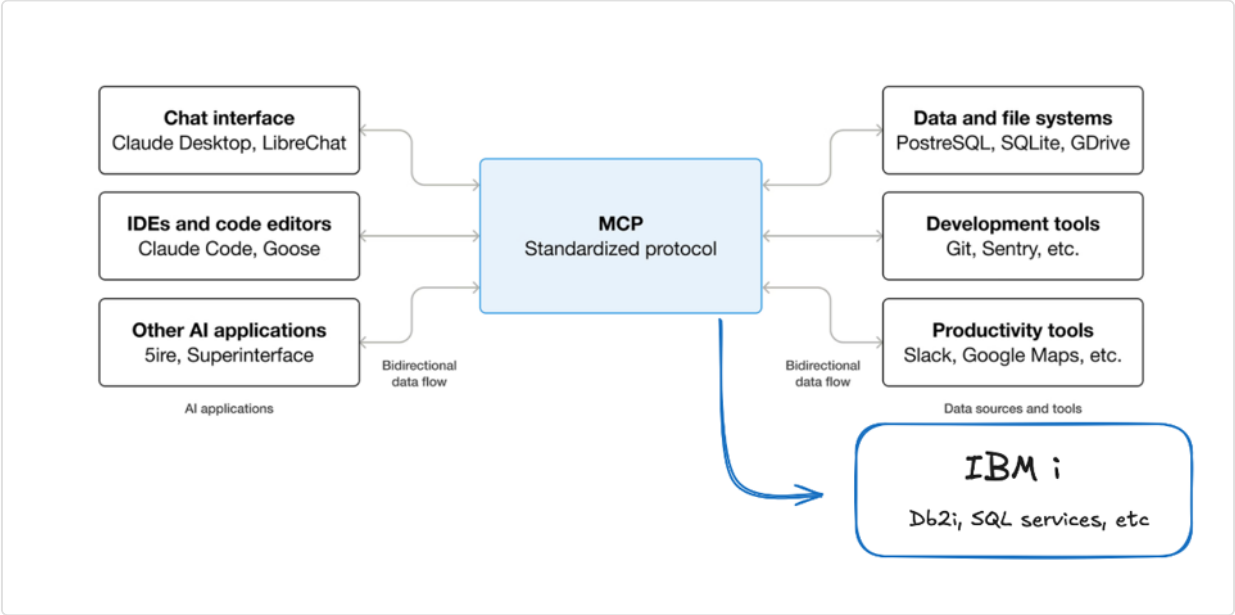


Figure: MCP Overview Diagram

> **Note:** this diagram comes from IBM's Adam Shedivy via https://ibm-d95bab6e.mintlify.app/

Below are some of the key benefits of MCP integration with IBM i:

Real-Time Access to System Information

MCP enables AI assistants to query IBM i databases, examine system configuration, review job logs, and access other system information in real-time. When troubleshooting an issue, the AI can examine relevant system logs and

job details directly. This real-time access means assistance is based on current, accurate information rather than assumptions or outdated documentation.

Context-Aware Code Assistance

With MCP access to your source code repositories and database schemas, AI assistants can provide suggestions that fit your actual environment, referencing your existing programs, understanding your naming conventions, and suggesting changes that integrate with your specific codebase.

This context awareness dramatically improves the relevance and usefulness of code assistance, reducing the time developers spend adapting generic suggestions to their specific situation.

Secure, Auditable Interactions

MCP implements security at its core, because of this AI assistants do not get blanket access to your systems. Instead, you control exactly what resources are accessible and what operations are permitted.

Every interaction can be logged and audited, providing transparency about what the AI accessed and what actions it performed. This security model makes it practical to provide AI assistance in production environments while maintaining the control and oversight that enterprise systems require.

Consolidating AI Investment Across IT Platforms

From a business investment point of view, it is worth noting that because MCP is an open, standardized protocol, investments you make integrating AI with IBM I, should be easily extended to AI integration with other business systems that you might have in your estate, regardless of the Hardware, Operating System, Application or Database they run on.

MCP Enables Practical AI Assistance for Daily Workflows.

The most immediate benefit of MCP integration is making AI assistance genuinely useful for daily work. Developers can ask questions about database schemas and get answers based on actual table definitions. They can request help with SQL queries and have the AI test those queries against real data to verify they work correctly. Operations staff can ask about system performance and have the AI examine actual metrics rather than providing generic optimization advice. This practical assistance reduces the time spent on routine tasks and helps teams work more efficiently.

You can then build upon this foundation to enable more sophisticated assistance. Imagine an AI assistant that monitors your IBM i system proactively, identifying potential issues before they become problems. Or an assistant that can analyze application logs to identify patterns that indicate emerging issues. These capabilities are achievable today by combining MCP's access to system information with AI's pattern recognition abilities.

MCP Preserves IBM i Security While Enabling Innovation

One of the most important aspects of MCP is how it enables AI integration without compromising security. Traditional approaches to AI integration often require exposing systems through broad APIs or granting extensive permissions. MCP's architecture allows fine-grained control, you can provide an AI assistant access to read database schemas without allowing data modification, or permit execution of specific commands while restricting others.

You can deploy multiple MCP servers on the same host, each with different restrictions in their abilities, data access, and permitted users. This security model makes it practical to experiment with AI assistance in production environments. This approach respects the reality that IBM i systems often contain sensitive business data while still enabling the benefits of AI assistance.

For example, you could start with read-only access to non-sensitive information, verify the value, and gradually expand access as you gain confidence. The audit trail ensures you can review what the AI accessed and identify any concerns.

MCP Creates a Foundation for Future AI Capabilities

While MCP's immediate benefits are compelling, its long-term value lies in creating a foundation for capabilities we're only beginning to imagine. As AI models become more sophisticated, the ability to interact with IBM i systems through a standardized protocol becomes increasingly valuable.

Consider potential future scenarios: AI assistants that can help migrate applications by understanding both source and target environments, intelligent monitoring systems that correlate system metrics with business outcomes, or development tools that suggest optimizations based on actual usage patterns.

In Summary

MCP represents a practical approach to bringing AI assistance into IBM i environments. It respects the security and reliability requirements of enterprise systems while enabling the context-aware assistance that makes AI genuinely useful.

For most IBM i shops, the question is not whether to integrate AI assistance, but how to do so safely and effectively. MCP provides the answer; a standardized, secure, and practical approach to bridge the gap between AI capabilities and IBM i systems.

Should I care about testing AI?

Absolutely—**testing AI is not optional**. AI systems are fundamentally different from traditional software because their outputs are **probabilistic and data-driven**, rather than strictly deterministic. This introduces new risks that testing must address.

Premise

Why do we test?

The obvious answer is to catch bugs before production, avoiding disruptions, losses, and damage to the company. Ultimately, we test because we are responsible for what we deliver, and this responsibility can extend to the top management and even to regulators.

Testing serves to reduce risk.

Until recently, testing could have been relatively limited, relying on deep expertise to mentally validate solutions before writing code. But with LLM models, much of the work is changing: AI can generate solutions quickly, but without the certainty of fully understanding the business and sometimes with errors.

This leaves us with a new role: validating what we haven't fully designed ourselves.

Testing, already repetitive, becomes even more central and clearly an ideal candidate for automation.

So the real question is: how do we test so we can detect even the smallest change in behavior and quickly decide whether it's correct or not?

Eventually you have to take into account: **TRUST - VALIDATION - LEGAL REGULATION**

Here’s why and how you should care:

2. Key Areas to Test in AI Systems

a) Functional Correctness

- Does the AI **perform its intended task correctly**?
- Example: a pricing model calculates expected discounts accurately.

b) Data Quality

- AI depends heavily on data.
- Test for **missing values, outliers, and inconsistencies** in the training and input data.

c) Bias & Fairness

- Check whether AI outputs are **unfairly biased** toward any group.
- Example: credit scoring AI that favors certain demographics.

d) Robustness

- Can the AI **handle unexpected or adversarial inputs**?
- Test edge cases, outliers, or intentionally malformed data.

e) Performance & Scalability

- How fast does the AI respond under **real workloads**?
- Does it scale for thousands of simultaneous predictions?

f) Explainability

- Can you **interpret AI decisions**?
- Especially important for regulated industries (finance, healthcare, insurance).

5. Why It Matters

- Regulatory compliance (e.g., GDPR, AI Act in the EU)
- Maintaining **trust in AI-driven decisions**
- Preventing **financial loss, legal liability, or reputational damage**

Even if the AI “works most of the time,” failures can be catastrophic in critical systems like finance, healthcare, or IBM i enterprise workflows.

Using Sample Prompts to Get Better Results from IBM Bob

IBM Bob's effectiveness scales directly with the quality of the instructions it receives. A vague prompt yields a generic result. A precise, context-rich prompt yields output that is targeted, standards-compliant, and immediately actionable. This section covers the core techniques IBM i developers can apply to construct prompts that consistently produce high-quality RPG, CL, and SQL output.

The Five Elements of an Effective Prompt

A well-structured prompt for IBM Bob contains five elements. Each narrows the solution space and directs output toward a precise result.

- **Context** establishes where the work fits—what platform, project, or existing codebase is involved. Without context, Bob generates solutions in a vacuum.
- **Task** states the explicit action required. Use direct imperatives: create, analyze, convert, review. Avoid ambiguous verbs such as "help with" or "look at."
- **Requirements** enumerate the specific behaviors the output must satisfy—RPG format, data structure patterns, SQL standards, and service programs to be called.
- **Constraints** define boundaries the output must not cross: performance targets, backward compatibility, library restrictions, or HIPAA data handling rules.
- **Format** specifies how output should be delivered—a compilable source member, a markdown document, or a structured analysis report.

Context: Part of the Phase II Price Transparency system; called by 18 formula programs.

Task: Create free-format RPGLE service program PTSP001 to calculate discrete percentile values from payment amounts.

Requirements: Accept PACKED(15:2) array; return 10th, 50th, 90th percentiles in a qualified data structure; handle empty arrays, single elements, and duplicate values.

Constraints: Follow .bob/rules/medhost-standards.md; use MONITOR/ ON-ERROR; include UTSP_LOG calls; no PHI in log messages.

Format: Complete compilable service program with header documentation.

Role-Based Prompting

Assigning a professional role at the start of a prompt is one of the most effective techniques for improving output quality. When IBM Bob is given a role, it draws on the reasoning patterns, priorities, and communication style associated with that expertise. A senior IBM i software engineer focuses on correctness, ILE conventions, and platform-specific idioms. An IBM i architect emphasizes coupling, activation groups, and long-term structure. A code reviewer surfaces violations, risks, and improvement opportunities with evidence.

The role instruction belongs at the very beginning of the prompt, before context or task. This ensures IBM Bob's reasoning posture is established before it interprets any other instruction.

```
Act as a senior IBM i software engineer with expertise in free-format
RPG and ILE service program architecture.

Review PTSP002.rpgle against the standards in
.bob/rules/medhost-standards.md. For each violation, provide:
  - Line number
  - Standard violated
  - Severity: Critical | Major | Minor
  - Corrected code snippet
```

Close with a summary count by severity and a production-readiness assessment.

Role Instruction	Best Applied To	Effect on Output
Senior IBM i software engineer	Code generation, bug fixes, modernization	Platform-idiomatic RPG; IBM i-specific constraints applied automatically
IBM i enterprise architect	System design, service program structure	Emphasis on ILE coupling, activation groups, and long-term modularity
IBM i performance engineer	SQL tuning, batch optimization	Set-based operations preferred; index strategy and commitment control highlighted
Senior RPG code reviewer	Pre-commit review, standards enforcement	Systematic violation reporting with line numbers, severity, and corrected examples
Technical documentation specialist	Header docs, API reference, guides	Structured documentation aligned with IBM i object and procedure conventions

> **Best Practice:** Assign a role at the start of every non-trivial IBM Bob interaction. Match the role to the task type: engineer for implementation, architect for design, reviewer for analysis, documentation specialist for technical writing.

Referencing Standards Documents

When a team maintains coding standards in a project rules file—such as `.bob/rules/medhost-standards.md` —those standards can be invoked by reference rather than restated in every prompt. This reduces prompt length, eliminates reliance on the developer's recollection, and ensures IBM Bob reads the authoritative source directly. Combined with a role instruction, a standards reference produces output that is both expert in character and precisely aligned with team expectations.

```
Act as a senior IBM i software engineer.

Create free-format RPGLE program PA5P06A for Formula 17 billing codes.
Follow all standards in .bob/rules/medhost-standards.md.

Call service programs: PTSP001 (percentile), PTSP002 (payment
matching), PTSP003 (PT\_MRF update).
```

Iterative Refinement

Even well-structured prompts benefit from correction passes. When IBM Bob's output contains specific issues, the most efficient response is a targeted correction instruction—not a full prompt rewrite. State the line number, the problem, and the required fix. Direct IBM Bob to correct only those items.

The procedure calculatePercentile has three issues:

1. Line 87: Replace bubble sort with IBM i array sort using %LOOKUP.
2. Line 112: Add empty-array guard; return zeros when p_count = 0.
3. Missing UTSP_LOG call at procedure exit; add severity INFO.

Correct these three items only. Do not regenerate the full program.

Summary

IBM Bob's value to an IBM i development team scales directly with prompt quality. The five-element structure—context, task, requirements, constraints, and format—provides a repeatable foundation. Role-based prompting shifts Bob's output from generic to expert, with minimal additional effort. Standards references eliminate the need to restate conventions in every interaction. Iterative refinement corrects specific issues without discarding work already done. Applied together, these techniques produce output that reflects the judgment and precision expected of IBM i enterprise software development.

> **Note:** The prompt patterns in this section are starting points, not rigid templates. Teams should adapt them based on their environment, standards documents, and the task types their developers encounter most frequently.

Key Takeaways

- **Prompt quality determines AI quality** Clear, structured, and contextual prompts produce more accurate and useful results in IBM i environments.
- **Prompt sampling is an iterative process** Testing multiple prompt variations and refining them over time leads to better and more consistent AI output.
- **Strong prompts follow a structured pattern** Effective prompts include a defined role, clear task, system context, expected output format, and supporting input such as RPG, CL, or SQL code.
- **Context is critical in IBM i development** Including IBM i version, DB2 for i details, program type, and environment constraints ensures AI responses are technically accurate and compatible.
- **Reusable prompts increase team productivity** Maintaining a shared prompt library standardizes AI usage and reduces time spent rewriting instructions.
- **Prompt sampling supports multiple development activities** AI prompts can be used for code analysis, SQL optimization, documentation generation, troubleshooting, and modernization efforts.
- **Prompt libraries should be treated as development assets** Storing prompts in version control alongside source code enables repeatable and scalable AI workflows.

- **AI becomes more valuable with structured usage** Organizations that standardize prompt sampling achieve more reliable and efficient AI-assisted development on IBM i.
- **Continuous refinement is essential** Prompts should evolve alongside system upgrades, new development practices, and emerging AI capabilities.
- **The goal is consistency and trust** Well-designed prompts transform AI from a general assistant into a dependable IBM i development partner.

Test Driven Development with Automation

Testing Fundamentals

Why test IBM i applications?

Testing applications is a critical part of development that ensures a product is reliable, secure, and user-friendly before it reaches the public. Without a thorough testing process, a simple error could lead to massive financial losses or irreparable damage to a brand's reputation.

The Three Laws of Testing (ARO)

Any test, to be definable as such, must be:

1. Autonomous
2. Reproducible
3. Observable

Autonomous

A test should be isolated from events outside its perimeter. For example, if a test uses a file, this file cannot be updated while the test is running from a source outside of the test perimeter.

Reproducible

A reproducible experiment refers to the ability to replicate the same methodology and procedures, ensuring that the results can be independently verified by others. The initial condition must be identical, and the sequence of events must also be identical. Compliance with initial conditions requires the initialization of the database (DB), for example. The sequence of events is brought by the script.

Observable

Observability is the ability to observe the internal states of a system. You must be able to observe that the final state is identical between the expected and obtained states.

> **Note:** Any process that does not comply with any of these 3 laws will not be considered a formal test.

Prevent Regression in Legacy Code

IBM i applications often contain **decades of business logic**, frequently written in languages like:

- RPG
- COBOL
- CL
- SQL
- Java

- PHP

But this can be mixed as well with:

- Node.js
- Python

Over time, systems become complex and tightly coupled. Without testing, a small change can unintentionally affect other parts of the system.

Regression testing helps ensure **existing functionality keeps working after modifications**.

Enable Safer Continuous Delivery

Organizations adopting DevOps practices need testing to enable:

- automated builds
- continuous integration
- faster releases

Automated tests allow teams to **deploy changes with confidence instead of relying on manual checks**.

Ensure Compliance and Auditability

Many IBM i environments run in regulated industries like:

- banking
- insurance
- manufacturing
- healthcare

Testing provides **evidence that systems behave correctly**, which is often required for audits and compliance.

AI Large Language Models integrate into software development, making auditable and compliant workflows a critical industry standard. Organizations always need to handle the following aspects:

- How do you verify changes?
- How do you prove systems behave correctly?

Because AI LLMs automate code changes, you need to automate testing to keep pace with the code being produced. You also need to implement a testing framework to provide traceable validation evidence.

In short: Testing IBM i applications protects mission-critical systems, enables modernization, reduces risk, and supports faster development cycles. Automating testing becomes crucial for the Software Development Life Cycle (SDLC) and critical for AI LLM integration.

Testing Methodologies

The Core Development Cycle

TDD follows a repeating development loop:

1. Write a Test

The process starts by writing a test for a specific feature or behavior. Because the feature has not yet been implemented, the test will initially fail.

2. Run the Test and Confirm Failure

Running the test ensures it actually detects missing functionality. This step confirms that the test is meaningful and capable of catching errors.

3. Implement the Simplest Code

The developer then writes just enough code to make the test pass. The goal is not perfection but **the simplest possible working solution**.

If you want to add extra code, add a test first that validates the new code.

4. Verify All Tests Pass

Once the new code is written, the test suite is executed again to confirm that the new test and all previous tests pass successfully.

5. Refactor the Code

After functionality is confirmed, the code is improved. This might include:

- removing duplication
- improving naming
- restructuring logic
- splitting large methods into smaller ones

Refactoring is safe because the automated tests quickly reveal if a change breaks existing behavior.

6. Repeat the Cycle

The same process is repeated for the next feature or improvement.

This iterative loop is often summarized as **Red → Green → Refactor**:

- **Red:** failing test
- **Green:** working code
- **Refactor:** improved design.

Structuring a Good Test

A clear structure improves the readability and reliability of tests. A common pattern includes four stages:

1. **Setup** Prepare the system or component so the test can run.
2. **Execution** Trigger the behavior being tested.
3. **Validation** Verify that the results match the expected outcome.
4. **Cleanup** Reset the environment so other tests can run independently.

This structure ensures tests remain understandable and repeatable.

Benefits of Test-Driven Development

Adopting TDD can lead to several advantages:

- better software design
- clearer understanding of requirements
- faster detection of defects
- safer refactoring
- a growing suite of automated regression tests

Because every feature is accompanied by tests, the system gradually builds a reliable safety net for future changes.

Summary

Test-Driven Development is a disciplined process where:

1. Tests are written before code

2. Code is implemented to satisfy those tests
3. The design is improved while tests guarantee correctness

Over time, this process produces **well-tested, maintainable, and reliable software**.

For more information, see Test Driven Development.

What is IBMiUnit?

IBMiUnit is a unit testing framework designed specifically for the IBM i platform. It allows developers to write and run automated unit tests for RPG. The intention for IBMiUnit is for IBM i shops to start pivoting toward the test-driven development paradigm.

Key characteristics

Inspired by xUnit – follows the same conventions as JUnit, NUnit, and similar frameworks (test cases, assertions, test suites) Written in RPG – tests are written as RPG service programs, making it natural for IBM i developers Assertion-based – provides assertion procedures like assert(), assertEqual(), assertNotEqual(), AssertNumeric() etc.

Installation

Clone Git Repository

This is the easiest way to obtain IBMiUnit if you have a Git client.

This may not currently be the easiest way because the source still has a dependency on the OSSILE product. This will be removed in a future version of IBMiUnit but for now we assume and require the OSSILE library.

1. Navigate to the directory where you want to place the new `IBMiUnit` folder on the i Series
2. `git clone https://github.com/MarinaSchwenk/IBMiUnit.git`
3. Open QShell in the `IBMiUnit` folder
4. Run the `build` script; there are 6 optional parameters:

- `-l <obj-lib>` (that's a letter "L") install IBMiUnit to the specified library; default is `IBMIUNIT`
- `-s <src-lib>` copy source to the given library; default is to leave the source on the IFS and compile from there
- `-o <obj-owner>` the user profile that will own the new objects; default is `QPGMR`
- `-f` copy the obj-lib back into the repository as a save file; this is only useful to the repository contributors
- `-t` compile the test suite over IBMiUnit
- `-v` verbose output; this may be helpful or necessary when errors are encountered

You can keep up-to-date with IBMiUnit by pulling from the repository and re-building.

From Save File (part 1)

1. Create a save file on the IBM i, i.e. `CRTSAVF FILE(xxx/IBMIUNIT)`
2. Download `IBMIUNIT.SAVF` from the IBMiUnit repository
3. Choose one of the transfer options, then continue to part 2

Transfer Using CPYFRMSTMF

1. Copy the downloaded `IBMIUNIT.SAVF` to the IFS
2. Copy from the IFS to the *SAVF, for example:

```
CPYFRMSTMF FROMSTMF('/path/to/file/IBMIUNIT.SAVF') TOMBR('/QSYS.LIB/xxx.LIB/IBMIUNIT.FILE') MBROPT(*REPLAC
```

Transfer Using FTP

1. Go to a command window on your PC
2. Go to the location on the PC where `IBMIUNIT.SAVF` is located
3. Enter one of the following: `ftp (ibmi_ip_address)` or `ftp (ibmi_name)`
4. Enter your IBM i username and password for that system when prompted
5. Ensure the transfer mode is binary (there will be no conversion) with `bin`
6. Go to the library on the IBM i where the save file is stored using something like `cd /QSYS.LIB/xxx.LIB`
7. Transfer the save file from the PC to the empty save file on your IBM i with `put IBMiUnit.savf IBMiUnit.savf`
8. When the transfer is complete, exit FTP with `quit`
9. Close the MS-DOS shell with `exit`

From Save File (part 2)

1. Create the library for the objects, for example, `CRTLIB yyy`
2. Restore from the *SAVF, i.e. `RSTLIB SAVLIB(IBMIUNIT) DEV(*SAVF) SAVF(xxx/IBMIUNIT) RSTLIB(yyy) MBROPT(*ALL) ALWOBJDIF(*ALL)`

Examples

Program Source

```
ctl-opt bndDir( 'IBMIUNIT/IBMIUNIT' );

/copy IBMiUnit/QRPGLESRC,IBMiUnit_H

// test initialization/registration

IBMiUnit_setupSuite( 'TextUtil Tests' );

IBMiUnit_addTestCase( %pAddr( toUpperCase_fromLower ) : 'toUpperCase_fromLower' );

IBMiUnit_teardownSuite();
return;

// test cases

dcl-proc toUpperCase_fromLower;

   assertCharEquals( 'MARINA SCHWENK'
                   : toUpperCase( 'marina schwenk' ) // defined in service program
                   );

end-proc;
```

Running

A successful test example:

```
> ibmiunit/runtest qtehdr_t
  All 2 tests ran successfully
```

A test with a failure:

```
> ibmiunit/runtest qtehdr_t
  assertion failure in calculateSurcharges_changeOrder: Number of tariff
    items expected:<1> but was:<2>
  Done: 0 errors, 1 failures, 1 successful test(s)
```

FAQ

1. What version of IBMi OS does IBMiUnit Support? As of 2025 we still go back to V7r3. However moving forwards, any enhancements will be for V7R4 and above.
2. When will there be a GUI version? The plan has always been to integrate IBMiUnit into RDi or as of the last couple of years, VsCode. However this process takes time and we do not have an ETA on the extension for vsCode yet.
3. How hard is it to pivot to unit testing? Unit testing is part of the test-driven development paradigm. However you do not have to implement every piece of that paradigm and can pick up unit testing pretty quickly.
4. Can you unit test subfiles? No, as of right now we don't have support for validating subfiles and including them into the framework.
5. Can I unit test existing RPG programs. Technically yes, however that does not fit the true unit testing paradigm. The process of unit testing is to take small pieces and test them individually. Service program procedures fits beautifully for this task, however you can just take straight logic and unit test those pieces. For example:

   ```
   ctl-opt bndDir( 'IBMIUNIT/IBMIUNIT' );

   /copy IBMiUnit/QRPGLESRC,IBMiUnit_H

   // test initialization/registration

   IBMiUnit_setupSuite( 'TextUtil Tests' );

   IBMiUnit_addTestCase( %pAddr( myLogic) : 'myLogic' );

   IBMiUnit_teardownSuite();
   return;

   // test cases

   dcl-proc myLogic;

       dcl-s myString  varchar( 50 ) inz( 'Testing plain logic' );

      assertCharEquals( 'Testing'
                      : %subst( myString, 1 7  )
                      );

   end-proc;
   ```

6. Do I need a special compile command for the unit test programs? No, IBMiUnit uses the standard compile commands and don't need anything special.

What is integration testing?

Purpose of Integration Testing

The main goal is to detect problems that occur when modules work together, such as:

- incorrect data exchange
- interface mismatches

- incorrect assumptions between components
- communication failures

Even if individual modules work perfectly on their own, errors can appear when they interact.

Types of Integration Testing

Big Bang Integration Testing

All modules are combined at once and tested together.

Advantages:

- simple setup

Disadvantages:

- difficult to isolate failures.

Integration Testing vs Unit Testing

Testing Type	Focus
Unit testing	Tests a single component in isolation
Integration testing	Tests how components interact
System testing or end to end testing	Tests the complete system

Integration testing sits **between unit testing and full system testing**.

Takeaway

Integration testing verifies that **different parts of a system work together correctly when combined**. Their implementation is sometimes complex and the use of an automated framework is recommended.

What is test automation?

Test automation is the practice of using software tools and scripts to **automatically execute tests, verify results, and report outcomes**, instead of performing these checks manually. An advanced automation will include test script generation.

Automated tests run predefined test cases against an application to confirm that it behaves as expected.

Purpose of Test Automation

The main goals of test automation are to:

- reduce repetitive manual testing
- detect defects earlier
- increase testing speed and reliability
- enable frequent releases

Automated tests can run **whenever the software changes**, helping teams catch problems quickly.

How Test Automation Works

Following the typical steps/actions required to do a test, except for Test Case Preparation, all actions are candidates for automation.

Industry studies indicate that developers spend between 30% and 50% of their time on testing activities. By assigning a typical duration (0 to 5 time units) to each testing action, analysis shows that up to 88% of this effort can be automated.

Category	Step / Action	Description	Time taken with manual execution (time units)	Time taken (time units) with automated execut
Setup	Test Case Preparation	Define and sequence all test case actions (One-time)	3	3
	Access	Manage authentication and login	1	0
Data Management	Identification	Identify data tables involved in the test	5	0
	Initialization	Reset data from a known state (e.g., Backup)	4	0
	Backup	Save data states before and after execution (One-time)	3	0
Execution	Initial Run	Execute the test for the first time	2	2
	Iteration	Repeat execution with controlled reinitialization	5	0
Quality	Coverage	Measure code coverage	5	0
	Merge Coverage	Merge code coverage	5	0
Analysis	Data Interception	Capture input and output data flows	5	0
	Outcome Tracking	Track resulting database states	4	0
	I/O Comparison	Compare input/output data before and after	3	0
	Table Comparison	Compare database tables data before and after across runs	3	0
Reporting	Documentation	Generate reports	5	2
	History	Maintain history	4	0
	Metrics	Produce KPIs and performance indicators	4	0
		TOTAL	61	7

Key points to consider

- Data volume: Evaluate the volume of data implied for any tests requiring data initialization. Explore possibilities to shrink the data whenever possible to enable the automation and frequency at scale.
- Batch runtime: For interactive processes, typically 5250 scenarios, explore framework or tool possibilities to run them in batch. Otherwise, the process is not truly automated.
- Within an AI context, shrunk data volume and batch runtime will be critical for outcome validations automation.

Types of Tests That Can Be Automated

Test automation can be applied at different testing levels:

- **Unit tests** – verify individual functions or components
- **Integration tests** – verify interactions between modules
- **End-to-end tests** – validate complete business workflows
- **Regression tests** – ensure new changes do not break existing functionality
- **Workload tests** - verify performance based on processing scale and volume

Automation is particularly valuable for **regression testing**, which must be repeated frequently.

Benefits of Test Automation

Organizations adopt test automation because it provides:

- faster testing cycles
- improved test coverage
- consistent and repeatable tests
- early detection of defects
- support for continuous integration and delivery.

Takeaway

Test automation uses software tools to **run tests automatically and verify that an application works correctly without manual intervention**. It will considerably and consistently save time for any programmers or QA staff and reduce risk. For AI automated code generation it is a must for outcome validations.

What is Behavior-Driven Development (BDD)?

Behavior-Driven Development (BDD) is a software development approach where **tests are written in terms of system behavior from the user's perspective**. Instead of focusing on low-level code details, BDD describes **how the system should behave in real business scenarios**.

BDD evolved from **Test-Driven Development (TDD)** but emphasizes **communication between developers, testers, and business stakeholders**. Often called "The Three Amigos": Business, Development and Testing. During software development, The Three Amigos should meet regularly to discuss how the product will be developed.

The Given–When–Then Structure

BDD scenarios are usually written using a simple structure:

Given – When – Then

- **Given**: the initial context or system state
- **When**: the action or event that occurs
- **Then**: the expected outcome

Example:

```
Scenario: Apply customer discount

Given a customer with a 10% discount
When the customer places an order for 100€
Then the order total should be 90€
```

This scenario describes the **business rule**, not the implementation.

Relationship with Test-Driven Development

BDD can be seen as an extension of **Test-Driven Development**.

- **TDD** focuses on testing small units of code.
- **BDD** focuses on verifying system behavior and business rules.

Both approaches encourage writing tests before implementation, but BDD emphasizes **business understanding and system behavior** rather than internal code structure.

In simple terms: Behavior-Driven Development is a practice where **software behavior is described in business-friendly scenarios that become automated tests**, ensuring the system behaves exactly as users expect.

Takeaway

From The Three Amigos, the Development "Amigos" can be an AI LLM.

What is End-to-End (E2E) Testing?

End-to-End testing verifies that an entire application works correctly **from start to finish**, simulating real user workflows across all system components.

Instead of testing individual modules or integrations, E2E testing validates that **the complete system behaves correctly in a real-world scenario**.

Example

Consider an online order process. An end-to-end test might verify the entire workflow:

1. A user logs into the system
2. The user creates an order
3. The system calculates the price
4. Payment is processed
5. Inventory is updated
6. A confirmation is generated

The test confirms that **the full business process works correctly from beginning to end**.

Difference from Other Testing Levels

Testing Level	Focus
Unit testing	Individual functions or components
Integration testing	Interaction between modules
End-to-End testing	Complete system workflows

E2E testing sits at the **top of the testing pyramid**, validating the full user journey.

Takeaway

End-to-End testing ensures that **a complete business process works correctly across the entire system**, just as it would for a real user. Their implementation is sometimes complex and the use of an automated framework is recommended.

Test Implementation

How do I implement TDD for RPG?

Implementing **Test-Driven Development (TDD)** for programs written in **RPG** requires adapting the traditional TDD workflow to the architecture of **IBM i** applications. Because many IBM i systems are database-centric and built from large programs, the key is **designing code so it can be tested in small units**.

Below is a practical way teams typically implement TDD in RPG environments.

2. Write the Test First

Following TDD principles, the developer writes a test describing the expected behavior.

Example: testing a pricing calculation.

Test procedure (simplified):

```
dcl-proc testCalculateDiscount;

   dcl-s result packed(7:2);

   result = calculateDiscount(100 : 10);

   assertEquals(90 : result);

end-proc;
```

Since the function `calculateDiscount()` does not exist yet, the test fails.

This is the **“Red” phase** of TDD.

4. Refactor the Code

Once the test passes, the code can be improved without changing behavior.

Refactoring might include:

- splitting logic into procedures
- improving variable names
- removing duplication
- extracting reusable service programs

Tests ensure that changes do not break functionality.

6. Control Database Dependencies

Many RPG programs interact heavily with **Db2 for i** tables.

To make tests reliable:

- create dedicated **test libraries**
- populate controlled test data
- clean up data between test runs

This ensures repeatable results.

8. Start Small (Especially with Legacy Code)

Most existing IBM i applications were not built for TDD.

A realistic adoption strategy is:

1. Add tests when fixing bugs
2. Add tests when modifying code
3. Use TDD for **new modules**

Over time, test coverage grows naturally.

Takeaway

Pure TDD is **rare at the start**. Most teams first adopt **automated regression testing**, and only later transition to true TDD for new development.

How to Write RPGUnit Tests

What This Covers

This guide walks you through the process of writing your first RPGUnit test cases. You'll learn how to structure test programs, write test procedures, use assertions to verify behavior, and run your tests to validate RPG code functionality.

Before You Start

- ☐ RPGUnit framework is installed on your IBM i system
- ☐ You have access to a development library where you can create test programs
- ☐ You have basic knowledge of RPG ILE programming
- ☐ You have identified the RPG program or procedure you want to test
- ☐ You have RDi (Rational Developer for i) or another IBM i development environment set up

Modern Tooling Options

While RPGUnit tests can be run from the command line, modern development tools provide enhanced testing experiences:

VSCode with IBM i Testing Extension: The IBM i Testing extension for Visual Studio Code (VS Code) integrates RPGUnit directly into your editor. It provides visual test runners, inline test results, and the ability to run tests with a single click. This extension works seamlessly with the Code for IBM i extension, bringing modern IDE features to RPG development.

IBM Bob: IBM Bob™ is an AI-powered development assistant that understands RPG and can help generate test cases, explain test failures, and suggest improvements to your testing strategy. Bob can accelerate test creation by generating boilerplate test code and helping identify edge cases you might have missed.

Steps

1. **Create a Test Service Program**
   - Create a new RPG source member with a name ending in 'T' (e.g., `MYPROGT` for testing `MYPROG`)
   - Use `**FREE` format for modern RPG syntax
   - Include the RPGUnit copy book: `ctl-opt nomain;` and `/copy RPGUNIT1,TESTCASE`
   - This establishes your source as a test service program that RPGUnit can discover

2. **Define Your Test Procedures**
   - Create procedures with names starting with `test` (e.g., `testCalculateTotal`, `testValidateInput`)
   - Each test procedure should focus on testing one specific behavior or scenario
   - Use the following structure:

```
dcl-proc testCalculateTotal export;
  // Test code goes here
end-proc;
```

   - The `export` keyword makes the procedure visible to RPGUnit

3. **Write Test Setup (Optional)**
   - If your tests need common initialization, create a `setup` procedure
   - This runs before each test procedure:

```
dcl-proc setup export;
  // Initialize test data
  // Open files if needed
  // Set up test environment
end-proc;
```

   - Use `teardown` procedure for cleanup after each test

4. **Add Assertions to Verify Behavior**
   - Use RPGUnit assertion procedures to check expected outcomes:
     - `assert(condition : message)` - Verify a condition is true
     - `assertEqual(expected : actual : message)` - Check two values match
     - `assertNotEqual(expected : actual : message)` - Check values differ
     - `fail(message)` - Force a test failure with a message
   - Example:

```
dcl-s result packed(7:2);
result = calculateTotal(100 : 0.10);
assertEqual(110.00 : result : 'Total should include 10% markup');
```

5. **Test Edge Cases and Error Conditions**
   - Write separate test procedures for boundary conditions
   - Test with empty inputs, null values, maximum values, and invalid data
   - Verify error handling works correctly:

```
dcl-proc testEmptyInput export;
  dcl-s result varchar(100);
  result = processData('');
  assertEqual('ERROR' : result : 'Should return error for empty input');
end-proc;
```

6. **Compile the Test Service Program**
   - Compile your test source using `CRTRPGMOD` to create a module
   - Create a service program using `CRTSRVPGM`:

     ```
     CRTRPGMOD MODULE(MYLIB/MYPROGT) SRCFILE(MYLIB/QRPGLESRC)
     CRTSRVPGM SRVPGM(MYLIB/MYPROGT) MODULE(MYLIB/MYPROGT) EXPORT(*ALL)
     ```

   - Ensure the service program is in a library in your library list
7. **Run Your Tests**
   - Use the RPGUnit command to execute tests: `RUCALLTST MYPROGT`
   - Or run all tests in a library: `RUCALLTST MYLIB/*ALL`
   - Review the output in the job log or spool file for test results
8. **Analyze Test Results**
   - Green/Pass: Test assertions succeeded - the code works as expected
   - Red/Fail: Test assertions failed - review the failure message to identify the issue
   - Error: Test couldn't run - check for compilation errors or missing dependencies
   - Fix any failing tests by either correcting the code under test or adjusting test expectations

Verify It Worked

Your tests are working correctly when:

- All test procedures execute without compilation errors
- Test results clearly show pass/fail status for each test
- Failing tests provide meaningful error messages that help identify issues
- You can intentionally break code and see corresponding test failures
- Running tests becomes part of your regular development workflow

Key Takeaways

- Test procedures must start with "test" and be exported to be discovered by RPGUnit
- Use assertions to verify expected behavior - they're the heart of your tests
- Write focused tests that check one specific behavior per test procedure
- Test both happy paths and edge cases to ensure robust code coverage
- Setup and teardown procedures help manage test data and environment
- Regular test execution catches regressions early and builds confidence in code changes
- Good test names and assertion messages make failures easy to diagnose and fix

How do I test SQL stored procedures?

How do I implement tests?

In the classical **testing pyramid**, which includes, from bottom-up:

- **unit tests**
- **integration tests**
- **end-to-end tests**

The most real world level is **end-to-end tests** and this is what you will want to simulate ultimately. To verify and demonstrate that you've exercised enough application scenarios, you need to use the system's code coverage feature, either manually or with a suitable tool.

Typical steps/actions required to do a test

Your tests implementation will take part or all of the following actions:

Category	Step / Action	Description
Setup	Test Case Preparation	Define and sequence all test case actions (One-time)
	Access	Manage authentication and login
Data Management	Identification	Identify data tables involved in the test
	Initialization	Reset data from a known state (e.g., Backup)
	Backup	Save data states before and after execution (One-time)
Execution	Initial Run	Execute the test for the first time
	Iteration	Repeat execution with controlled reinitialization
Quality	Coverage	Measure code coverage
	Merge Coverage	Merge code coverage
Analysis	Data Interception	Capture input and output data flows
	Outcome Tracking	Track resulting database states
	I/O Comparison	Compare input/output data before and after
	Table Comparison	Compare database tables data before and after across runs
Reporting	Documentation	Generate reports
	History	Maintain history
	Metrics	Produce KPIs and performance indicators

> **Note:** Sometimes the SETUP can include a `FIXTURE` to prepare specific configurations or environments. Sometimes the EXECUTION can include a `TEARDOWN` at the end of the test to clean or reset configurations or environments.

Warnings

1. **New silent test cases** in case of transformed code.
    - The new application code might use new structures, methods, languages or environments and in some case the new methods will require new tests. For example for concurrent database access: A stateful RPG pgm might use a direct RLA access to update a record and naturally locks the record at the initial read access. If this code is transformed in a stateless SQL access the method could be a pessimistic or optimistic lock that would require an extra columns in the table for timestamp update. When, on the original RPG RLA no tests were required to insure that concurrent update behave correctly, with the new structure, formal tests would be required to insure concurrent updates behave the same way.
2. **Organisation & Governance** for testing.
    - Segregation of duties: Operate independent testing from whoever (or whatever) developed the code to overcome operational risks, as a test cannot be auto-referenced.
3. **The 3 Laws of testing** (ARO): Autonomous, Reproducible, Observable. Ref. chapter **Test Driven Development with Automation / Why test IBM i applications?**
    - Ensure that the 3 laws are respected

Takeaway

Either manually or automatically you will have to follow all those rules to successfully implement your tests. Many activities can be automated and if you are using AI to generate code you will have to automate testing to leverage AI at scale.

How do I automate test execution?

Automating **test execution** means setting up a system where tests run automatically—without manual intervention—whenever certain events occur (such as a code change, build, or scheduled run). The goal is to ensure tests run **consistently, frequently, and reliably**.

> **Tip:** Tools are highly valuable here. (Ref. chapter **Test Driven Development with Automation / What testing tools work with IBM i?**)

Below are practical considerations for automated test execution.

Use a Build or Automation Tool

Next, create scripts that trigger test execution automatically.

The **Typical steps/actions required to do a test** are:

Category	Step / Action	Description
Setup	Test Case Preparation	Define and sequence all test case actions (One-time)
	Access	Manage authentication and login
Data Management	Identification	Identify data tables involved in the test
	Initialization	Reset data from a known state (e.g., Backup)
	Backup	Save data states before and after execution (One-time)
Execution	Initial Run	Execute the test for the first time
	Iteration	Repeat execution with controlled reinitialization
Quality	Coverage	Measure code coverage
	Merge Coverage	Merge code coverage
Analysis	Data Interception	Capture input and output data flows
	Outcome Tracking	Track resulting database states
	I/O Comparison	Compare input/output data before and after
	Table Comparison	Compare database tables data before and after across runs
Reporting	Documentation	Generate reports
	History	Maintain history
	Metrics	Produce KPIs and performance indicators

Except for the first action, all other action executions must be automated. Build tools, scripts, or use a tool to coordinate these steps.

Each test must be executable **from an API or CLI**.

You can then evaluate & integrate different levels of automation, operability and AI integration described in **Test Driven Development with Automation / Level of automation with or without AI**

Schedule or Trigger Test Runs

Automated tests can be triggered in several ways:

Code-based triggers

- every commit
- pull request validation

- build completion.

Time-based triggers

- nightly test runs
- scheduled regression tests.

Environment triggers

- before deployment
- after configuration changes.

Identify the Behavior to Describe

Start by defining **business behaviors**, not technical features.

Focus on questions like:

- What should happen when a user performs an action?
- What business rules apply?
- What outcome should the system produce?

Example behavior:

> “Preferred customers receive a 10% discount when placing an order.”

These behaviors are then turned into **scenarios**.

Implement Step Definitions

Each step in the scenario must be mapped to code that performs the action or validation.

Conceptual example: `Given a preferred customer`

Step definition:

```
createCustomer(type="preferred")
```

```
When the customer places an order of 100€
```

Step definition:

```
order = createOrder(customer, 100)
```

```
Then the order total should be 90€
```

Step definition:

```
assert(order.total == 90)
```

These step definitions connect **human-readable behavior descriptions** with the actual system.

Integrate BDD into the Development Workflow

BDD should be part of the regular development process.

Typical workflow:

1. Discuss behavior with stakeholders
2. Write scenarios together
3. Implement automation for those scenarios
4. Write the code that satisfies the scenarios
5. Run tests continuously

This ensures development always aligns with expected behavior.

Example End-to-End Flow

1. Business defines behavior
2. Team writes a BDD scenario
3. Scenario is automated with a framework
4. Developers implement the feature
5. Automated scenarios validate behavior.

Testing Tools and Evaluation

What testing tools work with IBM i?

How to evaluate the right tool

Choosing the right solution depends on your context. The most effective approach is to run a **proof of concept (POC)** using your own applications and constraints.

Evaluation should be based on how well a tool supports and automates the **full set of testing actions**, including:

Category	Step / Action	Description	Quality criteria to consider
Setup	Test Case Preparation	Define and sequence all test case actions (One-time)	(1)
	Access	Manage authentication and login	(2) -Enable test replay with different login
Data Management	Identification	Identify data tables involved in the test	(3) -Autonomous process -Level of automation
	Initialization	Reset data from a known state (e.g., Backup)	(4) -Autonomous & global process
	Backup	Save data states before and after execution (One-time)	(5) -Autonomous & global process
Execution	Initial Run	Execute the test for the first time	(6) -Script generation automation
	Iteration	Repeat execution with controlled reinitialization	(7) -Batch execution enabled -Workload testing -Objects integrity
Quality	Coverage	Measure code coverage	(8) -Level of automation -Reporting -SQL based
	Merge coverage	Merge code coverage	(9) -Level of automation -Reporting -SQL based
Analysis	Data Interception	Capture input and output data flows	(10)-Level of automation
	Outcome Tracking	Track resulting database states	(11)-Autonomous process & global process
	I/O Comparison	Compare input/output data before and after	(12)-Level of automation
	Table Comparison	Compare database tables data before and after across runs	(13)-Level of automation -Level and interface details
Reporting	Documentation	Generate reports	(14)-Level of automation & Dashboard support
	History	Maintain history	(15)-Level of automation & Dashboard support
	Metrics	Produce KPIs and performance indicators	(16)-Level of automation -Integration support

1. Defining what the test scenario will do.
2. Test authority access from different user profile login in your test replay.
3. ◦ Capability to identify tables used by the test autonomously (on the fly).
   ◦ Level of automation behind the identification to copy, restore or compare the table when appropriate.
4. Capability to initialize data autonomously per single test replay and globally with shared data.
5. Capability to save data autonomously per single test or globally with shared data.
6. Capability to monitor the test and to generate an execution script automatically.

7. ◦ Some test scenario can run interactively and the capability to replay the scenario in batch is determinant to set a full automation.
   ◦ Workload testing with dynamic variable settings (If you run in parallel the same test scenario and your scenario includes an update, the first session will lock the record therefore you will want to run any parallel sessions with different access records).
   ◦ Object ownership and attributes integrity when replay (By default not all original authority attributes and privileges are restored within the object, you must ensure they are handled).
8. ◦ Capability to set code coverage.
   ◦ Capability to report it only in RDi or IFS or VS Code.
   ◦ Capability to report it in SQL.
9. ◦ Capability to aggregate & merge several code coverages per programs or modules.
   ◦ Capability to report it only in RDi or IFS or VS Code.
   ◦ Capability to monitor & report it in SQL.
10. Capability to capture input and output data.
11. Capability to capture initial and ending database data autonomously per single test or globally with shared data.
12. Capability to parse & compare I/O data across runs.
13.

- Capability to compare database table mismatches. - Capability to drill down in records and columns details.

14. Support for documentation and dashboard (including logging scripts and all context metadata).
15. History & dashboard support.
16.

- KPIs and performance indicators. - Query and integration capability with external pipelines.

Other points to consider

- Compliance with the 3 Laws of testing (ARO): Autonomous, Reproducible, Observable. Ref. chapter **Test Driven Development with Automation / Why test IBM i applications?**
- Test Suite with multiple tests
- Multiples jobs encapsulation capability within 1 test scenario
- Shrink option with database table subset with only records used for the test
- Test script capitalisation:
  ◦ Resiliency of the test after a structural change has occurred in a table or in the UI. (for example: capability to replay a test automatically if a database field size has changed)
  ◦ Re-usability of existing scripts
  ◦ Centralisation of all tests (unit test, integration & end to end tests)
- CI/CD integration
- AI & MCP integration
  ◦ Capability to test AI outcome
  ◦ AI governance
- (For CI/CD & MCP, batch execution is relevant for full automation and the shrink database option for high frequency test replay)

Advanced Testing Concepts

What is Workload Testing?

Workload testing is a type of performance testing that measures how a system behaves under **expected and peak workloads**. The goal is to ensure the system can **handle real-world usage** without performance degradation or failure.

It is particularly important in enterprise environments like **IBM i**, where systems often support **high-volume transactions, batch jobs, and multiple users**.

Key Characteristics

- **Simulates real-life usage**: not just stress, but actual workloads
- **Focuses on throughput and response times**
- **Identifies bottlenecks** in applications, programs, or databases
- **Helps plan capacity and scalability**

How Workload Testing Differs from Other Tests

Test Type	Focus	Example
Unit Test	Single function correctness	A discount calculation returns 10%
Integration Test	Interaction between modules	Order program updates inventory correctly
End-to-End Test	Complete workflow	Customer places order → inventory & billing updated
Workload Test	System performance under load	500 users place orders simultaneously
Stress Test	System behavior under extreme load	Double normal traffic until failure

> Workload testing ensures **the system performs well under expected load**, while stress testing pushes it **beyond limits**.

Summary

Workload testing ensures that an application or system can **handle expected business activity reliably**. It is essential for **capacity planning, performance optimization, and risk mitigation**, especially in critical systems like **IBM i**.

Level of automation with or without AI

When discussing **levels of test automation**, it is useful to distinguish between **manual automation approaches, traditional automated testing, and automation for AI**. These levels reflect how much of the testing process is handled automatically versus how much still requires human intervention.

Below is a practical maturity model that includes **modern testing automation concepts**.

Levels of automation:

We can identify up to 7 levels of automation:

1. Automated replay with automated data initialization & comparison
    - For any test that includes data, this is the process to initialize data support and compare their resulting states at the end of the test.
    - (refer to "Reproducible" and "Observable" laws from the 3 laws of testing)
2. Automated script code generation
    - Processes that can generate script code by:
        - monitoring user or system activities
        - reading prompt or any other support instructions.
3. Automated batch running for interactive processes
    - This is the process of running a test scenario in batch mode that originally ran in interactive mode.
4. Automated data subset extrapolation
    - This is the process of shrinking data to only what is necessary and sufficient to run the test scenario.
5. Automated code coverage reporting & merging
    - This is the process of getting code coverages reporting (per program or module) and being able to merge them as well with multiple variations of the same test scenario (i.e. calling the same program with different inputs values).
6. Automated integration of objects security attributes in replay mode
    - When objects are restored, their ownership and authority settings may be altered by the system.
    - A process is therefore required to reapply and align all object attributes and security settings with their original configuration.
7. Automated workload with differentiated records flow
    - This process involves running test scenarios in parallel, where each workload operates on different sets of records within the same table.
    - By distributing record access, it minimizes contention and helps prevent locking issues during concurrent updates.

Levels of operability:

We can identify up to 3 levels of operability:

1. Single job encapsulation
    - Test scenarios using a single job, typically 5250 or a single batch job.
2. Multiple jobs encapsulation
    - Test scenarios using multiple jobs, typically web environment, multiple environments coexistence or multiple threads.
3. Multiple platform encapsulation & hybrid systems
    - Test scenarios running over multiple platforms.

Levels of integration with AI:

We can identify up to 3 levels of integration with AI:

1. Testing to give meta-data to AI
    - Tests that drive AI.
2. AI to give indications for testing
    - AI that drives testing.
3. Autonomous testing (AI + MCP)
    - AI that autonomously calls testing.
        - To test AI outcome (when AI generates code).
        - To test other processes.

Why Automation Maturity Matters

Moving up these levels enables organizations to:

- release software faster
- improve test coverage
- reduce maintenance effort
- detect defects earlier
- leverage AI with confidence.

For enterprise platforms such as **IBM i** there is a multi-dimensional view of test automation maturity, structured across three key axes:

Levels of automation (from manual to highly optimized execution) Levels of operability (from single-job to multi-platform orchestration) Levels of integration with AI (from support to full autonomy)

At its core, the model highlights a fundamental shift: Organizations shift from manual, effort-heavy validation to automated, intelligent, and eventually autonomous testing systems.

> **Tip:** Tools are highly valuable here. For more information, see Test-driven development with automation and What testing tools work with IBM i?.

IBM i Performance Optimization Tools and Techniques

Database Performance Analysis Tools

What This Covers

You'll learn how to leverage the Index Advisor tool to identify and create optimal indexes for your Db2 for i queries. This guide covers accessing index recommendations, understanding the advice provided, and implementing indexes that improve query performance while avoiding common pitfalls.

Before You Start

- ☐ Access to IBM i Access Client Solutions (ACS) installed on your workstation

- ☐
  Authority to view database schemas and create indexes on target tables
- ☐
  Basic understanding of SQL queries and Db2 for i index concepts
- ☐
  Familiarity with your application's query patterns and performance requirements

Steps

1. Access the System-wide Index Advisor

The Index Advisor runs automatically on your system, collecting recommendations as queries execute. To access the advice:

- Open IBM i Access Client Solutions (ACS)
- Navigate to the Schemas tool
- Right-click on your database name, schema, or specific table
- Select the "Index Advisor" task from the context menu

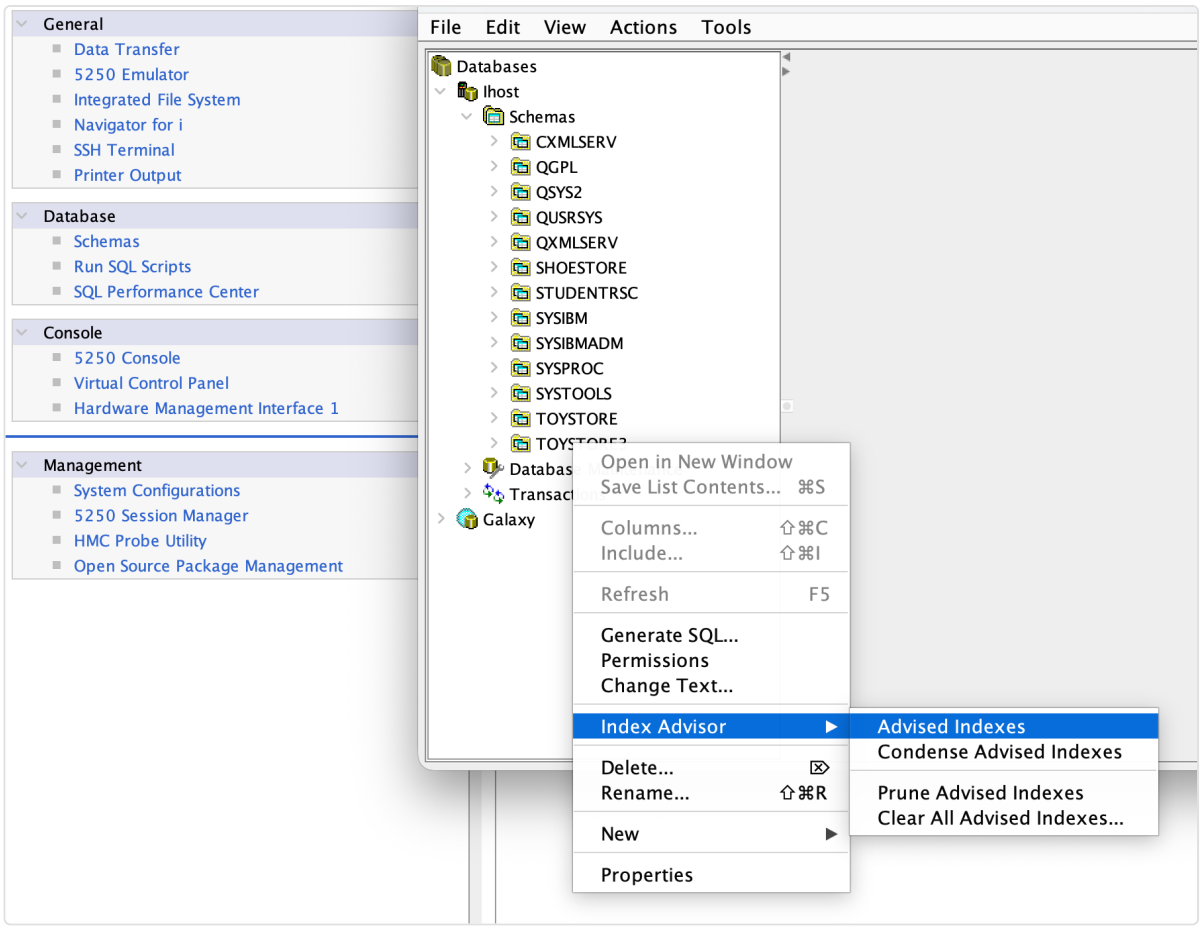


Figure. Index Advisor context menu

What to expect: A window displays all index recommendations with columns showing the advised index keys, table name, schema, index type (radix or EVI), number of times advised, and last advised timestamp. The interface allows sorting and filtering to prioritize which indexes to create first.

2. Analyze and Prioritize Index Recommendations

Review the index advice systematically to identify high-impact opportunities:

- **Sort by "Times Advised"** to find indexes recommended most frequently—these typically provide the broadest benefit across multiple queries

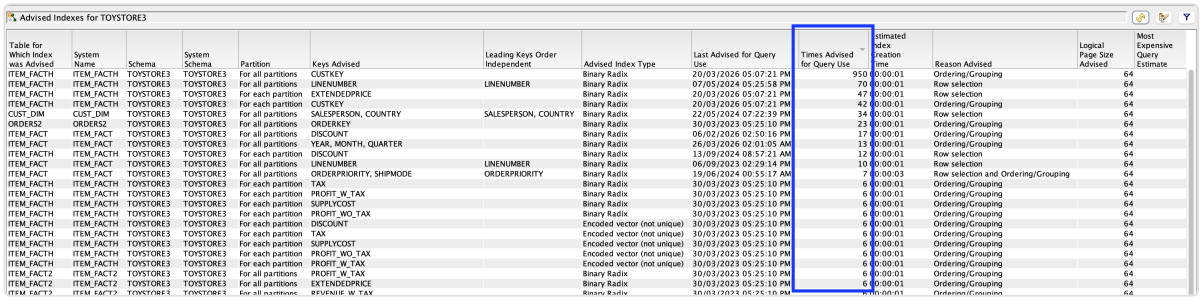

Advised Indexes for TOYSTORE3

Table for Which Index was Advised	System Name	Schema	System Schema	Partition	Keys Advised	Leading Keys Order Independent	Advised Index Type	Last Advised for Query Use	Times Advised for Query Use	Estimated Index Creation Time	Reason Advised	Logical Page Size Advised	Most Expensive Query Estimate
ITEM_FACTH	ITEM_FACTH	TOYSTORE3	TOYSTORE3	For all partitions	CUSTKEY		Binary Radix	20/03/2026 05:07:21 PM	950	00:00:01	Ordering/Grouping	64	
ITEM_FACTH	ITEM_FACTH	TOYSTORE3	TOYSTORE3	For all partitions	LINENUMBER	LINENUMBER	Binary Radix	07/05/2024 05:25:58 PM	70	00:00:01	Row selection	64	
ITEM_FACTH	ITEM_FACTH	TOYSTORE3	TOYSTORE3	For each partition	EXTENDEDPRICE		Binary Radix	20/03/2026 05:07:21 PM	47	00:00:01	Row selection	64	
ITEM_FACTH	ITEM_FACTH	TOYSTORE3	TOYSTORE3	For each partition	CUSTKEY		Binary Radix	20/03/2026 05:07:21 PM	42	00:00:01	Ordering/Grouping	64	
CUST_DIM	CUST_DIM	TOYSTORE3	TOYSTORE3	For all partitions	SALESPERSON, COUNTRY	SALESPERSON, COUNTRY	Binary Radix	22/05/2024 07:22:39 PM	34	00:00:01	Row selection	64	
ORDERS2	ORDERS2	TOYSTORE3	TOYSTORE3	For all partitions	ORDERKEY		Binary Radix	30/03/2023 05:25:10 PM	23	00:00:01	Ordering/Grouping	64	
ITEM_FACT	ITEM_FACT	TOYSTORE3	TOYSTORE3	For all partitions	DISCOUNT		Binary Radix	06/02/2026 02:50:16 PM	17	00:00:01	Ordering/Grouping	64	
ITEM_FACT	ITEM_FACT	TOYSTORE3	TOYSTORE3	For all partitions	YEAR, MONTH, QUARTER		Binary Radix	26/03/2026 02:01:05 AM	13	00:00:01	Ordering/Grouping	64	
ITEM_FACTH	ITEM_FACTH	TOYSTORE3	TOYSTORE3	For each partition	DISCOUNT		Binary Radix	13/09/2024 08:57:21 AM	12	00:00:01	Row selection	64	
ITEM_FACT	ITEM_FACT	TOYSTORE3	TOYSTORE3	For all partitions	LINENUMBER	LINENUMBER	Binary Radix	06/09/2023 02:29:14 PM	10	00:00:01	Row selection	64	
ITEM_FACT	ITEM_FACT	TOYSTORE3	TOYSTORE3	For all partitions	ORDERPRIORITY, SHIPMODE	ORDERPRIORITY	Binary Radix	19/06/2024 00:55:17 AM	7	00:00:03	Row selection and Ordering/Grouping	64	
ITEM_FACTH	ITEM_FACTH	TOYSTORE3	TOYSTORE3	For each partition	TAX		Binary Radix	30/03/2023 05:25:10 PM	6	00:00:01	Ordering/Grouping	64	
ITEM_FACTH	ITEM_FACTH	TOYSTORE3	TOYSTORE3	For each partition	PROFIT_W_TAX		Binary Radix	30/03/2023 05:25:10 PM	6	00:00:01	Ordering/Grouping	64	
ITEM_FACTH	ITEM_FACTH	TOYSTORE3	TOYSTORE3	For each partition	SUPPLYCOST		Binary Radix	30/03/2023 05:25:10 PM	6	00:00:01	Ordering/Grouping	64	
ITEM_FACTH	ITEM_FACTH	TOYSTORE3	TOYSTORE3	For each partition	PROFIT_WO_TAX		Binary Radix	30/03/2023 05:25:10 PM	6	00:00:01	Ordering/Grouping	64	
ITEM_FACTH	ITEM_FACTH	TOYSTORE3	TOYSTORE3	For each partition	DISCOUNT		Encoded vector (not unique)	30/03/2023 05:25:10 PM	6	00:00:01	Ordering/Grouping	64	
ITEM_FACTH	ITEM_FACTH	TOYSTORE3	TOYSTORE3	For each partition	TAX		Encoded vector (not unique)	30/03/2023 05:25:10 PM	6	00:00:01	Ordering/Grouping	64	
ITEM_FACTH	ITEM_FACTH	TOYSTORE3	TOYSTORE3	For each partition	SUPPLYCOST		Encoded vector (not unique)	30/03/2023 05:25:10 PM	6	00:00:01	Ordering/Grouping	64	
ITEM_FACTH	ITEM_FACTH	TOYSTORE3	TOYSTORE3	For each partition	PROFIT_WO_TAX		Encoded vector (not unique)	30/03/2023 05:25:10 PM	6	00:00:01	Ordering/Grouping	64	
ITEM_FACTH	ITEM_FACTH	TOYSTORE3	TOYSTORE3	For each partition	PROFIT_W_TAX		Encoded vector (not unique)	30/03/2023 05:25:10 PM	6	00:00:01	Ordering/Grouping	64	
ITEM_FACT2	ITEM_FACT2	TOYSTORE3	TOYSTORE3	For all partitions	PROFIT_W_TAX		Binary Radix	30/03/2023 05:25:10 PM	6	00:00:01	Ordering/Grouping	64	
ITEM_FACT2	ITEM_FACT2	TOYSTORE3	TOYSTORE3	For all partitions	EXTENDEDPRICE		Binary Radix	30/03/2023 05:25:10 PM	6	00:00:01	Ordering/Grouping	64	

Figure. Sort by Times Advised

- **Check "Last Advised" dates** to identify recently recommended indexes that address current performance issues

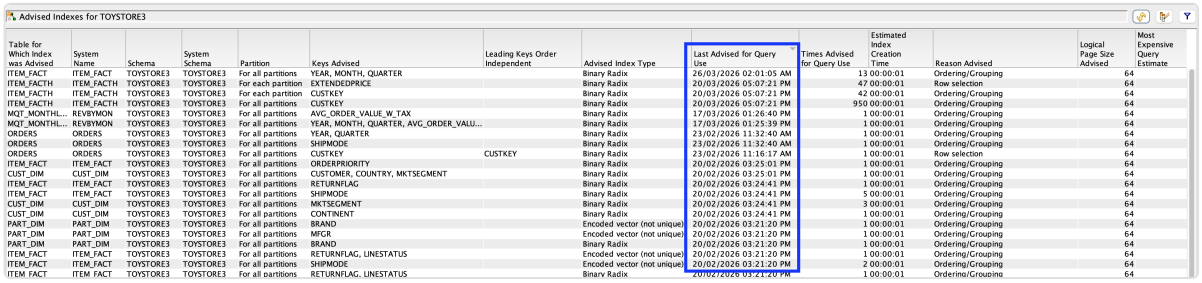

Advised Indexes for TOYSTORE3

Table for Which Index was Advised	System Name	Schema	System Schema	Partition	Keys Advised	Leading Keys Order Independent	Advised Index Type	Last Advised for Query Use	Times Advised for Query Use	Estimated Index Creation Time	Reason Advised	Logical Page Size Advised	Most Expensive Query Estimate
ITEM_FACT	ITEM_FACT	TOYSTORE3	TOYSTORE3	For all partitions	YEAR, MONTH, QUARTER		Binary Radix	26/03/2026 02:01:05 AM	13	00:00:01	Ordering/Grouping	64	
ITEM_FACTH	ITEM_FACTH	TOYSTORE3	TOYSTORE3	For each partition	EXTENDEDPRICE		Binary Radix	20/03/2026 05:07:21 PM	47	00:00:01	Row selection	64	
ITEM_FACTH	ITEM_FACTH	TOYSTORE3	TOYSTORE3	For each partition	CUSTKEY		Binary Radix	20/03/2026 05:07:21 PM	42	00:00:01	Ordering/Grouping	64	
ITEM_FACTH	ITEM_FACTH	TOYSTORE3	TOYSTORE3	For all partitions	CUSTKEY		Binary Radix	20/03/2026 05:07:21 PM	950	00:00:01	Ordering/Grouping	64	
MQT_MONTHL...	REVBYMON	TOYSTORE3	TOYSTORE3	For all partitions	AVG_ORDER_VALUE_W_TAX		Binary Radix	17/03/2026 01:26:40 PM	1	00:00:01	Ordering/Grouping	64	
MQT_MONTHL...	REVBYMON	TOYSTORE3	TOYSTORE3	For all partitions	YEAR, MONTH, QUARTER, AVG_ORDER_VALU...		Binary Radix	17/03/2026 01:25:39 PM	1	00:00:01	Ordering/Grouping	64	
ORDERS	ORDERS	TOYSTORE3	TOYSTORE3	For all partitions	YEAR, QUARTER		Binary Radix	23/02/2026 11:32:40 AM	1	00:00:01	Ordering/Grouping	64	
ORDERS	ORDERS	TOYSTORE3	TOYSTORE3	For all partitions	SHIPMODE		Binary Radix	23/02/2026 11:32:40 AM	1	00:00:01	Ordering/Grouping	64	
ORDERS	ORDERS	TOYSTORE3	TOYSTORE3	For all partitions	CUSTKEY	CUSTKEY	Binary Radix	23/02/2026 11:16:17 AM	1	00:00:01	Row selection	64	
ITEM_FACT	ITEM_FACT	TOYSTORE3	TOYSTORE3	For all partitions	ORDERPRIORITY		Binary Radix	20/02/2026 03:25:01 PM	1	00:00:01	Ordering/Grouping	64	
CUST_DIM	CUST_DIM	TOYSTORE3	TOYSTORE3	For all partitions	CUSTOMER, COUNTRY, MKTSEGMENT		Binary Radix	20/02/2026 03:25:01 PM	1	00:00:01	Ordering/Grouping	64	
ITEM_FACT	ITEM_FACT	TOYSTORE3	TOYSTORE3	For all partitions	RETURNFLAG		Binary Radix	20/02/2026 03:24:41 PM	1	00:00:01	Ordering/Grouping	64	
ITEM_FACT	ITEM_FACT	TOYSTORE3	TOYSTORE3	For all partitions	SHIPMODE		Binary Radix	20/02/2026 03:24:41 PM	5	00:00:01	Ordering/Grouping	64	
CUST_DIM	CUST_DIM	TOYSTORE3	TOYSTORE3	For all partitions	MKTSEGMENT		Binary Radix	20/02/2026 03:24:41 PM	3	00:00:01	Ordering/Grouping	64	
CUST_DIM	CUST_DIM	TOYSTORE3	TOYSTORE3	For all partitions	CONTINENT		Binary Radix	20/02/2026 03:24:41 PM	1	00:00:01	Ordering/Grouping	64	
PART_DIM	PART_DIM	TOYSTORE3	TOYSTORE3	For all partitions	BRAND		Encoded vector (not unique)	20/02/2026 03:21:20 PM	1	00:00:01	Ordering/Grouping	64	
PART_DIM	PART_DIM	TOYSTORE3	TOYSTORE3	For all partitions	MFGR		Encoded vector (not unique)	20/02/2026 03:21:20 PM	1	00:00:01	Ordering/Grouping	64	
PART_DIM	PART_DIM	TOYSTORE3	TOYSTORE3	For all partitions	BRAND		Binary Radix	20/02/2026 03:21:20 PM	1	00:00:01	Ordering/Grouping	64	
ITEM_FACT	ITEM_FACT	TOYSTORE3	TOYSTORE3	For all partitions	RETURNFLAG, LINESTATUS		Encoded vector (not unique)	20/02/2026 03:21:20 PM	1	00:00:01	Ordering/Grouping	64	
ITEM_FACT	ITEM_FACT	TOYSTORE3	TOYSTORE3	For all partitions	SHIPMODE		Encoded vector (not unique)	20/02/2026 03:21:20 PM	2	00:00:01	Ordering/Grouping	64	
ITEM_FACT	ITEM_FACT	TOYSTORE3	TOYSTORE3	For all partitions	RETURNFLAG, LINESTATUS		Binary Radix	20/02/2026 03:21:20 PM	1	00:00:01	Ordering/Grouping	64	

Figure. Check Last Advised dates

- **Review key column order** to understand which predicates the optimizer considers most selective—note that leading-order-independent keys allow the optimizer to use any combination of key columns for index probes

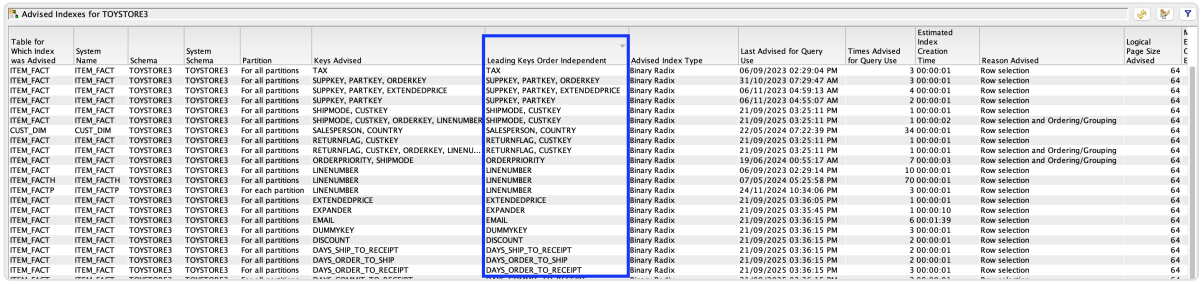
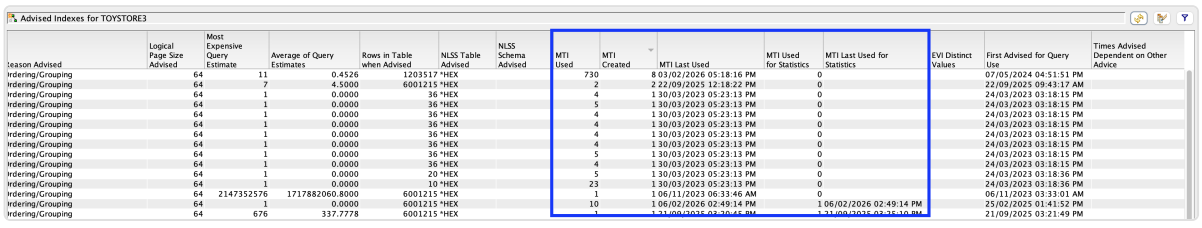

Advised Indexes for TOYSTORE3

Table for Which Index was Advised	System Name	Schema	System Schema	Partition	Keys Advised	Leading Keys Order Independent	Advised Index Type	Last Advised for Query Use	Times Advised for Query Use	Estimated Index Creation Time	Reason Advised	Logical Page Size Advised
ITEM_FACT	ITEM_FACT	TOYSTORE3	TOYSTORE3	For all partitions	TAX	TAX	Binary Radix	06/09/2023 02:29:04 PM	3	00:00:01	Row selection	64
ITEM_FACT	ITEM_FACT	TOYSTORE3	TOYSTORE3	For all partitions	SUPPKEY, PARTKEY, ORDERKEY	SUPPKEY, PARTKEY, ORDERKEY	Binary Radix	31/10/2023 07:29:47 AM	3	00:00:01	Row selection	64
ITEM_FACT	ITEM_FACT	TOYSTORE3	TOYSTORE3	For all partitions	SUPPKEY, PARTKEY, EXTENDEDPRICE	SUPPKEY, PARTKEY, EXTENDEDPRICE	Binary Radix	06/11/2023 04:59:13 AM	4	00:00:01	Row selection	64
ITEM_FACT	ITEM_FACT	TOYSTORE3	TOYSTORE3	For all partitions	SUPPKEY, PARTKEY	SUPPKEY, PARTKEY	Binary Radix	06/11/2023 04:55:07 AM	2	00:00:01	Row selection	64
ITEM_FACT	ITEM_FACT	TOYSTORE3	TOYSTORE3	For all partitions	SHIPMODE, CUSTKEY	SHIPMODE, CUSTKEY	Binary Radix	21/09/2025 03:25:11 PM	1	00:00:01	Row selection	64
ITEM_FACT	ITEM_FACT	TOYSTORE3	TOYSTORE3	For all partitions	SHIPMODE, CUSTKEY, ORDERKEY, LINENUMBER	SHIPMODE, CUSTKEY	Binary Radix	21/09/2025 03:25:11 PM	1	00:00:02	Row selection and Ordering/Grouping	64
CUST_DIM	CUST_DIM	TOYSTORE3	TOYSTORE3	For all partitions	SALESPERSON, COUNTRY	SALESPERSON, COUNTRY	Binary Radix	22/05/2024 07:22:39 PM	34	00:00:01	Row selection	64
ITEM_FACT	ITEM_FACT	TOYSTORE3	TOYSTORE3	For all partitions	RETURNFLAG, CUSTKEY	RETURNFLAG, CUSTKEY	Binary Radix	21/09/2025 03:25:11 PM	1	00:00:01	Row selection	64
ITEM_FACT	ITEM_FACT	TOYSTORE3	TOYSTORE3	For all partitions	RETURNFLAG, CUSTKEY, ORDERKEY, LINENU...	RETURNFLAG, CUSTKEY	Binary Radix	21/09/2025 03:25:11 PM	1	00:00:01	Row selection and Ordering/Grouping	64
ITEM_FACT	ITEM_FACT	TOYSTORE3	TOYSTORE3	For all partitions	ORDERPRIORITY, SHIPMODE	ORDERPRIORITY	Binary Radix	19/06/2024 00:55:17 AM	7	00:00:03	Row selection and Ordering/Grouping	64
ITEM_FACT	ITEM_FACT	TOYSTORE3	TOYSTORE3	For all partitions	LINENUMBER	LINENUMBER	Binary Radix	06/09/2023 02:29:14 PM	10	00:00:01	Row selection	64
ITEM_FACTH	ITEM_FACTH	TOYSTORE3	TOYSTORE3	For all partitions	LINENUMBER	LINENUMBER	Binary Radix	07/05/2024 05:25:58 PM	70	00:00:01	Row selection	64
ITEM_FACTP	ITEM_FACTP	TOYSTORE3	TOYSTORE3	For each partition	LINENUMBER	LINENUMBER	Binary Radix	24/11/2024 10:34:06 PM	3	00:00:01	Row selection	64
ITEM_FACT	ITEM_FACT	TOYSTORE3	TOYSTORE3	For all partitions	EXTENDEDPRICE	EXTENDEDPRICE	Binary Radix	21/09/2025 03:36:05 PM	1	00:00:01	Row selection	64
ITEM_FACT	ITEM_FACT	TOYSTORE3	TOYSTORE3	For all partitions	EXPANDER	EXPANDER	Binary Radix	21/09/2025 03:35:45 PM	1	00:00:10	Row selection	64
ITEM_FACT	ITEM_FACT	TOYSTORE3	TOYSTORE3	For all partitions	EMAIL	EMAIL	Binary Radix	21/09/2025 03:36:15 PM	6	00:01:39	Row selection	64
ITEM_FACT	ITEM_FACT	TOYSTORE3	TOYSTORE3	For all partitions	DUMMYKEY	DUMMYKEY	Binary Radix	21/09/2025 03:36:15 PM	3	00:00:01	Row selection	64
ITEM_FACT	ITEM_FACT	TOYSTORE3	TOYSTORE3	For all partitions	DISCOUNT	DISCOUNT	Binary Radix	21/09/2025 03:36:15 PM	2	00:00:01	Row selection	64
ITEM_FACT	ITEM_FACT	TOYSTORE3	TOYSTORE3	For all partitions	DAYS_SHIP_TO_RECEIPT	DAYS_SHIP_TO_RECEIPT	Binary Radix	21/09/2025 03:36:15 PM	2	00:00:01	Row selection	64
ITEM_FACT	ITEM_FACT	TOYSTORE3	TOYSTORE3	For all partitions	DAYS_ORDER_TO_SHIP	DAYS_ORDER_TO_SHIP	Binary Radix	21/09/2025 03:36:15 PM	2	00:00:01	Row selection	64
ITEM_FACT	ITEM_FACT	TOYSTORE3	TOYSTORE3	For all partitions	DAYS_ORDER_TO_RECEIPT	DAYS_ORDER_TO_RECEIPT	Binary Radix	21/09/2025 03:36:15 PM	3	00:00:01	Row selection	64

Figure. Review key column order

- **Look for temporary index indicators** (MTI columns) showing indexes the optimizer has already created automatically


Advised Indexes for TOYSTORE3

ason Advised	Logical Page Size Advised	Most Expensive Query Estimate	Average of Query Estimates	Rows in Table when Advised	NLSS Table Advised	NLSS Schema Advised	MTI Used	MTI Created	MTI Last Used	MTI Used for Statistics	MTI Last Used for Statistics	EVI Distinct Values	First Advised for Query Use	Times Advised Dependent on Other Advice
rdering/Grouping	64	11	0.4526	1203517	*HEX		730	8	03/02/2026 05:18:16 PM	0			07/05/2024 04:51:51 PM	
rdering/Grouping	64	7	4.5000	6001215	*HEX		2	2	22/09/2025 12:18:22 PM	0			22/09/2025 09:43:17 AM	
rdering/Grouping	64	1	0.0000	36	*HEX		4	1	30/03/2023 05:23:13 PM	0			24/03/2023 03:18:15 PM	
rdering/Grouping	64	1	0.0000	36	*HEX		5	1	30/03/2023 05:23:13 PM	0			24/03/2023 03:18:15 PM	
rdering/Grouping	64	1	0.0000	36	*HEX		4	1	30/03/2023 05:23:13 PM	0			24/03/2023 03:18:15 PM	
rdering/Grouping	64	1	0.0000	36	*HEX		4	1	30/03/2023 05:23:13 PM	0			24/03/2023 03:18:15 PM	
rdering/Grouping	64	1	0.0000	36	*HEX		4	1	30/03/2023 05:23:13 PM	0			24/03/2023 03:18:15 PM	
rdering/Grouping	64	1	0.0000	36	*HEX		4	1	30/03/2023 05:23:13 PM	0			24/03/2023 03:18:15 PM	
rdering/Grouping	64	1	0.0000	36	*HEX		5	1	30/03/2023 05:23:13 PM	0			24/03/2023 03:18:15 PM	
rdering/Grouping	64	1	0.0000	36	*HEX		4	1	30/03/2023 05:23:13 PM	0			24/03/2023 03:18:15 PM	
rdering/Grouping	64	1	0.0000	20	*HEX		5	1	30/03/2023 05:23:13 PM	0			24/03/2023 03:18:36 PM	
rdering/Grouping	64	1	0.0000	10	*HEX		23	1	30/03/2023 05:23:13 PM	0			24/03/2023 03:18:36 PM	
rdering/Grouping	64	2147352576	1717882060.8000	6001215	*HEX		1	1	06/11/2023 06:33:46 AM	0			06/11/2023 03:33:01 AM	
rdering/Grouping	64	1	0.0000	6001215	*HEX		10	1	06/02/2026 02:49:14 PM	1	06/02/2026 02:49:14 PM		25/02/2025 01:41:52 PM	
rdering/Grouping	64	676	337.7778	6001215	*HEX		1	1	21/09/2025 03:35:45 PM	1	21/09/2025 03:35:45 PM		21/09/2025 03:21:49 PM	

Figure. Temporary index indicators

What to expect: You'll identify patterns in the recommendations. Multiple similar indexes for the same table may indicate opportunities for a single, more comprehensive index. High "Times Advised" counts signal indexes that will benefit many queries. Remember that Index Advisor shows what the optimizer considered (costed) during query planning, not necessarily what it will use—you may see advice even for small tables where indexes won't significantly impact performance.

3. Validate Index Recommendations Before Creating

Index Advisor shows what the optimizer considered during query planning, not guarantees of usage. Before creating advised indexes:

- **Examine the underlying query** using Visual Explain or the SQL Plan Cache to understand why the index was advised and whether it will actually be used
- **Check existing indexes** using the Index Evaluator to avoid creating duplicate or redundant indexes
- **Consider data characteristics**: If the advised index assumes high selectivity but your data is evenly distributed, the optimizer may choose table scans instead
- **Evaluate index costs**: Each index consumes disk storage and requires CPU cycles to maintain during INSERT, UPDATE, and DELETE operations
- **Use the Condense Advised Indexes feature** to eliminate overlapping recommendations and identify comprehensive indexes that serve multiple queries

What to expect: You may discover that some advised indexes won't provide value due to data distribution or table size. The condensed view helps identify the minimal set of indexes needed for maximum benefit while minimizing storage and maintenance overhead.

4. Create and Test Indexes Strategically

Implement indexes methodically to measure their impact:

- **Start with high-frequency recommendations** that appear across multiple queries
- **Create indexes one at a time** or in small groups to isolate performance improvements
- **Leverage parallel processing** by using Db2 SMP (Symmetric Multiprocessing) to reduce total index creation time —multiple processors can work simultaneously to build indexes faster
- **Use the "Create Index" option** by right-clicking an advised index in ACS, or generate SQL scripts for version control
- **Include the WITH n DISTINCT VALUES clause** for EVIs to optimize creation and maintenance performance
- **Test query performance** before and after index creation using Visual Explain or the SQL Performance Monitor

What to expect: Advised indexes may or may not improve performance depending on data distribution, table size, and query patterns. The optimizer considers many factors when choosing access methods. Monitor query plans after index creation to confirm whether the new index is being utilized and whether performance actually improves.

5. Monitor and Maintain Your Index Strategy

Index Advisor is an ongoing tool for continuous optimization:

- **Review index usage** regularly using the Index Evaluator to identify unused indexes consuming storage and CPU maintenance overhead
- **Check for new recommendations** weekly or monthly as application usage patterns evolve
- **Replace temporary indexes (MTIs)** with permanent indexes to maintain performance after system restarts—temporary indexes are automatically deleted during IPLs
- **Reset usage counts** after creating new indexes to measure their actual utilization
- **Balance index benefits against costs**: Each index requires disk space and CPU cycles during data modifications
- **Document index decisions** including why specific indexes were created or rejected for future reference

What to expect: Your index strategy will evolve as applications change. Regular monitoring ensures indexes remain relevant and identifies opportunities for further optimization. Unused indexes should be removed to reduce storage consumption and maintenance overhead during INSERT, UPDATE, and DELETE operations.

Verify It Worked

Confirm your index implementation was successful:

- **Check Visual Explain** to verify the optimizer now uses your new index in query plans
- **Compare query execution times** before and after index creation using the SQL Performance Monitor
- **Review the Index Evaluator** to confirm the new index shows usage in the "Query Use Count" column
- **Monitor system performance** to ensure index maintenance overhead doesn't negatively impact transaction processing
- **Validate with the SQL Plan Cache** that queries using the new index show improved estimated costs

Key Takeaways

- **Index Advisor shows what the optimizer considered, not what it will use**—the advice represents query execution plans that were costed during optimization, so always validate recommendations against actual query plans and performance
- **The System-wide Index Advisor runs automatically**, collecting recommendations from all queries without requiring manual monitoring activation, making it easy to identify optimization opportunities
- **Understand that advice appears even for small tables** where indexes may not significantly impact performance —this design helps developers see all optimization considerations
- **Balance index benefits against costs**: Each index consumes disk storage and requires CPU cycles to maintain during data modifications, so prioritize high-frequency recommendations using the Condense feature
- **Leverage leading-order-independent keys** which allow the optimizer to use any combination of key columns for index probes, and use parallel processing to reduce index creation time on systems with multiple processors

What Key Features Exist Within Visual Explain?

What This Covers

This section provides an overview of the many key features available within Visual Explain.

What Is Visual Explain?

Visual Explain is a diagnostic and analysis tool that graphically represents how the SQL Query Engine (SQE) chooses to execute a query. The result of SQE's query implementation is known as an *access plan*. Visual Explain displays the access plan as an interactive diagram, making SQE's decisions easier to understand, analyze, and act upon.

Tuning and troubleshooting SQL performance is an important cornerstone of application performance. By using the many features within Visual Explain, database engineers can inspect SQL query access plans to ensure they run with speed and stability.

What Features Does Visual Explain Offer?

Tuning and troubleshooting SQL query performance often requires careful inspection of access plans to identify root causes. Visual Explain offers a variety of features and tools that allow users to deeply analyze query implementation. These features include:

- Row count statistics for each node
- Runtime statistics
- Index information, including:
    - Index Advisor

    - Indexes considered

- Node highlighting and arrow labeling

Row Count Statistics

Visual Explain displays both the estimated and actual number of rows processed and selected by each node. Estimated row counts are determined during query optimization, where SQE uses statistical models to estimate how a query processes the requested data. Actual row counts are collected during query execution and represent the true number of rows processed.

The *selected* row count of a node indicates the number of rows returned, while the *processed* row count indicates the number of rows a given node has received as input. Rows may be processed but not selected for a variety of reasons, including predicate filtering, aggregation or distinct processing, and more.

When generating a Visual Explain diagram, users may decide between three options:

- Explain
- Run and Explain
- Explain While Running

The *Explain* option produces an access plan and displays it to the user—the query does not execute. Like *Explain*, *Run and Explain* produces an access plan and displays it to the user. In addition, *Run and Explain* executes the query and returns a result set. For both *Explain* and *Run and Explain*, only **estimated** row count statistics are displayed. Finally, *Explain While Running* functions similarly to *Run and Explain*, where the query is executed alongside producing and displaying an access plan. However, *Explain While Running* collects and displays **actual** row count statistics. Actual row count statistics are also displayed when Visual Explain is launched from a Performance Monitor (Database Monitor) or from the Live Plan Cache or Plan Cache Snapshot tools.

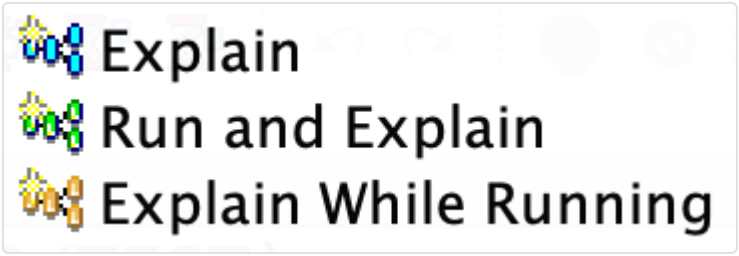


Figure. Visual Explain types

By examining row counts for each node, a database engineer can identify where potential hot spots may occur within a query. As more rows are processed, CPU usage and database read and write activity may increase. When tuning an SQL query, nodes with the largest row counts often provide the greatest opportunity for performance improvement.

Runtime Statistics

Many metrics are captured during query execution, including:

- Query runtime
- CPU time
- Database read, write, and page fault counts
- Time spent waiting on object, record, and seize locks

Actual Runtime Information	
Optimization Time (ms)	1,585
Longest Key Range Estimate (ms)	0
Key Range Estimate Timed Out	No
Average Run Time (ms)	737
Average Statement Open Time (ms)	Not Available
Average Statement Fetch Time (ms)	737
Average Statement Close Time (ms)	Not Available
Average Rows Fetched	1,443
Total Times Query Was Run	4
Total Time For All Runs (ms)	2,949
Average CPU Time (ms)	81
Average Synchronous Database Reads	3
Average Asynchronous Database Reads	1,523
Average Page Faults	15
Average Object Lock Wait Time (ms)	0
Average Object Lock Wait Count	0
Average Record Lock Wait Time (ms)	0
Average Record Lock Wait Count	0
Average Seize Wait Time (ms)	0
Average Seize Wait Count	0
SMP Degree Adjusted During Runtime	N/A
Re-Optimization Skipped by Optimizer	No

Figure. Runtime statistics displayed by Visual Explain

The **Final Select** node in every Visual Explain diagram displays these metrics on a query-wide basis.

These statistics can help database engineers understand why a particular SQL query is experiencing performance issues. For example, a query experiencing a significant degradation in performance may exhibit a large quantity of page faults, high record lock contention, higher-than-expected CPU time, or a myriad of other problems. The underlying cause could include memory pool sizing issues, poor SQL query coding techniques, an inadequate indexing strategy, or many other possibilities.

Every SQL query performance scenario is different, and these runtime statistics play a valuable role in diagnosing the particular type of problem a query is facing.

Index Information

Developing a strong indexing strategy is a cornerstone of SQL performance, and index advice can help identify potential shortcomings in an existing strategy.

The Index Advisor contains entries known as *index advice*. Index advice is generated by SQE when an index does not exist, but query optimization could potentially benefit if the advised index were available.

If index advice is followed and additional indexes are created, the number of optimization techniques available to SQE increases. A greater number and variety of techniques can significantly improve SQL performance. However, users should exercise caution when creating new indexes based on index advice, as a new optimization technique is not guaranteed to result in improved runtime performance. Additional indexes also require additional resources to maintain.

In addition to Index Advisor output, various table access nodes within Visual Explain provide a right-click option to **Show Indexes Considered**. This tool displays all indexes evaluated by SQE during optimization. Indexes may be considered but not used for a variety of reasons, most commonly because they provide suboptimal performance compared to alternative indexes or table access methods. When an index is not selected, the reason code and reason text are displayed. The below screenshot shows an example output of Show Indexes Considered. By understanding why an index was not used, database engineers can adjust their indexing strategy accordingly.

Index	Reason	Description
STAR1G.CUSTEVI	21	The access path cannot be used because not all referenced columns had an Encoded Vector Index.
STAR1G.ITEM_123	4	The cost to use this access path, as determined by the optimizer, was higher than the cost associated
STAR1G.ITEM_EVI1	21	The access path cannot be used because not all referenced columns had an Encoded Vector Index.
STAR1G.ITEM_EVI2	21	The access path cannot be used because not all referenced columns had an Encoded Vector Index.
STAR1G.ITEM_EVI4	21	The access path cannot be used because not all referenced columns had an Encoded Vector Index.
STAR1G.ITEM_EVI5	21	The access path cannot be used because not all referenced columns had an Encoded Vector Index.
STAR1G.ITEM_EVI6	21	The access path cannot be used because not all referenced columns had an Encoded Vector Index.
STAR1G.ITEM_EVI7	21	The access path cannot be used because not all referenced columns had an Encoded Vector Index.
STAR1G.ITEM_EVI8	21	The access path cannot be used because not all referenced columns had an Encoded Vector Index.
STAR1G.ITEM_IDX2	0	The access path was used to implement the query.
STAR1G.PARTEVI	17	The left-most key of the access path did not match any fields specified for the selection criteria. Ther
STAR1G.PKEYSKEY	4	The cost to use this access path, as determined by the optimizer, was higher than the cost associated

Figure. Show Indexes Considered

Node Highlighting and Arrow Labeling

Visual Explain includes node highlighting and arrow labeling—two features that improve navigation within access plan diagrams.

The labels on node-connecting arrows can be configured to display various metrics, including estimated and actual row counts, SMP degree, and more. These labels enable at-a-glance analysis of commonly used metrics within an access plan.

Node highlighting can also be enabled to color-code nodes based on selected attributes. This feature is especially useful in large diagrams, making it easier to pinpoint potential performance issues. This example Visual Explain diagram shows two highlighted nodes - highest actual row count (brown) and index advice (blue). Additionally, both the estimated and actual row count arrow labels are displayed.

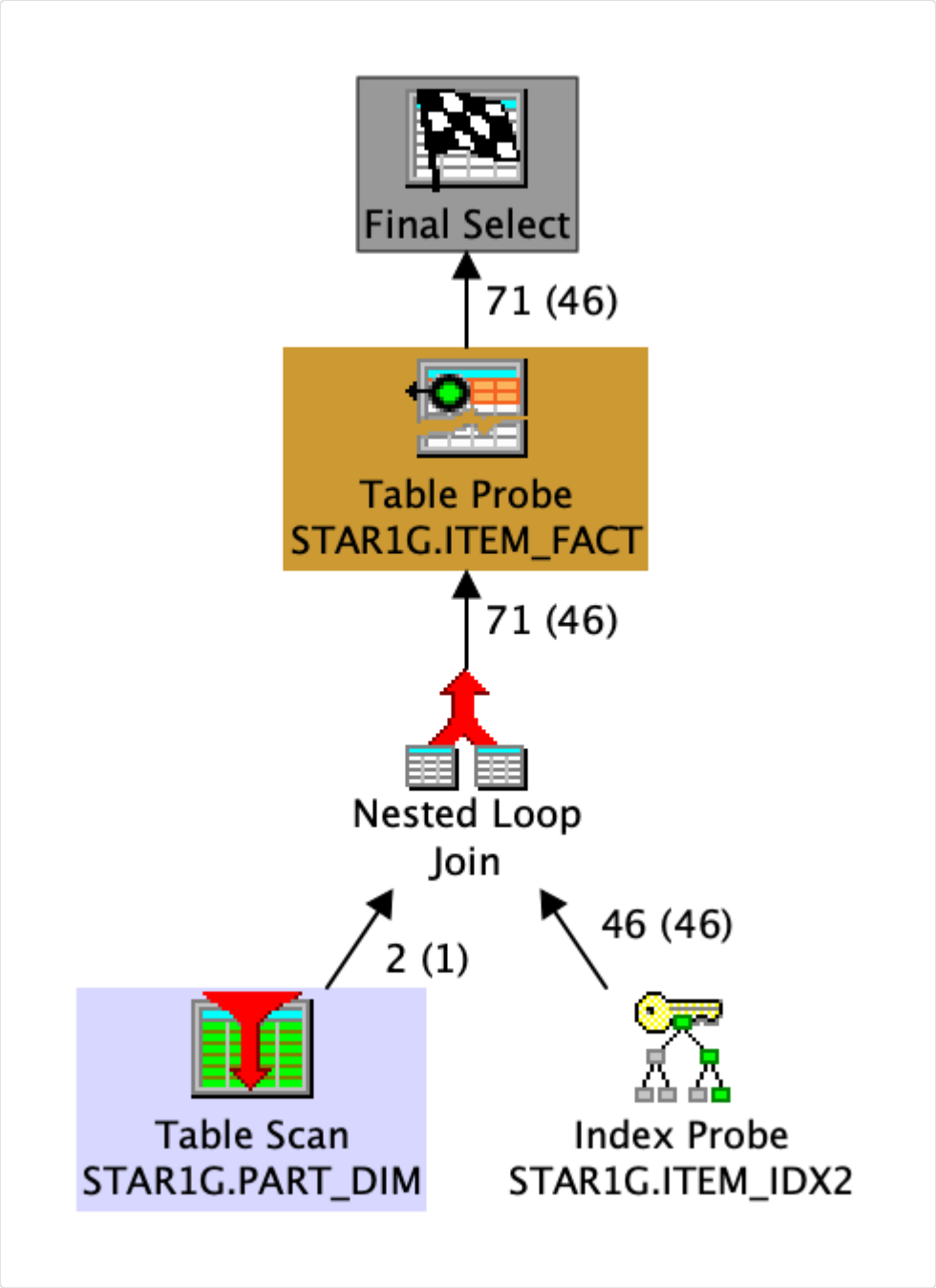


Figure. Node highlighting and arrow labeling in Visual Explain

Key Takeaways

- Visual Explain provides a rich set of features for deep analysis of SQL queries
- Tools such as Index Advisor and Show Indexes Considered help database engineers evaluate and refine indexing strategies
- Row count and runtime statistics provide insight into performance symptoms and where they occur within an access plan

What is a Materialized Query Table (MQT)?

A materialized query table is a table whose definition is based on the result of a query, and whose data is in the form of precomputed results that are taken from the table or tables on which the materialized query table definition is based.

What This Covers

This section provides an overview of MQT technology.

What It Is

An MQT differs from an SQL View in that the MQT (when refreshed) contains the data that is returned by the query over which it is defined. MQTs are user maintained, meaning that you need to decide when (and how frequently) to repopulate (refresh) the contents of the MQT. When an end consumer queries the MQT, they experience excellent performance because the data is already resident within the MQT.

However, keep in mind that the MQT data is not live data. The data in the MQT is captured when the MQT is repopulated using the REFRESH TABLE (SQL) statement.

The ideal use of an MQT is to find performance critical business queries where the query is reasonably or completely answered with data that is not live data. For example, some business intelligence (BI) queries are based upon the history of transactional data, and a nightly refresh of the MQT provides adequate currency for the data, and enables ideal performance for BI applications.

MQTs also have the possibility of being enabled for SQL Query Engine (SQE) optimization and consideration. If you want to explore opening up MQTs beyond direct consumption, you should read the deep details found at the following resources:

IBM Documentation

- Materialized query table optimization
- QAQQINI Options
- SYSMQTSTAT - Db2 for i Catalog

Why It Matters

MQTs provide improved performance for some SQL queries by expanding the data model. Since MQTs contain data, there is a storage impact when establishing one or more MQTs.

How It Works

A simple example of how to create, populate, and manage an MQT appears below. After the CREATE TABLE (SQL) statement, the mqt_monthly_revenue_summary table exists, but there are no rows in the table. When the REFRESH TABLE (SQL) statement is executed, the MQT query is extracted and executed, and the resulting data is inserted in the MQT.

A subsequent query to examine the data is a simple query, in part, because the MQT query already extracted, grouped, and transformed the data. Lastly, there's an example of how to schedule the daily refresh of the MQT.

```
--
-- Create the MQT
--
CREATE TABLE toystore3.mqt_monthly_revenue_summary for system name revbymon AS
      (SELECT year,
              month,
              quarter,
              SUM(quantity) AS total_quantity,
              SUM(revenue_w_tax) AS total_revenue_w_tax,
              SUM(profit_w_tax) AS total_profit_w_tax,
              COUNT(DISTINCT orderkey) AS order_count,
              CASE
                WHEN COUNT(DISTINCT orderkey) > 0
                THEN SUM(revenue_w_tax) / COUNT(DISTINCT orderkey)
                ELSE 0
              END AS avg_order_value_w_tax
          FROM toystore3.item_fact
          GROUP BY year,
                   month,
                   quarter)
      DATA INITIALLY immediate
      REFRESH DEFERRED
      MAINTAINED BY USER
      ENABLE QUERY OPTIMIZATION;

--
-- Execute the MQT's query and populate the data into the MQT
--
refresh table toystore3.mqt_monthly_revenue_summary;

--
-- Average Order value breakdown
--
SELECT *
  FROM toystore3.mqt_monthly_revenue_summary
  ORDER BY avg_order_value_w_tax DESC
  LIMIT 10;

--
-- Refresh the MQT nightly
--
cl:ADDJOBSCDE JOB(REFRESHIT) CMD(RUNSQL SQL('refresh table toystore3.mqt_monthly_revenue_summary') COMMIT(
```

Key Takeaways

- MQTs provide a nice performance advantage for some queries
- MQTs include a query optimization option
- MQTs represent data at the point in time that the MQT was refreshed

What's the Best tool for the task?

Questions about IBM i performance can be answered using a wide variety of tools and techniques. This topic exists to help you navigate the many approaches and settle on the best tool for your current performance task.

What This Covers

An overview of the most popular and powerful tools to use to understand performance-related topics on the IBM i.

IBM i on Power - Performance FAQ

The IBM i on Power - Performance FAQ is the best resource for understanding the various IBM i performance tools, what they provide, and how to find more detail. There are two resources found at this site, one that focuses on IBM i Performance FAQ and answers to those questions, and another that delves into techniques and ideas for optimizing performance on IBM i.

What is Collection Services?

Collection Services (CS) is an IBM i performance monitoring tool that automatically collects system performance data at regular intervals. It gathers detailed information about:

- System resources: CPU usage, memory, disk I/O, and communication line activity
- Job performance: Job execution statistics, wait times, and resource consumption
- Database activity: File operations, SQL queries, and database I/O
- Transaction data: Response times and throughput metrics

The data is collected automatically by the operating system on an interval basis, and can be analyzed using IBM i Performance Tools or queried directly via SQL. CS data can help you observe performance trends, anomalous behavior, and quickly isolate a very complex topic down to a specific job, workload, or query.

Use the following SQL statements to review the CS configuration on your IBM i.

```
--
-- Review the Collection Services (CS) configuration
--
select *
  from QSYS2.COLLECTION_SERVICES_INFO;

--
-- Shred the CS categories and interval settings
--
select a.*
  from QSYS2.COLLECTION_SERVICES_INFO, lateral (select * from JSON_TABLE(CATEGORY_LIST, 'lax $.category_li
  columns(cs_category clob(1k) ccsid 1208 path 'lax $."category"',
          cs_interval clob(1k) ccsid 1208 path 'lax $."interval"'))) a;
```

Performance Monitoring and Investigation

IBM iDoctor for IBM i

iDoctor is a licensed, Windows-based, suite of performance tools that can be used for analysis of Collection Services, Job Watcher, PEX, Disk Watcher, and SQL Plan Cache Snapshots.

IBM iDoctor for IBM i is used by the performance expert to collect, investigate, and analyze performance data on IBM i. The tools are used to monitor overall system health at a high overview level or to drill down to the performance details within job(s), disk unit(s) or programs over data collected during performance situations.

Performance Data Investigator

IBM i Performance Data Investigator (PDI), is a feature within IBM Navigator for i that provides a web-based GUI that allows interactive analysis of performance data through charts and tables. Using PDI, you can view and analyze data collected by Collection Services, IBM i Job Watcher, and IBM i Disk Watcher. Additionally, PDI offers chart options for Health Indicators, Database, System Monitors, and Graph History.

The PDI support within Navigator for i provides a feature-rich, graphical interface to explore performance data.

Db2 for IBM i Performance Tools

Db2 for i includes many facilities and resources for current and historical database performance topics. The following is a list of some of the most commonly used facilities:

- *SQL Plan Cache* - The SQL Plan Cache includes not only query plans from previously executed queries, but also deep performance details on the usage and execution of previous queries. As such, the SQL Plan Cache is an excellent resource to identify queries or workloads to focus on to achieve database performance improvements.
- *SQL Performance Monitor (aka Database Monitor)* - Gathers real-time information for database operation in one or more jobs. The collected information includes details about optimization and runtime behavior. The detail can be studied via the SQL Performance Center in IBM i Access Client Solutions (ACS) or directly queried using SQL.
- *Visual Explain* - Visual Explain (VE) is a diagnostic and analysis tool that graphically represents how the SQL Query Engine (SQE) chooses to execute a query. VE is found within ACS and Visual Studio Code (VS Code) for IBM i - Database Extension. Use VE to study a specific SQL query, and its related SQE implementation.
- *Query Supervisor* - The Db2 for i SQL Query Engine (SQE) provides a Query Supervisor which enables real-time monitoring of resource consumption by SQL and native queries. Query supervisor can be used to recognize and manage queries that are exceeding a real time consumption of a specific IBM i resource threshold.
- *Index Advisor* - The Index Advisor contains instances where the query optimizer wanted to cost a query implementation that included an index that does not exist. By evaluating the historical instances of index advice, it becomes possible to identify indexes that merit being created as permanent indexes.
- *Db2 for i SQL Services for Performance* - While ACS and VS Code for i are excellent tools to use to interact with Db2 for i performance tools, many Db2 for i SQL services exist that can be called or coded to directly.

IBM i Performance Resources:

- IBM Power performance papers and resources
- IBM iDoctor for IBM i
- QSYS2.COLLECTION_SERVICES_INFO view
- Performance Data Investigator in Navigator for i
- Query Supervisor
- Index Advisor
- Db2 for i SQL Services for Performance

Key Takeaways

- Be aware of the many tools that can be used to understand performance topics on the IBM i
- Many of the performance tools are available for use, without additional licensing requirements or investment
- Choose wisely and become conversant in many of these tools

What is the SQL Performance Center?

What It Is

IBM i Access Client Solutions (ACS) contains a full function dialog entitled the "SQL Performance Center". As shown in the following figure, the SQL Performance Center can be launched from the ACS main dialog.

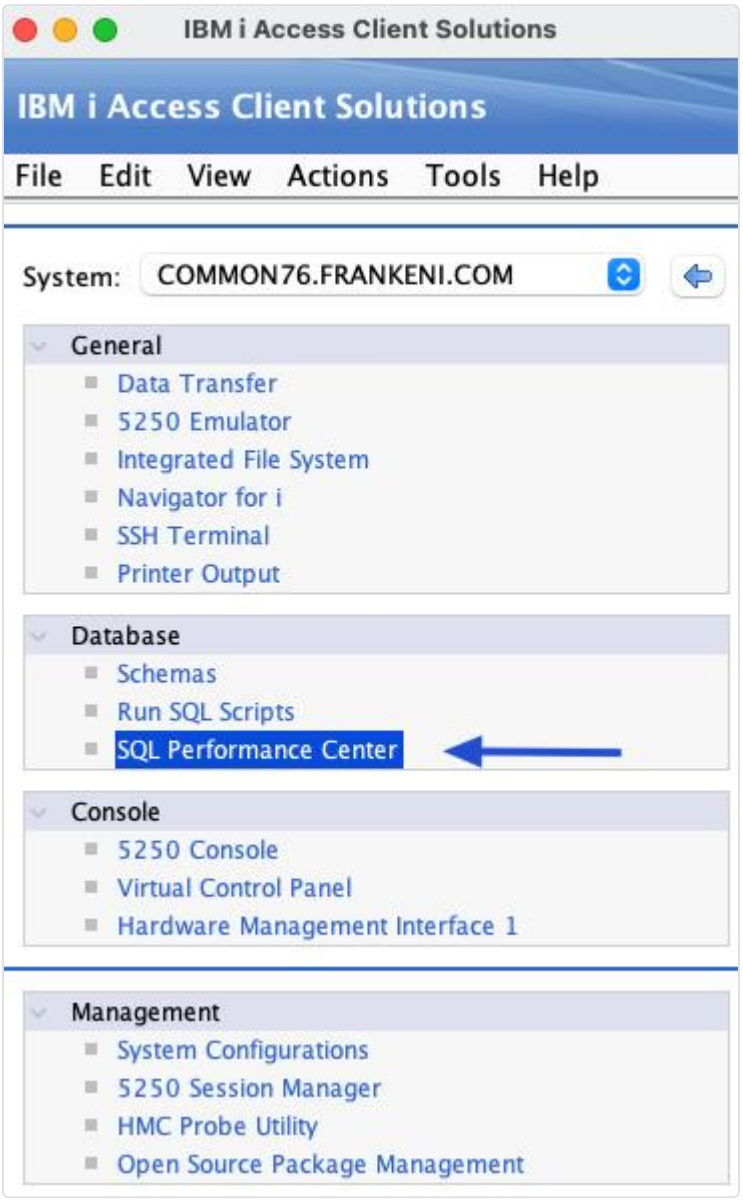


Figure. Launching the SQL Performance Center

The SQL Performance Center provides current and historical information pertaining to the execution of SQL statements on this IBM i, since the last IPL. The information can be used by Database Engineers (DBEs) and others to explore and analyze database performance characteristics. The following figure shows the SQL Performance Center primary dialog.

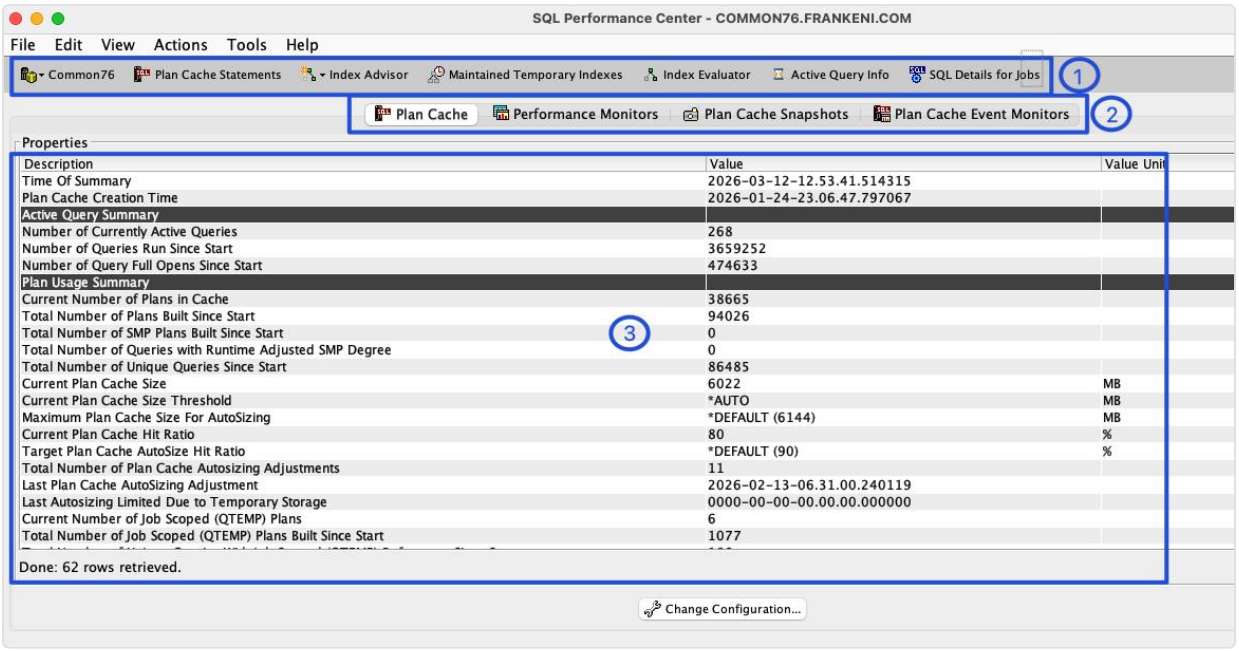


Figure. SQL Performance Center primary dialog

Three feature sets are highlighted:

1. Launch strip for database performance tools
2. Tabs to allow switching between the live SQL Query Engine (SQE) plan cache properties, and historical captures of SQL performance detail
3. The live SQL Query Engine (SQE) plan cache property details

Why should an application developer care?

The SQL Performance Center can help an application developer capture SQL Performance Monitors for any workload. The performance monitors help you identify the slowest performing queries in a workload. It also lets you see many other dimensions of insight, such as SQL statements that failed, SQL statements that used options different from the application strategy, and many other insights.

From any SQL statement within an SQL Performance Monitor, the user can easily extract the entire SQL statement text to Run SQL Scripts, for further analysis. Another popular option would be to launch into Visual Explain.

Other interesting features include the Index Advisor and Maintained Temporary Indexes (MTI) dialogs. By reviewing the different dimensions of index advice and MTI existence, the developer can make informed decisions on revisions to the index strategy.

How It Works

Most of the data seen within the SQL Performance Center can be accessed directly via Db2 for i Services. Are these services the tools of the DBE? Indeed they are! That being said, anyone responsible for application development can also gain valuable insight by using these tools!

Db2 for i Performance Services

These services include procedures that provide interfaces to work with indexes and a view to see information about SQL performance monitors. For more information, see Performance Services.

SQE Plan Cache Services

These procedures allow for programmatic access to the SQL plan cache and can be used for tasks such as scheduling plan cache captures or pre-starting an event monitor. For more information, see Plan Cache Services.

Key Takeaways

- ACS includes all the tools for the DBE, but the APPDEV team can use them too!
- While ACS is great, most of the data can be accessed directly via Database (SQL) Services

PDI Performance Data Investigator

What This Covers

There may come a time when you need to understand how your IBM i environment is performing; this information is crucial when you are optimizing your code or modelling the impact of scaling your workload.

The chapter describes the native IBM i Performance Data Investigation (PDI) tool that can assist in this process, covering what it is, how it works and how to access it.

At the end of this chapter, there are some useful links to IBM should you want to know more about PDI.

What It Is

The Performance Data Investigator (PDI) is a web-based, graphical tool within the IBM Navigator for i that analyzes IBM i performance data. PDI enables interactive analysis of Collection Services, Job Watcher, and Disk Watcher data through customizable charts and tables. It provides in-depth system performance, workload, and resource usage insights to aid in troubleshooting, often used for identifying bottlenecks, monitoring, and trend analysis.

Key Aspects of PDI: Analytical Capabilities: Allows deep-dive investigation into performance data, including database SQL plan cache, performance explorer, and historical data. Features include: Customization: Users can edit perspectives to tailor charts and tables. Drill-down: Enables zooming into specific metrics from higher-level views. Context Management: Allows selecting specific collections, libraries, and time intervals. Data Sources: Integrates with Collection Services, Job Watcher, Disk Watcher, and Performance Explorer (PEX).

Why It Matters

PDI is essential for monitoring system resources, such as memory and CPU usage, and helps analyze how workload changes impact overall performance. It allows you to view the system work load as a whole or at an individual job, in both cases you can see in details how the job is performing and what if anything the job is waiting on (Processor, storage, journalling, etc) this makes it much easier to understand why your job is performing the way it is and where to focus to improve it.

How It Works

By default, your IBM i system will collect performance data automatically and will retain 30 days of data, captured every 15 minutes. PDI can be used to query this performance data and visualise the system performance. There are hundreds of views in PDI that have been created to help you understand your system performance. PDI can be accessed via the "Investigate Data" task in the IBM Navigator for i graphical interface. The Navigator for i interface itself can be access either from the built-in link to your system in ACS (Access Client Solutions) client or direct from a web browser using `https://your_system:2003`.

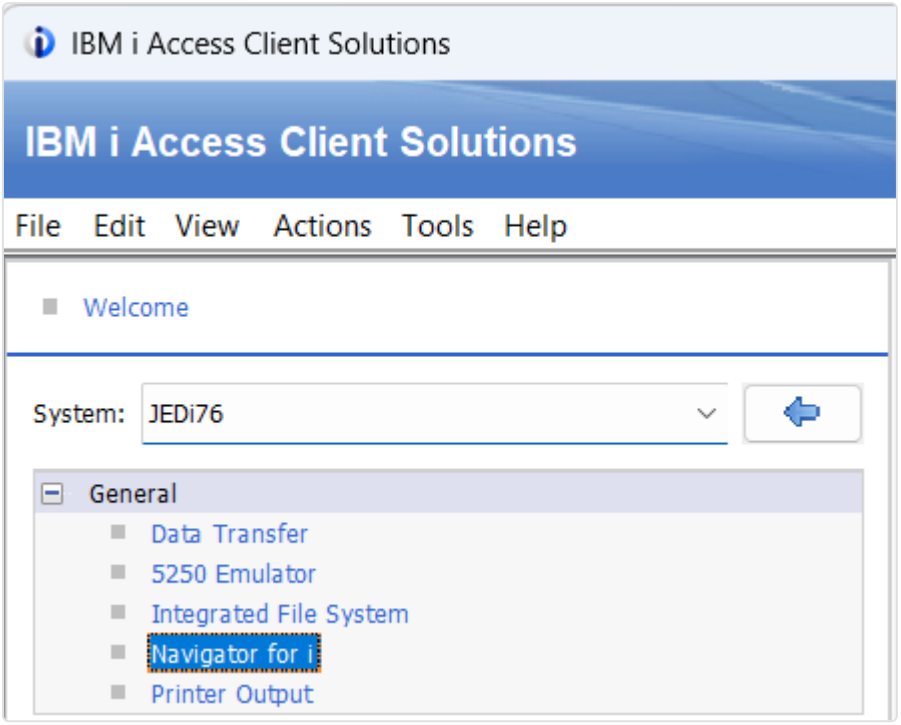


Figure. Accessing Navigator via ACS

You will need to authenticate with IBM i to use the Navigator interface using an IBM i user profile that has necessary authority.

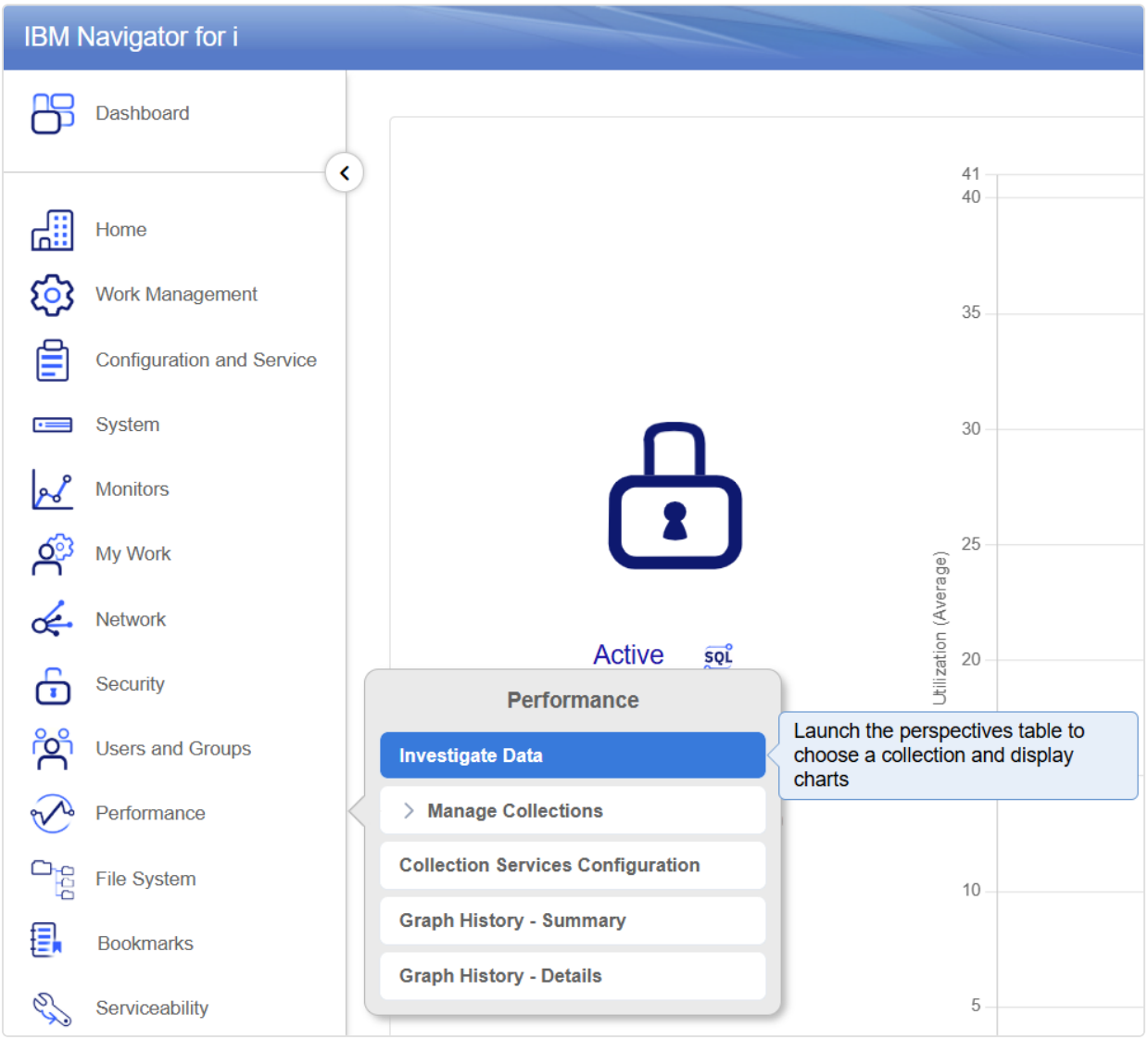


Figure. Accessing PDI via Navigator

From the Investigate Data page, you can select the performance data you wish to investigate.

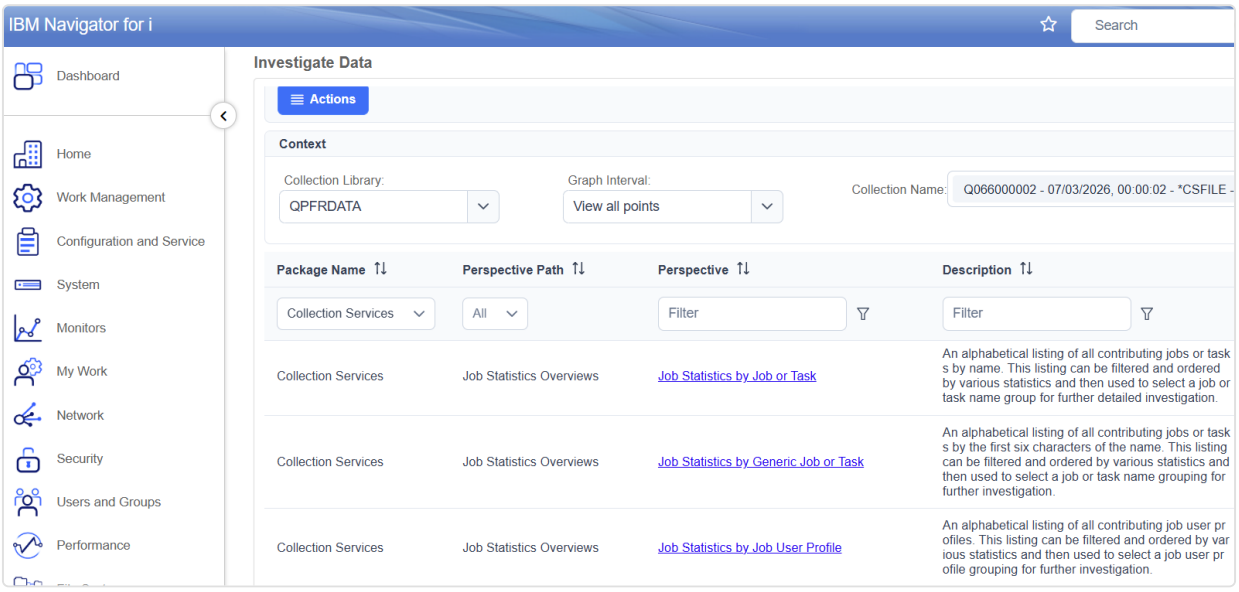


Figure. Selecting Performance Data

From here you can start to drill down to find out how a specific task is performing and wait if anything is causing the delay in performance. This data can be exported in graph or table form, and should you find the data presented useful, you can bookmark the view or export the SQL used to select this data for later use.

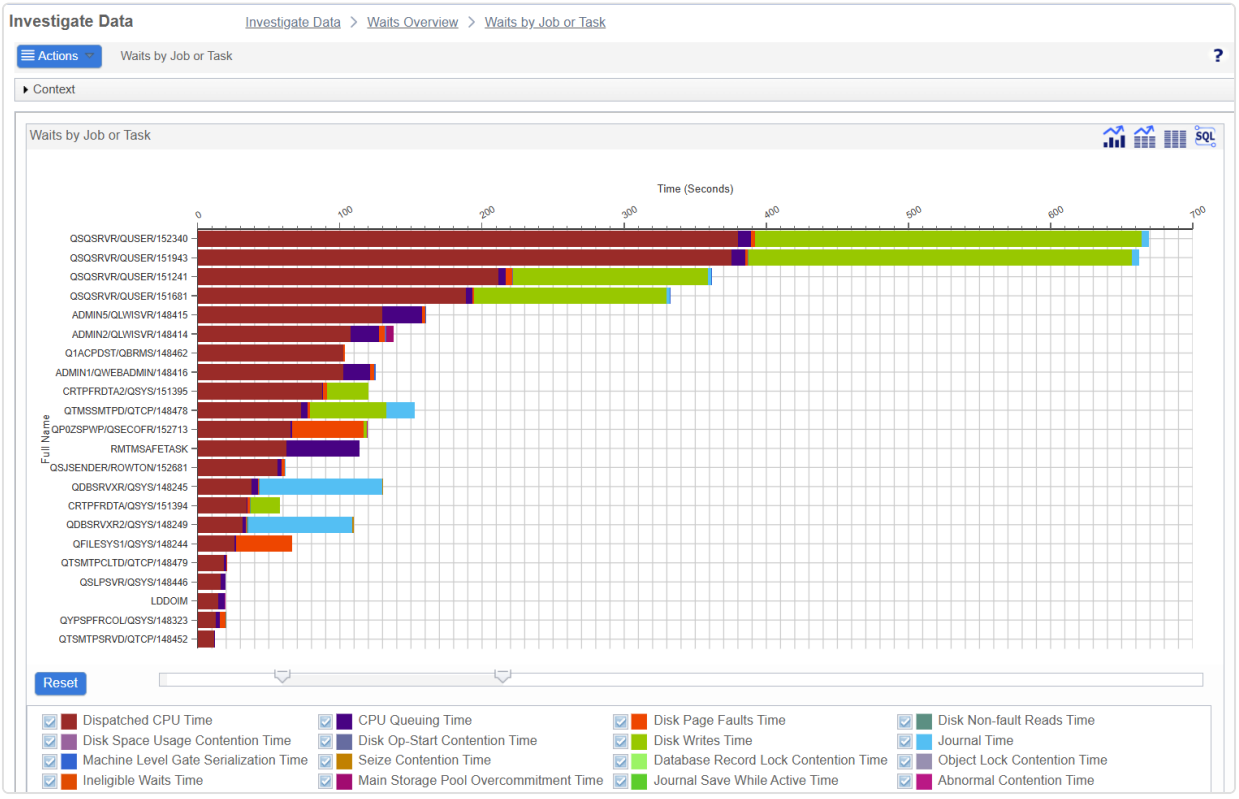


Figure. Waits by Job or Task

ProTip: By default, PDI will keep 30 days of performance data, captured every 15 minutes, the setting can be tailored to suit your needs, you may wish to reduce this interval from 15 to 5 minutes to get a more granular view of a specific job or jobs you wish to analyze. It is also worth noting that it is possible to save performance data from one system to analyze on another.

Key Takeaways

PDI is the first place to look if you need to understand how a particular system is performing as a whole or for individual jobs on their own. The information is collected by default and there is no additional cost for the PDI tool.

If the data displayed in PDI is useful to you it can be exported in graphical or CSV form and you can export the SQL used to return the data from any PDI screen (via the SQL button) so that you can access the PDI data directly outside of the PDI tool.

You can save your performance data from one IBM i system and analyse it on another.

If you would like to know more about PDI, the following link should prove useful:

- Performance Data Investigator
- IBM i Performance Data Investigator
- IBM Support: Performance Data Investigator in Navigator for i