Patterns: Self-Service Application Solutions Using WebSphere V5.0

Integrate Web applications with the enterprise tier
Explore Web services, J2EE Connectors, and JMS solutions
Learn by example with sample scenarios

Mark Endrei
Min Luo
Margo Pulles

ibm.com/redbooks
Patterns: Self-Service Application Solutions Using WebSphere V5.0

May 2003
First Edition (May 2003)

This edition applies to IBM WebSphere Application Server V5.0, IBM WebSphere Studio Application Developer V5.0, and IBM WebSphere MQ V5.3, for use with IBM AIX 5.1, Red Hat Linux Advanced Server V2.1, and Microsoft Windows 2000.

© Copyright International Business Machines Corporation 2003. All rights reserved.
Note to U.S. Government Users Restricted Rights -- Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

**Note:** Before using this information and the product it supports, read the information in “Notices” on page xi.
Contents

Notices ................................................................. xi
Trademark .......................................................... xii

Preface ............................................................... xiii
The team that wrote this redbook ................................ xiii
Become a published author ........................................ xv
Comments welcome ................................................ xv

Part 1. Self-Service patterns ........................................ 1

Chapter 1. Patterns for e-business ................................. 3
1.1 The Patterns for e-business layered asset model .............. 5
1.2 How to use the Patterns for e-business ......................... 6
  1.2.1 Business, Integration, or Composite pattern, or a custom design ... 7
  1.2.2 Selecting Application patterns ................................ 12
  1.2.3 Review Runtime patterns ..................................... 13
  1.2.4 Review Product mappings ..................................... 16
  1.2.5 Review guidelines and related links ......................... 16
1.3 Summary .......................................................... 17

Chapter 2. The Self-Service business pattern ..................... 19
2.1 Self-Service applications ......................................... 20
2.2 Self-Service application patterns ................................ 20
2.3 Application patterns used in this book ......................... 23

Chapter 3. Runtime pattern ........................................ 25
3.1 An introduction to the node types .............................. 26
  3.1.1 Web application server node ............................... 26
  3.1.2 User node .................................................... 26
  3.1.3 Public Key Infrastructure (PKI) ........................... 27
  3.1.4 Domain Name System (DNS) node ......................... 27
  3.1.5 Protocol firewall node ..................................... 27
  3.1.6 Domain firewall node ....................................... 28
  3.1.7 Directory and security services node ....................... 28
  3.1.8 Database server node ....................................... 28
  3.1.9 Existing applications and data node ......................... 28
  3.1.10 Web server redirector node ................................ 29
  3.1.11 Application server node .................................... 29
3.2 Runtime patterns for Stand-Alone Single Channel ............ 29
10.2.1 Component model .......................... 284
10.2.2 Object model ............................ 285
10.3 Enterprise application design .................. 288
10.4 XML messaging design and development .... 289
  10.4.1 XML message structure .................... 289
  10.4.2 XML DTD and XML Schema ................. 291
  10.4.3 XML and JavaBean .......................... 293
10.5 Application development using Web services .......... 294
  10.5.1 Preparing for development ................ 294
  10.5.2 Approach to developing the sample Web services components 295
  10.5.3 Creating the enterprise-tier application .... 297
  10.5.4 Creating the Web service .................. 306
  10.5.5 Using the Web Service wizard .......... 307
  10.5.6 Examining the generated files ................ 315
  10.5.7 SOAP router servlets ................. 316
  10.5.8 Viewing the deployed Web services .......... 317
  10.5.9 Web service client proxy ............. 320
  10.5.10 Web service test client ............. 322
  10.5.11 Creating the Web service requester .......... 323

Chapter 11. J2EE Connector Architecture scenario .......... 333
  11.1 Architectural overview model .................. 334
  11.2 System design overview ..................... 334
    11.2.1 Component model .......................... 335
    11.2.2 Object model ............................ 336
  11.3 Low-level design considerations ................ 339
    11.3.1 Connection management ...................... 340
    11.3.2 Creating the input and output record ........ 342
    11.3.3 Data conversion ............................ 344
    11.3.4 Executing the enterprise application ........ 344
    11.3.5 Transaction management ..................... 345
    11.3.6 Security ................................. 346
  11.4 Application development using J2EE Connectors .......... 346
    11.4.1 Installing the J2EE Connector resource adapter .......... 346
    11.4.2 Create a session EJB ....................... 347
    11.4.3 Testing the session EJB ................... 353
  11.5 Runtime configuration .......................... 355
    11.5.1 Runtime configuration design .............. 356
    11.5.2 WebSphere Application Server configuration .... 358
    11.5.3 CICS Transaction Server configuration .......... 360

Chapter 12. Java Message Service scenario .................. 365
  12.1 Architectural overview model .................. 366
12.2 System design overview ........................................... 366
  12.2.1 Component model ........................................ 367
  12.2.2 Object model ........................................... 367
12.3 Low-level design considerations ........................................ 370
  12.3.1 Point-to-point messaging model ..................... 371
  12.3.2 Message structure ....................................... 371
  12.3.3 JMS resource lookups using JNDI .................... 371
  12.3.4 Managing JMS resources ............................... 372
  12.3.5 Synchronous vs. asynchronous .......................... 373
  12.3.6 Message selectors ....................................... 374
  12.3.7 Message time to live .................................. 375
  12.3.8 Persistent vs. non-persistent messages ............ 375
  12.3.9 Enterprise application design ....................... 376
  12.3.10 Alternate publish/subscribe design ................. 376
12.4 Development environment configuration .......................... 377
  12.4.1 Adding JMS resources to the server configuration ... 378
  12.4.2 Adding a Listener Port to the server configuration ... 380
  12.4.3 Co-located WebSphere Application Server ........... 381
12.5 Runtime configuration ........................................ 381
  12.5.1 JNDI name spaces ....................................... 382
  12.5.2 IBM WebSphere MQ logical configuration ............ 384
  12.5.3 IBM WebSphere MQ configuration for Windows 2000 .... 385
  12.5.4 IBM WebSphere MQ configuration for AIX and Linux ... 394
  12.5.5 IBM WebSphere Application Server V5.0 JMS configuration .... 401
  12.5.6 Enterprise simulation application setup ............. 409

Part 4. Appendixes ................................................... 411

Appendix A. Lab environment scenarios .................................. 413
  Windows lab environment ...................................... 414
  IBM AIX lab environment ..................................... 415
  Linux lab environment ........................................ 417

Appendix B. PDK sample setup ......................................... 421
  Patterns for e-business downloads ............................ 422
    Patterns Development Kit Lite (PDK Lite) ................. 422
    How to download and install the PDK Lite ................... 422
  IBM WebSphere Application Server base V5.0 ................... 423
    Before you start ........................................... 423
    Installing the PDK sample .................................. 425
    Running the PDK sample .................................... 426
  IBM WebSphere Studio Application Developer .................. 427
    Changing the default settings ................................ 431
    Selecting different topologies ............................... 432
Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area.
Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:
IBM Director of Licensing, IBM Corporation, North Castle Drive Armonk, NY 10504-1785 U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:
This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.
Trademarks

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

- AIX®
- Crossworlds®
- CICS®
- Domino™
- DB2 Universal Database™
- DB2®
- developerWorks®
- e-business on demand™
- IBM®
- ibm.com®

- IMS™
- Language Environment®
- Lotus®
- MQSeries®
- MVS™
- OS/2®
- OS/390
- OS/400
- pSeries™
- RACF®
- Redbooks™

- Redbooks (logo)™
- RS/6000®
- S/390®
- SecureWay®
- VSE/ESA™
- VisualAge®
- WebSphere®
- z/OS™
- zSeries®

The following terms are trademarks of International Business Machines Corporation and Lotus Development Corporation in the United States, other countries, or both:

- ClearCase®
- Rational Software Corporation®
- Rational Unified Process®
- Rational®
- XDE®

The following terms are trademarks of other companies:

ActionMedia, LANDesk, MMX, Pentium and ProShare are trademarks of Intel Corporation in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

C-bus is a trademark of Corollary, Inc. in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

SET, SET Secure Electronic Transaction, and the SET Logo are trademarks owned by SET Secure Electronic Transaction LLC.

Other company, product, and service names may be trademarks or service marks of others.
Preface

The Patterns for e-business are a group of proven, reusable assets that can be used to increase the speed of developing and deploying Web applications. This IBM® Redbook focuses on the Self-Service::Stand-Alone Single Channel application pattern for facilitating user access to business sites, and the Self-Service::Directly Integrated Single Channel application pattern for including one or more point-to-point connections with back-end applications.

Part 1 of this publication guides you through the process of selecting an Application and Runtime pattern. Next, the platform-specific Product mappings are identified based upon the selected Runtime pattern.

Part 2 of this publication provides a set of guidelines for building your Web application and for integrating it with the enterprise tier using Web services, J2EE Connectors, and JMS. These guidelines include technology options, application design, application development, and systems management.

Part 3 of this publication teaches you by example how to design and build sample solutions using IBM WebSphere® Application Server V5.0 with Web services, J2EE Connectors, and IBM CICS®, and JMS and IBM WebSphere MQ.

The team that wrote this redbook

This redbook was produced by a team of specialists from around the world working at the International Technical Support Organization, Raleigh Center.
Mark Endrei is an IT Architect at the International Technical Support Organization, Raleigh Center. He writes about all areas of WebSphere. Before joining the ITSO early in 2001, Mark worked in IBM Global Services Australia as an IT Architect. He holds a Bachelor’s degree in Computer Systems Engineering from the Royal Melbourne Institute of Technology, and an MBA in Technology Management from Deakin University/APESMA.

Dr. Min Luo is a Senior Certified IT Architect in IBM Global Services’ National Architecture and Technology Center of Excellence. He has over 15 years of IT industry experience, and has taught undergraduate and graduate Computer Science courses for over seven years. He is proficient in all major aspects of information technology, especially application architecture, design, integration, planning, and management. He has successfully designed and implemented solutions for transportation, financial, and manufacturing industries, and large-scale government social services projects. He has led and managed whole life cycle projects across major platforms, operating systems, database management systems, and development environments. Dr. Luo received a PhD in Electrical Engineering from the Georgia Institute of Technology in 1992, specializing in Network Simulation and Optimization. He also holds an MS in Computer Science (1987) and BS in Computer Information Systems (1981).

Margo Pulles is a Technical Designer in the Tax Office Information and Communications Technology Centre (B/CICT) in the Netherlands. She has five years of experience with application development and infrastructure design. She
has worked on numerous projects dealing with these technologies, ranging in scope from ICT-architectural design to implementation, as well as proofs of concept. Her areas of expertise include database technologies, J2EE design, application server, and e-business infrastructure.

Thanks to the following people for their contributions to this project:

**Jonathan Adams**  
IBM UK

**Michele Galic, Peter Kovari, Margaret Ticknor**  
IBM ITSO Raleigh

**Phil Wakelin**  
IBM UK

**Julie Czubik**  
IBM ITSO Poughkeepsie

**Arun Thakore**  
IBM Austin

**Become a published author**

Join us for a two- to six-week residency program! Help write an IBM Redbook dealing with specific products or solutions, while getting hands-on experience with leading-edge technologies. You'll team with IBM technical professionals, Business Partners and/or customers.

Your efforts will help increase product acceptance and customer satisfaction. As a bonus, you'll develop a network of contacts in IBM development labs, and increase your productivity and marketability.

Find out more about the residency program, browse the residency index, and apply online at:  
[ibm.com/redbooks/residencies.html](http://ibm.com/redbooks/residencies.html)

**Comments welcome**

Your comments are important to us!

We want our Redbooks™ to be as helpful as possible. Send us your comments about this or other Redbooks in one of the following ways:
▶ Use the online **Contact us** review redbook form found at:
   ibm.com/redbooks

▶ Send your comments in an Internet note to:
   redbook@us.ibm.com

▶ Mail your comments to:
   IBM Corporation, International Technical Support Organization
   Dept. HZ8  Building 662
   P.O. Box 12195
   Research Triangle Park, NC 27709-2195
Part 1 guides you through the process of selecting Application and Runtime patterns. The platform-specific Product mappings are identified based upon the selected Runtime pattern.

Included in Part 1 are the following chapters:

- Chapter 1, "Patterns for e-business" on page 3
- Chapter 2, "The Self-Service business pattern" on page 19
- Chapter 3, "Runtime pattern" on page 25
- Chapter 4, "Product mappings" on page 37
Chapter 1. Patterns for e-business

This publication is part of the Patterns for e-business series. In this introductory chapter we provide an overview of how IT architects can work effectively with the Patterns for e-business.

The role of the IT architect is to evaluate business problems and to build solutions to solve them. To do this, the architect begins by gathering input on the problem, an outline of the desired solution, and any special considerations or requirements that need to be factored into that solution. The architect then takes this input and designs the solution. This solution can include one or more computer applications that address the business problems by supplying the necessary business functions.

To enable the architect to do this better each time, we need to capture and reuse the experience of these IT architects in such a way that future engagements can be made simpler and faster. We do this by taking these experiences and using them to build a repository of assets that provides a source from which architects can reuse this experience to build future solutions, using proven assets. This reuse saves time, money, and effort, and in the process helps ensure delivery of a solid, properly architected solution.

The IBM Patterns for e-business helps facilitate this reuse of assets. Their purpose is to capture and publish e-business artifacts that have been used, tested, and proven. The information captured by them is assumed to fit the majority, or 80/20, situation.
The IBM Patterns for e-business are further augmented with guidelines and related links for their better use.

The layers of patterns plus their associated links and guidelines allow the architect to start with a problem and a vision for the solution, and then find a pattern that fits that vision. Then by drilling down using the pattern’s process, the architect can further define the additional functional pieces that the application will need to succeed. Finally he can build the application using the coding techniques outlined in the associated guidelines.
1.1 The Patterns for e-business layered asset model

The Patterns for e-business approach enables architects to implement successful e-business solutions through the reuse of components and solution elements from proven successful experiences. The Patterns approach is based on a set of layered assets that can be exploited by any existing development methodology. These layered assets are structured in a way such that each level of detail builds on the last. These assets include:

▶ Business patterns that identify the interaction between users, businesses, and data.
▶ Integration patterns that tie multiple Business patterns together when a solution cannot be provided based on a single Business pattern.
▶ Composite patterns that represent commonly occurring combinations of Business patterns and Integration patterns.
▶ Application patterns that provide a conceptual layout describing how the application components and data within a Business pattern or Integration pattern interact.
▶ Runtime patterns that define the logical middleware structure supporting an Application pattern. Runtime patterns depict the major middleware nodes, their roles, and the interfaces between these nodes.
▶ Product mappings that identify proven and tested software implementations for each Runtime pattern.
▶ Best-practice guidelines for design, development, deployment, and management of e-business applications.

These assets and their relation to each other are shown in Figure 1-1 on page 6.
Patterns for e-business Web site

The Patterns Web site provides an easy way of navigating top down through the layered Patterns’ assets in order to determine the preferred reusable assets for an engagement.

For easy reference to Patterns for e-business refer to the Patterns for e-business Web site at:


1.2 How to use the Patterns for e-business

As described in the last section, the Patterns for e-business is a layered structure where each layer builds detail on the last. At the highest layer is Business patterns. These describe the entities involved in the e-business solution.
Composite patterns appear in the hierarchy shown in Figure 1-1 on page 6 above the Business patterns. However, Composite patterns are made up of a number of individual Business patterns, and at least one Integration pattern. In this section, we discuss how to use the layered structure of Patterns for e-business assets.

1.2.1 Business, Integration, or Composite pattern, or a custom design

When faced with the challenge of designing a solution for a business problem, the first step is to take a high-level view of the go, and each element should be matched to an appropriate IBM Pattern for e-business. You may find, for example, that the total solution requires multiple Business and Integration patterns, or that it fits into a Composite pattern or custom design. For example, suppose an insurance company wants to reduce the amount of time and money spent on call centers that handle customer inquiries. By allowing customers to view their policy information and to request changes online, they will be able to cut back significantly on the resources spent handling this by phone. The objective is to allow policyholders to view their policy information stored in legacy databases.

The Self-Service business pattern fits this scenario perfectly. It is meant to be used in situations where users need direct access to business applications and data. Let's take a look at the available Business patterns.

Business patterns

A Business pattern describes the relationship between the users, the business organizations or applications, and the data to be accessed.
There are four primary Business patterns, as explained in Figure 1-2.

<table>
<thead>
<tr>
<th>Business Patterns</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Self-Service (User-to-Business)</td>
<td>Applications where users interact with a business via the Internet or intranet</td>
<td>Simple Web site applications</td>
</tr>
<tr>
<td>Information Aggregation (User-to-Data)</td>
<td>Applications where users can extract useful information from large volumes of data, text, images, etc.</td>
<td>Business intelligence, knowledge management, Web crawlers</td>
</tr>
<tr>
<td>Collaboration (User-to-User)</td>
<td>Applications where the Internet supports collaborative work between users</td>
<td>E-mail, community, chat, video conferencing, etc.</td>
</tr>
<tr>
<td>Extended Enterprise (Business-to-Business)</td>
<td>Applications that link two or more business processes across separate enterprises</td>
<td>EDI, supply chain management, etc.</td>
</tr>
</tbody>
</table>

*Figure 1-2  The four primary Business patterns*

It would be very convenient if all problems fit nicely into these four slots, but reality says that things will often be more complicated. The patterns assume that most problems, when broken down into their most basic components, will fit more than one of these patterns. When a problem requires multiple Business patterns, the Patterns for e-business provide additional patterns in the form of Integration patterns.

**Integration patterns**

Integration patterns allow us to tie together multiple Business patterns to solve a business problem. The Integration patterns are outlined in Figure 1-3 on page 9.
These Business and Integration patterns can be combined to implement installation-specific business solutions. We call this a custom design.

**Custom design**
We can represent the use of a custom design to address a business problem through the iconic representation shown in Figure 1-4.

If any of the Business or Integration patterns are not used in a custom design, we show that in the figures by making a block(s) lighter than another. For example, Figure 1-5 shows a custom design that does not have a Collaboration business pattern or an Extended Enterprise business pattern for a business problem.
A custom design may also be a Composite pattern if it recurs many times across domains with similar business problems. For example, the iconic view of a custom design in Figure 1-5 can also describe a Sell-Side Hub composite pattern.

**Composite patterns**

Several common uses of Business and Integration patterns have been identified and formalized into Composite patterns. The identified Composite patterns are shown in Figure 1-6 on page 11.
<table>
<thead>
<tr>
<th>Composite patterns</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Electronic Commerce</strong></td>
<td>User-to-online-buying.</td>
<td>• <a href="http://www.macys.com">www.macys.com</a></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <a href="http://www.amazon.com">www.amazon.com</a></td>
</tr>
<tr>
<td><strong>Portal</strong></td>
<td>Typically designed to aggregate multiple information sources and applications to provide uniform, seamless, and personalized access for its users.</td>
<td>• Enterprise intranet portal providing self-service functions such as payroll, benefits, and travel expenses</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Collaboration providers who provide services such as e-mail or instant messaging</td>
</tr>
<tr>
<td><strong>Account Access</strong></td>
<td>Provides customers with around-the-clock account access to their account information.</td>
<td>• Online brokerage trading apps.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Telephone company account manager functions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Bank, credit card and insurance company online apps</td>
</tr>
<tr>
<td><strong>Trading Exchange</strong></td>
<td>Allows buyers and sellers to trade goods and services on a public site.</td>
<td>• Buyer's side: Interaction between buyer's procurement system and commerce functions of e-Marketplace</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Seller's side: Interaction between the procurement functions of the e-Marketplace and its suppliers</td>
</tr>
<tr>
<td><strong>Sell-Side Hub</strong></td>
<td>The seller owns the e-Marketplace and uses it as a vehicle to sell goods and services on the Web.</td>
<td><a href="http://www.carmax.com">www.carmax.com</a> (car purchase)</td>
</tr>
<tr>
<td>(Supplier)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Buy-Side Hub</strong></td>
<td>The buyer of the goods owns the e-Marketplace and uses it as a vehicle to leverage the buying or procurement budget in soliciting the best deals for goods and services from prospective sellers across the Web.</td>
<td><a href="http://www.wre.org">www.wre.org</a> (WorldWide Retail Exchange)</td>
</tr>
<tr>
<td>(Purchaser)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 1-6  Composite patterns**

The makeup of these patterns is variable in that there will be basic patterns present for each type, but the Composite can easily be extended to meet additional criteria. For more information on Composite patterns, refer to *Patterns for e-business: A Strategy for Reuse* by Jonathan Adams, Srinivas Koushik, Guru Vasudeva, and George Galambos.
1.2.2 Selecting Application patterns

Once the Business pattern is identified, the next step is to define the high-level logical components that make up the solution and how these components interact. This is known as the Application pattern. A Business pattern will usually have multiple possible Application patterns. An Application pattern may have logical components that describe a presentation tier for interacting with users, an application tier, and a back-end application tier.

Application patterns break the application down into the most basic conceptual components, identifying the goal of the application. In our example, the application falls into the Self-Service business pattern, and the goal is to build a simple application that allows users to access back-end information. The Self-Service::Directly Integrated Single Channel application pattern shown in Figure 1-7 fulfills this requirement.

![Figure 1-7 Self-Service::Directly Integrated Single Channel](image)

The Application pattern shown in Figure 1-7 consists of a presentation tier that handles the request/response to the user. The application tier represents the component that handles access to the back-end applications and data. The multiple application boxes on the right represent the back-end applications that contain the business data. The type of communication is specified as synchronous (one request/one response, then next request/response) or asynchronous (multiple requests and responses intermixed).
Suppose that the situation is a little more complicated than that. Let us say that the automobile policies and the homeowner policies are kept in two separate and dissimilar databases. The user request would actually need data from multiple, disparate back-end systems. In this case there is a need to break the request down into multiple requests (decompose the request) to be sent to the two different back-end databases, then to gather the information sent back from the requests, and then put this information into the form of a response (recompose). In this case the Self-Service::Decomposition application pattern shown in Figure 1-8 would be more appropriate.

![Figure 1-8  Self-Service::Decomposition](image)

This Application pattern extends the idea of the application tier that accesses the back-end data by adding decomposition and recomposition capabilities.

### 1.2.3 Review Runtime patterns

The Application pattern can be further refined with more explicit functions to be performed. Each function is associated with a runtime node. In reality these functions, or nodes, can exist on separate physical machines or may coexist on the same machine. In the Runtime pattern this is not relevant. The focus is on the logical nodes required and their placement in the overall network structure. As an example, let us assume that our customer has determined that his solution fits into the Self-Service business pattern and that the Directly Integrated Single Channel pattern is the most descriptive of the situation. The next step is to determine which Runtime pattern is the most appropriate for his situation.
He knows that he will have users on the Internet accessing his business data, and he will therefore require a measure of security. Security can be implemented at various layers of the application, but the first line of defense is almost always one or more firewalls that define who and what can cross the physical network boundaries into his company network.

He also needs to determine the functional nodes required to implement the application and security measures. The Runtime pattern shown in Figure 1-9 is one of his options.

By overlaying the Application pattern on the Runtime pattern, you can see the roles that each functional node will fulfill in the application. The presentation and application tiers will be implemented with a Web application server, which combines the functions of an HTTP server and an application server. It handles both static and dynamic Web pages.

Application security is handled by the Web application server through the use of a common central directory and security services node.
A characteristic that makes this Runtime pattern different from others is the placement of the Web application server between the two firewalls. The Runtime pattern shown in Figure 1-10 is a variation of this. It splits the Web application server into two functional nodes by separating the HTTP server function from the application server. The HTTP server (Web server redirector) will serve static Web pages and redirect other requests to the application server. It moves the application server function behind the second firewall, adding further security.

These are just two examples of the possible Runtime patterns available. Each Application pattern will have one or more Runtime patterns defined. These can be modified to suit the customer’s needs. For example, she may want to add a load-balancing function and multiple application servers.
1.2.4 Review Product mappings

The last step in defining the network structure for the application is to correlate real products with one or more runtime nodes. The Patterns Web site shows each Runtime pattern with products that have been tested in that capacity. The Product mappings are oriented toward a particular platform, though more likely the customer will have a variety of platforms involved in the network; in that case, it is simply a matter of mix and match. For example, the runtime variation in Figure 1-10 on page 15 could be implemented using the product set depicted in Figure 1-11.

![Diagram of Directly Integrated Single Channel application pattern: Windows 2000 Product mapping](image)

1.2.5 Review guidelines and related links

The Application patterns, Runtime patterns, and Product mappings are intended to guide you in defining the application requirements and the network layout. The actual application development has not been addressed yet. The Patterns Web site provides guidelines for each Application pattern, including techniques for developing, implementing, and managing the application based on the following:

- Design guidelines give you tips and techniques for designing the applications.
Development guidelines take you through the process of building the application, from the requirements phase all the way through the testing and rollout phases.

System management guidelines address the day-to-day operational concerns, including security, backup and recovery, application management, etc.

Performance guidelines give information on how to improve the application and system performance.

1.3 Summary

The IBM Patterns for e-business are a collective set of proven architectures. This repository of assets can be used by companies to facilitate the development of Web-based applications. They help an organization understand and analyze complex business problems and break them down into smaller, more manageable functions that can then be implemented.
The Self-Service business pattern

Businesses have traditionally invested a lot of resources into making information available to customers, vendors, and employees. These resources took the form of call centers, mailings, etc. They have also maintained information about their customers in the form of customer profiles. Updates to these profiles were handled over the phone or by mail.

The concept of self-service puts this information at the fingertips of the customers through a user interface, whether that interface is a Web site, a personal digital assistant (PDA), or some other client interface. An e-business application makes the information accessible to the right audience in an easy-to-access manner, thus reducing the need for human interaction and increasing user satisfaction.
2.1 Self-Service applications

Key elements of an application that provides self-service for a customer would include clear navigational directions, extended search capabilities, and useful links. A popular aspect is a direct link to the online representatives who can answer questions and offer a human interface if needed.

The following are examples of self-service applications:

- An insurance company makes policy information available to users and allows them to apply for a policy online.
- A mortgage company publishes information about its loan policies and load rates online. Customers can view their current mortgage information, change their payment options, or apply for a mortgage online.
- A scientific organization makes research papers available to interested users by putting the papers online.
- A bank allows customers to access their accounts and pay bills online.
- A well-known and respected group of technical writers makes its work available online. The group recruits technical participants for its projects by listing the upcoming projects online and allowing possible participants to apply online.
- A company allows its employees to view current human resource policies online. Employees can change their medical plan, tax withholding information, stock purchase plan, etc., online without having to call the Human Resources office.

2.2 Self-Service application patterns

As you can see in Figure 2-1 on page 21, the Self-Service business pattern covers a wide range of uses. Applications of this pattern can range from the very simple function of allowing users to view data built explicitly for one purpose, to taking requests from users, decomposing them into multiple requests to be sent to multiple, disparate data sources, personalizing the information, and recomposing it into a response for the user. For this reason, there are currently seven defined Application patterns that fit this range of functions. We summarize these for you here. More detailed information can be found in *Patterns for e-business: A Strategy for Reuse*, by Jonathan Adams, Srinivas Koushik, Guru Vasudeva, and George Galambos.
Below we go through the steps shown in Figure 2-1.

1. Stand-alone Single Channel application pattern

Provides for stand-alone applications that have no need for integration with existing applications or data. It assumes one delivery channel, most likely a Web client, although it could be something else. It consists of a presentation tier that handles all aspects of the user interface, and an application tier that contains the business logic to access data from a local database. The communication between the two tiers is synchronous. The presentation tier passes a request from the user to the business logic in the Web application tier. The request is handled and a response is sent back to the presentation tier for delivery to the user.
2. Directly Integrated Single Channel application pattern

Provides point-to-point connectivity between the user and the existing back-end applications. As with the Stand-alone Single Channel application pattern, it assumes one delivery channel, and the user interface is handled by the presentation tier. The business logic can reside in the Web application tier and in the back-end application. The Web application tier has access to local data that exists primarily as a result of this application, for example, customer profile information or cached data. It is also responsible for accessing one or more back-end applications. The back-end applications contain business logic and are responsible for accessing the existing back-end data. The communication between the presentation tier and Web application tier is synchronous. The communication between the Web application tier and the back-end can be either synchronous or asynchronous, depending on the characteristics and capabilities of the back-end application.

3. As-is Host application pattern

Provides simple direct access to existing host applications. The application is unchanged, but the user access is translated from green-screen type access to Web browser-based access. This is very quickly implemented but does nothing to change the appearance of the application to the user. The business logic and presentation are both handled by the back-end host. Because the interface is still host driven, this is more suited to an intranet solution where employees are familiar with the application.

4. Customized Presentation to Host application pattern

This is one step up from the As-is Host pattern. The back-end host application remains unchanged, but a Web application now translates the presentation from the back-end host application into a more user-friendly, graphical view. The back-end host application is not aware of this translation.

5. Router application pattern

The Router application pattern provides intelligent routing from multiple channels to multiple back-end applications using a hub-and-spoke architecture. The interaction between the user and the back-end application is a one-to-one relation, meaning the user interacts with applications one at a time. The router maintains the connections to the back-end applications and pools connections when appropriate, but there is no true integration of the applications themselves. The router can use a read-only database, most probably to look up routing information. The primary business logic still resides in the back-end application tier.
This pattern assumes that the users are accessing the applications from a variety of client types such as Web browsers, voice response units (VRUs), or kiosks. The Router application pattern provides a common interface for accessing multiple back-end applications and acts as an intermediary between them and the delivery channels. In doing this, the Router application pattern may use elements of the Integration patterns.

6. Decomposition application pattern

The Decomposition application pattern expands on the Router application pattern, providing all the features and functions of that pattern and adding recomposition/decomposition capability. It provides the ability to take a user request and decompose it into multiple requests to be routed to multiple back-end applications. The responses are recomposed into a single response for the user. This moves some of the business logic into the decomposition tier, but the primary business logic still resides in the back-end application tier.

7. Agent application pattern

The Agent pattern includes the functions of the decomposition tier, plus it incorporates personalization into the application to provide a customer-centric view. The agent tier collects information about the users, either from monitoring their habits or from information stored in a CRM. It uses this information to customize the view presented to users and can perform cross-selling functions by pushing offers to the users.

2.3 Application patterns used in this book

The rest of this redbook concentrates on e-business solutions based on the following Self-Service Application patterns:

- Stand-Alone Single Channel application pattern
- Directly Integrated Single Channel application pattern
The Stand-Alone Single Channel application pattern and Directly Integrated Single Channel application pattern represent a starting point for delivering e-business applications. In the simplest case (Stand-Alone) there is no interaction with back-end systems. In the more advanced case (Directly Integrated), there are connections to new or legacy back-end systems. In both cases, there is a single delivery channel assumed.

The next step is to choose Runtime patterns that most closely match the requirements of the application. A Runtime pattern uses nodes to group functional and operational components. The nodes are interconnected to solve a business problem. Each Application pattern leads to one or more underpinning Runtime patterns.
3.1 An introduction to the node types

A Runtime pattern consists of several nodes representing specific functions. Most Runtime patterns consist of a core set of common nodes, with the addition of one or more nodes unique to that pattern. To understand the Runtime pattern, you will need to review the node definitions described in the following sections.

3.1.1 Web application server node

A Web application server node is an application server that includes an HTTP server (also known as a Web server) and is typically designed for access by HTTP clients and to host both presentation and business logic.

The Web application server node is a functional extension of the informational (publishing-based) Web server. It provides the technology platform and contains the components to support access to both public and user-specific information by users employing Web browser technology. For the latter, the node provides robust services to allow users to communicate with shared applications and databases. In this way it acts as an interface to business functions, such as banking, lending, and Human Resources (HR) systems.

The node can contain these data types:
- HTML text pages, images, multimedia content to be downloaded to the client browser
- Servlets, JavaServer Pages
- Enterprise beans
- Application program libraries, such as Java applets for dynamic download to client workstations

See also “Public Key Infrastructure (PKI)” on page 27.

3.1.2 User node

The user node is most frequently a personal computing device (PC) supporting a commercial browser, for example, Netscape Navigator and Internet Explorer. The browser is expected to support SSL and some level of DHTML. Increasingly, designers need to also consider that this node might be a pervasive computing device, such as a personal digital assistant (PDA).
3.1.3 Public Key Infrastructure (PKI)

PKI is a system for verifying the authenticity of each party involved in an Internet transaction, protecting against fraud or sabotage, and for nonrepudiation purposes to help consumers and retailers protect themselves against denial of transactions. Trusted third-party organizations called certificate authorities issue digital certificates, which are attachments to electronic messages that specify key components of the user’s identity.

During an Internet transaction using signed, encrypted messages, the parties can verify that the other’s certificate is signed by a trusted certificate authority before proceeding with the transaction. PKI can be embedded in software applications or offered as a service or a product. e-Business leaders agree that PKI is critical for transaction security and integrity, and the software industry is moving to adopt open standards for their use.

3.1.4 Domain Name System (DNS) node

The DNS node assists in determining the physical network address associated with the symbolic address (URL) of the requested information. The Domain Name Server node provides the technology platform to provide host-to-IP address mapping; that is, to allow for the translation of names (referred to as URLs) into IP addresses and vice versa.

3.1.5 Protocol firewall node

A firewall is a hardware/software system that manages the flow of information between the Internet and an organization’s private network. Firewalls can prevent unauthorized Internet users from accessing private networks connected to the Internet, especially intranets, and can block some virus attacks, as long as those viruses are coming from the Internet. A firewall can separate two or more parts of a local network to control data exchange between departments. Components of firewalls include filters or screens, each of which controls the transmission of certain classes of traffic. Firewalls provide the first line of defense for protecting private information, but comprehensive security systems combine firewalls with encryption and other complementary services, such as content filtering and intrusion detection.

Firewalls control access from a less trusted network to a more trusted network. Traditional implementations of firewall services include:

- Screening routers (the protocol firewall)
- Application gateways (the domain firewall)
A pair of firewall nodes provides increasing levels of protection at the expense of increasing computing resource requirements. The protocol firewall is typically implemented as an IP router.

3.1.6 Domain firewall node

The domain firewall is typically implemented as a dedicated server node. See “Protocol firewall node” on page 27 for a description of firewalls.

3.1.7 Directory and security services node

The directory and security services node supplies information on the location, capabilities, and attributes (including user ID/password pairs and certificates) of resources and users known to this Web application system. This node can supply information for various security services (authentication and authorization) and can also perform the actual security processing, for example, to verify certificates. The authentication in most current designs validates the access to the Web application server part of the Web server, but this node also authenticates for access to the database server.

3.1.8 Database server node

The function of this node is to provide persistent data storage and retrieval in support of the user-to-business transactional interaction. The data stored is relevant to the specific business interaction, for example, bank balance, insurance information, and current purchase by the user.

It is important to note that the mode of database access is perhaps the most important factor determining the performance of this Web application in all but the simplest cases. The recommended approach is to collapse the database accesses into a single or few calls. One approach for achieving this is by coding and invoking stored procedure calls on the database.

3.1.9 Existing applications and data node

Existing applications are run and maintained on nodes, which are installed in the internal network. These applications provide for business logic that uses data maintained in the internal network. The number and topology of these existing application and data nodes is dependent on the particular configuration used by these legacy systems.
3.1.10 Web server redirector node

In order to separate the Web server from the application server, a so-called Web server redirector node (or just redirector for short) is introduced. The Web server redirector is used in conjunction with a Web server. The Web server serves HTTP pages and the redirector forwards servlet and JSP requests to the application servers. The advantage of using a redirector is that you can move the application server behind the domain firewall into the secure network, where it is more protected than within the DMZ.

3.1.11 Application server node

The application server node provides the infrastructure for application logic and can be part of a Web application server. It is capable of running both presentation and business logic but generally does not serve HTTP requests. When used with a Web server redirector, the application server node can run both presentation and business logic. In other situations, it can be used for business logic only.

See also “Web server redirector node” on page 29.

3.2 Runtime patterns for Stand-Alone Single Channel

The Stand-Alone Single Channel application pattern addresses a situation where the data accessed by the application resides on a local database server. In the Runtime patterns, this node is represented by the database server node.

3.2.1 Basic Runtime pattern

This Runtime pattern, shown in Figure 3-1 on page 30, provides an initial implementation with an entry-level footprint. It is a simple yet effective way to make the solution available. The basic pattern uses a minimum of runtime nodes, yet provides a measure of security by putting all sensitive persistent data behind a firewall. While it does not provide scalability or failover capabilities, it is a good starting point from which you can easily progress to Runtime patterns that do provide these functions.

The Runtime pattern does not differentiate between intranet and Internet implementations. However, you should be aware of certain issues:

- Bandwidth is usually greater in an intranet, allowing the use of more network-intensive solutions, such as thick clients.
Security may be less of an issue in an intranet. However, protecting the network is still important and firewalls protecting resources from unauthorized access are still advisable.

Due to corporate rules, you may expect certain browsers to be used, allowing you to exploit all available features and not code to the least common denominator.

Figure 3-1  Stand-Alone Single Channel application pattern::Runtime pattern

The presentation logic and business logic of the application are provided by a single Web application server node in a demilitarized zone (DMZ). The data to be accessed from the business logic is behind the domain firewall in the internal network.

In addition to the network security provided by the firewalls, application-level security is provided by the Web application server node. The user information required for authentication and authorization is stored in the directory and security services node behind the domain firewall in the internal network.
### 3.2.2 Runtime pattern: Variation 1

As shown in Figure 3-2, this variation to the basic Runtime pattern uses one Web server redirector containing the Web server and one application server, effectively splitting the function of a Web application server across two machines. In this case the application server resides in the internal network to provide it with more security. The application server node will run both presentation and business logic. The Web server remains in the DMZ and serves static pages. A Web server redirector is used to forward the requests from the Web server to the application server.

![Figure 3-2](image)

We focus on this variation of the Stand-Alone Single Channel application pattern throughout this book.
3.2.3 Runtime pattern: Other variations

Other variations of the Stand-Alone Single Channel application pattern::Basic runtime pattern can be used to address the availability and performance requirements of your application. The Patterns for e-business define Non-Functional Requirements custom designs that enable high-availability and high-performance e-business applications, using a variety of middleware configurations within the demilitarized zone or external network.

The high-availability patterns provide the node redundancy and failover capabilities needed to eliminate single-point-of-failure for the end-to-end production system. The following Non-Functional Requirements::High Availability: Runtime patterns are defined:

- High Availability: Basic Runtime pattern
- High Availability: Runtime pattern: Single load balancer
- High Availability: Runtime pattern: Load balancer hot standby
- High Availability: Runtime pattern: Mutual high availability
- High Availability: Runtime pattern: Wide area load balancing

The high performance patterns provide the scalability and workload management capabilities needed to meet performance and throughput requirements. The following Non-Functional Requirements::High Performance: Runtime patterns are defined:

- High Performance: Basic Runtime pattern
- High Performance: Runtime pattern: Redirectors
- High Performance: Runtime pattern: Separation
- High Performance: Runtime pattern: Caching proxy

See the IBM Redbook Patterns for the Edge of Network, SG24-6822, for details on how these Non-Functional Requirements custom designs are applied to the Stand-Alone Single Channel application pattern::Runtime pattern.

3.3 Runtime patterns for Directly Integrated Single Channel

The Directly Integrated Single Channel application pattern extends the Stand-Alone Single Channel application pattern by providing connections to one or more back-end systems.
3.3.1 Basic Runtime pattern

The primary Runtime pattern represents one solution for the Self-Service pattern. Based on the Enterprise Solution Structure (ESS) Thin Client Transactional pattern, this runtime is a starting point for extending business to the Web.

In this basic Runtime pattern, shown in Figure 3-3 on page 34, the Web application server (where the Web server and the application server are running on the same machine) is in the demilitarized zone. The business logic is implemented on both the Web application server and existing applications in the internal network. User information, needed for authentication and authorization, is stored in the directory and security services node behind the domain firewall in the internal network. The existing applications and data accessed by the business logic are in the internal network behind the domain firewall.

The Runtime pattern does not differentiate between intranet and Internet implementations. However, you should be aware of certain issues:

- Bandwidth is usually greater in an intranet, allowing the use of more network-intensive solutions, such as thick clients.
- Security may be less of an issue in an intranet. However, protecting the network is still important and firewalls protecting resources from unauthorized access are still advisable.
- Due to corporate rules, you may expect certain browsers to be used, allowing you to exploit all available features and not code to the least common denominator.
Example

A discount brokerage firm wishes to establish a Web sales channel. They may select the Directly Integrated Single Channel application pattern because real-time integration with their back-end applications is critical. In implementing this application they could use the Runtime pattern shown in Figure 3-3. The Web application server would host the presentation and some limited business logic. The majority of the business logic would continue to reside on existing applications and data sources. The directory and security services node is used to implement features including authentication and authorization.

3.3.2 Runtime pattern: Variation 1

As shown in Figure 3-4 on page 35, this variation to the basic Runtime pattern uses one Web server redirector containing the Web server and one application server, effectively splitting the function of a Web application server across two machines. In this case the application server resides in the internal network to
provide it with more security. The application server node will run both presentation and business logic. The Web server remains in the DMZ and serves static pages. A Web server redirector is used to forward the requests from the Web server to the application server.

The discount brokerage application described earlier can be deployed on this Runtime pattern variation to achieve a higher level of security. The discount brokerage application developed complex investment tools within the Web application. These tools need to be protected behind a firewall. This approach helps secure their highly sensitive business logic.

We focus on this variation of the Directly Integrated Single Channel application pattern throughout this book.
3.3.3 Runtime pattern: Other variations

Other variations to the Directly Integrated Single Channel application pattern::Basic runtime pattern can be used to address the availability and performance requirements of your application.

As with the Stand-Alone Single Channel application pattern::Runtime pattern, the Non-Functional Requirements custom designs can be applied to the Directly Integrated Single Channel application pattern::Basic runtime. See 3.2.3, “Runtime pattern: Other variations” on page 32, for details.
Product mappings

The next step after choosing a Runtime pattern is to determine the actual products and platforms to be used. It is suggested that you make the final platform recommendation based on the following considerations:

- Existing systems and platform investments
- Customer and developer skills available
- Customer choice

The platform selected should fit into the customer's environment and ensure quality of service, such as scalability and reliability, so that the solution can grow along with the e-business.

Our sample application, based on the Stand-Alone Single Channel and Directly Integrated Single Channel application patterns, has been implemented using IBM WebSphere Application Server V5.0 in Microsoft Windows 2000, IBM AIX®, and Red Hat Linux environments.

The following alternatives are detailed for implementation of the connection between the front-end application server and the back-end enterprise application:

- For the Stand-Alone Single Channel application pattern:
  - No back-end integration
For the Directly Integrated Single Channel application pattern:

- Web services using the Web services support provided with IBM WebSphere Application Server base V5.0
- J2EE Connectors using IBM CICS
- Java Message Service (JMS) using IBM WebSphere MQ

This chapter introduces the major products used in the application and provides an overview of the products as they apply to these two Self-Service Runtime patterns.

**Note:** You only need a subset of the products detailed in this chapter, depending on the back-end application connectivity needed (no back-end, Web services, J2EE Connectors, or JMS).

You can also use any combination of the options concurrently. In Part 3, “Technical scenarios” on page 251, our sample application uses all options concurrently.

Please refer to the Product mapping diagrams to determine which products are needed.
4.1 IBM WebSphere Application Server

WebSphere Application Server V5.0 represents a continuation of the evolution to a single, integrated, cost-effective, Web services-enabled, J2EE server foundation for applications that offer customers:

- One deployment model
- One administration point
- One programming model
- One integrated application development environment

With WebSphere Application Server V5.0, IBM enables customers to expand their business opportunities and productivity through a world class infrastructure ready for e-business on demand.

IBM WebSphere Application Server V5.0 comes in a number editions, each offering a unique combination of features geared toward different customer needs.

4.1.1 IBM WebSphere Application Server Express V5.0

IBM WebSphere Application Server Express V5.0 provides a combination of development tools and application servers in a single integrated package geared toward developing Web page-centric applications. It provides a simplified programming model that allows you to create new Web applications and to convert existing static applications to dynamic applications.

It provides a cost-effective starting point for businesses that want to have a presence on the Web. As your business needs grow, the WebSphere family provides a smooth migration path to higher-end WebSphere Application Server and WebSphere Studio configurations.

More information about IBM WebSphere Application Server Express V5.0 can be found at:

http://www.ibm.com/software/webservers/appserv/express

4.1.2 IBM WebSphere Application Server base V5.0

IBM WebSphere Application Server base V5.0 provides a robust application deployment environment for single-server light production situations.

It contains a base application server that supports the full J2EE environment. It allows a full range of enterprise integration and offers enhanced security, performance, availability, connectivity, and scalability options. Administration is done through a Web-based interface or through a scripting tool.
More information about IBM WebSphere Application Server base V5.0 can be found at:

http://www.ibm.com/software/webservers/appserv/was/

4.1.3 IBM WebSphere Application Server Network Deployment V5.0

IBM WebSphere Application Server Network Deployment V5.0 is designed to add non-programming enhancements to the features provided in the base edition. These enhancements add scalability features, allowing you to run applications on multiple servers and on multiple physical nodes.

In addition to the features included with the base edition of WebSphere Application Server you get:

- The Deployment Manager, allowing you to centrally manage a number of different application server instances and clustering for workload management and failover.
- The Network Dispatcher and Caching Proxy Server. These features provide the edge of network functions required to set up a classic DMZ in front of the application server.
- A private UDDI registry for easier deployment of internal Web services applications and a Web Services Gateway.

More information about IBM WebSphere Application Server Network Deployment V5.0 can be found at:

http://www.ibm.com/software/webservers/appserv/was/network/

4.1.4 IBM WebSphere Application Server Enterprise V5.0

IBM WebSphere Application Server Enterprise V5.0 provides the features in IBM WebSphere Application Server Network Deployment V5.0 plus programming model extensions for sophisticated application designs.

It offers advanced capabilities such as advanced application adapters, application workflow composition and choreography, extended messaging, dynamic rules-based application adaptability, internationalization, and asynchronous processing.

WebSphere MQ is bundled with the package (except on z/OS™).

More information about IBM WebSphere Application Server Enterprise V5.0 can be found at:

http://www.ibm.com/software/webservers/appserv/enterprise/
4.2 IBM CICS

IBM Customer Information Control System (CICS) is a family of application servers and connectors that provides industrial-strength, online transaction management and connectivity for mission-critical applications. Existing CICS installations process more than $1 trillion in transactions each day.

Our J2EE Connector scenario makes use of the following IBM CICS components:

- IBM CICS Transaction Server
- CICS Transaction Gateway (CICS TG)

More information can be found at the IBM CICS Web site:

http://www.ibm.com/software/ts/cics

4.2.1 IBM CICS Transaction Server

The IBM CICS Transaction Server is an online transaction processing (OLTP) environment for hosting business applications. It provides transaction management services, operating system services, database services, security services, and a variety of client access services. It allows resource managers to join transactions, and provides them with notification of transaction events, such as commit or rollback.

It supports numerous application development environments and models including COBOL, PL/I, Java, EJB, and object oriented (OO), in any combination. CICS Transaction Server runs on z/OS, OS/390®, and VSE/ESA™.

4.2.2 CICS Transaction Gateway

The CICS Transaction Gateway (CICS TG) has a long heritage as a Java connector for CICS, originally being provided as the CICS Gateway for Java for use with the CICS Client. Since then, CICS TG has advanced along with the Java world and now provides three principal interfaces for communication with CICS:

- Base classes
- Common Connector Framework API
- J2EE Common Client Interface

CICS TG is a well-proven and established product, available on multiple platforms, including AIX, Windows, z/OS, Solaris, HP-UX, and Linux on zSeries®.
4.3 IBM WebSphere MQ

IBM WebSphere MQ provides assured once-only delivery of messages across more than 35 industry platforms using a variety of communications protocols.

The transportation of message data through a network is made possible through the use of a network of WebSphere MQ queue managers. Each queue manager hosts local queues that are containers used to store messages. Through remote queue definitions and message channels, data can be transported to its destination queue manager.

To use the services of a WebSphere MQ transport layer, an application must make a connection to a WebSphere MQ queue manager, the services of which will enable it to receive (get) messages from local queues, or send (put) messages to any queue on any queue manager. The application's connection may be made directly (where the queue manager runs locally to the application) or as a client to a queue manager that is accessible over a network.

Dynamic workload distribution is another important feature of WebSphere MQ. This feature shares the workload among a group of queue managers that are part of the same cluster. This allows WebSphere MQ to automatically balance the workload across available resources, and provide hot standby capabilities if a system component fails. This is a critical feature for companies that need to maintain round-the-clock availability.

WebSphere MQ supports a variety of application programming interfaces (MQI, AMI, JMS), which provide support for several programming languages as well the point-to-point and publish/subscribe communication models. Beside support for application programming, WebSphere MQ provides a number of connectors and gateways to a variety of other products, such as Microsoft Exchange, Lotus® Domino™, SAP/R3, CICS, and IMS™ to name a few.

More information can be found at the IBM WebSphere MQ Web site:

http://www.ibm.com/software/ts/mqseries

4.4 Product mappings for Stand-Alone Single Channel

Figure 4-1 on page 43 shows a Stand-Alone Single Channel application pattern Product mapping based on the Windows 2000 operating system platform and Variation 1 of the Runtime pattern (see 3.2.2, “Runtime pattern: Variation 1” on page 31). We used this combination of runtime nodes and products to implement the sample application created for this book.
By using a Web server redirector node, we can place the majority of the business logic in the internal network, placing it behind two firewalls. The redirector is implemented using the IBM HTTP Server and WebSphere Application Server Web server plug-in. The redirector serves static HTML pages and forwards requests for dynamic content to a WebSphere application server using the HTTP protocol.

Figure 4-2 on page 44 shows this same implementation using AIX as the primary platform.
Figure 4-2  Stand-Alone Single Channel application pattern: AIX Product mapping

Figure 4-3 on page 45 shows this same implementation using Linux as the primary platform.
As shown in Figure 4-4 on page 46, the network protocols used for the Windows 2000, AIX, and Linux implementations are:

- **HTTP/HTTPS**: Hypertext Transfer Protocol (HTTP V1.1), or Hypertext Transfer Protocol Secure (HTTPS - HTTP V1.1/SSL V3), is used from the user’s Web browser to the HTTP server in the Web server redirector node. HTTP, or HTTPS, is also used from the WebSphere Web server plug-in in the Web server redirector node to the Web container in the Application server node.

- **LDAP**: The application server uses Lightweight Directory Access Protocol (LDAP V3) to access the LDAP server in the Directory and Security Services node.

- **JDBC**: The application server uses a Java Database Connectivity (JDBC V2.0) driver to access the database.
4.5 Product mappings for Directly Integrated Single Channel

Figure 4-5 on page 47 shows a Directly Integrated Single Channel application pattern Product mapping based on the Windows 2000 operating system platform and Variation 1 of the Runtime pattern (see 3.3.2, “Runtime pattern: Variation 1” on page 34). We used this combination of runtime nodes and products to implement the sample application created for this book.
By using a Web server redirector node, we can place the majority of the business logic in the internal network, placing it behind two firewalls. The redirector is implemented using the IBM HTTP Server and WebSphere Application Server Web server plug-in. The redirector serves static HTML pages and forwards requests for dynamic content to a WebSphere application server using the HTTP protocol.

As described at the start of this chapter, we examine the following alternatives for implementing the connection between the application server and the back-end enterprise application:

- **Web services**: This is the “Web services option” in Figure 4-5.
- **J2EE Connectors**: These are the “J2C option” in Figure 4-5.
- **JMS**: This is the “JMS option” in Figure 4-5.

For the Web services option, our implementation uses the Web services support provided by WebSphere Application Server. The front-end Java application uses SOAP/HTTP to communicate with the existing enterprise application.
For the J2C option, our implementation uses the CICS Transaction Gateway TCP protocol to communicate with the CICS Transaction Gateway on the zSeries enterprise system. The front-end Java application uses a CICS ECI J2EE Connector adapter to access the existing CICS enterprise application, via the CICS Transaction Gateway.

For the JMS option, WebSphere MQ provides the transport mechanism for JMS messages. In our implementation, we use an WebSphere MQ queue manager on each server to transport the messages. The Java application uses JMS to place messages on a local queue. WebSphere MQ is then responsible for ensured delivery of this message to the proper destination, in our case, the WebSphere MQ queue manager on the existing enterprise application.

The existing, back-end systems for our application consisted of a WebSphere Web service provider, a CICS enterprise application written in the C programming language, and the WebSphere JMS application for WebSphere MQ.

Figure 4-6 shows this same implementation using AIX as the primary platform.

Figure 4-7 shows this same implementation using Linux as the primary platform.
As shown in Figure 4-8 on page 50, the network protocols used for the Windows 2000, AIX, and Linux implementations are:

- **HTTP/HTTPS**: Hypertext Transfer Protocol (HTTP V1.1), or Hypertext Transfer Protocol Secure (HTTPS - HTTP V1.1/SSL V3), is used from the user’s Web browser to the HTTP server in the Web server redirector node. HTTP, or HTTPS, is also used from the WebSphere Web server plug-in in the Web server redirector node to the Web container in the Application server node.

- **LDAP**: The application server uses Lightweight Directory Access Protocol (LDAP V3) to access the LDAP server in the Directory and Security Services node.

- **JDBC**: The application server uses a Java Database Connectivity (JBDC V2.0) driver to access the database.

The protocol alternatives used to connect to the enterprise tier are:

- **SOAP/HTTP for the Web services option**:
  
  Simple Object Access Protocol (SOAP 1.1) and HTTP V1.1 are used between the Web services client in the application server node and the Web service provider in the existing enterprise tier.
Message data is passed using XML V1.0 with UTF-8 encoded character strings. Messages are validated using the message’s XML schema definition.

- **CICS TG TCP for the J2C option:**
  The proprietary CICS Transaction Gateway V5.0 TCP protocol is used from the J2C adapter in the application server to the CICS Transaction Gateway in the existing enterprise tier.
  Message data is passed using a byte array representing the CICS COMMAREA. Message data uses EBCDIC encoding (codepage 037).

- **WebSphere MQ for the JMS option:**
  The proprietary WebSphere MQ protocol is used from the WebSphere MQ Server on the application server node to the WebSphere MQ Server in the existing enterprise tier.
  Message data is passed using XML V1.0 with UTF-8 encoded character strings. Messages are validated using the message’s XML schema definition.

![Diagram of Directly Integrated Single Channel application pattern: Protocol mapping](image-url)
Guidelines

The chapters in Part 2 provide a set of guidelines for building your Web application, and for integrating it with the enterprise tier using Web services, J2EE Connectors, and JMS. These guidelines include technology options, application design, application development, and systems management.

Included in Part 2 are the following chapters:

- Chapter 5, “Technology options” on page 53
- Chapter 6, “Application and system design” on page 87
- Chapter 7, “Application development” on page 175
- Chapter 8, “Systems management” on page 205
Technology options

We take a look at the Web application technology options you should consider in this chapter. The recommendations are guided by the demands of reuse, flexibility, and interoperability, and subsequently are based on the open industry standards outlined by Java 2 Platform, Enterprise Edition (J2EE). Many of the choices continue to evolve and expand as the J2EE specification matures to include a broader view of the enterprise architecture. These recommendations are based on the J2EE1.3 specification and parts of the J2EE1.4 specification.

Our discussion of technology options is organized along the enterprise application tiers shown in Figure 5-1 on page 54:

- Web client technologies for providing client-side presentation
- Web application server technologies for providing server-side presentation and business logic
- Integration technologies for providing access to the enterprise tier
Figure 5-1  Self-Service application tiers
5.1 Web client

Figure 5-2 shows the recommended technologies for Web clients.

The clients are “thin clients” with little or no application logic. Applications are managed on the server and downloaded to the requesting clients. The client portions of the applications should be implemented in HTML, dynamic HTML (DHTML), XML, and Java applets.

The selection of client-side technologies used in your design will require consideration for the server side, such as whether to store, or dynamically create, elements for the client side.

The following sections outline some of the possible technologies that you should consider, but remember that your choices may be constrained by the policy of your customer or sponsor. For example, for security reasons, only HTML is allowed in the Web client at some government agencies.
We also touch on some of the current technology choices in the wireless area.

5.1.1 Web browser

A Web browser is a fundamental component of the Web client. For PC-based clients, the browser typically incorporates support for HTML, DHTML, JavaScript, and Java. Some browsers are beginning to add support for XML as well. Under user control, there is a whole range of additional technologies that can be configured as plug-ins, such as RealPlayer from RealNetworks or Macromedia Flash.

As an application designer, you must consider the level of technology you can assume will be available in the user’s browser, or you can add logic to your application to enable slight modifications based upon the browser level. For Internet users, this is especially true. With intranet users, you can assume support for a standard browser. Regarding plug-ins, you need to consider what portion of your intended user community will have that capability.

Cross-browser strategies are required to ensure robust application development. Although many of these technology choices are maturing, they continue to be inconsistently supported by the full range of browser vendors. Developers must know browser compatibility for all features being exploited by the application. In general, developers will need to code to a lowest denominator, or at least be able to distinguish among browser types using programmatic techniques. The key decision here is to determine the application requirements and behavior when handled by old browsers, other platforms such as Linux and Mac, and even the latest browsers.

In the J2EE model, the Web browser plays the role of client container. The model requires that the container provide a Java Runtime Environment as defined by the Java 2 Platform, Standard Edition (J2SE). However, for an e-business application that is to be accessed by the broadest set of users with varying browser capabilities, the client is often written in HTML with no other technologies. On an exception basis, limited use of other technologies, such as using JavaScript for simple edit checks, can then be considered based on the value to the user and the policy of the organization for whom the project is being developed.

The emergence of pervasive devices introduces new considerations to your design with regard to the content streams that the device can render and the more limited capabilities of the browser. For example, Wireless Application Protocol (WAP) enabled devices render content sent in Wireless Markup Language (WML).
5.1.2 HTML

HyperText Markup Language (HTML) is a document markup language with support for hyperlinks, that is, rendered by the browser. It includes tags for simple form controls. Many e-business applications are assembled strictly using HTML. This has the advantage that the client-side Web application can be a simple HTML browser, enabling a less capable client to execute an e-business application.

The HTML specification defines user interface (UI) elements for text with various fonts and colors, lists, tables, images, and forms (text fields, buttons, checkboxes, and radio buttons). These elements are adequate to display the user interface for most applications. The disadvantage, however, is that these elements have a generic look and feel, and they lack customization. As a result, some e-business application developers augment HTML with other user-interface technologies to enhance the visual experience, subject to maintaining access by the intended user base and compliance with company policy on Web client technologies.

Because most Web browsers can display HTML Version 3.2, this is the lowest common denominator for building the client side of an application. To ensure compatibility, developers should be unit testing pages against a validator tool. Free tools, such as the W3C HTML Validation Service, are available at:

http://validator.w3.org/

5.1.3 Dynamic HTML

DHTML allows a high degree of flexibility in designing and displaying a user interface. In particular, DHTML includes Cascading Style Sheets (CSS, which enable different fonts, margins, and line spacing for various parts of the display to be created. These elements can be accurately positioned using absolute coordinates. See 5.1.4, “Cascading Style Sheets (CSS)” on page 58, for details on Cascading Style Sheets.

Another advantage of DHTML is that it increases the level of functionality of an HTML page through a document object model and event model. The document object enables scripting languages such as JavaScript to control parts of the HTML page. For example, text and images can be moved about the window, and hidden or shown, under the command of a script. Also, scripting can be used to change the color or image of a link when the mouse is moved over it, or to validate a text input field of a form without having to send it to the server.
Unfortunately, there are several disadvantages to using DHTML. The greatest of these is that two different implementations (Netscape and Microsoft) exist and are found only on the more recent browser versions. A small, basic set of functionality is common to both, but differences appear in most areas. The significant difference is that Microsoft allows the content of the HTML page to be modified by using either JScript or VBScript, while Netscape allows the content to be manipulated (moved, hidden, shown) using JavaScript only.

Due to varying levels of browser support, cross-browser design strategies must be used to ensure appropriate presentation and behavior of DHTML elements. In general this technology is not recommended unless its features are needed to meet usability requirements.

5.1.4 Cascading Style Sheets (CSS)

Cascading Style Sheets allow you to define a common look and feel for HTML documents. This specification describes how Web documents are to be presented in print and online.

CSS is defined as a set of rules that are identified by selectors. When processed by the client browser, the selectors are matched to specific HTML tags and then are applied against the properties of the tag. This allows for global control over colors, fonts, margins, and borders. More advanced commands allow for control over pixel coordinates. Related stylesheet commands can be grouped and then externalized as a separate template file to be referenced by a multitude of Web pages.

CSS is defined as level 1 and level 2 specifications. Level 1 was written with HTML in mind, while level 2 was expanded to include general markup styles for XML documents. Developers using CSS should unit test against a validator tool, such as the W3C CSS Validation Service at:

http://jigsaw.w3.org/css-validator/

Due to varying levels of browser support, cross-browser design strategies must be used to ensure appropriate presentation and behavior of CSS elements. In general, this technology should be used with great attention to support of specification elements.
5.1.5 **JavaScript**

JavaScript is a cross-platform, object-oriented scripting language. It has great utility in Web applications because of the browser and document objects that the language supports. Client-side JavaScript provides the capability to interact with HTML forms. You can use JavaScript to validate user input on the client and help improve the performance of your Web application by reducing the number of requests that flow over the network to the server.

ECMA, a European standards body, has published a standard (ECMA-262) that is based on JavaScript (from Netscape) and JScript (from Microsoft), called ECMAScript. The ECMAScript standard defines a core set of objects for scripting in Web browsers. JavaScript and JScript implement a superset of ECMAScript.

To address various client-side requirements, Netscape and Microsoft have extended their implementations of JavaScript in Version 1.2 by adding new browser objects. Because Netscape’s and Microsoft’s extensions are different from each other, any script that uses JavaScript 1.2 extensions must detect the browser being used, and select the correct statements to run.

One caveat is that users can disable JavaScript on the client browser, but this can be programmatically detected.

The use of JavaScript on the server side of a Web application is not recommended, given the alternatives available with Java. Where your design indicates the value of using JavaScript, for example for simple edit checking, use JavaScript 1.1, which contains the core elements of the ECMAScript standard.

5.1.6 **Java applets**

The Java applet offers the most user interface (UI) technology flexibility that can be run in a Web browser. Java provides a rich set of UI elements that include an equivalent for each of the HTML UI elements. In addition, because Java is a programming language, an infinite set of UI elements can be built and used. There are many widget libraries available that offer common UI elements, such as tables, scrolling text, spreadsheets, editors, graphs, charts, and so on.

You can use either the awt or the Swing classes to build a Java applet. But while designing your applet, you should keep in mind that Swing is supported only by later browser versions.

A Java applet is a program written in Java that is downloaded from the Web server and run on the Web browser. The applet to be run is specified in the HTML page using an APPLET tag:

```html
<APPLET CODEBASE="/mydir" CODE="myapplet.class" width=400 height=100>
```
For this example, a Java applet called “myapplet” will run. An effective way to send data to an applet is with the use of the PARAM tag. The applet has access to this parameter data and can easily use it as input to the display logic.

Java can also request a new HTML page from the Web application server. This provides a function equivalent to the HTML FORM submit function. The advantage is that an applet can load a new HTML page based upon the obvious (a button being clicked) or the unique (the editing of a cell in a spreadsheet).

A characteristic of Java applets is that they seldom consist of just one class file. On the contrary, a large applet may reference hundreds of class files. Making a request for each of these class files individually can tax any server and also tax the network capacity. However, packaging all of these class files into one file reduces the number of requests from hundreds to just one. This optimization is available in many Web browsers in the form of either a JAR file or a CAB file. Netscape and HotJava support JAR files simply by adding an ARCHIVE="myjarfile.jar" variable within the APPLET tag. Internet Explorer uses CAB files specified as an applet parameter within the APPLET tag. In all cases, executing an applet contained within a JAR/CAB file exhibits faster load times than individual class files. While Netscape and Internet Explorer use different APPLET tags to identify the packaged class files, a single HTML page containing both tags can be created to support both browsers. Each browser simply ignores the other’s tag.

JavaScript can be used to invoke methods on an applet using the SCRIPT tag in the applet’s HTML page.

A disadvantage of using Java applets for UI generation is that the required version of Java must be supported by the Web browser. Thus, when using Java, the UI part of the application will dictate which browsers can be used for the client-side application. Note that the leading browsers support variants of the JDK 1.1 level of Java, and they have different security models for signed applets.

Using Java plug-ins, you can extend the functionality of your browser to support a particular version of Java. Java plug-ins are part of the Java Runtime Environment (JRE) and are installed when the JRE gets installed on the computer. You can specify certain tags in your Web page to use a particular JRE. This will download the particular JRE if it is not found on the local computer. This can be done in HTML through either of the following:

- Conventional APPLET tag
- OBJECT tag instead of APPLET tag for Internet Explorer or the EMBED tag with the APPLET tag for Netscape
A second disadvantage of Java applets is that any classes that are not included as part of the Java support in the browser (such as widgets and business logic) must be loaded from the Web server as they are needed. If these additional classes are large, the initialization of the applet may take from seconds to minutes, depending upon the speed of the connection to the Internet.

Using HTTP tunneling, an applet can call back on the server without reloading the HTML page. For users who are behind a restrictive firewall, HTTP tunneling offers a bidirectional data connection to connect to a system outside the firewall.

Because of the above shortcomings, the use of Java applets is not recommended in environments where mixed levels and brands of browsers are present. Small applets may be used in rare cases where HTML UI elements are insufficient to express the semantics of the client-side Web application user interface. If it is absolutely necessary to use an applet, care should be taken to include UI elements that are core Java classes whenever possible.

5.1.7 XML (client side)

XML allows you to specify your own markup language with tags specified in a Document Type Definition (DTD) or XML Schema. Actual content streams are then produced that use this markup. The content streams can be transformed to other content streams by using Extensible Stylesheet Language (XSL), which is based on CSS.

For PC-based browsers, HTML is well established for both document content and formatting. The leading browsers have significant investments in rendering engines and a Document Object Model (DOM) based on HTML for manipulation by JavaScript.

XML seems to be evolving as a complementary role for active content within HTML documents for the PC browser environment.

For new devices such as WAP-enabled phones and voice clients, the data content and formatting is being defined by new XML schema: WML for WAP phone and VoiceXML for voice interfaces.

For most Web application designs, you should focus your attention on the use of XML on the server side. See 5.2.4, “XML” on page 68, for additional discussion of the server-side use of XML.
5.1.8 XHTML 1.1 (HTML 4.01)

Extended HyperText Markup Language (XHTML) is an extension to HTML 4, which supports document types that are XML based. It is intended to be used as a language for XML-conforming content as well as for HTML 4-conforming user agents.

The advantages of XHTML are:
- Since XHTML documents are XML conforming, they can be viewed, edited, and validated with standard XML tools.
- XHTML documents can be used to traverse either the HTML Document Object Model or the XML Document Object Model.

Some issues with XHTML are:
- XHTML documents are not as easy to create as HTML documents because XHTML is validated more strictly than HTML.
- HTML is already used so widely that it is difficult for XHTML to attract the attention of most Web developers.
- Browser support is not usually an issue as documents can be created using HTML-compatible XHTML that is understood by most browsers. There are also utilities that can be used to convert HTML documents to HTML-compatible XHTML.
- Development tool support for XHTML is also improving. The Page Designer tool in IBM WebSphere Studio Application Developer V5.0, for example, allows visual authoring of XHTML pages.

XHTML Basic is designed for Web clients that do not support the full set of XHTML features. It is meant to serve as a common language and share basic content across mobile phones, pagers, car navigation systems, vending machines, etc.

Some of the common features found in Wirless Markup Language (WML) and other subsets of HTML have been used as the basis for developing XHTML Basic:
- Basic text
- Basic forms and tables
- Hyperlinks

Some HTML 4 features have been found inappropriate for non-desktop devices, so extending and building on XHTML Basic will help to bridge that gap.
5.1.9 VoiceXML

VoiceXML is a dialog markup language that leverages the other specifications for creating dialogs that feature synthesized speech, digitized audio, speech recognition, DTMF (touch tone) input, etc.

5.1.10 XForms

XForms is W3C’s specification for Web forms that can be used with desktop computers, hand-held devices, etc. The disadvantage of the HTML Web forms is that there is no separation of purpose from presentation. XForms separates the data and logic of a form from its presentation. Also, XForms are device independent.

XForms uses XML for transporting the data that is displayed on the form and the data that is submitted from the form. HTML is used for the data display.

Currently, the main issue with XForms is that it is still an emerging technology, so browser and server support is not yet standard.

5.1.11 Mobile clients

Mobile clients include wireless devices such as phones, pagers, and PDAs. The challenges these devices bring as Web clients are based primarily on the very limited computer resources of the supporting platform. The goal, however, is to overcome these limitations to provide access to information and application services from anywhere by leveraging and extending the existing Web server architectures.

**Devices**

Mobile devices include wireless desktop PCs, WAP devices, i-mode devices, PDAs, and Phone w/Voice. PDA devices cannot run the major operating systems that run on desktop PCs and consequently there are various mobile device-specific platforms. Palm devices use Palm-OS. WinCE/PocketPC devices use a version of Microsoft Windows called Windows CE.

**Voice**

Voice-enabled applications allow for a hands-free user experience unencumbered by the limitations of computer interface controls.

Voice technology fall into two categories: Those that recognize speech and those that generate speech. The ability to recognize human voice by computers is called Automatic Speech Recognition (ASR). The ability to generate speech from written text is called speech synthesis or Text-to-Speech (TTS).
Architecture
Support for mobile clients impacts the runtime topology and therefore must be designed and implemented using best practices for system architecture. The good news is that any past investment in Web architecture to support Internet-based applications can be extended to support mobile clients.

A Wireless Application Protocol (WAP) gateway is used between the mobile client device and the Web server. The gateway translates requests from the wireless protocol into HTTP requests and, conversely, converts HTTP requests into the appropriate device format.

WAP
WAP is the Wireless Application Protocol. This is the standard for the presentation and delivery of information to wireless devices, which are platform, device, and network neutral. The goal of this protocol is to provide a platform for global, secure access through mobile phones, pagers, and other wireless devices.

Microbrowser
WAP microbrowsers run on mobile clients. They are responsible for the display of Web pages written in WML and can execute WMLScripts. These play the same role as HTML browsers that run on a PC.

WML
The Wireless Markup Language (WML) is based on XML and HTML 4.0 to fit small hand-held devices. It is a tag-based language that handles formatting static text and images, can accept data input, and can follow hyperlinks.

WMLScript
This is the companion language to WML, in the same way as JavaScript is a companion language to HTML. WMLScript allows for procedural programming such as loops, conditional, and event handling. It has been optimized for a small memory footprint and small devices. This language is derived from JavaScript.

Bluetooth
Bluetooth is a set of technical specifications for low-range (up to 30 feet) wireless devices that define standards such as power use and radio frequency. The goal of this technology is to connect a wide range of computing and telecommunication devices easily and simply in a peer-to-peer manner.
5.2 Web application server

Figure 5-3 shows the recommended technology model for a Web application server.

We assume in this book that you are using a Web application server and server-side Java. While there are many other models for a Web application server, this is the one that is experiencing widespread industry adoption.

Before looking at the technologies and APIs available in the Web application programming environment, first let us have a word about two fundamental operational components on this node: The HTTP server and the application server. For production applications, they should be chosen for their operational characteristics in areas such as robustness, performance, and availability.
We follow the well-known Model-View-Controller (MVC) design structure so often used in user interfaces. For the Web application programming model:

- The Model represents the data of the application, and the business rules and logic that govern the processing of the data. In a J2EE application, the model is usually represented to the View and the Controller via a set of JavaBean components.

- The View is a visual representation of the model. Multiple views can exist simultaneously for the same model, and each view is responsible for making sure that it is presenting the most current data by either subscribing to state change events or by making periodic queries to the model. With J2EE, the view is generally implemented using JavaServer Pages (JSP).

- The Interaction Controller decouples the visual presentation from the underlying business data and logic by handling user interactions and controlling access to the model. It processes the incoming HTTP requests and invokes the appropriate business or UI logic. Using J2EE, the controller is often implemented as a servlet.

See 6.1, “Application structure” on page 89, for more on MVC.

### 5.2.1 Java servlets

Servlets are Java-based software components that can respond to HTTP requests with dynamically generated HTML. Servlets are more efficient than CGI for Web request processing since they do not create a new process for each request.

Servlets run within a Web container as defined by the J2EE model and therefore have access to the rich set of Java-based APIs and services. In this model, the HTTP request is invoked by a client such as a Web browser using the servlet URL. Parameters associated with the request are passed into the servlet via the HttpServletRequest, which maintains the data in the form of name/value pairs. Servlets maintain state across multiple requests by accessing the current HttpSession object, which is unique per client and remains available throughout the life of the client session.

Acting as an MVC Controller component, a servlet delegates the requested tasks to beans that coordinate the execution of business logic. The results of the tasks are then forwarded to a View component, such as a JSP, to produce formatted output.

One of the attractions of using servlets is that the API is a very accessible one for a Java programmer to master. The specification of the J2EE 1.3 platform requires Servlet API 2.3 for support of packaging and installation of Web applications.
Servlets are a core technology in the Web application programming model. They are the recommended choice for implementing the Interaction Controller classes that handle HTTP requests received from the Web client.

### 5.2.2 JavaServer Pages (JSPs)

JSPs were designed to simplify the process of creating Web pages by separating Web presentation from Web content. In the page construction logic of a Web application, the response sent to the client is often a combination of template data and dynamically generated data. In this situation it is much easier to work with JSPs than to do everything with servlets. The JSP acts as the View component in the MVC model.

The chief advantage JSPs have over standard Java servlets is that they are closer to the presentation medium. A JavaServer Page is developed as an HTML page. Once compiled it runs as a servlet. JSPs can contain all of the HTML tags that Web authors are familiar with. A JSP may contain fragments of Java code that encapsulate the logic that generates the content for the page. These code fragments may call out to beans to access reusable components and enterprise data.

JSP technology uses XML-like tags and scriptlets written in Java programming language to encapsulate the conditional logic that generates dynamic content for an HTML page. In the runtime environment, JSPs are compiled into servlets before being executed on the Web application. Output is not limited to HTML but also includes WML, XML, cHTML, DHTML, and VoiceXML. The JSP API for J2EE 1.3 is JSP 1.2.

JSPs are the recommended choice for implementing the view that is sent back to the Web client. For those cases where the code required on the page is a large percentage of the page, and the HTML minimal, writing a Java servlet will make the Java code much easier to read and, therefore, maintain.

### 5.2.3 JavaBeans

JavaBeans is an architecture developed by Sun Microsystems, Inc. describing an API and a set of conventions for reusable, Java-based components. Code written to Sun's JavaBeans architecture is called JavaBeans or just beans. One of the design criteria for the JavaBeans API was support for builder tools that can compose solutions that incorporate beans. Beans may be visual or non-visual.
Beans are recommended for use in conjunction with servlets and JSPs in the following ways:

- As the client interface to the Model layer. An Interaction Controller servlet will use this bean interface.
- As the client interface to other resources. In some cases this may be generated for you by a tool.
- As a component that incorporates a number of property-value pairs for use by other components or classes. For example, the JavaServer Pages specification includes a set of tags for accessing JavaBeans properties.

### 5.2.4 XML

Extensible Markup Language (XML) and XSL stylesheets can be used on the server side to encode content streams and parse them for different clients, thus enabling you to develop applications for both a range of PC browsers and for the emerging pervasive devices. The content is in XML and an XML parser is used to transform it to output streams based on XSL stylesheets that use CSS.

This general capability is known as transcoding and is not limited to XML-based technology. The appropriate design decision here is how much control over the content transforms you need in your application. You will want to consider when it is appropriate to use this dynamic content generation and when there are advantages to having servlets or JSPs specific to certain device types.

XML is also used as a means to specify the content of messages between servers, whether the two servers are within an enterprise or represent a business-to-business connection. The critical factor here is the agreement between parties on the message schema, which is specified as an XML DTD or Schema. An XML parser is used to extract specific content from the message stream. Your design will need to consider whether to use an event-based approach, for which the SAX API is appropriate, or to navigate the tree structure of the document using the DOM API.

IBM's XML4J XML parser was made available through the Apache open source organization under the Xerces name. For open source XML frameworks, see:

http://xml.apache.org/

### Defining XML documents

XML documents are defined using DTDs or XML Schemas.

DTDs are a basic XML definition language, inherited from the SGML specification. The DTD specifies what markup tags can be used in the document along with their structure.
DTDs have two major problems:

- Poor data typing: In DTDs elements can only be specified as EMPTY, ANY, element content, or mixed element-and-text content, and there is no standard way to specify null values for elements.
- Data typing like date formats, numbers, or other common data types cannot be specified in the DTD, so a XML document may comply with the DTD but still have data type errors that can only be detected by the application.
- Not defined in XML: DTD uses its own language to define XML syntax that is not compliant to the XML specification. This makes it difficult to manipulate a DTD.

To solve these problems, the World Wide Web Consortium (W3C) defined a new standard to define XML documents called XML Schema. XML Schema provides the following advantages over DTDs:

- Strong typing for elements and attributes
- Standardized way to represent null values for elements
- Key mechanism that is directly analogous to relational database foreign keys
- Defined as XML documents, making them programmatically accessible

Even though XML Schema is a more powerful technology to define XML documents, it is also a lot harder to work with, so DTDs are still widely used to define XML documents. Additionally, simple, not hard-typified documents can be easily defined using DTDs with similar results to using XML Schema.

Whether to use one or the other will depend on the complexity of the messages and the validation requirements of the application. Actually in many cases both (a DTD and a XML Schema) are provided, so they can be used by the application depending on its requirements.

Note: We have to remember that the validation process of an XML document using XML Schemas is an expensive process. Validation should be performed only when it is necessary.

XSLT
Extensible Stylesheet Language Transformations (XSLT) is a W3C specification for transforming XML documents into other XML documents. The XSLT is built on top of the Extensible Stylesheet Language (XSL), which, like CSS2 seen in 5.1.4, "Cascading Style Sheets (CSS)" on page 58, is a stylesheet language for XML. Unlike CSS2, XSL is also a transformation language.

A transformation expressed in the XSLT language defines a set of rules for transforming a source tree to a result tree, and it is expressed in the form of a stylesheet.
An XSLT processor is used for transforming a source document to a result document. There are currently a number of XSLT processors available on the market. DataPower has introduced an XSL just-in-time (JIT) compiler, which speeds up the time taken for the XSL transformation.

The XSLT processor has a performance overhead, so online processing of larger documents can be slow.

**XML security**

XML security is an important issue, particularly where XML is being used to by organizations to interchange data across the Internet. Several new XML security specifications are working their way through three standards bodies—the World Wide Web Consortium (W3C), Internet Engineering Task Force (IETF), and Organization for the Advancement of Structured Information Standards (OASIS). We highlight a few of them here:

- **XML Signature Syntax and Processing** is a specification for digitally signing electronic documents using XML syntax. According to the W3C, “XML Signatures provide integrity, message authentication, and/or signer authentication services for data of any type, whether located within the XML that includes the signature or elsewhere.”

  A key feature of the protocol is the ability to sign parts of an XML document rather than the document in its entirety. This is necessary because an XML document might contain elements that will change as the document is passed along or various elements that will be signed by different parties.

  WebSphere Studio provides you with the ability to create (using a wizard) and verify XML digital signatures.

- **XML encryption** will allow encryption of digital content, such as Graphical Interchange Format (GIF) images or XML fragments. XML Encryption allows parts of an XML document to be encrypted while leaving other parts open, encryption of the XML itself, or the super-encryption of data (that is, encrypting an XML document when some elements have already been encrypted).

- **XML Key Management Specification (XKMS)** establishes a standard for XML-based applications to use Public Key Infrastructure (PKI) when handling digitally signed or encrypted XML documents. XML signature addresses message and user integrity, but not issues of trust that key cryptography ensures.

- **Security Assertion Markup Language (SAML)** is the first industry standard for secure e-commerce transactions using XML. It aims to standardize the exchange of user identities and authorizations by defining how this information is to be presented in XML documents, regardless of the underlying security systems in place.
For further discussion, see Sun ONE article *Riddle Me This: Is Your XML Data Safe?* by Brett Mendel:

http://sunonedev.sun.com/building/tech_articles/xmldata.html

## Advantages of XML

There are many advantages of XML in a broad range of areas. Some of the factors that influenced the wide acceptance of XML are:

- **Acceptability of use for data transfer**
  
  XML is a standard way of putting information in a format that can be processed and exchanged across different hardware devices, operating systems, software applications, and the Web.

- **Uniformity and conformity**
  
  XML gives you an common format that could be developed upon and is accepted industry-wide.

- **Simplicity and openness**
  
  Information coded in XML is human readable.

- **Separation of data and display**
  
  The representation of the data is separated from the presentation and formatting of the data for display in a browser or other device.

- **Industry acceptance**
  
  XML has been accepted widely by the information technology and computing industry. Numerous tools and utilities are available, along with new products for parsing and transforming XML data to other data, or for display.

## Disadvantages of XML

Some XML issues to consider are:

- **Complexity**
  
  While XML tags can allow software to recognize meaningful content within documents, this is only useful to the extent that the software reading the document knows what the tagged content means in human terms, and knows what to do with it.

- **Standardization**
  
  When multiple applications use XML to communicate with each other they need to agree on the tag names they are using. While industry-specific standard tag definitions often do exist, you can still declare your own non-standard tags.
- Large size
  XML documents tend to be larger in size than other forms of data representation.

### 5.2.5 Enterprise JavaBeans

Enterprise JavaBeans is Sun's trademarked term for its EJB architecture (or *component model*). When writing to the EJB specification, you are developing enterprise beans (or, if you prefer, EJBs).

Enterprise beans are distinguished from JavaBeans in that they are designed to be installed on a server and accessed remotely by a client. The EJB framework provides a standard for server-side components with transactional characteristics.

The EJB framework specifies clearly the responsibilities of the EJB developer and the EJB container provider. The intent is that the "plumbing" required to implement transactions or database access can be implemented by the EJB container. The EJB developer specifies the required transactional and security characteristics of an EJB in a deployment descriptor (this is sometimes referred to as declarative programming). In a separate step, the EJB is then deployed to the EJB container provided by the application server vendor of your choice.

There are three types of Enterprise JavaBeans:

- Session beans
- Entity beans
- Message-driven beans

A typical session bean has the following characteristics:

- Executes on behalf of a single client.
- Can be transactional.
- Can update data in an underlying database.
- Is relatively short lived.
- Is destroyed when the EJB server is stopped. The client has to establish a new session bean to continue computation.
- Does not represent persistent data that should be stored in a database.
- Provides a scalable runtime environment to execute a large number of session beans concurrently.

A typical entity bean has the following characteristics:

- Represents data in a database.
- Can be transactional.
- Shared access from multiple users.
- Can be long lived (lives as long as the data in the database).
- Survives restarts of the EJB server. A restart is transparent to the client.
- Provides a scalable runtime environment for a large number of concurrently active entity objects.

A typical message-driven bean has the following characteristics:
- Consumes messages sent to a specific queue.
- Is asynchronously invoked.
- Is stateless.
- Can be transaction aware.
- May update shared data in an underlying client message.
- Executes upon receipt of single client message.
- Has no component or home interface.
- Is removed when the EJB container crashes. The container has to re-establish a new message-driven object to continue computation.

Typically, an entity bean is used for information that has to survive system restarts. In session beans, on the other hand, the data is transient and does not survive when the client's browser is closed. For example, a shopping cart containing information that may be discarded uses a session bean, and an invoice issued after the purchase of the items is an entity bean.

An important design choice when implementing entity beans is whether to use Bean Managed Persistence (BMP), in which case you must code the JDBC logic, or Container Managed Persistence (CMP), where the database access logic is handled by the EJB container.

The business logic of a Web application often accesses data in a database. EJB entity beans are a convenient way to wrap the relational database layer in an object layer, hiding the complexity of database access. Because a single business task may involve accessing several tables in a database, modeling rows in those tables with entity beans makes it easier for your application logic to manipulate the data.
An important change to the specification in EJB 2.0 is the addition of a new enterprise bean type, the message-driven bean (MDB). The message-driven bean is designed specifically to handle incoming JMS messages. The EJB container uses message properties and a bean deployment descriptor to select the bean to invoke when a message arrives, so your application logic only needs to process the message contents.

The J2EE 1.3 platform requires support for EJB 2.0. As a tool provider the WebSphere Application Server V5.0 supports J2EE 1.3 and therefore supports EJB 2.0. EJBs are packaged into EJB modules (JAR files) and then combined with Web modules (WAR files) to form an enterprise application (EAR file). EJB deployment requires generating EJB deployment code specific to the target application server.

5.2.6 Additional enterprise Java APIs

The J2EE specification defines a set of related APIs that work together. Here are the remainder not discussed so far:

- **JNDI**: Java Naming and Directory Interface. This package provides a common API to a directory service independent of any directory access protocol. This allows for easy migration to new directory services. Through this interface, component providers can store and retrieve Java object instances by name. Service provider implementations include those for JDBC data sources, LDAP directories, RMI, and CORBA object registries. Sample uses of JNDI include:
  - Accessing a user profile from an LDAP directory
  - Locating and accessing an EJB home
  - Locating a driver-specific data source

- **RMI-IIOP**: Remote Method Invocation (RMI) and RMI over IIOP are part of the EJB specification as the access method for clients to access EJB services. From the component provider point of view, these calls are local. The EJB container takes care of calling the remote methods and receiving the response. To use this API, component providers create an IDL description of the EJB interface and then compile it to generate the client-side and server-side stubs. The stubs connect the object implementations with the Object Request Broker (ORB). ORBs communicate with each other through the Internet Inter-ORB Protocol (IIOP). RMI can also be used to implement limited-function Java servers.

- **JTA**: Java Transaction API. This Java API for working with transaction services is based on the XA standard. With the availability of EJB servers, you are less likely to use this API directly.

- **JAF**: JavaBeans Activation Framework. This API is not intended for typical application use, but it is required by the JavaMail API.
JavaMail: This is a set of classes for supporting e-mail. Functionally it provides APIs for reading, sending, and composing Internet mail. This API models a mail delivery system and requires the SMTP for sending mail and POP3 or IMAP for receiving mail. Special data wrapper classes are provided to view and edit data in the mail content. Support for MIME data is delegated to the JAF-aware beans.

JAXP: API for parsing and transforming XML documents.

JAAS: Java Authentication and Authorization Service.

5.3 Integration technologies

With the continuous progress of enterprise computing, more and more enterprises are finding the need to quickly adopt new technologies and integrate with existing applications. Furthermore, it is often not feasible for enterprises to completely discard their existing infrastructure, due to limitations in cost and human resources.

Enterprise application integration (EAI) allows disparate applications to communicate with each other. Some points you should consider while deciding on the integration technology between your application and the enterprise tier applications are as follows:

- The current infrastructure
  Do you already have a messaging system on the enterprise tier? Then it makes sense to go for JMS. Or if you have a legacy system, such as CICS or IMS, J2EE Connectors might be the better choice.

- Time to market
  Web service enabling an application is relatively fast with the Web services development tools available.

- Future expansion plans
  If you plan to expand your enterprise systems in the future, you need to keep in mind the integration with your current infrastructure and your planned infrastructure. Web services may provide the most cost-effective migration path in such a case.

- Reliability
  JMS with WebSphere MQ, for example, can be used to provide assured transfer of data, even when the enterprise application is unavailable.
5.3.1 Web services

The W3C’s Web Services Architecture Working Group has jointly come to agreement on the following working definition of a Web service:

“A Web service is a software application identified by a URI, whose interfaces and bindings are capable of being defined, described, and discovered as XML artifacts. A Web service supports direct interactions with other software agents using XML-based messages exchanged via Internet-based protocols.”

Basic Web services combine the power of two ubiquitous technologies: XML, the universal data description language, and the HTTP transport protocol widely supported by browser and Web servers.

Web services = XML + transport protocol (such as HTTP)

Let us take a closer look:

- Web services are self-contained.
  On the client side, no additional software is required. A programming language with XML and HTTP client support, for example, is enough to get you started. On the server side, merely a Web server and a servlet engine are required. It is possible to Web service enable an existing application without writing a single line of code.

- Web services are self-describing.
  Neither the client nor the server knows or cares about anything besides the format and content of request and response messages (loosely coupled application integration).
  The definition of the message format travels with the message. No external metadata repositories or code generation tools are required.

- Web services are modular.
  Web services are a technology for deploying and providing access to business functions over the Web; J2EE, CORBA, and other standards are technologies for implementing these Web services.
Web services can be published, located, and invoked across the Web. The standards required to do so are:

- Simple Object Access Protocol (SOAP), also known as service-oriented architecture protocol, an XML-based RPC and messaging protocol
- Web Service Description Language (WSDL) is a descriptive interface and protocol binding language
- Universal Description, Discovery, and Integration (UDDI), a registry mechanism that can be used to look up Web service descriptions

Web services are language independent and interoperable. The interaction between a service provider and a service requester is designed to be completely platform and language independent. This interaction requires a WSDL document to define the interface and describe the service, along with a network protocol (usually HTTP). Because the service provider and the service requester have no idea what platforms or languages the other is using, interoperability is a given.

Web services are inherently open and standards based. XML and HTTP are the technical foundation for Web services. A large part of the Web service technology has been built using open source projects. Therefore, vendor independence and interoperability are realistic goals.

Web services are dynamic. Dynamic e-business can become a reality using Web services because, with UDDI and WSDL, the Web service description and discovery can be automated.

Web services are composable. Simple Web services can be aggregated to more complex ones, either using workflow techniques or by calling lower-layer Web services from a Web service implementation.

WebSphere V5.0 provides support for Web services. WebSphere applications can send and receive SOAP messages and also communicate with UDDI registries to publish and find services.

For detailed information on Web services, check out the following:

IBM Redbooks:
- WebSphere Version 5 Web Services Handbook, SG24-6891
- Web Services Wizardry with WebSphere Studio Application Developer, SG24-6292
The World Wide Web Consortium (W3C) Web site at:

http://www.w3.org/

Static and dynamic Web services

There are two ways of binding to Web services: Static and dynamic.

- In the static process, the binding is done at design time. The service requester obtains a service interface and implementation description through a proprietary channel from the service provider (by e-mail, for example), and stores it in a local configuration file. No private, public, or shared UDDI registry is involved.

- The dynamic binding occurs at runtime. While the client application is running, it dynamically locates the service using a UDDI registry and then dynamically binds to it using WSDL and SOAP.

  This requires that the contents of the UDDI registry be trusted. Currently, only private UDDI networks can provide such control over the contents.

Web services and the service-oriented architecture

Service-oriented architectures (SOAs) support a programming model that allows service components residing on a network to be published, discovered, and invoked by each other in a platform, network protocol, and language-independent manner.

The origin of SOA can be traced back to Remote Procedure Calls (RPCs), distributed object protocols such as CORBA and Java RMI, and component-based architecture such as J2EE/EJBs (Sun) and (D)COM/COM+/.Net (Microsoft).

Using XML over HTTP, Web services extend the SOA programming model into the global Internet allowing the publication, deployment, and discovery of service applications over the Internet.

For more information on SOA and Web services, refer to:


This Web site provides a collection of IBM resources on this topic. For example, you can find an introduction to the SOA in a white paper titled Web Services Conceptual Architecture (WSCA 1.0).
Advantages of Web services

Web services technology enables businesses to:

- Deliver new IT solutions faster and at lower cost by focusing development effort on core business applications, while consuming Web services applications provided by others for non-core business applications.
- Protect their investment in IT legacy systems by using Web services to wrap legacy software systems for integration with modern IT systems.
- Integrate their business processes with customers and partners at less cost. Web services make this integration feasible by allowing businesses to share processes without sharing technology. With lower costs, even small business will be able to participate in B2B integration.
- Enter new markets and widen their customer base. Web services listed in UDDI registries can be “discovered” and thus are “visible” to the entire Web community.

Disadvantages of Web services

Some Web services issues to consider are:

- Binding to Web services dynamically requires that the contents of the UDDI registry be trusted. Currently, only private UDDI networks can provide such control over the contents.
- The SOAP server footprint is significant and the technology is relatively new, so adding the Web service provider stack to existing enterprise systems can be a problem.

5.3.2 J2EE Connector Architecture

The J2EE Connector Architecture is aimed at providing a standard way of accessing enterprise applications from a J2EE-based Java application. It defines a set of Java interfaces, through which application developers can access heterogeneous EIS systems, for example legacy systems such as CICS, and Enterprise Resource Planning (ERP) applications.

J2EE Connector Architecture 1.0 support is a requirement of the J2EE 1.3 specification. It provides access to a range of systems through a common client interface API (CCI). Application programmers code to the single API rather than having unique interfaces for each proprietary system. The link from the API to the enterprise system is called a resource adapter and is provided by a third-party vendor. This is somewhat analogous to the model for JDBC drivers. Resource adapters are packaged as resource adapter archive (RAR) files.
IBM WebSphere Application Server V5.0 supports the J2EE Connector Architecture 1.0, as required by the J2EE 1.3 specification. The administrative console supports J2EE Connector resource adapter configuration. The administrative console allows the association of connection factories for the resource adapter that encapsulate the pooling attributes. Component providers request a connection for an enterprise information system (EIS) from the connection factory through the JNDI lookup mechanism. IBM supplies resource adapters for enterprise systems such as CICS, HOD, IMS, SAP, and Crossworlds® as separate products.

IBM WebSphere Studio Application Developer V5.0 supports application development using J2EE Connectors, and development of custom J2EE Connectors.

**CICS resource adapter**

The CICS Transaction Gateway (CICS TG) V5 is a set of client and server software components that allow a Java application to invoke services in a CICS region.

The CICS TG offers three basic interfaces for Java clients:

- External Call Interface (ECI) for COMMAREA-based CICS applications
- External Presentation Interface (EPI) for 3270-based transactions
- External Security Interface (ESI) for password management in order to verify and change user IDs and passwords

The CICS resource adapter is covered in detail in the following chapters.

**IMS resource adapter**

The IMS Connector for Java provides a way to create Java applications that can access IMS transactions. The IMS Connector for Java uses IMS Connect to access IMS. IMS Connect is a facility that runs on the host IMS machine and supports TCP/IP and Local Option communication to IMS. A Java application or servlet accesses IMS Open Transaction Manager Access (OTMA) through IMS Connect. IMS Connect accepts messages from its TCP/IP clients and routes them to IMS OTMA using the Cross-System Coupling Facility (XCF).

The runtime component of IMS Connector for Java is provided as a component of IMS Connect, Version 1 Release 2 (Program Number 5655-E51). The J2EE Connector implementation of this runtime component is also referred to as the IBM WebSphere Adapter for IMS. It is packaged as a RAR file, imsico.rar, for deployment into a WebSphere Application Server. The RAR file is installed to a target directory from the IBM IMS Connect, Version 1 Release 2.
Advantages of J2EE Connectors

Some reasons to use J2EE Connectors are:

- The common client interface simplifies application integration with diverse EISs. This common interface makes it easy to plug third-party or home-grown resource adapters into your applications.
- Each EIS requires just one implementation of the resource adapter since there is no need to custom develop an adapter for every application.
- J2EE Connectors facilitate scalability and provide Quality of Service features transparently to the client application.
- J2EE Connector Architecture-compliant resource adapters are portable across J2EE application servers. If a vendor provides a resource adapter for WebLogic, for example, it should also work with the WebSphere Application Server.
- J2EE Connectors have low intrusion on the enterprise system because native client interfaces are utilized.

Disadvantages of J2EE Connectors

Some J2EE Connector issues to consider are:

- J2EE Connector Architecture has support only for synchronous communication. (The CICS adapter does offer support for non-blocking calls.) Support for asynchronous communications is expected in the J2EE Connector Architecture 1.5 specification.
- The J2EE Connectors standard is still relatively new, and performance compared with previous alternatives has not been firmly established. For example, some customers may prefer to continue with the well-proven non-J2EE Connector CICS TG base classes.
- Though J2EE Connector Architecture promises an abstraction to access any legacy system, with J2EE Connector Architecture 1.0, parts of the client application need to have resource adapter-specific implementation. This means that if you have to change the resource adapter (move to a different enterprise system, which provides a different adapter), the client application will be impacted.

For more information on J2EE Connectors and CICS, refer to the following redbooks:

- Java Connectors for CICS: Featuring the J2EE Connector Architecture, SG24-6401
- Revealed! Architecting Web Access to CICS, SG24-5466
5.3.3 Java Message Service

Messaging middleware is a popular choice for accessing existing enterprise systems in an asynchronous manner. It is one of the options if you are implementing a solution based on the Directly Integrated Single Channel application pattern.

A standard way for using messaging middleware from a Java application is using the Java Message Service (JMS) interface. JMS offers Java programmers a common way to create, send, receive and read enterprise messages. The JMS specification was developed by Sun Microsystems with the active involvement of IBM, other enterprise messaging vendors, transaction processing vendors, and RDBMS vendors.

In IBM WebSphere Application Server V5.0, the J2EE 1.3 specification is implemented, which includes JMS 1.0 and EJB 2.0.

According to the JMS 1.0 specification, a message provider is integrated in an application server. As shown in Figure 5-4, the integrated message provider makes it possible to communicate asynchronously with other WebSphere applications, without installing separate messaging software like IBM WebSphere MQ. WebSphere's integrated JMS server is based on IBM WebSphere MQ.

![Figure 5-4   Integrated JMS Provider](image)

An important new feature of EJB 2.0 is message-driven beans (MDB). As described in 5.2.5, “Enterprise JavaBeans” on page 72, message-driven beans are designed specifically to handle incoming JMS messages.
What messaging is
Messaging is a form of communication between two or more software applications or components. One strength of messaging is application integration. Messaging communication is loosely coupled, as compared to tightly coupled technologies such as Remote Method Invocation (RMI) or Remote Procedure Calls (RPC). The sender does not need to know anything about the receiver for communication. The message to be delivered is sent to a destination (queue) by a sender component, and the recipient picks it up from there. Moreover, the sender and receiver do not both have to be available at the same time to communicate.

JMS has two messaging styles, or in other words two domains:
- One-to-one, or point-to-point model
- Publish/subscribe model

JMS and IBM WebSphere MQ
When you want to integrate with an application not based on WebSphere Application Server V5.0, an external JMS Provider is needed. IBM WebSphere MQ V5.3 includes built-in JMS Provider support with enhanced performance features for integrating JMS applications with other applications.

WebSphere MQ enables application integration by allowing business applications to exchange information across different platforms, sending and receiving data as messages. WebSphere MQ takes care of network interfaces, assures once and once only delivery of messages, deals with communications protocols, dynamically distributes workload across available resources, and handles recovery after system problems.

Advantages of JMS
The JMS standard is important because:
- It is the first enterprise messaging API that has achieved wide cross-industry support.
- It simplifies the development of enterprise applications by providing standard messaging concepts and conventions that apply across a wide range of enterprise messaging systems.
- It leverages existing, enterprise-proven messaging systems.
- It allows you to extend existing message-based applications by adding new JMS clients that are integrated fully with their existing non-JMS clients.
- Developers have to learn only one common interface for accessing diverse messaging systems.
Disadvantages of JMS
Though JMS provides a common interface for Java applications to interact with messaging systems, it might lose out on some specific functionality offered by the messaging vendor. In that case, you might still have to write vendor-specific code to access such functionality.

JMS only provides asynchronous messaging so the design is more complex when addressing response correlation, error handling, and data synchronization.

Further information on JMS can be found in the IBM Redbook *MQSeries Programming Patterns*, SG24-6506.

5.3.4 Others
In this section we briefly touch on a few other integration technologies, including:

- RMI/IIOP
- CORBA

RMI/IIOP
Remote Method Invocation (RMI) APIs allow developers to build distributed applications in the Java programming language. They enable an object running in one Java Virtual Machine to access another object running in a different Java Virtual Machine.

The Internet Inter-ORB (Object Request Broker) Protocol (IIOP) is a protocol used for communication between CORBA object request brokers. An object request broker is a library that enables CORBA objects to locate and to communicate with one another.

RMI/IIOP is an implementation of the RMI API over IIOP that allows developers to write remote interfaces in the Java programming language.

CORBA
Common Object Request Broker Architecture (CORBA) is a platform-, language-, and vendor-neutral standard for writing distributed object systems. The CORBA standard was developed by the Object Management Group (OMG), a consortium of companies founded in 1989. CORBA offers a broad range of middleware services, including naming service, relationship service, and so on.

CORBA can be used for integration with legacy applications. This is done by creating a CORBA wrapper for the existing application, which can then be invoked by other applications.
CORBA is just a specification, and there are a number of vendors (such as IONA or Borland) that implement it. Each vendor will provide additional value-added services such as persistence, security, and so on, which can be leveraged by CORBA developers.

The disadvantage of CORBA is in the steep learning curve involved. Also, CORBA is slow-moving; it takes a long time for the OMG to adopt a new feature.

5.4 Where to find more information

For more information on topics discussed in this chapter, see:

- WebSphere Version 5 Web Services Handbook, SG24-6891
- Web Services Wizardry with WebSphere Studio Application Developer, SG24-6292
- Java Connectors for CICS: Featuring the J2EE Connector Architecture, SG24-6401
- Revealed! Architecting Web Access to CICS, SG24-5466
- MQSeries Programming Patterns, SG24-6506
- Maruyama, Hiroshi, Kent Tamura and Naohiko Uramoto, XML and Java: Developing Web Applications, Addison-Wesley 1999
- IBM CICS
  http://www.ibm.com/software/ts/cics
- IBM WebSphere MQ
  http://www.ibm.com/software/ts/mqseries
- ECMA Script language specification
  http://www.ecma-international.org/publications/standards/ECMA-262.HTM
- Java APIs and technology
  http://java.sun.com/products
- Validator tools
  http://validator.w3.org/
  http://jigsaw.w3.org/css-validator/
- Bluetooth Web site
  http://www.bluetooth.com
- Bluetooth Applications in Pervasive Computing white paper
- World Wide Web Consortium (W3C) site
  http://www.w3.org/
- Open source XML frameworks
  http://xml.apache.org/
- Sun ONE article, *Riddle Me This: Is Your XML Data Safe?* by Brett Mendel
  http://sunonedev.sun.com/building/tech_articles/xmldata.html
- Service-oriented architecture and Web services
Chapter 6. Application and system design

e-Business application design presents some unique challenges compared to traditional application design and development. The majority of these challenges are related to the fact that traditional applications were primarily used by a defined set of internal users, whereas e-business applications are used by a broad set of internal and external users such as employees, customers, and partners. Web applications must be developed to meet the varied needs of these end users. The following list provides key issues to consider when designing e-business applications:

- The user experience and the look and feel of the site need to be constantly enhanced to leverage emerging technologies and to attract and retain site users.
- New features have to be constantly added to the site to meet customer demands.
- Such changes and enhancements will have to be delivered at record speed to avoid losing customers to the competition.
- e-business applications in essence represent the corporate brand online. Developers have to work closely with the marketing department to ensure that the digital brand effectively represents the company image. Such intra-group interactions usually present content management challenges.
It is hard to predict the runtime load of e-business applications. Based on the marketing of the site, the load can increase dramatically over time. If the load increases, the design must allow such applications to be deployed in various high-volume configurations. It is important to be able to move Web applications between these runtime configurations without making significant changes to the code.

Security requirements are significantly higher for e-business applications compared to traditional applications. In order to execute traditional applications from the Web, a special set of security-related software may be needed to access private networks.

The emergence of the personal digital assistant (PDA) market and broadband Internet market will require the same information to be presented in various user interface formats. PDAs will require a lightweight presentation style to accommodate the low network bandwidth. Broadband users, on the other hand, will demand a highly interactive, rich graphical user interface.

In order to meet these challenges, it is critical to design Web applications to be flexible. This chapter helps you understand some of these design challenges and presents various design patterns that promote loosely coupled design to provide a maximum degree of flexibility in a Web application. We also provide application integration design guidelines and best practices for Web services, J2EE Connectors, and JMS.
6.1 Application structure

A Self-Service Web application can be viewed as a set of interactions between the browser and the Web application server. The interaction usually begins with an initial request by the browser for the welcome page of the application. This is most often done by the user typing in the welcome page URL on the browser. All subsequent interactions are initiated by the user by clicking a button or a link. This causes a request to be sent to the Web application server. The Web application server processes the request and dynamically generates a results page and sends it back to the client along with a set of buttons and links for the next request.

A closer examination of these interactions reveals a common set of processing requirements that need to be considered on the server side. These interactions can be easily mapped to the classical Model-View-Controller design pattern. As outlined in the book Design Patterns: Elements of Reusable Object-Oriented Software, the relationship between the MVC triad classes are composed of Observer, Composite, and Strategy design patterns.

We start this section with a detailed look at the MVC design pattern. After that, we examine some design patterns for interaction between MVC components and some implementation considerations for the MVC components, including:

- Result bean design pattern
- View bean design pattern
- Formatter beans design pattern
- Command bean design pattern
- Frameworks including Struts and EAD4J
- WebSphere command framework with EJBs
- Best practices for EJBs

6.1.1 Model-View-Controller design pattern

Over the years, a number of GUI-based client/server applications have been designed using the Model-View-Controller (MVC) design pattern. This powerful and well-tested design pattern can be extended to support Self-Service Web applications.

As shown in Figure 6-1 on page 90, Model represents the application object that implements the application data and business logic. The View is responsible for formatting the application results and dynamic page construction. The Controller is responsible for receiving the client request, invoking the appropriate business logic, and based on the results, selecting the appropriate view to be presented to the user.
A number of different types of skills and tools are required to implement various parts of a Web application. For example, the skills and tools required to design an HTML page are vastly different from the skills and tools required to design and develop the business logic part of the application. In order to effectively leverage these scarce resources and to promote reuse, we recommend structuring Web applications to follow the Model-View-Controller design pattern.

Throughout this chapter, \textit{Model} is often referred to as business logic, \textit{View} is referred to as page constructor or display page, and \textit{Controller} is referred to as interaction controller. This section further outlines the responsibilities of each of these components and discusses what technologies could be used to implement the same.

See Chapter 9, “PDK sample overview” on page 253, for a sample implementation of the MVC pattern using Struts.

\textbf{Interaction controller (Controller)}

The easiest way to think about the responsibility of the interaction controller is that it is the piece of code that ties protocol-independent business logic to a Web application. This means that the interaction controller's primary responsibility is mapping HTTP protocol-specific input into the input required by the protocol-independent business logic (that might be used by several different types of applications), scripting the elements of business logic together and then
delegating to a page construction component that will create the response page to be returned to the client. Here's a list of typical functions performed by the interaction controller:

- Validate the request and session parameters used by the interaction.
- Verify that the client has the necessary privileges to access the requested business task.
- Transaction demarcation.
- Invoke business logic components to perform the required tasks. This includes mapping the request and session parameters to the business logic component's input properties, using the output of the components to control logic flow, and correctly chaining the business logic.
- Call the appropriate page construction component based on the output of one or more of the business logic commands.

Interaction controllers can be implemented using either Java servlets or JSPs. It is important to note that interaction controller code is primarily Java code, and Java code is easy to develop and maintain using Java Integrated Development Environments (IDEs) such as VisualAge® for Java or IBM WebSphere Studio Application Developer. Since servlets are also Java classes it is possible to leverage such IDEs to write, compile, and maintain servlets.

JSPs, on the other hand, provide a simple script-like environment. Even though JSPs are primarily used for dynamic page construction, they can be used to code interaction controller logic. Due to their simplicity, JSPs have a broad appeal to script programmers, and many of the tools available today for JSPs are primarily targeted toward dynamic page construction. The latest application development tool from IBM, the IBM WebSphere Studio Application Developer, is designed to support the development of every facet of a Web application, including both servlets and JSPs.

Finally, it is up to the project team to decide whether to use JSPs, servlets, or both for coding interaction controller logic. What is much more important is to recognize the need for the separation between interaction controller logic and page construction logic. Such a separation is necessary under the following conditions:

- Display pages need to be reusable because they can be called by multiple interaction controllers. For example, Figure 6-2 on page 92 shows that an error page may be called by more than one interaction controller. In such a scenario, if we were to combine the error page construction logic and the interaction controller logic, then the error page logic would need to be duplicated in several places throughout the application.
The interaction controller is required to do page selection. There are several reasons for this, such as the need to include different display pages depending on runtime results, national language support, client browser types, customer types, etc. For example, Figure 6-3 shows an interaction controller calling a page constructor for either an administrator page or a page for normal users, based on the user type.

If there is a need to use different tools and skills for coding interaction controllers and display pages then it is good to separate the two to simplify the development process. Failing to do so could result in multiple people having to write different parts of the same file, thus complicating the version control and code management process.
Another common design issue to consider is the relationship between interactions and interaction controllers. The following is an overview of the options:

- **One interaction to one interaction controller**
  For every unique interaction, there is a unique interaction controller. For example, login is handled by loginServlet, and logoff is handled by logoffServlet.

- **One interaction group to one interaction controller**
  A group of related Web interactions are all handled by the same interaction controller. The interaction controller for the group is passed a parameter to differentiate which interaction within the group is being performed. For example, login and logoff both could be handled by authenticateServlet, which can get a parameter called action type that could be either set to login or logoff.

- **All interactions to one interaction controller**
  This approach extends the interaction group to all interactions and builds a monolithic interaction controller. This choice is also referred to as a gateway, and is best used in conjunction with a suite of interaction handler classes to which the servlet may delegate individual business requests.

We recommend a combination of these choices rather than any one extreme. One-to-one will not scale for very large applications. As a rule, a servlet can handle all of the pages related to one use case. In the case of fairly complex use cases, these can be broken up into a suite of servlets handling specific interactions, or employ the use of a gateway model with several interaction handlers.

**Page constructor (View)**
The page constructor is responsible for generating the HTML page that will be returned to the client; it is the view component of the application. Like interaction controllers, WebSphere allows display pages to be implemented as either servlets or JSPs. JSPs allow template pages to be developed directly in HTML, with scripting logic inserted for dynamic elements of the page and jsp:include actions for multipart pages. Hence, a JSP is the best choice for implementing page construction components.
In many cases, the interaction controller will pass the dynamic data as JavaBeans to the display page for formatting. In other cases, the display page will invoke business logic directly to obtain dynamic data. It makes sense to have the interaction controller pass the data when it has already obtained it and when the data is an essential component of the contract between the interaction controller and the display page. In other cases, the data needed for display is not an essential part of the interaction and can be obtained independently by inserting calls to business logic directly in the display page. However, such direct access to business logic from the page construction component increases the complexity of the display page, since the page designer must know the details of the business logic methods. For this reason, care must be taken to minimize such direct access to business logic from the display pages. Ideally, the source data the page constructor requires can be pre-packaged by the controller, which eliminates direct manipulation of the Model by the View.

Once the page constructor has obtained the dynamic data (either from the interaction controller or via its own logic), it will typically format the data. This can be done in two ways. The simplest mechanism is to format the data using simple scripting inside the page constructor. An alternative is to develop reusable formatting components called formatter beans that will take a data set and return formatted HTML.

**Business logic (Model)**

The business logic part of a Web application is the piece of code ultimately responsible for satisfying client requests. As a result, the business logic must address a wide range of potential requirements which include ensuring transactional integrity of application components, maintaining and quickly accessing application data, supporting the coordination of business workflow processes, and integrating new application components with existing applications. To address these requirements, WebSphere supports business logic written in Java and uses the full facilities of the Java runtime, including support for servlets, beans, enterprise beans, JDBC, CORBA, LDAP, JMS, and JCA connectors to CICS, IMS, and other enterprise services.

We recommend that the business logic be wrapped with JavaBeans or EJBs. Such a separation of business logic from the Web-specific interaction controller and display page logic isolates the business logic from the details of Web programming, increasing the reusability of the business logic in both Web and non-Web applications.

**Advantages and disadvantages of the MVC design pattern**

To summarize, the MVC design pattern recognizes various types of program logic involved in implementing a typical Web application and advocates the separation of business logic, page construction logic, and interaction controller
logic. We recommend using servlets for implementing interaction controllers, JSPs for implementing dynamic display pages, and simple JavaBeans and/or enterprise beans for implementing business logic. Such a separation provides the following advantages:

- **Leverages different skill sets**
  
  As discussed earlier, the skill sets required to design an HTML page are vastly different from the skill sets required to code the business logic in Java. The separation of concerns outlined here allows for the effective use of skilled resources.

- **Increases reusability**
  
  In a non-trivial application, there are usually display pages that can be called from multiple interaction controllers. For example, an error page may be called as a result of many interactions. Similarly, based on some conditions there might be a need to perform page selection. For example, one might have to display different pages for administration users and normal users. Finally, the business logic could be used by several interactions or applications. For example, you may have to display the current weather information on multiple pages of a Web application. Under these conditions, a clear separation of concerns would increase the potential for reuse.

- **Can support multiple user interfaces**
  
  e-Business applications often support multiple user interfaces such as HTML clients for the Internet, application clients for the call center, wireless hand-held PDAs, and voice response units (VRUs). Separating the presentation logic from the business logic allows reuse of the business logic component among these user interface environments. In addition to providing higher reusability, such a separation also ensures the consistency of the business logic across these applications.

- **Improves maintainability of the site**
  
  In this scenario it is easy to make changes to the user interface without affecting the business logic, and vice versa. For example, the user interface can be changed to leverage a new HTML standard such as CSS without affecting the business logic components, making it easy to respond to the demands of the business in record speed.

- **Reduces complexity**
  
  Any non-trivial application implemented without clear separation of concerns could result in large and complex code. For such applications, the MVC separation reduces the complexity.
On the other hand, it is important to recognize the following disadvantages of the MVC design pattern:

- **Could be overkill for small applications**
  
  The MVC design pattern can introduce extra artifacts that may not be necessary for very simple cases and may in fact increase the complexity of the application. However, if the application is likely to evolve over time, then it still may be beneficial to “pay now versus pay later” to gain the flexibility provided by the MVC design pattern.

- **High level of communication requirements between various groups**
  
  Since various groups would typically be responsible for implementing the various parts of the application, there is a need for a defined communication plan. For example, page developers need to know interaction controller names and vice versa. They have to agree upon a naming convention for various parameters and attributes, and interaction control developers need to know the business logic, etc.

### 6.1.2 Result bean design pattern

If the interaction controller expects more than one field as a result of executing the business logic, then we recommend returning a data bean that wraps all the result fields. Since this data bean represents the result of executing business logic, we call it a **result bean**. A result bean effectively defines the contract between a particular piece of business logic and a particular interaction controller. Result beans are usually implemented as simple JavaBeans. Since JavaBeans are by definition serializable, they can be passed by value between EJB-based business logic implementations. Whenever possible, result bean properties should be implemented as read-only properties. A constructor could be used to initialize these properties during instantiation. This prevents the interaction controllers and page constructors from inadvertently updating the data.

It is important to note that a result bean can be reused by multiple business logic methods or objects. For example, based on some condition, the controller may decide to call two different business logic methods. If it is appropriate, both methods could use the same result bean to return the results. The reverse can also be true. Multiple controllers may call the same business logic that returns the same result bean to all controllers.

Figure 6-4 on page 97 demonstrates the relationship between MVC components, result beans, and view beans. (See 6.1.3, “View bean design pattern” on page 97, for our discussion on view beans.)
Advantages of result beans

Some reasons to use result beans are:

- Result beans clearly define what the controller expects back from the business logic.

- Once a result bean is defined, the developers of the Controller and the Model can develop their components independently. This simplifies and optimizes the development process and allows for parallel development.

- Since result beans can be serialized, they can be sent to remote servers or received from remote servers, such as EJB-based distributed applications. In addition, they can be stored in a file for persistence purposes.

- The result bean data structure can be reused by multiple business logic objects and interaction controller objects.

6.1.3 View bean design pattern

A view bean defines the contract between the controller and the view. It lists all the attributes the JSP can display. The main benefit of defining such a view bean
is to make it easy for the JSP page designer to get all the required data in one place. The display page often contains the data from the following sources:

- Result bean properties (returned by the business logic)
- HTTP request data (including attributes, parameters, cookies, URL string)
- Session state
- Servlet context

Figure 6-4 on page 97 demonstrates the relationship between MVC components, view beans, and result beans. (See 6.1.2, “Result bean design pattern” on page 96, for our discussion on result beans.)

The controller is responsible for instantiating the view bean and initializing all of the properties of the bean. View beans can be designed to be responsible for view-specific transformations. For example, a view bean can be responsible for converting the monetary values into the user-preferred currency. Such a view bean can have two properties: The monetary value in a base currency and the currency display type. The controller can initialize both of these properties. The view bean can use this information to call a reusable currency conversion library and get the monetary value in the appropriate format.

Usually, view beans are tightly coupled with a JSP, since the primary purpose is to provide all the properties the JSP designer would need in one place. However, under special circumstances one can reuse the view beans by inheritance.

Both result and view beans are implemented using simple JavaBeans. In simple Web interactions, both a result bean and a view bean could be implemented by the same JavaBean.

**Advantages and disadvantages of view beans**

Some advantages of view beans are:

- Clearly defines all the fields a View (JSP) can display.
- Once a view bean is clearly defined, the developers of the Controller and the View can develop their components independently. This simplifies and optimizes the development process and allows for parallel development.
- The View (JSP) designer can get all the dynamic data from one view bean and use `<jsp:useBean>` and `<jsp:getProperty>` tags to insert these values. This allows the JSP designer to concentrate on the look and feel of the page rather than worry about gathering data from various sources (for example, sessions, cookies, result beans, etc.) and coding complex View-specific Java code. View beans effectively hide these complexities from the display page designer.
- Complete separation of the View-specific logic from the business logic; for example, currency conversion based on the user preference.
Using inheritance can promote view bean reuse and ensure that similar information is received by all users, for example, CSRs and customers.

View beans can be used with tools such as WebSphere Page Designer or IBM WebSphere Studio Application Developer, which allows a developer to insert JavaBean properties into JSPs.

JSPs and associated view beans can be exhaustively unit tested by the View developer prior to integration with the rest of the application.

On the other side, it is important to recognize the following disadvantages of introducing result beans and view beans:

- View beans are tightly coupled with display pages and interaction controllers. This implies that any changes to the dynamic content of the display page will require a change to the interaction controller. We depend on the interaction controller to gather all the required information in one view bean.
- For small applications, the introduction of view beans could result in too many individual pieces of code and could increase the complexity of application management.
- The number of artifacts to be coded, managed, and maintained will be increased.

### 6.1.4 Formatter beans design pattern

The view bean concept described above tries to minimize the need for inserting Java code directly inside a display page. In order to insert complex tables or drop-down lists, complex conditional loops may be needed. JSP API 1.1 does not define repeat tags that allow for looping through an indexed JavaBean property. One option to overcome this is to implement the complex table or drop-down list generation in Java and wrap it inside a bean. Such reusable beans are called formatter beans.

To summarize, a formatter bean is a bean that wrappers reusable HTML formatting logic inside a method.

#### Advantages and disadvantages of formatter beans

Some advantages of formatter beans are:

- Eliminates the need for inserting complex scripting logic inside a JSP.
- Promotes reusability of the formatting logic.
- Hides the complexity of scripting logic from view designers.
- Promotes the ability to drop common information on multiple pages, such as displaying the current weather information and current stock price on all the pages of the Web application.
A disadvantage of formatter beans is that some of the display page functionality that is best expressed in a JSP is now moved into Java code.

6.1.5 Command bean design pattern

The business logic part of a Web application must address a wide range of potential requirements, including transactional integrity, application data access, workflow support, and integration of new and legacy applications. In order to achieve this, business logic components may use various protocols, including JDBC, JNDI, IIOP, RMI, Web services, JCA, JMS, etc. to communicate with enterprise applications, enterprise data sources, and external applications. The Model is not only responsible for implementing the business logic, but also for hiding the details of the data and application access protocols. To simplify the implementation of the Model, we can implement one business logic bean per task. We call such beans *command beans*.

Command beans can be defined as JavaBeans that provide a standard way to invoke business logic. The following are the key characteristics of command beans:

- Each command bean corresponds to a single business logic task, such as a query or an update task.
- All command beans inherit from a single command interface. In essence, they implement the command interface.
- The inherited command bean defines business domain-specific properties such as account numbers.
- Command execution results are stored as properties of a command bean. Therefore, command beans also act as result beans.
- Commands have a simple, uniform usage pattern:
  a. Create an instance of the command bean.
  b. Initialize the bean by setting its properties.
  c. Cause the bean to execute its action by calling one of its methods.
  d. Access the results of command execution by inspecting its properties.
  e. Discard the command object.
- Commands can be serialized.

Figure 6-5 on page 101 shows how such a command bean interacts with the other components of the Web application.
6.1.6 Frameworks

In this section we look at the following Web application frameworks:

- The Apache Jakarta Project’s Struts framework
- IBM’s Enterprise Application Development frameworks for Java (EAD4J)

The goal of such frameworks is to make it easier to build and maintain Web applications with reusable components. There are some issues to consider when adopting a framework:

- Frameworks are restrictive: They are good for what they have been designed for, but nothing else. Framework choice is critical because customizing a framework can be difficult.
- Frameworks impose a way of thinking: Different ideas just do not fit. If the framework is well designed, this can be a good thing because it prevents bad practices.
- Learning curve: It takes time to get started with a framework, partly because there are more components to deal with and additional configuration is needed.

**Note:** The command bean contains the command execution results, so there is no need to introduce another result bean. In essence, the command bean encapsulates both the business logic and result data.
Struts

Struts is a Model II Servlet-JSP framework offered by the Apache Software Foundation. Struts supports application architectures based on the Model II approach, which is an implementation of the traditional MVC paradigm discussed earlier.

While J2EE APIs can be used to develop Web-based applications that implement the MVC pattern, there are a number of common problems that must be solved in every servlet project:

- Mapping HTTP parameters to JavaBeans: Servlet programmers need to map the HTTP parameters from the POST of an HTML form to a JavaBean for manipulation.
- Validation: Servlet programmers need to develop their own validation procedures to check that the form is filled in correctly.
- Error display: There is no standard way to display error messages in a JSP page or generate error messages in a servlet.
- Message internationalization: Internationalization can be difficult, as short error or informational messages are often spread throughout servlet and model.
- Hard coded JSP URIs: The URIs of JSP pages are usually coded directly into the code of the calling servlet. This means that it is impossible to reorganize the JSPs in a Web site, or even change their names, without updating Java code in the servlets.

Struts is an open-source framework for solving the kind of problems described above. Information on Struts, a set of installable JAR files, and the full Struts source code is available at the Apache Jakarta Project Struts Web site. Struts has been designed to be easy to use, modular (so that you can choose to use one part of Struts without having to use all of the others), and efficient. It has also been designed so that tool builders can easily write their tools to generate code that sits on top of the Struts framework.

Struts application design

True to the MVC design pattern, Struts applications have three major components:

- Controller implemented using the Struts ActionServlet and classes extending the Struts Action class
- View implemented using JavaServer Pages and Struts form beans
- Model implementing the application's business logic

The ActionServlet routes HTTP requests from the user to the appropriate action class. Action classes provide access to the application's business logic and
control how flow should proceed. Form beans are used to collect and validate form data from the user.

Figure 6-6 shows an example Struts form bean, TransferFundsForm, that has been defined in the struts-config.xml file and linked to an action mapping. When a request calls for the TransferFundsAction Struts action, the ActionServlet retrieves the form bean (or creates it if it does not exist), and passes it to the action.

![Figure 6-6  Struts action class diagram](image)

The action can then check the contents of the form bean before its input form is displayed, and also queue messages to be handled by the form. When ready, the action can return control with a forward to its output form, usually a JSP. The ActionServlet can then respond to the HTTP request and direct the client to the JSP. Figure 6-7 on page 104 summarizes this sequence.
See Chapter 9, “PDK sample overview” on page 253 for a sample implementation of the Struts framework.

For more information on Struts see:

- Redbook *Legacy Modernization with WebSphere Studio Enterprise Developer*, SG24-6806
- WebSphere Developer Domain for a number of articles on Struts: [http://www7b.software.ibm.com/wsdd/](http://www7b.software.ibm.com/wsdd/)

**EAD4J**

EAD4J is a enterprise application Java framework that is J2EE compliant and may be licensed from IBM Global Services. It represents IBM Global Services best practices for custom enterprise application development in the Java space. Harvested and hardened from hundreds of e-Business engagements, EAD4J is not limited to the Model II space but instead is a full feature, end-to-end J2EE framework aimed at the enterprise customer.
EAD4J is comprised of a set of components, each responsible for a different portion of the J2EE space, for example:

- EAD4J.Jade is based on the Model II Servlet-JSP pattern.
- EAD4J.Topaz provide support in the model/persistence areas.
- EAD4J.Opal handles logging.
- EAD4J.Ruby handles the XML/XSLT transcoding issues.

EAD4J is more than a Web application solution framework. EAD4J provides solutions for batch processing, interfaces with thick clients, back-office applications, or integration into existing applications using only a single component of the EAD4J suite. In addition to the code, EAD4J provides documentation, UML models, extensive Javadoc, reference applications, and training—required by customers who need an application infrastructure for mission critical and enterprise scale applications.

### 6.1.7 WebSphere command framework with EJBs

This section discusses the motivation for using the shippable command model in e-business applications with enterprise beans. We discuss how the command framework can be used with session and entity beans for the application model.

For a full discussion of the WebSphere command framework, see the following publications:

- *Design and Implement Servlets, JSPs, and EJBs for IBM WebSphere Application Server*, SG24-5754
- *Self-Service Patterns using WebSphere Application Server V4.0*, SG24-6175

More information on the command pattern is available in *Design Patterns: Elements of Reusable Object-Oriented Software*.

**Command model**

In their most basic form, commands simply encapsulate some request for information or action. Commands are particularly useful for significant program boundaries, such as the boundary between presentation code and business logic.

To use commands you must perform the following steps:

1. Create (instantiate).
2. Initialize by setting some of the command’s properties. This may be done in one of the command’s constructors (all commands must have a no-argument constructor to comply with the JavaBeans standard, but they may also have convenience constructors that take initial values for the command’s
properties) by calling some of the command's set methods, or by a combination of two mechanisms.

3. Call the command's execute() method. This is a no-argument method that makes the command's output properties ready for access.

And optionally:

4. Inspect the command's output properties by calling get methods. Depending on how you implement your commands, some operations may not require any output parameters.

The command package
WebSphere V5.0 includes support for the command model as formalized in the command package (com.ibm.webSphere.command) and extended to accommodate command shipping (called TargetableCommands). The concept behind command shipping is to intercept the execute method, ship the command to a better execution point (say on a remote server), execute it there, and then ship it back to the caller.

The command package is available to any WebSphere Java application. For example, you can implement command shipping using an entity bean. When execute() is called on a command, performExecute(TargetableCommand targetableCommand) is called on the entity bean. Or, for example, you could implement command shipping using a catcher servlet. In this approach, the CommandTarget class would construct and send an HttpServletRequest to a servlet on the EJB server. The servlet would retrieve the command from the request, execute it, then store the executed command in the HttpServletResponse object for return to the CommandTarget.

Both of these approaches are valid. The choice of which to use may depend on whether you want your servlet and EJB environments completely separate. Security is also a factor. The EJB approach transports over IIOP, which may present a problem in environments with strict firewall rules. The servlet catcher uses HTTP for the protocol, but in some environments Internet protocols are not allowed to pass through the firewall from the presentation layer to the application layer.

Advantages and disadvantages of command shipping
The command shipping model has several compelling advantages:

- It is a direct extension of the base command model and therefore maintains the same programming style and tooling advantages.
- It isolates application logic from communication protocols and routing policies. This allows the best protocol to be selected without requiring extensive application changes. Indeed, using techniques such as dynamic class
loading, new protocols can be supported “on-the-fly” without the need to change or recompile existing code.

- It supports an agent-oriented service definition model in which the service provider provides a functional interface without consideration for distribution overhead. The service client then defines commands based on the service interface. The commands are shipped to the service and run there. This allows the service client to control the granularity of remote communication and avoids many of the performance and complexity issues associated with remote interfaces.

- In an EJB environment, command shipping allows multiple EJB calls to be made without the need for multiple round trips to the EJB server. All calls are made locally by the command server.

There are also some disadvantages.

- The simplest implementation of command shipping uses Java serialization to generate the messages that flow between servers. This may hamper the use of a messaging infrastructure such as MQSI, since it is difficult to interpret a serialized bean at an intermediate point. However, command shipping does not dictate an encoding for requests. It is perfectly reasonable to provide a command target that encodes the command in XML, or even SOAP, for transport. Currently this will require per-command encoding logic, but this logic would be required on a per-request basis with any approach.

- The simplest implementation of command shipping uses the same class for both the server-side and the client-side implementation of the command. Thus, if the server-side implementation of the execute method needed to be changed, it would be necessary to redeploy the command class to all clients as well, for example, by including the command classes in a deployed EJB JAR file. Given the goal of agent-oriented service definitions, this does not seem like a serious issue. However, if this is a concern, then a simple dynamic delegation pattern can be followed where the execute method of a command is implemented by delegation to a dynamically linked (via classForName) helper class. In this way the helper class can be changed at any time with no impact on the client code.

**Command caching**

Command caching is beyond the scope of this book. However, it warrants a brief discussion to show future direction and to further justify the use of the command model.

Command caching extends the command model to allow executed commands to be saved in a cache and then retrieved when they are needed, thus avoiding the cost of re-executing the command. To do this, commands are extended with IDs and other metadata such as dependencies. The usage model for cacheable
commands is exactly like that for non-cacheable commands. However, when execute() is called on the command, the caching infrastructure checks to see if a command with exactly the same ID is already in the cache. If it is, then the contents of the cached command are copied into the newly executed command using the setOutputProperties() method added by TargetableCommand. Execute() then simply returns without really executing the command.

**Advantages and disadvantages of command caching**

The advantages of command caching are:

- Caching is transparent to application code.
- It is a true caching model. The application works correctly if items are not in the cache. Just as an application using a command does not know or care how the action is carried out, an application using a caching command does not know or care if the action is carried out. It interacts with the command in the same way, regardless of implementation.
- It provides a unified caching model. The model is the same for expensive computations, remote requests, database queries, etc.
- A consistent caching model helps to contain the complexity of invalidation.

The disadvantages of command caching are:

- It mixes logic and data solution by using data objects for output properties of commands.
- J2EE container services does not provide a command manager cache. It requires the application developer to implement invalidation logic. However:
  - Timeouts work very well for most non-user specific commands.
  - There is a special pattern for user-specific commands that keeps things quite simple.
  - A consistent, regular framework is better than ad hoc caching, which is the only real alternative.

**Command classes**

The complete command hierarchy is shown in Figure 6-8 on page 109. This shows the command interface as the base for all commands. Each command has to implement at least the Command interface. When using the base command interface, the command is executed locally in the same JVM as the calling servlet. An application that requires a command to be executed remotely (a shippable command) needs to implement the TargetableCommand and TargetableCommandImpl interfaces. Finally, if a command is to undo the work done by another command, then it must implement the CompensableCommand interface.
6.1.8 Best practices for EJBs

The following is a brief collection of best practices for developers of Enterprise JavaBeans:

- Use session beans to represent large-grained tasks in the business process.
  Session beans are used by one client and may or may not have associated properties.

See Chapter 9, “PDK sample overview” on page 253, for a sample implementation of the command framework.
- Use entity beans to represent fine-grained business domain elements.
  Entity beans are shared by many clients and typically have persistent data.
- Use JavaBeans as helper objects to get work done.
  Use JavaBeans as general-purpose utility elements, but avoid using helper objects in create() and finder methods in entity beans. This makes reuse of the bean challenging and deployment difficult.
- Wherever possible use stateless session beans.
  Stateless session beans can be pooled by the EJB container for efficiency, since they do not retain data between client invocations.
- Use session beans as facades to entity beans.
  This prevents a new transaction from being created on every method call, and eliminates the need to code business logic to back out a failed transaction.
- Use Container Managed Persistence (CMP) entity beans in most cases.
  Use Bean Managed Persistence (BMP) for situations requiring complex SQL or relational joins. When using BMP, always use WebSphere data source objects and connection pooling.
- Cache read-only objects in a Singleton JavaBean.
  Cache read-only objects or objects that do not change state on a per-JVM basis by storing them in a Singleton JavaBean. Access these objects using a stateless session bean.
- Use session beans for write-only objects.
  For write-only objects that do not need to be read into memory, such as a log entries, use session beans—or consider using asynchronous messaging.
- Cache the EJB home interface.
  Looking up the context, the remote home, and the remote interface is expensive, so cache the home interface to improve performance. Beware of stale handles.
- EJB local vs. remote interface.
  Only provide a local interface for entity EJBs. This ensures that fine-grained entity access is only available within the JVM. Provide remote coarse-grained entity access using a session bean as a facade.

See also the best practice recommendations in publication *EJB 2.0 Development with WebSphere Studio Application Developer*, SG24-6819, for further details.
6.2 Design guidelines for Web services

In this section we discuss the following Web services topics:

- Web services architecture
- Web services design considerations
- The key challenges in Web services
- Best practices for Web services
- Web services and Microsoft .NET

6.2.1 Web services architecture

Web services are deployed on the Web by service providers. The functions provided by the Web service are described using the Web Services Description Language (WSDL). Deployed services are published on the Web by service providers.

A service broker helps service providers and service requestors find each other. A service requestor uses the Universal Discovery Description and Integration (UDDI) API to ask the service broker about the services it needs. When the service broker returns the search results, the service requestor can use those results to bind to a particular service.

As we can see in Figure 6-9:

- Web service descriptions can be created and published by service providers.
- Web services can be categorized and searched by specific service brokers.
- Web services can be located and invoked by service requesters.

![Figure 6-9 Web services roles and operations](image)
Let us now look at the building blocks of Web services:

- **SOAP**
- **UDDI**
- **WSDL**

**SOAP**
SOAP is a network-, transport-, and programming language-neutral protocol that allows a client to call a remote service. The message format is XML. The currently adopted standard is W3C's SOAP 1.1 specification, while SOAP 1.2 is in the review process.

SOAP has the following characteristics:

- SOAP is designed to be simple and extensible.
- All SOAP messages are encoded using XML.
- SOAP is transport protocol independent. HTTP is one of the supported transports. Hence, SOAP can be run over an existing Internet infrastructure.
- There is no distributed garbage collection. Therefore, call by reference is not supported by SOAP; a SOAP client does not hold any stateful references to remote objects.
- SOAP is operating system independent and not tied to any programming language or component technology. It is object model neutral.

Due to these characteristics, it does not matter what technology is used to implement the client, as long as the client can issue XML messages. Similarly, the service can be implemented in any language, as long as it can process XML messages.

**WSDL**
The Web Services Description Language (WSDL) is an XML-based interface and implementation description language. WSDL 1.1 provides a notation to formally describe both the service invocation interface and the service location.

WSDL allows a service provider to specify the following characteristics of a Web service:

- The name of the Web service and addressing information
- The protocol and encoding style to be used when accessing the public operations of the Web service
- Type information, including operations, parameters, and data types comprising the interface of the Web service, plus a name for the interface

A WSDL specification uses XML syntax; therefore, there is an XML schema for it.
UDDI

UDDI stands for Universal Description Discovery and Integration. UDDI is both a client-side API and a SOAP-based server implementation that can be used to store and retrieve information on service providers and Web services.

UDDI is a *technical discovery* layer. It defines:

- The structure for a registry of service providers and services
- The API that can be used to access registries with this structure
- The organization and project defining this registry structure and its API

UDDI is a search engine for application clients rather than human beings; however, there is a browser interface for human users as well.

For detailed information on SOAP, UDDI and WSDL, refer to the publication *WebSphere Version 5 Web Services Handbook*, SG24-6891.

Next we look at the roles a business and its Web service-enabled applications can take. Three roles can be identified:

- Service broker
- Service provider
- Service requester

**Service broker**

The Web service broker is responsible for creating and publishing the UDDI registry. UDDI registries can be provided in two forms:

- Public registries, such as the IBM UDDI Business Registry and the IBM UDDI Business Test Registry:
- Private registries, such as the UDDI registry provided with IBM WebSphere Application Server Network Deployment V5.0.

The service broker does not have to be a public UDDI registry; there are other alternatives, for example a direct document exchange link between the service provider and the service requester.

**Service provider**

The service provider creates a Web service and publishes its interface and access information to the service registry.

Figure 6-10 on page 114 shows in more detail the application architecture of a Web service provider. Note that we have introduced the concept of a Web service provider bean. This bean can be a simple JavaBean, an EJB (stateless
session bean) or a simple Java class. This bean is a facade for the actual business logic that is present in the form of business objects. The Web service provider bean exposes methods that may mirror methods in the actual business objects, or the bean may expose methods that call a number of business objects. In any case, think of this as a facade bean for the real business objects. Using this architecture, we do not have to change the existing business logic or business objects in order to create a Web service from the existing enterprise business objects.

A WSDL specification consists of two parts, the service interface and the service implementation. Hence, service interface provider and service implementation provider are the two respective sub-roles for the service provider. The two roles can, but do not have to be taken by the same business.

**Service requester**

The service requester locates entries in the broker registry using various find operations and then binds to the service provider in order to invoke one of its Web services.

Figure 6-10 on page 115 shows the architecture of a Web service requester. Note that the architectural model follows the Model-View-Controller (MVC) pattern, with the servlet as the main component of the controller; the JSP the main component of the View; and the commands and the Web services residing the Model layer. Web services provide a link to another system within the Model layer.
This is the pattern that our PDK sample application follows and is considered a best practice for building Web-based applications. We can see that Web services fit very easily into this model. We will explore the Web service requester in more detail later in our sample Web services implementation in Chapter 10, “Web services scenario” on page 281.

6.2.2 Web services design considerations

This section describes architectural decisions that need to be made when designing Web service providers and requesters. We will describe each of the decisions that need to be made and then talk about some of the technical issues involved in these decisions and give some guidelines for system architects on how to make the appropriate choices for a given application.

Transmission patterns

The first design option we should look at for designing Web services is the Transmission pattern we expect to use. These patterns represent different types of operations that can be defined in a WSDL file. The four basic patterns are:

- Request-response
- One-way
- Solicit-response
- Notification operation
**Request-response**
The request-response transmission primitive is shown in Figure 6-12.

![Request-response transmission](image)

**Note:** A request-response operation is an abstract notion. A particular binding must be consulted to determine how the messages are actually sent: Within a single communication (such as a HTTP request/response), or as two independent communications (such as two HTTP requests).

**One-way**
The one-way transmission primitive is shown in Figure 6-13.

![One-way transmission](image)

**Solicit-response**
The solicit-response transmission primitive is shown in Figure 6-14 on page 117.
**Notification operation**

The notification operation transmission primitive is shown in Figure 6-15.

**SOAP messaging mechanisms**

The next design point in architecting a Web service is to choose the SOAP messaging mechanism to use. Figure 6-16 on page 118 shows the two general categories of Web services SOAP messaging mechanisms:

- SOAP RPC-based Web services
- SOAP message-oriented Web services
RPC vs. message-oriented

The advantages and disadvantages of the SOAP RPC approach versus the SOAP message-oriented Web service approach can be summarized as follows:

- **SOAP RPC advantages:**
  - Simpler development

- **SOAP RPC disadvantages:**
  - Requester is too dependent on the availability of the Web service provider

- **SOAP message-oriented advantages:**
  - Less dependency on the Web service provider availability
  - Works well for exchanging large documents
  - Works well from a non-repudiation perspective since documents can be digitally signed and stored at both ends
  - Enables extended enterprise electronic workflow and business process integration using asynchronous integration

- **SOAP message-oriented disadvantages:**
  - Relatively more complex development since it uses assured delivery of asynchronous messages and may require compensating transactions
Static vs. dynamic Web services discovery

Our next design point is to decide if the Web service requester will use static or dynamic discovery of available Web services. The requester has to begin with the WSDL file that describes the interface and implementation specification of the Web service to be invoked. This WSDL file can be retrieved dynamically using a service registry, or statically, as shown in Figure 6-17.

Three types of discovery methods for requesters can be identified. They import interface and implementation information at different points in time (build time vs. runtime):

- **Static service**
  
  No public, private, or shared UDDI registry is involved. The service requester obtains a service interface and implementation description through a proprietary channel from the service provider (an e-mail, for example), and stores it in a local configuration file.

- **Provider-dynamic**
  
  The service requester obtains the service interface specification from a public, private, or shared UDDI registry at build time and generates proxy code for it. The service implementation document identifying the service provider is dynamically discovered at runtime (using the same or another UDDI registry).
Type-dynamic

The service requester obtains both the service interface specification and the service implementation information from a public, private, or shared UDDI registry at runtime. No proxy code is generated; the service requester directly uses the more generic SOAP APIs to bind to the service provider and invoke the Web service.

Message structure

The Web services specification does not mandate any particular message structure. The message structure is defined by the service provider. Message structures can be anything from simple strings to complex XML documents.

SOAP Encoding vs. literal encoding

SOAP Encoding, the infamous set of rules often referred to as "Section 5 encoding" after its location in the specification, was introduced to provide standard rules for encoding data within SOAP envelopes. Simple services and clients could simply agree to follow a set of rules for encoding data to XML in order to make writing services and clients easier. But SOAP Encoding is only a suggestion in the specification. Thus, when a product claims SOAP compatibility, it is not explicitly claiming SOAP Encoding compatibility. This is why the available higher-level APIs cannot be correct in general when they do automatic marshalling of datatypes. Just as the extensibility of SOAP with regard to transports often causes implicit assumptions that create compatibility problems, so does this extensibility with regard to payload data encoding. To know whether the client API will generate SOAP envelopes that a specific Web service will understand, we must be explicitly aware of the data encoding that the Web service expects and whether that encoding is supported by the client API.

An alternative is to use literal encoding where the payload of a SOAP message is defined completely by a specific schema, often an XML Schema. Instead of having the Web service and the client agree to follow a set of rules for serializing the data, they agree on the exact format of the data. If the format is described by using an XML Schema, a development tool can read it and provide automatic marshalling of the data from the native language structures into XML. In this case, all the toolkit has to understand is the entire XML Schema specification instead of the combination of the particular encoding rules as well as the chosen type system. The only issue left with using literal encoding is how the tool finds the particular service's XML Schema. This issue is solved by WSDL.
Currently, no machine-understandable standard exists for describing data models for use with SOAP Section 5 encoding, so developers are leaning towards literal encoding with XML Schema. Development tools often provide features such as syntax assistance and data model validation with XML Schema. This may change soon, however, as the Web Services Description working group is considering creating a language to describe SOAP Section 5 data models as part of the WSDL binding for SOAP in WSDL version 1.2.

**Synchronous vs. asynchronous Web services**

Our next design point is selecting the kind of messaging we wish to implement. Our choices are synchronous or asynchronous. The Web services specifications define synchronous operations only. However, Web services are by their very nature somewhat asynchronous. From the perspective of the Web service provider it must, in effect, be a listener and be prepared to accept requests asynchronously from the requester. From the consumer or requester side, the application can be designed for either synchronous or asynchronous operation. Although Web services defines synchronous operations only, there is nothing in the specifications to preclude asynchronous operations. Generally, the Web services requester has no guarantee of when, or if, it will receive a response. Beyond that, there are also situations where the Web service provider needs to perform some external operation, or wait for human intervention, or call another service that would introduce a delay in the response.

Synchronous types of Web services are suitable when the Web service provider can provide the required response instantaneously (such as when getting a stock quote). Here we are in effect using Web services as another RPC mechanism. Current tools are more focused on this type of Web service. Asynchronous types of Web services are suitable when the Web service provider is unable to provide the required response instantaneously, for a variety of reasons as mentioned above. Asynchronous operations are usually driven by the asynchronous nature of the business transaction itself. Asynchronous Web services are suitable for document exchange between enterprises. It is important to separate this design point from the reliability of the underlying transport mechanism. We will discuss this in further detail in the next sections.

The designer of a Web services requester needs to decide how to handle asynchronous responses and how to ensure that his or her implementation is compatible with the way in which a service provider supports asynchronous operations. One option for the requester is to issue a request and then block its thread of execution waiting for a response, but for obvious reasons this is not a good alternative; among other problems, it results in resource inefficiencies and raises transactional and scalability issues. The preferred solution is to build asynchronous behavior into the Web services requester. The requester makes a request as part of one transaction and carries on with the thread of execution. The response message is then handled by a different thread within a separate
transaction. In this model, the requester requires a notification mechanism and a registered listener component to receive responses. Likewise, there must be a correlator (a correlation or transaction ID) exchanged between the service requester and the service provider for associating responses with their requests.

**Transports and local interfaces**
The transports that can be used for Web services communications vary in their capabilities to facilitate the support of asynchronous operations. Thus, it is not only Web services behavior that can be described as either asynchronous or synchronous; the transport used for exchanging Web services messages also falls into one category or the other. Transports whose interfaces inherently support the correlation of response messages to request messages for application use and support a push and pull type of message exchange are often described as being asynchronous transports. Synchronous transports do not provide these facilities and, when used for asynchronous operations, require that the applications (the client and service provider, for the purposes of this discussion) manage the correlation of messages exchanged by not only defining how the correlator will be passed within each message, but by also matching responses with requests. Examples of transports that can be used in support of asynchronous operations are listed in Table 6-1.

<table>
<thead>
<tr>
<th>Asynchronous transports</th>
<th>Synchronous transports</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTPR</td>
<td>HTTP</td>
</tr>
<tr>
<td>JMS</td>
<td>HTTPS</td>
</tr>
<tr>
<td>IBM WebSphere MQ Messaging</td>
<td>RMI/IIOP</td>
</tr>
<tr>
<td>MS Messaging</td>
<td>SMTP</td>
</tr>
</tbody>
</table>

Typically, when business partners are utilizing Web services to integrate their business processes, they will prefer to use HTTP, HTTPS, and HTTPR as transports for communications across the Internet.

**Correlation ID**
Regardless of the transport being used for an asynchronous operation, since the response to a request is expected to be received at a later time, there must be a mechanism to correlate the response with the request. Web services requesters and providers must agree upon a correlation ID scheme. They also must agree upon who is responsible for generating the correlation ID.

**Return address**
In addition there must be an agreed-upon mechanism to identify the return address to send the response to. One could set up a return address in a profile database or its return address could be part of every request.
The asynchronous transports enable a client to continue processing on its thread of execution immediately after requesting a service invocation; they also provide mechanisms to enable a client to determine the status of its Web service requests, and to retrieve responses to those requests.

Web service implementations that do not provide the ability to initiate the transmission of a response on a separate thread of execution cannot be used for asynchronous operations. Examples of such implementations would be those that use EJBs to front-end database applications or implementations that provide access to enterprise systems through the use of local interfaces such as JCA.

**Asynchronous Web services approaches**

When implementing an asynchronous mechanism in Web Services, the preferred solution is to build asynchronous behavior into the Web services requester. The requester makes a request as part of one transaction and carries on with the thread of execution. The response message is then handled by a different thread within a separate transaction. In this model, the requester requires a notification mechanism and a registered listener component to receive responses. Likewise, there must be a correlator (a correlation or transaction ID) exchanged between the service requester and the service provider for associating responses with their requests.

A typically asynchronous scenario would include the following:

- Production and transmission of a request message by a service requester
- Consumption of the request message by the service provider
- Production and transmission of a response message by the service provider
- Consumption of the response message by the service requester

We examine the two approaches for asynchronous Web services:

- Decoupled publication-subscription
- Polling

**Decoupled publication-subscription**

This approach requires partners A and B to be both Web service provider and consumer. They alternate roles. This means both need the SOAP server footprint. Figure 6-18 on page 124 illustrates the steps that need to be taken to implement publication-subscription asynchronous Web services messaging.
Polling

Here the consumer polls on a periodic or event basis to retrieve the response for an earlier request. Partner A remains as a pure Web service consumer and does not need the SOAP server footprint. Figure 6-19 illustrates how to implement a polling approach to asynchronous Web services.

Development strategies for Web service provider

A service provider can choose between three different development styles when defining the WSDL and the Java implementation for his/her Web service:

- Top down
  
  When following the top-down approach, both the server-side and client-side Java code are developed from an existing WSDL specification.
Bottom up
If some server-side Java code already exists, the WSDL specification can be generated from it. The client-side Java proxy is still generated from this WSDL document.

Meet in the middle
The meet in the middle (MIM) development style is a combination of the two previous ones. There are two variants:

- MIM variant 1
  Some server-side Java code is already there. Its interface, however, is not fully suitable to be exposed as a Web service. For example, the method signatures might contain unsupported data types. A Java wrapper is developed and used as input to the WSDL generation tools in use.

- MIM variant 2
  There is an existing WSDL specification for the problem domain; however, its operations, parameters, and data types do not fully match with the envisioned solution architecture. The WSDL is adopted before server-side Java is generated from it.

In the near future, we expect most real-world projects to follow the meet in the middle approach, with a strong emphasis on its bottom-up elements. This is MIM variant 1, starting from and modifying existing server-side Java and generating WSDL from it.

Level of integration between requester and provider
In a homogeneous environment, client and server (requester and provider) use the same implementation technology, possibly from the same vendor. They might even run in the same network.

In such an environment, runtime optimizations such as performance and security improvements are possible. We expect such additional vendor-specific features to become available as the Web services technology evolves.

We do not recommend enabling such features, however, because some of the main advantages of the Web service technology such as openness, language independence, and flexibility can no longer be exploited. Rather, you should design your solution to loosely couple requester and provider, allowing heterogeneous systems to communicate with each other.

6.2.3 The key challenges in Web services
Web services can potentially revolutionize application integration by providing a layer of abstraction between the technology that requests a service and the
technology that provides the service. In order to achieve this, though, there are still technical challenges that have to be addressed. This section briefly describes a few key issues, such as the Extended Web Services Architecture, security, interoperability, quality of service, and distributed transactions.

Extended Web Services Architecture

In November 2002, W3C released a draft Web Service Architecture specification that identifies the functional components, the relationships among those components, and establishes a set of constraints to guide the desired properties of the overall architecture.

The proposed architecture consists of a basic architecture that defines the interactions between service requesters and service providers, as discussed in 6.2.1, “Web services architecture” on page 111.

The proposed architecture also defines an Extended Web Services Architecture that incorporates additional features and functionality. These additional features include:

- Asynchronous messaging
- Attachments typically used to include binary data in SOAP messages
- Caching
- Message exchange pattern (MEP)
- Correlation
- Long running transactions
- Reliable messages
- Message authentication
- Message confidentiality
- Message integrity
- Message routing
- Management messages
- Session

See the W3C Web Services Architecture specification for more detail:

http://www.w3.org/TR/2002/WD-ws-arch-20021114/

Security

Security concerns are the main limitation of current Web services initiatives. The Internet and many of its prevalent technologies were not designed with security in mind. Web services security must also be compatible with the foundational technologies (SOAP, WSDL, XML Digital Signature, XML Encryption, and SSL/TLS).

We will further discuss the latest development in Web services security in 8.1.3, “Web service security specifications” on page 210.
Interoperability

By utilizing open standards, Web services can, in theory, enable any two software components to communicate—no matter what technologies or platforms are used to create or deploy them. Interoperability is one of the key value propositions of Web services. Unfortunately, there is still no common, agreed-upon definition of what a Web service is, and there are still many needed standards in their infancy and some are still competing against each other. Such fragmentation and “niching” of Web services standards, tools, and APIs could really jeopardize the applicability and thus the wide adoption of Web services.

To address the potential problems, the Web Services Interoperability (WS-I) Organization released the Web Services Basic Profile 1.0 on October 17, 2002. It is an important milestone for the technology as a published description of what standards and technologies will be required for interoperability between Web services implementations on different software and operating system platforms.

As an architect or a developer, you need to monitor WS-I’s progress and participate if you can. In addition, try to follow the use cases and scenarios introduced in the current draft documentation as design guidance.

See the WS-I Basic Profile Version 1.0 specification for more detail:
http://www.ws-i.org/Profiles/Basic/2002-10/BasicProfile-1.0-WGD.htm

Quality of Service (Qos)

Quality of Service is one of the top issues in the minds of those considering Web services. WSDL specifies the syntactic signature for a service but does not specify any semantics or non-functional aspects. QoS-enabled Web services require a separate QoS language for Web services to answer questions such as:

▶ What is the expected latency?
▶ What is the acceptable round-trip time?

A programmer needs to understand the QoS characteristics of Web services while developing applications that invoke Web services.

Ideally, a QoS-enabled Web services platform should be capable of supporting a multitude of different types of applications:

▶ With different QoS requirements
▶ Using different types of communication and computing resources

When considering QoS-aware Web services, the interface specification should provide QoS statements that can be associated to the whole interface or individual operations and attributes. In the case of a service requestor, these statements describe the required QoS associated with the service required by
the client. From a service provider’s perspective, these statements describe the offered QoS associated with the service offered by the server object.

See the IBM developerWorks article *Understanding quality of service for Web services*:  

**Distributed transactions**
Almost every party agrees that we need a standard that accommodates both classical ACID (XA or database-style transactions) and long-running, compensating transactions. But there is still sharply divided opinion on where such standards fit in the Web services stack.

The Business Transaction Protocol (BTP) from OASIS was backed by a number of smaller vendors (BEA, HP, Choreology, Oracle) and the Version 1.0 was released in May 2002. BTP tries to adopt XML-based technology for business transactions on the Internet and tackles such challenges as transactions that span multiple enterprises and long lasting transactions.

BTP has been criticized as being too complex, and still lacks backing from an industry heavyweight (like IBM or Microsoft). In August 2002, IBM, Microsoft, and BEA published two drafted specifications:

- **WS-Coordination** is a general purpose and extensible framework for providing protocols that coordinate the actions of distributed transactions. The defined framework enables an application service to create a context needed to propagate an activity to other services and to register for coordination protocols. The framework also enables existing transaction processing, workflow, and other systems for coordination to hide their proprietary protocols and to operate in a heterogeneous environment. It can be used with message sequencing and state machine synchronization.


- **WS-Transaction** includes support for the two types of transactions. It describes coordination types that are used with the extensible coordination framework as described in WS-Coordination. Two coordination types are defined: Atomic Transaction (AT) and Business Activity (BA). WS-Transaction is a building block used with other specifications (for example, WS-Coordination, WS-Security) and application-specific protocols that are able to accommodate a wide variety of coordination protocols related to the coordination actions of distributed applications.

While these proposals and specifications are still evolving, it is recommended that we, as architects and developers, actively participate in, review, comment, and help improve the specifications. Also evaluate early implementations for inclusion in corporate architecture standards and possible application implementation. If there is urgent need for designing and implementing a Web services-based transactional infrastructure and related business services, we recommend using the principles behind the new specifications.

6.2.4 Best practices for Web services

Web services constitute a distributed computer architecture made up of different computers communicating over a network to form one system. At the current time there are two competing application paradigms being put forward. One is the J2EE by Sun Microsystems and adopted by IBM and many other companies, and the other is the .NET by Microsoft. For example, in the J2EE paradigm, Sun Microsystems has a model called the Sun Open Net Environment (Sun ONE), an open framework that supports "smart" Web services where the J2EE framework plays a fundamental role. In fact, using Web services, either .NET or J2EE Web services can be accessed by a Web services requester. However, there are still issues and problems when communicating between J2EE-based and .NET-based applications. Thus, the first best practice that we suggest is to avoid mixing these application paradigms if at all possible. Having said that, keep in mind that many .NET services are being successfully accessed by J2EE-based applications, and vice versa.

In this section, we focus on best practices for Web services development and deployment within a J2EE environment, that is, Web services that are built using servlets, JSP pages, EJB architecture, and all the other standards that are part of the J2EE technology.

Apply distributed computing principles

Think of Web services as another technology for developing distributed systems. All of the best-practice principles used in developing distributed systems apply to Web services. All of the considerations that would go into any enterprise systems design apply to Web services, such as high availability, high throughput, clustering, hardware management, and network topology.

The main difference between most distributed systems and Web services is that Web services are newer. Most Web services software is less than a year old. As a rule, there is not the same level of reliability, security, or performance that you would find with other distributed systems software that have been around longer. Another factor is that Web services are built on a set of technologies (SOAP, XML, WSDL, UDDI) that are still evolving and are being evolved by separate standards organizations and vendors in parallel. It will be some time before all of
these standards will be able to converge (especially given the Sun versus Microsoft debate). Because of the lack of a solid set of standards, implementation details are left for individuals. Still, some common principles can be adopted as best practices at this time:

- **Design systems that are layered.**

  This is the same principle that you would apply to any distributed, component architecture. It is especially important in Web services applications where we do not have control over some components (services) that we access in our application.

- **Design coarse-grained Web services.**

  Web services have all of the same issues as those of distributed systems when it comes to requesting a remote service. Requesting a service from a machine over the network is more expensive than a local operation. With this in mind, keep the request as coarse grained as possible when requesting a Web service from a remote machine.

  Existing JavaBeans or EJBs with fine-grained methods or operations should be aggregated into a single coarse-grained Web service, wherever possible. This technique avoids unnecessary network traffic and overhead on the communication stack. This also makes it possible to push the transaction integrity requirements to the Web service provider making for a cleaner design. In other words, if a coarse-grained request did not successfully complete, then the Web service provider can roll back that entire transaction.

- **Design for “loosely coupled” components.**

  A Web service by definition is an interface to a loosely coupled component on a remote system. Therefore, it is very important to be cognizant of the impact of integrating loosely coupled components. With this in mind, define clear contracts between layers and services, but utilize the “Parameter List” paradigm where possible.

- **Limit dependency on other components.**

  Managing dependencies is one of the key challenges in utilizing Web services in an intranet or extranet scenario. Common dependencies that occur in an application design are:

  - **Call flow dependency**

    Business processes implemented by systems are not typically within the domain of one business component.
Object association dependency

Using object-oriented techniques, it is easy to model a business problem by associating objects together. However, from an implementation perspective, doing so increases the linkage from one component to another. Use interfaces where possible.

Implement all cross “domain” business processes in a “control” or “workflow” layer.

The flexibility of an application is increased if all business processes that cross multiple business domains are implemented in a workflow layer. In doing so, the application architecture has more flexibility in what is called, when it is called, managing the call (such as exception handling), and performing any translation on the data that is passed in or out.

Utilize standard XML structures to pass data

One of the biggest challenges to effectively using Web services is the need to not only pass data, but also the meaning of data between services.

The standard currency for Web services in this case is an XML document. Of course, simply using XML is not enough. In order for a Web service to understand the elements (or metadata) of a document, the structure and meaning must be standardized:

- Design components to store metadata as well as data. Understanding the meaning and value of the information passed is critical to implementing any distributed system.

- Manage data within the Web service component responsible for the data. By exposing a business process as a Web service that is available to users within and outside an organization, it is best to make that service as transparent as possible. Web service requesters should not have detailed knowledge of database structures or data formats.

- When designing XML schema to define business-related, more advanced data types, try not to use complex XML schema constructs (such as choice), and decompose them into simple and clean interfaces with primitive data types. Also avoid using specific techniques (for example, INOUT parameter passing) that are not widely supported.

- Whenever possible, provide complete XML Schema and WSDL definitions in one WSDL file rather than importing them from various locations.

Use existing Web services tools

This begins with utilizing standards-based tools for service lookup. While UDDI promises to provide the capability for dynamic lookup of services to call, the reality may be closer to a dynamic binding capability to a known service.
Use an IDE that supports Web services, such as IBM WebSphere Studio Application Developer. This allows you to expose assets and services using WSDL and proxy-generation tools, which shield you from the underlying XML messages in Web services.

Web services tools will eventually make XML transparent, but for now it is essential that you become comfortable with XML standards based on SOAP, WSDL, UDDI, or ebXML.

The future will bring tool suites that will incorporate standard API for creating GUIs, tools for generated Web services-enabled JSP pages and servlets, and collections of XML-based Java APIs for XML processing registries, messaging, binding, and remote procedure calls (RPC). Also look for application servers such as IBM WebSphere, to have the ability to automatically publish an EJB as a Web service in future releases.

**Use Web scalability principles**

Many of the same principles that apply to any Web application can be applied to Web services when it comes to scalability. For example:

- Maintain state information on the Web service provider side.
- Use a session ID to access persistent data.
- Use existing Web techniques for increasing throughput:
  - Acceleration hardware
  - Load balancing
- Use caching wherever possible:
  - Despite being dynamic, caching of responses for certain transactions may be feasible.
  - Caching works well for non-user specific data, especially if validation is simple.
  - User-specific data can be cached if combined with server affinity.

**XML performance**

Concerns about Web services performance being impacted by transmitting and parsing XML have been overstated. The time to parse XML is usually negligible and can be dependent on an XML parser library. Still, in order to minimize the effect on performance of XML parsing, some steps can be taken:

- Use SOAP implementations that allow for pluggable XML parsers to leverage the latest advances in parser technology.
- Avoid chaining services if possible, since this will increase path lengths.
- Design for coarse-grained, document-based interactions.
Balance architecting service reuse with number of invocations per transaction.

**Interoperability**
Consider the following interoperability principles:
- Web services should be used when you need interoperability across heterogeneous platforms. That is, when you need to expose all or part of your application to other applications on different platforms.
- There will be some issues with vendor compliance, albeit less than CORBA. Such issues usually involve .NET and Java camps. Beware of vendor-specific extensions.
- Use Apache as your “benchmark” for SOAP compliance.
- Use low-level XML APIs to adjust SOAP requests to overcome glitches:
  - MS SOAP API
  - JDOM or other Java XML API
- Build sample stubs that use WSDL for all supported platforms to share with partners.

**Maintainability**
Consider the following maintainability principles:
- Standardize on SOAP rather than roll your own XML message structures.
- Utilize the flexibility of XML to release updates to a Web service.
- Employ a version/release number on each request in the SOAP header so that service can be routed for backwards compatibility.

**Reuseability**
Consider the following reuseability principles:
- Leverage reuse of services by establishing an enterprise-wide service-oriented architecture.
- Reduce cost/complexity of maintaining multiple distributed infrastructures:
  - Phase out CORBA/others with Web services where applicable.
  - Expose coarse-grained services for application integration. This approach helps to promote reuse of components.
  - Use CORBA for lightweight, tightly coupled, fine-grained access to applications if necessary.
- Limit the use of complex types in distributed environments.
Reliability
There are no Quality of Service agreements in Web services or guaranteed
delivery. With Web services you will be accessing services that are not under
your control. Therefore, plan for the cases where the service you are requesting
fails.

Web services support both asynchronous and synchronous RPC-style
architectures as well as messaging. You can use a loosely coupled,
asynchronous architecture that allows functions to continue without network
response. However, you cannot count on the reliability of many of the services
you call because they are not under your control. You will need to provide backup
plans for important processes.

6.2.5 Web services and Microsoft .NET

Using IBM WebSphere Studio Application Developer, it is possible to create and
run a Web service and to invoke the Web service from client applications created
using the Microsoft .NET Framework SDK. Using the Web Service wizard within
Application Developer, you can generate WSDL proxies for consumption by
these Microsoft .NET clients in both the C# and JScript programming languages,
which are both currently supported by the Microsoft .NET Framework. It is even
possible to combine proxies created in both languages into single executables.

Microsoft .NET generates, by default, document-style Web service, using literal
encoding. Although the Web service uses document/literal, the .NET framework
makes the service appear to be RPC style to .NET clients. This is not necessarily
so for other Web service frameworks, and many users may find it difficult to
access the Web service using an RPC style client. If you are writing an RPC Web
service, force your framework to generate the RPC style WSDL.

For details on how to create these integrated applications, see the WebSphere
Developer Domain article Developing Microsoft .NET Web Service Clients for
EJB Web Services with IBM WebSphere Studio Application Developer and the
Microsoft .NET Framework SDK:

en&t=851,t=gr

Note: C#, pronounced “C sharp,” is a Microsoft programming language similar
to Java and C/C++.
6.3 Design guidelines for J2EE Connectors

The following section outlines application and system design guidelines for application integration using the J2EE Connector Architecture. We discuss the Common Connector Interface (CCI) API, the CICS resource adapters, and the IBM WebSphere Application Server V5.0 J2EE Connector environment.

6.3.1 Components of J2EE Connector Architecture

As shown in Figure 6-20 on page 136, Version 1.0 of the J2EE Connector Architecture defines a number of components and interfaces that make up this architecture:

- **Common Client Interface (CCI)**
  
The CCI defines a common API for interacting with resource adapters. It is independent of a specific EIS. A Java developer communicates to the resource adapter using this API.

- **System contracts**
  
  A set of system-level contracts between an application server and EIS. These extend the application server to provide:

  - Connection management
  - Transaction management
  - Security management

  These system contracts are transparent to the application developer, that is, they do not implement these services themselves.

- **Resource adapter deployment and packaging**

  A resource adapter provider develops a set of Java interfaces/classes as part of its implementation of a resource adapter. The Java interfaces/classes are packaged together with a deployment descriptor to create a Resource Adapter Archive (represented by a file with an extension of .rar). This Resource Adapter Module is used to deploy the resource adapter into the application server.
6.3.2 Managed and non-managed environments

There are two different types of environments that a Java application using J2EE Connectors can run in:

- Managed environment
  
The Java application accesses a resource adapter through an application server such as WebSphere Application Server. Management of connections, transactions, and security is provided by this application server. The Java application developer does not have to code this management manually.

- Non-managed environment
  
In a non-managed environment, you do not have to use an application server. Instead, the Java application directly uses the resource adapter to access an EIS. In this case management of connections, transactions, and security must be handled manually by the Java application. You can find further details in the WebSphere Developer Domain article Using J2EE Resource Adapters in a Non-managed Environment at:

6.3.3 Common Connector Interface

The Common Client Interface (CCI) defines a standard client API so that application components can access multiple resource adapters, as shown in Figure 6-21. This API can be used directly, or enterprise application integration frameworks can be used to generate EIS access code for the developer. The CCI is designed to be an EIS independent API, so that an enterprise application development tool can produce code for any J2EE-compliant resource adapter that implements the CCI interface. Such tools include the Enterprise Service toolkit of IBM WebSphere Studio Application Developer Integration Edition, and VisualAge for Java Enterprise Access Builder.

![Figure 6-21 CCI with multiple resource adapters](image)

The CCI has the following characteristics:

- It is independent of a specific EIS. It forms a base-level API for EIS access on which higher-level functionality, specific to an EIS, can be built.
- It provides an API that is consistent with other APIs in the J2EE platform, such as JDBC.
- It is targeted primarily towards application development tools and enterprise application integration frameworks, rather than Java developers using the CCI API directly.

One goal of the CCI is to complement, rather than replace, the JDBC API. The CCI programming model and JDBC programming model are aligned, but the APIs serve the following different purposes:

- The JDBC API is used to access relational databases.
- The CCI API is used to access an EIS (which is not a database).
6.3.4 The CCI classes

The CCI interface can be divided into the following parts:

- Connection-related interfaces
- Interaction-related interfaces
- Data representation-related interfaces
- Metadata-related interfaces
- Additional classes

A resource adapter provides an implementation of the CCI interfaces.

Connection interfaces

The connection interface provides an interface for the connection with an EIS application. The following classes are available:

- ConnectionFactory is the interface for getting a connection to the EIS.
- Connection is the application level connection handle that is used by a component to access the EIS.
- ConnectionSpec is for passing connection-specific properties for the Connection.
- LocalTransaction enables a component to demarcate resource manager local transactions.
Interaction interfaces
An interaction instance supports the following interaction with an EIS system:

- Interaction has an execute method, which executes an EIS function and gives the result back in an output record or as return value.
- InteractionSpec holds the properties for the interaction with the EIS system.

Data representation interfaces
Data representation interfaces are used to represent the data that is involved in an interaction with an EIS system.

- Record is a representation of the input or output record.
- MappedRecord is a key-value pair-based collection that represents a record.
- IndexedRecord is an ordered and indexed collection that represents a record.
- RecordFactory is the factory used for creating mapped or indexed records.
- Streamable enables an adapter to set input or get output data as a stream of bytes.
- ResultSet is a representation of tabular data containing retrieve and update methods.
- ResultSetMetaData provides meta information about the columns in the ResultSet.

Metadata related interfaces
The metadata related interfaces provide you meta information about a resource adapter implementation and an EIS connection.

- ConnectionMetaData provides information about an EIS instance connected through a connection instance.
- ResourceAdapterMetaData provides information about the capabilities of a resource adapter implementation.
- ResultSetInfo provides information over the support of the ResultSet interface of the resource adapter.

Additional classes
The additional classes consist of exception interfaces.

- ResourceException extends java.lang.Exception. It consists of a string describing the error, a specific error code, and a reference to another exception.
- ResourceWarning provides information on the warnings related to the interactions with the EIS system.
6.3.5 **System contracts**

To achieve ease of interaction between the application server and EIS, the J2EE Connector Architecture defines a set of system contracts. There are system contracts for transaction, security, and connection management. The application server uses a resource adapter to support these contracts. The resource adapter implements the system contracts to collaborate with the application server and uses an EIS-specific API to communicate with the EIS. Thus, a resource adapter is specific to an EIS. However, because it implements the system contracts, it can be plugged into any J2EE-compliant application server.

**Connection management**

The connection management contract gives an application component a connection to an EIS. To deliver performance and scalability, the connection management contract should support connection pooling and management.

When retrieving data from an EIS, a large portion of the time (from making the connection to receiving the data and closing the connection) is in the creation of the connection itself. Connection pooling alleviates this bottleneck. When you call for a connection, you are passed a handle to the next available connection that is in a ready-to-use state. This increases performance greatly by removing the actual connection time, and scalability is handled by predefining as many connections in the pool as you need.

Connection pooling is a *quality of service* offered by the application server. An application server uses the connection management contract to implement a connection pooling mechanism in its own implementation-specific way.

The J2EE Connector Architecture Specification V1.0 explains in detail how the connection contract is implemented by various connection management components of the application server and the resource adapter.

**Transaction management**

Before discussing transaction management, the following terms should be defined:

- **Resource manager**
  The resource adapter and underlying EIS. It may participate in transactions that are externally controlled by a transaction manager.

- **Transaction manager**
  Controls and coordinates transactions across multiple resource managers.
A resource manager has three options for supporting transactions:

- **No support**
  The resource manager does not support transactions.

- **Local transactions**
  These are transactions that are managed internally by the resource manager. The coordination of such transactions involves no external transaction manager.

- **Global transactions**
  There are multiple resources managers involved, and an external transaction manager must be used to coordinate the transaction using two-phase commit.

### Local transactions
These are transactions that are managed internally by the resource manager. The coordination of such transactions involves no external transaction manager and can be utilized when only one resource manager is involved. Local transactions only support one-phase commit, because they only reference one EIS.

To support local transactions, the resource adapter must implement the `javax.resource.spi.LocalTransaction` interface. This instance represents a physical connection to an EIS and acts as a factory of connection handlers. Next to this it allows the application server to hear and react to the following local transaction events:

- **begin**
- **commit**
- **rollback**

By listening for these events, the application server can perform various functions such as local transaction cleanup. For example, the CICS ECI resource adapter provides a local transaction interface called `CCILocalTransaction`, which allows application components to manipulate a CICS transaction. An instance of `LocalTransaction` can be obtained from `getLocalTransaction` on a `Connection` instance. Example 6-1 shows two enterprise applications being invoked from one unit of work.

```
Example 6-1  Local transaction example

// Start of a unit of work
conn.getLocalTransaction().begin();
...
ispec1.setFunctionName("CALCRATE")
execute(ispec1,record1,record1);
```
... ispec2.setFunctionName("CALCRAT2")
execute(ispec2,record2,record2);
...
// End of a unit of work
conn.getLocalTransaction().commit();

You can roll back the entire transaction if you issue a rollback to the LocalTransaction. Example 6-2 describes how a rollback is performed when a transaction abnormal end occurs at the enterprise application.

Example 6-2  Local transaction exception handling example

```java
try {
    ...
} catch (com.ibm.connector2.cics.CICSTxnAbendException re) {
    // CICS Transaction abend occurred. The transaction will be rolled back.
    conn.getLocalTransaction().rollback();
}
```

Global transactions
There are multiple resources managers involved, and an external transaction manager must be used to coordinate the transaction using two-phase commit.

Global transactions are also referred to in the J2EE Connector Architecture Specification as JTA transactions, and they are supported by the resource adapter implementing the javax.transaction.xa.XAResource interface.

In a managed environment, the application server uses a transaction manager to coordinate the transaction. The application server will provide the following functions:

- Inform the transaction manager when a transaction begins.
- Perform the work of the transaction.
- Tell the transaction manager to commit the transaction.

The transaction manager uses the XAResource interface of the resource adapters to coordinate the two-phase commit process across multiple resource managers.

In a managed environment, there are two options to decide where to control the scope and the behavior of the transaction, as follows:

- Component managed transaction demarcation.
  The transaction is explicitly managed by the application component accessing the enterprise tier.
- Container managed transaction demarcation.
  
  The transaction is implicitly managed by the container of an enterprise bean accessing the enterprise tier.

In a non-managed environment, the Java application is responsible for managing transactions, through the local transaction interface (providing that the resource adapter supports this). By using the managed environment, the programmer does not even need to think about managing the transaction, because the transaction manager is a provided quality of service.

**Local transaction optimization**

Local transaction optimization forces the use of one-phase commit in the situation when two-phase commit is not needed for a global transaction. This is when only one resource manager was referenced, so two-phase commit is an unnecessary overhead.

**Security management**

The J2EE Connector Architecture security contract extends the J2EE security model to provide secure connections to EIS. To create a connection to an EIS, there must be some form of signing on to the EIS, to authenticate the connection requester. Re-authentication can also take place if supported by the EIS. This occurs when the security context is changed after a connection is made. (For example, connection pooling could cause a re-authentication when the connection is redistributed.)

Performing the sign-on generally involves one or more of the following steps:

1. Determine the resource principal under whose security context the connection will be made.
2. Authenticate the resource principal.
3. Establish secure communications.
4. Determine authorization (access control).

The application component requests that a connection be established under the security context of a resource principal. For example, a CICS ECI application can specify the user name and password to the resource adapter that signs on to CICS to invoke the CICS program. Once a connection is established successfully, all application-level invocations to the EIS instance using the
connection happen under the security context of the resource principal. The application component has the following two choices related to EIS sign-on:

- Container-managed sign-on
  
The deployer sets up the resource principal and EIS sign-on information. For example, the deployer sets the user name and password for establishing a connection to an EIS instance.

- Component-managed sign-on
  
  Application code in the component performs the sign-on to an EIS by explicitly specifying the security information for a resource principal.

If you choose component-managed sign-on, you need to specify a user name and password at an instance of ConnectionSpec. Example 11-3 on page 341 describes the way to use ConnectionSpec when used with the CICS ECI resource adapter.

More information on CICS security is available in “CICS security” on page 224.

WebSphere V5.0 supports container-managed sign-on with the use of a user ID and password credential (Option A in the J2EE Connector Architecture Specification) and component-managed sign-on (Option C in the J2EE Connector Architecture Specification). Component-managed sign-on requires the component to pass user ID and password credentials through the ConnectionSpec to CICS. If the credentials in the ConnectionSpec are not set, then the credentials in the ManagedConnectionFactory are used to authenticate to CICS.

### 6.3.6 Migration issues

WebSphere Application Server V4.0.x provided an initial implementation of the J2EE Connector Architecture Specification Version 1.0. This implementation provided basic runtime support based on the final J2EE Connector V1.0 specification, but it was not a complete implementation.

WebSphere Application Server V5.0 provides a complete implementation of the J2EE Connector V1.0 specification. These features include support for:

- Connection sharing (res-sharing-scope)
- Get/use/close programming model for connection handles
- Get/use/cache programming model for connection handles
- XA, Local, and No Transaction models of resource adapters, including XA recovery
- J2EE Connector access from both EJB and Web components
6.3.7 CICS resource adapters

The CICS Transaction Gateway (CICS TG) is a set of client and server software components that allows a Java application to invoke services in a CICS region. The Java application can be an applet, a servlet, an enterprise bean, or any other Java application.

The CICS Transaction Gateway consists of the following components:

- Gateway daemon, a long running process that serves as a server for network-attached Java applications. It provides the three basic interfaces.
- Client daemon, provides client-server connectivity.
- Configuration tool, a GUI for configuring the client and the gateway daemon.
- Terminal servlet, a servlet that allows you to use a Web browser as an emulator for a 3270 CICS Application.
- Java class library, three basic interfaces: ECI, EPI, ESI.

![Figure 6-23 CICS Transaction Gateway](image)

Two J2EE Connector CICS resource adapters are provided with the IBM CICS Transaction Gateway (CICS TG):

- External Call Interface (ECI) is a call interface to COMMAREA-based CICS applications.
- External Presentation Interface (EPI) is an API to invoke 3270-based transactions.

**ECI resource adapter**

The CICS ECI resource adapter uses the External Call Interface (ECI) of the CICS Transaction Gateway to communicate with CICS. It can link to CICS.
programs, passing data in a buffer called a COMMAREA. The J2EE Connector resource adapter archive (RAR) file is cicseci.rar.

**Transaction management support**
The CICS ECI resource adapter implements the LocalTransaction interface and supports local transactions. If you use an application server that supports last participant support, such as IBM WebSphere Application Server Enterprise V5.0, the resource adapter can participate in a global transaction provided that it is the only local transaction resource in the global transaction.

**Security management support**
To communicate with a secure CICS region using the ECI, you must send a valid user name and password. These can be supplied by the application or by the container (if the container supports this).

**Connection management support**
Support is provided by the ECI resource adapter for pooling connections from the application server to the gateway daemon. If you are using the TCP protocol, the underlying socket connection from the Java application to the gateway daemon can be pooled for different ECI requests. However, if the CICS TG protocol is LOCAL, then there are no network connections and objects are pooled only in memory, thus having little impact on performance.

**EPI resource adapter**
The CICS EPI resource adapter uses the External Presentation Interface (EPI) of the CICS Transaction Gateway to communicate with CICS. It can start CICS transactions by interacting with a virtual terminal. The virtual terminal represents a 3270 terminal to the CICS TG user. The J2EE Connector RAR file is cicsepi.rar.

**Transaction management support**
The EPI deals only with the CICS 3270 interface; therefore, recoverable work is not performed on the client. For this reason the CICS EPI resource adapter does not support transactions.

**Security management support**
To communicate with a secure CICS region using the EPI, you must supply a valid user ID and password. If you are using signon-incapable terminals, then a user ID and password are required for each request. Support for signon-capable terminals is provided by the LogonLogoff interface, which allows a Java developer to define a sign-on procedure in CICS (such as starting the CESN transaction).
**Connection management support**

Connections are not pooled when using the EPI resource adapter, but support is provided by the EPI resource adapter for pooling terminals installed on the CICS region. This provides for optimization by reducing the amount of network flows and CICS terminal install operations required for a series of EPI calls.

### 6.3.8 Selecting a CICS resource adapter

In this section we consider the characteristics of the two CICS resource adapters (EPI/ECI) and the situations in which each would be selected:

- **External Call Interface (ECI)**
  
  ECI uses COMMAREA as an interface to a CICS enterprise application. If the enterprise application is not using COMMAREA as an interface, it needs to be modified to use COMMAREA. The development effort for a session bean that has an interface with a CICS ECI resource adapter is relatively small. This is because ECI has a simple calling type interface rather than the screen-oriented, conversational type interface of EPI. For this reason, we recommend that ECI be used for new enterprise applications that will be Web enabled. You can separate business logic in the enterprise tier from presentation logic residing in an application server.

- **External Presentation Interface (EPI)**
  
  EPI uses a 3270 data stream as an interface to a CICS 3270 application. If the enterprise application is a 3270 CICS application, EPI should be used for the resource adapter. There is no need to change the enterprise 3270 application at all. Using J2EE Connector Architecture CCI, the EPI application can use the same interface as ECI, but the underlying interface is conversational.

Table 6-2 shows the characteristics of ECI and EPI.

**Table 6-2  CICS ECI and EPI characteristics**

<table>
<thead>
<tr>
<th></th>
<th>ECI</th>
<th>EPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Protocol type</td>
<td>Remote call</td>
<td>Conversational</td>
</tr>
<tr>
<td>Interface</td>
<td>COMMAREA</td>
<td>3270 data stream</td>
</tr>
<tr>
<td>Max data length</td>
<td>32 KB</td>
<td>24x80 plus control characters</td>
</tr>
<tr>
<td>CICS TG JCA support</td>
<td>Distributed or z/OS</td>
<td>Distributed only</td>
</tr>
</tbody>
</table>
6.3.9 Asynchronous calls

You can avoid a blocking call when you invoke an enterprise application using asynchronous CCI calls. This allows your Java application to call a CICS program without blocking while waiting for the response from CICS.

Example 6-3 shows how to perform a non-blocking call using CCI. This call type option can be specified for the InteractionSpec using setInteractionVerb. The synchronous example using CICS ECI is shown in Example 11-8 on page 345.

Example 6-3  Non-blocking call using CCI

```java
// Create Interaction Spec
iSpec = new ECIInteractionSpec();
// Set the CICS program name to the function name
iSpec.setFunctionName("CALCRATE");
// Set call type to SYNC_SEND (send-without-waiting)
iSpec.setInteractionVerb(ECIInteractionSpec.SYNC_SEND);
...
// Create rateRecord
...
// Invoke the program in asynchronous mode
interaction.execute(iSpec, rateRecord, null);
...
// Other business logic
...
// Set commarea length for the reply
iSpec.setCommareaLength(rateRecord.getSize());
// Set call type to SYNC_RECEIVE to receive the asynchronous response
iSpec.setInteractionVerb(ECIInteractionSpec.SYNC_RECEIVE);
// Wait for the reply COMMAREA
interaction.execute(iSpec, rateRecord, rateRecord);
```

6.3.10 CICS ECI design considerations

Some application design considerations when selecting the CICS ECI resource adapter are:

- If your legacy CICS application does not use a COMMAREA interface, it must be changed to use COMMAREA.

<table>
<thead>
<tr>
<th>Recommendation</th>
<th>ECI</th>
<th>EPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use with new applications or existing COMMAREA-based applications</td>
<td>Use with existing 3270 applications only</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 6. Application and system design

6.3.11 Best practices for J2EE Connector Architecture

Some best practices for J2C application developers are:

- Use J2C in a managed environment.
  From an application developer’s perspective, the greatest benefit of utilizing J2C is the Quality of Services (QoSs) provided by the system contracts. In the managed environment, the application developer does not have to program transactions, security, concurrency, and distribution, but relies on a container to provide these services transparently. In a non-managed environment, the application client has to take responsibility for managing connections, transactions, and security by using the low-level APIs exposed by the resource adapter. Connection pooling, for example, provides advantages such as reduced network I/O and CPU utilization, and writing your own connection pooling code is not a simple task.

- Minimize the resource adapter-specific calls.
  J2C provides a set of common client programming interfaces, an EIS-independent API for coding the resource adapter-specific function calls. The application developer should not use the resource adapter-specific calls directly if the function is provided by CCI. However, even CCI has a resource adapter-specific interface such as ConnectionSpec or InteractionSpec. You can encapsulate the manipulating of these classes or the other resource adapter-specific calls into a method that makes the calling side of the method more generic and independent to the resource adapters. This technique should be considered if you need to invoke several resource adapters.

- Only agent optimization.
  If an application accesses a single resource manager using an XA transaction, it has a performance overhead compared to using a local transaction. The overhead is due to the involvement of an external transaction manager in the coordination of the XA transaction. To avoid the overhead of using an XA transaction in a single resource manager, the application component should use a local transaction instead of an XA transaction.
6.4 Design guidelines for Java Message Service

In this section we focus on the roles of the J2EE Java Message Service (JMS) and WebSphere MQ in enterprise messaging applications. JMS applications are composed of the following parts:

- JMS clients are Java programs that send and receive messages.
- Messages are defined for each application and used for communication.
- A JMS Provider is a message system that implements JMS in addition to the other administrative and control functionality required for a full-featured messaging product.
- Administated objects are preconfigured JMS objects created by an administrator for JMS clients. There are two types of administrative objects:
  - ConnectionFactory is used to create a connection with the provider.
  - Destination is used to access a source or destination of messages.

6.4.1 Message models

Each messaging model has a set of interfaces in JMS that define specialized operations for that model. There are two domains defined in the JMS specification for messaging applications:

- Point-to-point (PTP)
- Publish/subscribe (pub/sub)

Please see Figure 6-24 on page 151 for the JMS class diagram.
JMS is based on some common messaging concepts which are defined in JMS parent classes. Each messaging domain defines a customized set of these classes for its own domain. There are also classes defined that are transaction aware, like XAQueueConnectionFactory.

The JMS parent classes define the following basic message concepts:

- ConnectionFactory is an administrative object used by a client to create a connection.
- Connection is an active connection to a JMS Provider.
- Destination is an administrative object encapsulating the identity of a message destination.
- Session is a single-threaded context for sending and receiving messages.
- MessageProducer is an object created by a Session for sending messages to a Destination.
- MessageConsumer is an object created by a Session for receiving messages from a Destination.
Not all JMS objects can be used concurrently. Table 6-3 shows the objects that can be used concurrently and those that cannot.

Table 6-3 Concurrent JMS classes

<table>
<thead>
<tr>
<th>Object</th>
<th>Concurrent use</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConnectionFactory</td>
<td>Yes</td>
</tr>
<tr>
<td>Connection</td>
<td>Yes</td>
</tr>
<tr>
<td>Destination</td>
<td>Yes</td>
</tr>
<tr>
<td>Session</td>
<td>No</td>
</tr>
<tr>
<td>MessageProducer</td>
<td>No</td>
</tr>
<tr>
<td>MessageConsumer</td>
<td>No</td>
</tr>
</tbody>
</table>

There are two reasons for restricting concurrent access to Sessions. First, Sessions are the JMS entity that supports transactions. It is very difficult to implement transactions that are multi-threaded. Second, Sessions support asynchronous message consumption. If a Session has been set up with multiple, asynchronous consumers, it is important that these separate consumers do not execute concurrently.

6.4.2 JMS point-to-point model

Point-to-point (PTP) messaging involves working with queues of messages. The sender sends messages to a specific queue to be consumed normally by a single receiver. In point-to-point communication, a message has at most one recipient. A sending client addresses the message to the queue that holds the messages for the intended (receiving) client. You can think of the queue as a mailbox. Many clients might send messages to the queue, but a message is taken out by only one client. Like a mailbox, messages remain in the queue until they are removed. Thus the availability of the recipient client does not affect the ability to deliver a message. In a point-to-point system, a client can be a sender (message producer), a receiver (message consumer), or both. In JMS, PTP types are prefixed with “Queue”. 

Figure 6-25 JMS point-to-point model
In point-to-point messaging, there are generally three messaging patterns:

- Request/reply
- Send-and-forget model or message producer
- Message consumer

We look at these messaging patterns in more detail in 6.4.5, “Synchronous vs. asynchronous design considerations” on page 155.

### 6.4.3 JMS publish/subscribe model

In contrast to the point-to-point model of communication, the publish/subscribe model, shown in Figure 6-26, enables the delivery of a message to multiple recipients. A sending client addresses, or publishes, the message to a topic to which multiple clients can be subscribed. There can be multiple publishers, as well as subscribers, to a topic. A durable (or persistent) subscription, or interest, exists across client shutdowns and restarts. While a client is down, all objects that are delivered to the topic are stored and then sent to the client when it renews the subscription. A non-durable subscription will deliver messages when the consumer is connected, but discard messages when the consumer is not connected. In a publish/subscribe system, a client can be a publisher (message producer), a subscriber (message consumer), or both. In JMS, pub/sub types are prefixed with “Topic”.

![Publish/subscribe model](image)

JMS also supports the optional durability of subscribers and “remembers” that they exist while they are inactive. All an application has to do is send information it wants to share to a standard destination managed by IBM WebSphere MQ publish/subscribe, and let IBM WebSphere MQ publish/subscribe deal with the distribution. Similarly, the target application does not have to know anything about the source of the information it receives.

Another important aspect of the pub/sub model is that there is typically some latency in all pub/sub systems. This is because messages observed by subscribers may depend on the underlying JMS Provider’s capability to
propagate the existence of new subscribers and how long the messages are retained by the provider.

We look at the pub/sub pattern in more detail in 6.4.5, “Synchronous vs. asynchronous design considerations” on page 155.

6.4.4 JMS messages

Another design choice to use is JMS message type. As shown in Figure 6-27, JMS messages are composed of the following parts:

- **Header**: Contains information to identify and route messages.
- **Properties**: Custom values that can optionally be added to messages. Properties can be:
  - Application-specific: Properties added to messages, which are used by JMS applications
  - Standard: JMS properties
  - Provider-specific: Properties that are specific to a messaging provider
- **Body**: The message data.

![Figure 6-27 Message content](image)

JMS provides different *message types*. Each contains specific interfaces pertaining to its content and allows specific operations on the messages.

The message types that can be used in JMS are:

- **BytesMessage**: Contains operations for storing and accessing a stream of bytes.
- **StreamMessage**: Contains operations for storing and accessing a stream of Java primitive values. It is filled and read sequentially.
- **ObjectMessage**: Contains operations for storing and accessing a serialized Java object. If the application design requires more than one object to be serialized, then use a Collection object.
MapMessage: Contains operations for storing and accessing a set of key-value pairs from the message body. The keys must be strings and the values must be primitive types.

TextMessage: Contains operations for storing and accessing the body of a message as a string. Text messages can be used to store XML-data. This type of message can be used for sending messages to non-Java applications.

A couple of message settings are also important to look at:

- Delivery mode: When delivery needs to be assured by the business requirements, persistent messages are needed. But when this is not needed, performance can be gained by the use of non-persistent messages.
- Message expiration: When using non-persistent messages, message expiration can be used to discard messages that have remained on a queue or topic for longer than required. This prevents unprocessed messages from building up over time.

### 6.4.5 Synchronous vs. asynchronous design considerations

In the Web application environment, choosing an asynchronous or synchronous approach to JMS communication will significantly affect the design of the application. The effects could ripple as far as the user interface interaction (or user experience), or it could affect only the low-level design and behavior of the underlying application.

We look at both the user interaction differences and the system design considerations.

For the purposes of discussion, let us consider an example Web application that provides Web banking and needs to connect to an enterprise application that is hosting the bank account data.

First, it is important to go over some basic Web application principles. The Web is a stateless environment; typically a request is received and the reply sent back immediately within the same client session. A Web server is not normally able to initiate a connection to a Web client out of the blue. Information about the requesting client is retained while the request is being serviced and not lost until a reply is sent back. The Web is a typical request/reply model. Most Web applications are built using this model and this style of user interaction where the
user can expect a reply back from the server that will be the result of making a request.

Using our Web banking example, let us assume a Web request requires information from the enterprise application about the bank balance. The JMS interaction between the Web application and the enterprise application can be achieved using:

- Request/reply pattern
- Send-and-forget pattern
- Message consumer pattern
- Publish/subscribe pattern

**Request/reply pattern**

Using this approach, we fit the standard Web model by providing a complete round trip for the client request that results in a reply. The user does not have to visit another results page to see the results of his/her request.

As shown in Figure 6-28 on page 157, the Web application sends a request message, then waits for a reply. The response message needs to be linked to the request message using the request message ID as the correlation ID of the response message.

The overriding factor in a request/reply pattern is the time delay before a reply gets back. You should remember that request/reply is a synchronous communication over an asynchronous transport. For request/reply, two queues are needed, one for the sender to send messages and one for receiving the responses back. The request/reply consists of two units of work:

- Putting the message on a queue
- Receiving the response and, for example, inserting the message in a database

These actions can never be one unit of work because the real put of the message only takes place after the commit. No message will be sent without a commit, and when no request message is sent, no reply will arrive.

An example of a request/reply scenario is getting your account balance. A message is first sent with the account ID, then the application waits until a response message is sent back with the balance of the account, with the results logged to an application database.
Applications should not be designed without appropriate timeout or retry capability. Waiting indefinitely on a queue for the reply message to arrive will cause at best a poor user experience. At worst it will result in a consumption of system resources to a point where the entire application will fail or stop responding. The application design should therefore cover delayed replies and no reply at all. The use of non-persistent messages can become useful in these situations. If a reply has not been received within a given amount of time, an exception path is taken by the application, potentially resubmitting the request. When the response of the first request arrives, this response can be ignored. It is also possible to include a timeout time in a message. The message will then be destroyed when the timeout is reached. Typically the use of non-persistent messages will also depend on the type of application and business requirements. For example, if the application is a transfer of monetary funds then it is quite likely that persistent messages will be used. When a failure occurs, a check back later algorithm is needed.

However, implementation of this approach presents a few important issues. Specifically, if the message producer is implemented as a session EJB, then in the request/reply JMS model, the EJB must wait (or block) until the enterprise application has replied before it can continue processing. Blocking in an EJB is not generally recommended because it restricts the EJB container's ability to effectively manage its resources. Care must be taken to ensure that the EJB is not waiting indefinitely and that there is a timeout in place.

Send-and-forget pattern
In send-and-forget (or fire and forget), shown in Figure 6-29 on page 158, the Web application will initiate the request to the enterprise application using a JMS destination, but it will not wait for the outcome. This design has important
repercussions on the user interaction. A message consumer pattern could be used for receiving the reply from the enterprise application. The user must at some point go to a result page to see his or her bank balance when it has been retrieved from the enterprise application.

From an implementation point of view, the blocking EJB dilemma is avoided. However, a new page is required to allow the customer to come back to check his/her last balance request from the local database. This design has alleviated the need for the blocking EJB design. However, the user experience is drastically different from the request/reply model.

Figure 6-29  Send-and-forget pattern

**Send-and-forget design considerations**

One of the first things you should consider is whether you make use of persistent or non-persistent messages. Non-persistent messages do not survive process failure, but because their handling does not incur any disk I/O (for persistence), they can be processed much quicker by the JMS Provider. The decision to use persistent or non-persistent messages will generally be governed by the business requirements. In the case of getting a balance, no funds transfer occurs; as such, a lost message has little impact and may not warrant persistent messages. The application may be sending the message as part of a unit of work (transaction), which implies that at some point either the application or the application server will commit the transaction to affect the send.

Another point to consider is the application component used to implement the message producer. The Model-View-Controller model suggests that access to an enterprise resource (message infrastructure) should occur via session EJBs. If you want the message sent as part of a transaction that will be coordinated by the application server (for example, send message and update a local database within a transaction), then EJBs are required. However, it is not uncommon to find the message producer being implemented in servlets; this offers a simpler
programming framework (when compared to EJBs), but immediately precludes the use of global units of work, or access to any other features that the EJB container may offer.

**Message consumer pattern**

Message consumers can be implemented by message-driven beans, which are invoked by the container when a message arrives on a destination. When a message arrives at the destination, the EJB container passes the message to an instance of a user-developed message-driven bean.

This pattern can be used by a catalogue application receiving updates for changes in the online catalogue. In this scenario, a message-driven bean receives an incoming message and updates a database, as shown in Figure 6-30. The pattern is typically useful in a business-to-business situation where no user interaction is needed.

**Message consumer design considerations**

Use message-driven beans only to handle the message. Move the business logic to another bean that will be invoked by the message-driven bean. This way it is also possible to call the business logic out of another channel like a servlet that has been activated by a user.

Message-driven beans cannot throw exceptions to the user, so exceptions have to be logged in an error report.

**Publish/subscribe pattern**

Using this approach, we can provide the user with an immediate reply without him having to explicitly go to a separate Web page to see the results. However, this can only be achieved if a local copy of data is used.
The Web application will register interest in information from the enterprise application upon startup. Periodically the enterprise application will publish information to the subscribers (Web application). The message consumer pattern can be implemented at the subscriber site to receive the publications of the enterprise application. The Web application will store this information in a local database and use this information when a Web request is being serviced.

Using this approach the Web application can operate in its native modes (stateless and request/reply) and the user can see the results of his request within the same user transaction. However, the information may be slightly outdated.

**Publish/subscribe design considerations**

Never cache a non-durable subscription; use durable subscriptions instead.

For further information about how IBM WebSphere MQ can be used in the pub/sub model, refer to the publication *MQSeries Publish/Subscribe Applications*, SG24-6282.

**Selecting a messaging pattern**

None of these options is incorrect if *implemented* correctly. The user’s requirements and experience will dictate which decision is the correct one.

A request/reply JMS communication model is ideal in a Web environment. However, if EJBs are to be the implementers of the enterprise access, care needs to be taken during implementation to prevent blocking calls from EJBs. If the user is willing to accept a different user interaction model, then asynchronous fire-and-forget is also an acceptable option. The middle ground could be achieved using full publish and subscribe; however, the accuracy of the information may be at stake.

**Note:** The request/reply blocking stateless EJB *must* be implemented such that appropriate timeout and retry conditions are applied.

The EJB 2.0 specification does point out that only one client will have access to an instance of a stateless EJB while it is servicing a client-invoked method. If, however, a blocking wait occurs for an indefinite period, the container may run short of available instances of the specific EJB to service other clients and thus slow down the overall performance of the application.

For further information, please refer to:

http://java.sun.com/products/jms/tutorial/1_3_1-fcs/doc/jmsj2ee.html
6.4.6 Where to implement message producers and consumers

There are a number of options when considering where to implement your JMS message producers and consumers in the J2EE application architecture. We examine some of these options in this section.

**Producers**

If the Model-View-Controller (MVC) pattern is invoked, then the model is typically where the producer would be implemented. In J2EE application architecture, this is likely to be a session Enterprise JavaBean. However, it is possible to implement the message producer almost anywhere. A simple JavaBean could also implement the message producer and fit in with the MVC pattern.

If the producer is participating in a transaction of some kind, then session EJBs may be a better implementation choice. Transaction creation and management is gained almost for free within EJBs, whereas it would have to be explicitly created and managed within other implementation choices such as JavaBeans. In the same manner, EJBs typically will have access to other facilities or features within an EJB container.

Servlets can also be used as message producers. They offer a simpler programming model than EJBs. Servlets, however, are usually implementers of the controller aspect of the MVC pattern, and no advantages can be made of the container facilities for EJBs. A special case could be made for “utility or helper” servlets that are not being used as controllers.

**Consumers**

In the same way as producers, there are a number of implementation choices for consumers. When consumers are used in request-reply scenarios, it leaves the choice to implement this in a servlet or an EJB. When implementing the consumer in a EJB, there is the advantage of transaction management and security management of the container. The disadvantage is that an EJB thread will be occupied until a response arrives. Some extra programming is needed to disregard a response when it takes too much time.

Another option for the consumer is a message-driven bean. When using a message-driven bean, the request and reply will be loosely coupled, which makes it more complex. A message-driven bean is a good solution for subscription and message consumer patterns.

6.4.7 Message-driven beans

A message-driven bean (MBD) is an asynchronous message consumer. The onMessage method of the message-driven bean is invoked by the container on
arrival of a JMS message on a queue. Rather than writing application code to poll for messages on a queue, you can use a message-driven bean instead. The main difference between message-driven beans and other enterprise beans is that message-driven beans have only a bean class. There is no home or remote interface for a message-driven bean. Message-driven beans can only be invoked by the container.

The main components used with message-driven beans are shown in Figure 6-31. The deployed message-driven beans (MDB1 to 4) are invoked by listeners. These listeners listen to the ports that are defined for the different destinations where other applications put their messages. Each listener port defines the association between a connection factory, destination, and a deployed message-driven bean. Another component is the message listener service, which comprises a listener for each listener port, all controlled by the same listener manager.

The message listener service is an extension to the JMS functions of the JMS Provider. It includes a listener manager, which controls and monitors one or more JMS listeners. Each listener monitors either a JMS queue destination (for point-to-point messaging) or a JMS topic destination (for pub/sub messaging).
The connection factory is used to create connections with the JMS Provider for a specific JMS queue or topic destination. Each connection factory encapsulates the configuration parameters needed to create a connection to a JMS destination.

When a deployed message-driven bean is installed, it is associated with a listener port and the listener for a destination. When a message arrives on the destination, the listener passes the message to a new instance of a message-driven bean for processing.

The listener manager is initialized at the start of an application server based on the configuration data. The listener manager creates a dynamic session thread pool for use by listeners, creates and starts listeners, and during server termination controls the cleanup of listener message service resources. Each listener completes several steps for the JMS destination that it is to monitor, including:

- Creating a JMS server session pool and allocating JMS server sessions and session threads for incoming messages
- Interfacing with JMS ASF to create JMS connection consumers to listen for incoming messages
- If specified, starting a transaction and requesting that it is committed (or rolled back) when the EJB method has completed
- Processing incoming messages by invoking the onMessage() method of the specified enterprise bean

According to the EJB 2.0 specification, a transactional context may not be carried by a message, so a MDB will never execute within an existing transaction. However, a transaction may be started during the onMessage method execution if one of the following applies:

- The transaction attribute is "required" (container-managed transaction).
- It is explicitly started within the method (bean-managed transaction).

In the second case, the message receipt will not be part of the transaction. In the first case, the container will start a new transaction before de-queuing the JMS message (the receipt of which will thus be part of the started transaction); and enlist the resource manager associated with the arriving message, and all the resource managers accessed by the onMessage method. If the onMessage method invokes other enterprise beans, the container passes the transaction context with the invocation. The transaction started at the onMessage method execution can involve several operations such as accessing a database (via an entity bean or a JDBC data source), or sending messages (using a JMS connection factory). A message-driven bean instance has no state for a specific client. However, the instance variables of the message-driven bean instance can
contain a state across the handling of client messages. Examples of such states include an open database connection and an object reference to an EJB object.

**Message-driven beans design considerations**

A message-driven bean should not contain any business logic. It is better to put the business logic in another enterprise bean, as shown in Figure 6-32, which makes it possible to call the business logic from other components.

![Figure 6-32 Message-driven bean development](image)

Some other design issues are:

- The server does not know the client’s security identity. Messaging does not propagate this identity to the message-driven bean. With this kind of bean, all instances are the same.

- Performance considerations. Messaging becomes a middle layer between the client and the server. Even though message-driven beans are relatively lightweight, an extra layer can add time to your system response.

- Application complexity. An application that does not need asynchronous processing can be easier to code and debug.

### 6.4.8 Managing JMS objects

JMS Connection is the first point of access to JMS objects. JMS Connection is created from JMS ConnectionFactory. Once a connection is created, one or more sessions can be created in the context of the connection. JMS Sessions allow you to create message consumers and producers. When consumers or producers are created, the connection needs to be started to receive or send
messages. JMS Connections can be cached, in a similar way as EJB Home objects are cached, and reused by many clients.

JMS Sessions are designed for synchronous access only. A session can only be used by a single client and not shared among other clients. Similarly, an instance of either MessageConsumer and MessageProducer can only be used by a single client. JMS Sessions are opened for the duration of message sending or receiving; after this the session can be closed.

When a session is opened, the correct session acknowledgment must be selected from a performance perspective. In our sample scenario, we selected AUTO_ACKNOWLEDGE. This policy specifies that the message be delivered once and only once. The server must send an acknowledgment back, so the server incurs an overhead to implement this policy. The DUPS_OK_ACKNOWLEDGE setting resends the message until an acknowledgment is sent from the server. The server will operate in a lazy acknowledge mode, thereby reducing the overhead on the server but resulting in an increase in network traffic. With the most overhead of the three settings, CLIENT_ACKNOWLEDGE will cause the server to wait until a request for acknowledgment is sent from the client. Usually the client calls the sent message’s acknowledge method.

On completion of interaction with the message producer or consumer (sender or receiver), the session needs to be closed. If the connection is closed, the session belonging to this connection is automatically closed.

The message producers/consumers must also be closed when you finish sending/receiving messages. Again, if the connection is closed the producers and consumers are automatically closed.

Garbage collection of Java cannot be relied upon to clean out objects in a timely manner. It is always a good practice to call the close of any resource-bound object.

For further information, please read the JMS specification at:

http://java.sun.com/products/jms/docs.html

6.4.9 JMS and JNDI

The Java Naming and Directory Interface (JNDI) API implementation provides directory and naming functionality to programs developed in Java. This allows Java programs to discover and retrieve objects of any type from the JNDI name space.
JMS has two types of administered objects:

- ConnectionFactory
- Destination

An administrator can place objects of these types in the JNDI name space to be accessed by messaging applications.

Figure 6-33 shows the role of JMS and JNDI relative to a Java application. These two APIs sit above any specific service providers and encapsulate any vendor-specific information.

As a result, a developer using these technologies in a messaging-enabled application need only be familiar with the APIs, not the specific messaging systems.

So, how does an administrator put these objects in the JNDI name space? This step is vendor specific. If you are using WebSphere MQ V5.3 with WebSphere Application Server V5.0 you can administer these objects right from the WebSphere Administrative Console. If you are using another application server, WebSphere MQ V5.3 provides a tool called JMSAdmin for this purpose.

### 6.4.10 Embedded JMS Provider vs. WebSphere MQ

In line with the J2EE 1.3 specification, WebSphere Application Server V5.0 has an embedded JMS Provider, or messaging service, included in the application server. This internal JMS Provider can be used for asynchronous JMS communications with other WebSphere applications. The internal messaging service cannot be used for messaging with other messaging system, such as WebSphere MQ. If you need to communicate with other systems using WebSphere MQ, then you need to install WebSphere MQ as a JMS Provider on a WebSphere Application Server.
6.4.11 WebSphere to MQ connection options

A message placed on an IBM WebSphere MQ queue from an application server may originate directly from a servlet, or may be sent from a command bean or EJB. (We recommend the latter two methods and not so much from servlets.) Regardless of the method used, the messages are sent to a queue manager using one of the two available WebSphere MQ Java APIs by IBM WebSphere MQ. Each API has certain characteristics that make it appropriate for a situation, depending on the priorities you have. However, the API chosen can have an effect on the options you have for distributing the application components.

The two APIs that we discuss here are:

- The IBM WebSphere MQ for Java Message Service package, com.ibm.mq.jms.jar and com.ibm.jms
  IBM WebSphere MQ for JMS classes implements the J2EE Java Message Service (JMS) interface to enable JMS programs to access a subset of IBM WebSphere MQ features from a vendor-neutral point of view, as defined by the JMS specification. The JMS interface is implemented by a set of IBM WebSphere MQ classes for JMS.

- The IBM WebSphere MQ for Java package, com.ibm.mq.jar
  IBM WebSphere MQ for Java classes enable Java applets, applications, servlets, and EJBs to issue direct calls and queries to IBM WebSphere MQ using specific calls designed to take advantage of IBM WebSphere MQ features.

A JMS Java application uses the vendor-independent JMS interfaces to access the MQ-specific implementation of the JMS classes.

A key idea in JMS is that it is possible, and strongly recommended, to write application programs that use only references to the interfaces in javax.jms. All vendor-specific information is encapsulated in implementations of:

- QueueConnectionFactory
- TopicConnectionFactory
- Queue
- Topic

Coding outside the JMS interface to access WebSphere MQ-specific features will, of course, reduce the portability of the application, since it is now directly referencing WebSphere MQ-specific classes. If application portability, vendor independence, and location transparency are of importance, pure JMS is the obvious choice. JMS uses abstracted concepts of messaging to provide a vendor-independent API to messaging, while underneath lies the IBM WebSphere MQ implementation of the JMS interfaces. The real-world entities that are IBM WebSphere MQ queue managers and queues are accessed by
JMS clients through the use of the Java Directory and Naming Service (JNDI). The IBM WebSphere MQ entities are published to JNDI from the WebSphere Administrative Console, or through a tool called JMSAdmin. MQ JMS supports both the point-to-point and publish/subscribe models of JMS.

MQ base JMS classes provide two connection options to IBM WebSphere MQ:

- Bindings mode to connect to a queue manager directly
- Client mode using TCP/IP to connect to a queue manager (not supported on z/OS or OS/390)

All support connection pooling.

Java bindings mode

In binding mode, also known as server connection, the communication to the queue manager utilizes inter-process communications. One of the key factors that should be kept in mind is that binding mode is available only to programs running on the WebSphere MQ server that hosts the queue manager.

The key connection parameter in this case is the queue manager name.

Connecting to the local queue manager has several major advantages:

- The probability of establishing a connection to a queue manager in your own host is high, as opposed to a connection with a remote queue manager.
- The time it takes to establish a network connection to the queue manager is avoided.
- The local queue manager can distribute the work among multiple brokers. If connection performance is a high priority in your network, then using bindings mode is the clear choice.
Using bindings mode, you can also use WebSphere as an XA resource coordinator for units of work that involve WebSphere MQ updates and database updates, for databases and drivers that support the XOpen/XA standards.

**Java client mode**

Client connection uses a TCP/IP connection to the WebSphere MQ Server and enables communications with the queue manager. Programs using client connections can run on an WebSphere MQ client machine as well as on a WebSphere MQ server machine. Client connections use client channels on the queue manager to communicate with the queue manager. The client connection does not support XA transaction coordination by the queue manager.

Both the WebSphere MQ classes for Java and the WebSphere MQ classes for JMS are needed to use WebSphere MQ client mode. The key connection parameters are host name, TCP/IP port, and server connection channel name. If your code is using only the JMS interfaces (for maximum portability), then client mode cannot be achieved since there are no methods exposed in the JMS interface to select a host or port number.

The client mode is best used when you do not want IBM WebSphere MQ to reside on the same machine as the application server. It allows you to connect directly to a remote IBM WebSphere MQ queue manager.

![Figure 6-35 Client mode to remote brokers](image)

When you connect directly to a queue manager on a broker, as in Figure 6-35, you relinquish any workload distribution the queue manager offers. The application must decide which broker to send the work to and any workload distribution would have to be done in the application itself, which is not recommended. Even having the queue managers in a cluster does not help, since a queue manager will always send the work to the local instance of the broker. Another pitfall is that XOpen/XA facilities for coordinated commits to
WebSphere MQ/JMS and databases are no longer available using WebSphere as the transaction coordinator.

One way around this is to connect to a remote queue manager that does not have a broker instance, but is there purely for workload distribution, as shown in Figure 6-36.

Figure 6-36  Client mode to a remote queue manager

You still have the network connectivity time; in fact, you have made it a little worse by introducing an intermediate system. But you do have the advantage of the queue manager workload distribution and the ability to connect to a remote queue manager. Yet another way would be to use TCP/IP load balancing, but this is not a function of WebSphere MQ.

The client mode can also be used to connect to the local queue manager by passing through the internal TCP/IP stack. This is obviously not as efficient as using the bindings mode, but it does allow your program to be used in a generic environment where you do not know if the queue manager will be local or not. You can also make the different connection options parameter-driven so that the application is ready no matter the connection type. You do need to ensure that the correct parameters are passed so you get the connection type you desire. A database table would be a good place to store these if you are interacting with a database.

Both MQ JMS classes (not the pure JMS interface) and MQ base Java allow you to put messages from the Java application in WebSphere directly into the remote broker’s queue. If you are thinking of doing this, you should consider the performance implications. The cost of creating a network connection is added to the total cost of each request. For each request, an IBM WebSphere MQ-to-client session is created. There is no long-lasting network connection. This will impact the ability to run thousands of sessions in parallel. If you create a local IBM WebSphere MQ session for each request, the overhead will be much lower. The
network connection is now maintained by a sender-receiver channel pair and is long running. Ideally, long-running IBM WebSphere MQ sessions are preferable.

**IBM WebSphere MQ clustering**

WebSphere MQ offers the ability to create clusters. MQ clusters provide a number of benefits that are silently utilized by JMS applications. Clusters offer:

- **Simpler administration of logically related queue managers**
  
  Clustering allows communication between queue managers to promote information about the queues they offer. Once in a cluster, queues on remote queue managers are visible to all queue managers if the queues are defined as cluster queues. The number of explicit definitions within IBM WebSphere MQ administration is reduced with the use of clusters.

- **Workload and failover management**

  Adding queue managers to clusters allows access to WebSphere MQ workload and failover features.

  As shown in Figure 6-37, QM3 is able to load balance across the queue named ReplyQ, since it is available on both QM1 and QM2. Similarly, if QM1 is disabled, all messages for ReplyQ are routed to QM2.

![Figure 6-37  Cluster workload management](image)

None of these features can be controlled through the JMS interfaces. However, MQ will automatically utilize the workload and failover under JMS.

These and other features of MQ offer significant benefits and demonstrate that IBM WebSphere MQ is a reliable, scalable, and mature JMS Provider.
6.4.12 Best practices for JMS and IBM WebSphere MQ

This section briefly discusses a number of issues and common best practices for JMS and IBM WebSphere MQ.

**Design level**

Application design level best practices for JMS and IBM WebSphere MQ include:

- **Use message timeouts.**
  
  Using message timeouts avoids large numbers of messages remaining on a queue, thus reducing performance overheads. It also allows a relevancy aspect on the messages. For example, a message may not be relevant after a certain period of time because the information in the message has been superseded.

- **Use message selectors.**
  
  JMS provides a useful API that can reduce the need for complex message browsing code. Message selectors allow the underlying provider (through JMS) to browse messages before they are retrieved with little application code.

- **Persistent versus non-persistent.**
  
  The use of durable messages is often necessary when communication is between, for example, systems of a financial nature. However, if messages do not need to be persisted, then messages need to be explicitly set to non-persistent, since the default in JMS is persistent.

- **Clusters.**
  
  The use of IBM WebSphere MQ clusters allows simple administration, and automatic workload management and failover.

- **Message producers.**
  
  EJB message producers: In a request/reply scenario, it is important that the issue of blocking calls is dealt with correctly. Essentially, EJBs should only be used with appropriate request/reply timeouts and retries.

- **Message consumers.**
  
  Message-driven beans: Business logic should not be implemented in the message-driven bean. Implement the business logic in another component, such as a session bean, and use message-driven beans only for receiving the message.

  Never throw application exceptions in the onMessage method.
Consider using XML-based messages for inter-application integration.

XML is commonly used as a messaging structure that allows for a more portable inter-application integration model. Although it does add some overhead to the message payload size and requires XML parsers, it is quickly becoming a standard for inter-operability.

**Code level**

Application code level best practices for JMS and IBM WebSphere MQ include:

- **Explicitly close JMS resources that are no longer required.**

  Java garbage collection alone cannot release all IBM WebSphere MQ resources in a timely manner. This is especially true if the application needs to create many short-lived JMS objects at the Session level or lower. It is therefore important to call the close() methods of the various classes (QueueConnection, QueueSession, QueueSender, and QueueReceiver) when the resources are no longer required.

- **Handling errors.**

  Any runtime errors in a JMS application are reported by exceptions. The majority of methods in JMS throw JMSExceptions to indicate errors. It is good programming practice to catch these exceptions and report them to suitable output.

  Unlike normal Java exceptions, a JMSException may contain a further exception embedded in it. For JMS, this can be a valuable way to pass important details from the underlying transport. In the case of MQ JMS, when IBM WebSphere MQ raises an MQException, this exception is usually included as an embedded exception in a JMSException.

  The implementation of JMSException does not include the embedded exception in the output of its toString() method. Therefore, it is necessary to check explicitly for an embedded exception and print it out, as shown in Example 6-4.

**Example 6-4  Checking for JMSException**

```java
try {
    // code which may throw a JMSException
    ...
} catch (JMSException je) {
    System.err.println("caught "+je);
    Exception e = je.getLinkedException();
    if (e != null) {
        System.err.println("linked exception: "+e);
    }
}
```
Exception listeners.

For asynchronous message delivery, the application code cannot catch exceptions raised by failures to receive messages. This is because the application code does not make explicit calls to receive() methods. To cope with this situation, it is possible to register an ExceptionListener, which is an instance of a class that implements the onException() method. When a serious error occurs, this method is called with the JMSException passed as its only parameter. Further details are in the J2EE JMS specification.

6.4.13 More information

These documents and Web sites are further information sources:

- Java Message Service API documentation
- WebSphere Developer Domain article *Integrating IBM WebSphere Application Server and the WebSphere MQ Family*
- WebSphere Developer Domain article *IBM WebSphere and MQSeries® Integration Using Servlets and JavaServer Pages*
- IBM Redbook *Self-Service Applications using IBM WebSphere V4.0 and IBM MQSeries Integrator*, SG24-6160
- IBM Redbook *MQSeries Programming Patterns*, SG24-6506
- IBM Redbook *EJB 2.0 Development with WebSphere Studio Application Developer*, SG24-6819
- *WebSphere MQ Application Programming Guide*, SC34-6064
- *WebSphere MQ Using Java*, SC34-6066
Chapter 7. Application development

The development of a self-service application using Web services, J2EE Connectors, or JMS does not differ very much from the development of any object-oriented, client/server application. However, there are some special considerations, which we outline in the main body of this chapter. We start this chapter with a brief look at application development methodology in general, and conclude with development guidelines for XML messaging and an overview of design modeling using Rational XDE®.
7.1 Application development methodology

Today it is quite common in the industry to develop object-oriented software via an iterative and incremental process. This approach has different roots. For more information, refer to *Object-Oriented Analysis and Design with Applications* by Grady Booch, *Object-Oriented Software Engineering* by Ivar Jacobson, and *Object-Oriented Modeling and Design* by James Rumbaugh.

There is no defined standard process for development that everyone uses. Different teams typically adopt a recognized process using a vendor methodology or using their services team methodology. IBM Global Services has its own methodology used in customer engagements that covers the development process. Rational Software Corporation®, for example, uses its Rational Unified Process®.

These methodologies generally divide the development process into different phases. Each phase is done in a sequential manner and is subdivided into further smaller phases. Some phases are only run through once. Others are done over and over again, forming the iterative and incremental part of the development process. The actual process and which phases you use might differ slightly depending on the development team or organization that uses the process. We can divide the whole process into the following phases:

- Solution outline
- Macro design
- Micro design
- Build cycle
- Deployment

*Figure 7-1 Development process overview*

In the solution outline phase you decide the scope of the project, explore what the essential business needs are, come up with an idea of the base architecture, and get the commitment from the project sponsor to start.
Then you start with the macro design, which concentrates on the detailed requirements gathering, business process modeling, high-level analysis and design, the base architecture, and a plan for the subsequent development phases, including a development release plan. These two phases are usually done once in a project.

Now the iterative and incremental part of the development starts. For each release of the developed e-business application, the micro design, build cycle, and deployment phases are completed. Usually, a subset of use cases that has to be developed to meet a part of the system requirements make up a release. Use cases are grouped according to relevance and timeliness of functionality. The releases are defined in the project plan produced in the previous phase. A release can be an internal one that is not deployed to any users. This is quite common for early stages of big projects. Others, such as alpha or beta releases, might be deployed to a certain number of test users. It might take several iterations until a first official release of the application is deployed to the users. In turn, there are often several releases to the users until all requirements are met, plus maintenance releases to fix errors and other defects.

For more information on the application development process in the context of IBM Patterns for e-business, refer to the publication *Self-Service Patterns using WebSphere Application Server V4.0*, SG24-6175.

### 7.2 Development guidelines for Web services

In this section we describe some high-level application development procedures for developing Web service requesters and providers. In 6.2, “Design guidelines for Web services” on page 111, we discussed some of the design choices that an architect has to make when designing a Web service provider or requester. We look at procedures for implementing some of these design choices, including transmission patterns and SOAP messaging mechanisms. We also look at tools for developing, deploying, and hosting Web services, including WebSphere Studio and WebSphere Application Server.

#### 7.2.1 Transmission patterns

The Web services transmission patterns we identified in “Transmission patterns” on page 115 represent different types of operations. Examples of how to implement the operation in WSDL for each transmission pattern are:

> - Request-response (Example 7-1 on page 178)
> - One-way (Example 7-2 on page 178)
> - Solicit-response (Example 7-3 on page 178)
> - Notification operation (Example 7-4 on page 178)
Example 7-1  WSDL for request-response transmission

```xml
<wsdl:definitions>
  <wsdl:portType> *
    <wsdl:operation name="nmtoken" parameterOrder="nmtokens">
      <wsdl:input name="nmtoken" message="qname"/>
      <wsdl:output name="nmtoken" message="qname"/>
      <wsdl:fault name="nmtoken" message="qname"/>
    </wsdl:operation>
  </wsdl:portType>
</wsdl:definitions>
```

Example 7-2  WSDL for one-way transmission

```xml
<wsdl:definitions>
  <wsdl:portType> *
    <wsdl:operation name="nmtoken">
      <wsdl:input name="nmtoken" message="qname"/>
    </wsdl:operation>
  </wsdl:portType>
</wsdl:definitions>
```

Example 7-3  WSDL for solicit-response transmission

```xml
<wsdl:definitions>
  <wsdl:portType> *
    <wsdl:operation name="nmtoken" parameterOrder="nmtokens">
      <wsdl:input name="nmtoken" message="qname"/>
      <wsdl:output name="nmtoken" message="qname"/>
      <wsdl:fault name="nmtoken" message="qname"/>
    </wsdl:operation>
  </wsdl:portType>
</wsdl:definitions>
```

Example 7-4  WSDL for notification operation transmission

```xml
<wsdl:definitions>
  <wsdl:portType> *
    <wsdl:operation name="nmtoken">
      <wsdl:output name="nmtoken" message="qname"/>
    </wsdl:operation>
  </wsdl:portType>
</wsdl:definitions>
```
7.2.2 SOAP messaging mechanism

In this section we look at the steps needed to implement Web services using the following messaging mechanisms as identified in “SOAP messaging mechanisms" on page 117:

- SOAP RPC-based Web services
- SOAP message-oriented Web services

Creating RPC-based Web services
You can perform the following steps to build a SOAP RPC requester (client):

1. Obtain the WSDL service definition and interface files from the provider.
2. Create a RPC Call object.
3. Set the target URI.
4. Set the MethodName of the service to be called.
5. Create the Parameter objects to send.
6. Invoke the Web service.
7. Receive the Response object.

Our sample Web services application in Chapter 10, “Web services scenario” on page 281, illustrates in detail how to perform these steps to create a Web service client or requester.

To create a SOAP RPC provider service using WebSphere Studio, you would perform the following steps:

1. Create a Web project to host the Apache SOAP servlets for handling incoming requests.
2. Create or import an object (JavaBean, EJB, etc.) to be turned into a Web service.
3. Create a Web service that exposes the required methods.

The Web service methods that are exposed usually serve as a facade to the actual enterprise service that will provide the data requested. This enterprise service or services can be legacy or existing services residing on any enterprise platform and written in any format. Our sample Web services application in Chapter 10, “Web services scenario” on page 281, gives detailed instructions on how to perform these steps to create a Web service provider on an existing or new enterprise application.

Creating message-oriented Web services
You can perform the following steps to create a SOAP message-oriented requester (client):

1. Obtain the WSDL service definition and interface files from the provider.
2. Construct a SOAP envelope.
3. Build a SOAP header.
4. Create a Message object.
5. Invoke the message to send.
6. Retrieve the SOAP Transport object.
7. Parse the returned SOAP envelope.
8. Check for SOAP faults.

To create a message-oriented provider service, you would perform the following steps:

1. Process the SOAP envelope of the received request.
2. Generate a new SOAP envelope if the interaction is of the request-response type.
3. Extract the SOAP message body.
4. Create the message-oriented Web service interface.
5. Return the Web service response as a SOAP envelope or other data type.
6. Check and set up the SOAP exceptions: FAULT_CODE_CLIENT and FAULT_CODE_SERVER.

7.2.3 Web service application development steps

In this section we look at how to develop a Web services application from end to end. Figure 7-2 on page 181 illustrates the steps involved in a pseudo-UML collaboration diagram.
Let us walk through the diagram:

1. Develop the service provider-side Java artifact.

2. Generate the WSDL interface and implementation document from the programming language artifact, using a bottom-up approach. Alternatively, you can start with the definition of the WSDL specification (1’), and generate server-side Java from it (2’), using a top-down approach.

3. Generate the server-side SOAP deployment descriptor from the WSDL specification.

4. Publish/locate the WSDL document. When using dynamic discovery:
   a. Publish, unpublish, and update service registrations in the UDDI registry.
   b. Find service interface registrations in the UDDI registry.

   For static discovery, the obtain WSDL document using e-mail or file transfer, for example.

5. Generate the client-side SOAP stub and optionally generate Web service test clients.
6. Invoke and execute service.
   a. Dynamic lookup of service providers in the UDDI registry (provider
dynamic requester only).
   b. Service requester-side service invocation.
   c. Transport level communication between requester and provider.
   d. Service provider-side service invocation.

Even if all specifications are human readable (XML), there is a strong need for
tools supporting these development steps because many documents with
overlapping content are involved. It would be cumbersome and error prone to
define all these files without tools.

The IBM WebSphere Studio Application Developer provides tools and wizards to
simplify most of the tasks detailed above.

7.2.4 IBM WebSphere Studio Application Developer

WebSphere Studio provides the following tools to assist with Web services
development:

- **Discover.** Browse the UDDI Business Registries or WSIL documents to locate
  existing Web services for integration. The Web becomes an extension of
  WebSphere Studio.

- **Create or Transform.** Create Web services from existing artifacts, such as
  JavaBean, enterprise beans, URLs that take and return data, DB2® XML
  Extender calls, DB2 Stored Procedures, and SQL queries.

- **Build.** Wrap existing artifacts as SOAP and HTTP GET/POST accessible
  services and describe them in WSDL. The Web services wizards assist you in
  generating a Java client proxy to Web services described in WSDL and in
  generating JavaBean skeletons from WSDL.

- **Deploy.** Deploy Web services into the WebSphere Application Server or
  Tomcat test environments using Server Tools.

- **Test.** Test Web services running locally or remotely in order to get instant
  feedback.

- **Develop.** Generate sample applications to assist you in creating your own
  Web service client application.

- **Publish.** Publish Web services to a UDDI v2 Business Registry, advertising
  your Web services so that other businesses and clients can access them.
For more information, see the *WebSphere Version 5 Web Services Handbook*, SG24-6891, publication, or the WebSphere Studio help view on application development documentation.

In Chapter 10, “Web services scenario” on page 281, we will detail how we develop a sample PDK Web services application using IBM WebSphere Studio Application Developer.

### 7.2.5 WebSphere V5.0 support for Web services

Integration of SOAP support in WebSphere V5.0 provides both SOAP server and client application environments. It enables WebSphere applications to send and receive SOAP messages, and leverage WSDL.

Integration of UDDI4J provides a Java interface to UDDI registries. This enables WebSphere applications to communicate with UDDI-compliant registries to publish and find Web services.

Web services deployed on the WebSphere platform can utilize platform strengths such as security, transaction monitoring, and trace/debug functions.

**WebSphere SOAP support**

Apache SOAP Version 2.3 is integrated into WebSphere Application Server V5.0. Apache SOAP Version 2.3 is a Java-based implementation of the SOAP 1.1 specification with support for SOAP with attachments. SOAP with attachments allows binary data, such as images, to be passed with a SOAP message. WebSphere Application Server V5.0 allows you to expose the following artifacts as SOAP services:

- Standard Java classes
- Enterprise JavaBeans
- Bean Scripting Framework (BSF) supported scripts
- DB2 stored procedures

WebSphere provides tools to assist you with packaging and deploying these artifacts as Web services. When deploying your Web services in WebSphere, you can choose to enable the XML-SOAP administration tool, which allows you to manage your SOAP-enabled services.

WebSphere Application Server also contains an implementation of the security extensions for SOAP. These security extensions provide secure connections and enable digitally signed messages.
WebSphere UDDI support
UDDI4J is incorporated in IBM WebSphere Application Server base V5.0. UDDI4J is an open-source Java class library that provides an API to interact with a Universal Description, Discovery and Integration (UDDI) registry. UDDI4J contains an implementation of the client side of UDDI (everything your application needs to publish, find, and bind a Web service). It also includes the source code and the complete Javadoc for the APIs. IBM WebSphere Application Server Network Deployment V5.0 includes a UDDI Version 2 registry.

Web Services Gateway
The Web Services Gateway is provided with IBM WebSphere Application Server Network Deployment V5.0. This gateway can be seen as a kind of proxy that acts as an additional layer between Web service client and Web service provider. The gateway is useful for enabling a flexible way for calling Web services located in an intranet from the Internet, as well calling Internet Web Services from the intranet. Another function of the gateway is the possibility for protocol switching and security for Web Service calls.

WebSphere XML support
WebSphere Application Server V5.0 ships with Apache XML4J Version 4.0 (Xerces Version 2.0). This is a DOM2/SAX2/JAXP1.1 compatible, name space-aware XML parser, as required for Apache SOAP.

WebSphere SOAP EAR enabler tool
The SOAP Enterprise Archive enabler tool can be used to enable an application EAR file to use the SOAP environment on WebSphere. The SoapEarEnabler tool in the WebSphere <WAS_HOME>/bin directory takes the EAR file and the SOAP deployment descriptor file, and creates a SOAP-enabled EAR file to install in WebSphere. The SOAP deployment descriptor describes the service provided by the Web services application. The deployment descriptor contains:

- The identifier used by the Web service requesters
- The operations available from the service
- The class that implements the Web service

7.2.6 Other Web services packages from IBM
IBM provides several other packages supporting the development and hosting of Web services:

- WebSphere SDK for Web Services (WSDK) provides developer tools and a run-time environment for designing and executing Web service applications that are consistent with the industry-standard platform for Web services and the emerging standard for interoperability defined by the Web Services
Interoperability Organization (WS-I). Such applications can discover and collaborate in business transactions without programming requirements or human intervention. WSDK is based on Axis but does not expose the Axis programming model; it uses the WebSphere programming model based on JAX-RPC (JSR 101) and Web services for J2EE (JSR 109).

Axis is basically Apache SOAP 3.x, redesigned from scratch around a streaming model that uses SAX internally instead of DOM. The intention is to create a more modular, more flexible, and higher-performing SOAP implementation, relative to Apache SOAP 2.x.

- The Web Services Toolkit (WSTK) delivers early implementations of new Web services technology to developers, and acts as a consolidator of Web services technologies from various IBM development and research labs. Unlike WSDK, the WSTK does not have an embedded application server or private UDDI registry. The WSDK is therefore ideal for running services developed with WSTK. WSTK exposes the Axis programming model.
- The WebSphere Web Services for J2EE Technology Preview supports emerging Java Web services standards, such as Java API for XML-based RPC Version 1.0 (JAX-RPC) and Web services for J2EE. It also contains an implementation of the emerging WS-Security standard. It does not come with an application server and must be installed on top of WebSphere Application Server V5.0. The technology preview is based on Axis but does not expose the Axis programming model (same as for WSDK).

You can use WSDK without WSTK to develop industry-standard Web services applications, or you can use it with WSTK if you want to explore the latest emerging technology developments in Web services.

For information on these Web services packages see:
- IBM Redbook *WebSphere Version 5 Web Services Handbook*, SG24-6891
- IBM developerWorks article IBM *WebSphere SDK for Web Services (WSDK) Version 5.0*
  
- IBM alphaWorks Web Services Toolkit page
  
- IBM WebSphere Developer Domain Web Services Technology Preview
  
7.3 Development guidelines for J2EE Connectors

In this section we take a look at development guidelines for J2EE Connectors. Using IBM WebSphere Studio Application Developer, there are two ways to develop J2EE Connector applications:

- Using CCI
  
  An application component uses Common Client Interface (CCI), which is provided by a resource adapter. This is a standard way of developing J2EE Connector applications regardless of the development tools.

- Using the Enterprise services toolkit
  
  Using IBM WebSphere Studio Application Developer Integration Edition or WebSphere Studio Enterprise Developer, you can develop a J2EE Connector application as an Enterprise Service.

7.3.1 Creating a J2EE Connector application using native CCI

You can implement J2EE Connector connectivity using native CCI in your application with the following steps:

1. Configure your J2EE Connector resource adapter and connection factory in your integrated development environment.

2. Create an input and output Record class.
   
   These records implement javax.resource.cci.Record and perform the input and output conversions between application Java data structures and enterprise tier data structures. You can manually code classes using the CCI record framework or you can use a tool such as the VisualAge for Java Enterprise Access Builder. The Enterprise Access Builder allows you to create a CCI record from an existing C or COBOL structure.

3. Get an instance of the required J2EE Connector ConnectionFactory (usually through a JNDI lookup).

4. Get an instance of J2EE Connector Connection from the connection factory.

5. Create instances of input and output Record using classes developed in step 2.

Note: The basic configuration of IBM WebSphere Studio Application Developer does not provide a tool to import a C or COBOL structure into a CCI record. With the Enterprise Services toolkit in IBM WebSphere Studio Application Developer Integration Edition, you can import a C or COBOL structure into an enterprise service definition.
6. Create an Interaction instance from the Connection.
7. Create an InteractionSpec and set the required properties.
8. Execute the Interaction, passing the InteractionSpec, input Record, and output Record.

You can find detailed development steps for our sample J2EE Connector scenario in Chapter 11, “J2EE Connector Architecture scenario” on page 333.

### 7.3.2 Enterprise Services overview

This section provides an overview of Enterprise Services and how they are developed using IBM WebSphere Studio Application Developer Integration Edition.

#### Enterprise Services toolkit

IBM WebSphere Studio Application Developer Integration Edition includes a new set of tools and wizards, collectively referred to as the Enterprise Services toolkit. The Enterprise Services toolkit is a fully service-oriented development environment for business and enterprise application integration.

At the heart of the Enterprise Services toolkit programming model are Enterprise Services or Services for short. Services are used to model different kinds of service providers in a consistent way. Figure 7-3 shows the currently supported providers. Note that in the Enterprise Services world a Web service is just one form of service provider. J2EE Connectors are another.

![Figure 7-3 Services supported by WebSphere Studio Integration Edition](image)

#### What Enterprise Services are

Enterprise Services are used to model different kinds of service providers in a consistent way. The Enterprise Services toolkit uses the Web Services Description Language (WSDL) as its model for describing any kind of service.
In the top section of the WSDL document, you can see the abstract service interface definition. The service interface in WSDL is called a portType. portTypes consist of one or more operations with input and output. The input and output are described by messages. Service messages are typed using an XML schema.
The service location is described by a service provider-specific port extensibility element. The service implementation is described by service provider-specific extensibility elements in the binding section. The Enterprise Services toolkit supports the following service provider-specific bindings: SOAP, J2EE Connector, JavaBean, stateless session EJB, Flow, and Transform.

What WSIF is
The Web Services Invocation Framework (WSIF) provides a standard API to invoke services, no matter how or where the service is provided, as long it is described in WSDL. This API is used by tools such as WebSphere Studio Integration Edition and runtimes such as IBM WebSphere Application Server V5.0 to construct and manipulate services defined in WSDL documents. The architecture allows new bindings to be added at runtime. WSIF enables the user to move away from the native Java Connector Architecture services programming model of working directly with the J2EE Connector APIs, and towards a model where the user interacts with representations of the services. This allows the user to work with the same programming model regardless of how the service is implemented and accessed. WSIF is WSDL-driven and it provides a uniform interface to invoke services using WSDL documents.

J2EE Connector Enterprise Services
The J2EE Connector Tool Plug-in makes it possible to plug a J2EE Connector-compliant EIS resource adapter, such as the CICS ECI adapter, into the Enterprise Services toolkit provided with WebSphere Studio Integration Edition.

Using the Enterprise Services toolkit, there is no need to write any J2EE Connector code because the Service Definition wizard in WebSphere Studio Integration Edition guides you through the service definition. The wizard generates a WSDL file providing the WSIF with all the information needed to connect to the enterprise tier and to invoke the enterprise application.

7.3.3 Using Enterprise Services toolkit
You can implement J2EE Connector connectivity in your application using the Enterprise Services toolkit (provided in WebSphere Studio Integration Edition) with the following steps:

1. Create a J2EE Connector service definition from the corresponding resource adapter in the Service Provider Browser. In the Service Definition wizard you define the following:
   - Port and portType
   - Operation
   - Binding
Input and output message(s)

A message corresponds to a J2EE Connector Record. You can import the message definition from an existing C or COBOL structure.

2. Deploy the J2EE Connector service. The Service Deployment wizard allows you to deploy services into WebSphere. By default, the wizard deploys the service as a session EJB. It can also deploy the service as a SOAP service.

3. Build your application client for accessing the J2EE Connector service. Create an application EJB or servlet, for example, to access the deployed J2EE Connector enterprise service.

7.3.4 Data conversion

Data conversion is a key issue when writing Java applications to invoke CICS programs. CICS, which runs on IBM's zSeries or S/390® processors, grew up in an EBCDIC world, whereas Java is based on Unicode. Unicode is a double-byte character set derived from ASCII character sets that are used in the UNIX and PC worlds.

There are three different strategies you can use to perform data conversion:

- Conversion within Java

  The COMMAREA flowed to CICS in an ECI request object has to be a Java byte array (composed of single byte characters), whereas in Java, character data is usually stored in a String, which is Unicode. Each time you convert from a String to a byte array you convert each character from Unicode to a single-byte character and, therefore, you need to specify the encoding parameter to ensure consistent results.

  When converting from a String to a byte array, use the getBytes() method on the String class, passing the encoding of the byte array you wish to use. In this example, we specify the EBCDIC code page IBM037:

  ```java
  byte abCommarea[] = new byte[27];
  abCommarea = "abcd".getBytes("IBM037");
  ```

  When converting the byte array to a String, specify the correct encoding of the data on the String constructor as follows:

  ```java
  String strCommarea = new String(abCommarea,"IBM037");
  ```

  However, there is an alternative, which is to convert the data to ASCII within the JVM, and then convert from ASCII to EBCDIC within CICS. This is not as inefficient as it sounds, since data conversion from Unicode to ASCII is an efficient operation in Java, because it involves only the removal of the high-order byte, whereas conversion to EBCDIC requires a table lookup. This means the high cost of EBCDIC conversion can be transferred to CICS, thereby potentially improving performance within the JVM.
If the data being passed is numeric, it will be necessary to convert all integer values into a byte array before they can be passed to CICS.

- **Conversion by CICS: DFHCNV templates**

  ECI applications use the facilities of the CICS mirror program (DFHMIRS) to link to the specified user program, passing a buffer known as the COMMAREA for input and output. The CICS mirror program can invoke the services of the data conversion program (DFHCNV) to perform the necessary conversion of the inbound and outbound COMMAREA. Only if DFHCNV finds a conversion template in the DFHCNV table that matches the program name will it perform code page translation for the COMMAREA associated with the ECI request. Templates must specify either character or numeric data, since they are dealt with differently by the conversion program DFHCNV.

- **Data conversion with WebSphere Studio Integration Edition**

  If you are using the IBM WebSphere Studio Application Developer Integration Edition for developing your enterprise application, the Enterprise Services toolkit can do the conversion for you. The Enterprise Services toolkit allows the Java message structure and conversion code to be imported from the C or COBOL COMMAREA definition used in your CICS enterprise application.

Refer to the publication *Java Connectors for CICS: Featuring the J2EE Connector Architecture*, SG24-6401, for further details on CICS data conversion.

### 7.3.5 Migration to other J2EE Connector resource adapters

The J2EE Connector Architecture CCI provides a common programming interface to application component developers. When you migrate the application component to a different adapter, you only need to change the method calls that are specific to that resource adapter. The classes that are specific to a resource adapter are:

- **Input and output record classes**

  Depending on the enterprise application and resource adapter, the structure of an input record or an output record will change. You need to migrate the record classes to suit your enterprise application and resource adapter when you switch to another resource adapter.

- **ConnectionSpec**

  Refer to “Dealing with a connection” on page 341 for details.

- **InteractionSpec**

  Refer to 11.3.4, “Executing the enterprise application” on page 344“.
For example, IMSConnectionSpec has getGroupName and setGroupName, which are unique to the IMS resource adapter for specifying the IMS Group. Please refer to the resource adapter documentation for the details.

When you are developing an enterprise service using WebSphere Studio Integration Edition, all the code is generated by the tool. You should regenerate the code using a new resource adapter to migrate your application.

### 7.4 Development guidelines for JMS

In this section we cover the basics of creating a JMS application from Java coding steps to the involvement of various tools that can aid development. The aim is to provide a pointer and not necessarily complete walkthroughs of the tools. Detailed coding and tool usage information is provided in Chapter 12, “Java Message Service scenario” on page 365.

#### 7.4.1 Creating a JMS application

In this section we consider the steps necessary to add JMS connectivity to your application. To create the Java application:

1. Configure your JMS Provider and destination(s) in your Integrated Development Environment (IDE). For development you can use the internal JMS Provider, which is included in IBM WebSphere Studio Application Developer.
2. Get an instance of JMS javax.jms.ConnectionFactory (usually through a JNDI lookup).
3. Get an instance of JMS Connection from the ConnectionFactory.
4. Start the JMS Connection.
5. Get a JMS Session from the Connection object.
6. Using the Session object, create either a producer (QueueReceiver) or consumer (QueueSender) on a specified destination (IBM WebSphere MQ queue).
7. Use this producer or consumer to access the Destination.
8. Close the message consumer, session, and connection. Closing the connection will close the session and the message producers and consumers associated with it.

You can find detailed development steps for our sample JMS scenario in Chapter 12, “Java Message Service scenario” on page 365.
7.4.2 WebSphere Studio JMS development

IBM WebSphere Studio Application Developer, IBM WebSphere Studio Application Developer Integration Edition, and WebSphere Studio Enterprise Developer all support JMS development and testing. The steps required to develop JMS code in either version of Studio are the same.

WebSphere Studio also provides a WebSphere Application Server V5.0 test environment. You need to create the required JMS resources in the test environment.

We use WebSphere Studio for JMS development and testing for our sample JMS scenario in Chapter 12, “Java Message Service scenario” on page 365.

7.4.3 IBM WebSphere MQ client or server

With most middleware-oriented software, there are client and server components. The client is usually a smaller piece of software that provides local access to the remote server. The server implements all of the functionality, and the client provides a relatively light facade for accessing the server’s function.

The WebSphere MQ client provides access to all of the WebSphere MQ API and is typically used where there are limited hardware or system administration requirements. The WebSphere MQ client is also able to connect to multiple queue managers on different platforms.

The server provides richer administration functionality to take advantage of the full suite of JMS functionality, such as failover support or scalability configuration options.

During development, the client option is useful where machine resources are limited. Any complex changes to the WebSphere MQ configuration will, however, require the server version. The embedded WebSphere JMS Provider available in the WebSphere Studio test environment can also be used during development.

Tip: Remember to properly manage the JMS resources (opening and closing) and also to properly handle timeout if using EJBs as message consumers (see 6.4.6, “Where to implement message producers and consumers” on page 161).
7.5 Development guidelines for XML messaging

In this section we consider the design and development of XML message structure. The aim is to provide pointers rather than a complete walkthrough of the tools and steps to be taken.

7.5.1 Creating an XML Message

Before creating your own message structure from scratch, consider the XML schemas and DTDs that already exist for many business communities. More such domain-specific language catalogs are currently being defined by various industry consortium and standards bodies. Standard schemas make it easier for your application to communicate with other services or applications. A good place to start looking is http://www.xml.org.

If you expect your XML-based messaging system to evolve into a large system between several applications it might be worth considering a standardized message model. Standardizing the message model gives you the flexibility, for example, to implement new XML messages for new application versions. A generic model can also protect the message system from too many transformations, which takes a lot of time. A generic message model will end in a generic data model of all the data that is used within a company.

A few XML guidelines are:

- Use descriptive names in XML documents.
- Group elements in related sets and decrease the number of choices that can be made.
- Create a collection element for the context of repeated elements.
- Add elements that can be used in a later stage.
- Separate metadata from the real content. For example use head and body elements.
- Include a message version number in the message metadata.
- With large amounts of metadata put this data, in a different document.

7.5.2 Creating Java classes

Once you have designed and implemented an XML message structure you can create Java classes from this structure. These classes can be used for creating and validating the messages in a Java application.
There are several products on the market for generating Java classes from the DTD or XML schema. It is preferable to use a generating tool that is based on a standard DOM or SAX implementation.

### 7.5.3 WebSphere Studio XML tools

IBM WebSphere Studio Application Developer provides a comprehensive visual XML development environment. The tool set includes components for building DTDs, XML schemas, XML, and XSL files.

The following XML editor tools are available:

- The XML editor is a tool for creating XML files, either from scratch, existing DTDs, or existing XML schemas.
- The DTD editor is a tool for creating DTDs. You can generate XML Schema files and JavaBean for creating XML instances.
- The XML schema editor is a tool for creating XML Schemas. You can generate DTDs and JavaBean for creating XML instances.
- The XSL editor can be used to create new XSL files or to edit existing ones.
- The XML to XML mapping editor is a tool used to map one or more source XML files to a single target XML file.
- The RDB to XML mapping editor is a tool for defining the mapping between one or more relational tables and an XML file.

The following XML wizards are also available:

- XPath expression wizard to create XPath expressions
- XSL debugging and transformation tool to apply XSL files to XML files, transforming them into new XML, HTML, or text files
- XML and SQL query wizard to create an XML file from the results of an SQL query or take an XML file and store it in a relational table

### 7.6 Design modeling using Rational XDE

Rational XDE, or Rational Extended Development Environment, combines your design and Java development into a tightly integrated environment. The IBM WebSphere Studio Application Developer and Rational XDE allow you to work in a single Java environment, avoiding the need to switch between tools outside of your environment.
In this section we provide some simple examples to help you become familiar with design modeling using Rational XDE and IBM WebSphere Studio Application Developer. We describe how to use the design tools for the following basic tasks:

- Creating a model in Rational XDE
- Modeling behavior
- Modeling structure
- Round-trip engineering models and code

For more information on Rational XDE please refer to:

http://www.rational.com/products/xde/

### 7.6.1 Creating an XDE model

There are three steps needed to create a Rational XDE model:

1. Create a project for the model.
2. Create a model from one of the templates.
3. Create packages and classes in the model.

**Creating a project**

In this example, we create a Java project for our Rational XDE model. You can also create XDE models in other project types, such as Web projects or modeling projects. To create a Java project:

1. Start IBM WebSphere Studio Application Developer.
2. Click **File -> New -> Project** to start the New Project wizard.
3. In the left pane of the New Project wizard, click **Java**. In the right pane click **Java Project**, then click **Next**.
4. In the Project Name field, type a name for your new project. We called our project "Sample Java Project".
5. Click **Finish**.

**Creating a model from one of the templates**

Rational XDE provides a number of model templates for creating new models. To create a Java model:

1. Open the Modeling perspective. You can open a perspective from the WebSphere Studio main menu by selecting **Perspective -> Open -> Other**.
2. In the Navigator view, right-click the destination project name, **Sample Java Project** in our example, and select **New -> Model** from the pop-up menu.
3. In the New XDE Model window:
   a. Make the required selections in the File types and Templates lists. We selected **Java** file type and **Java Code Model** template.
   b. Enter the File name. We accepted the default Java Code Model.
   c. Click **Finish** to create the model.

Figure 7-5 shows our new model in the Model Explorer view. This view displays open modeling files and the packages, diagrams, classes, relationships, and other model elements that they contain.

![Figure 7-5 WebSphere Studio/Rational XDE Modeling perspective](image)

**Creating a package and classes in the model**

Next we create classes named Customer and Account in a package called `sample.package`:

1. In the Modeling perspective, Model Explorer view, create a new package:
   a. Right-click the destination model, **Java Code Model** in our case, and select **Add Java -> Package** from the pop-up menu.
   b. Enter a name for the package. We named our package `sample.package`. 
2. Create a new class:
   a. Right-click the destination package, **Package1** in our case, and select **Add Java -> Class** from the pop-up menu.
   b. Enter a name for the class. We named our first class **Customer**.
3. Create further packages/classes as required. We created another class named **Account** in the **Package1** package.
4. Add the required methods and attributes to the new classes. We added a method called **Operation1** to the **Account** class by right-clicking the **Account** class and selecting **Add Java -> Method** from the pop-up menu.
5. Add any required relationships between classes. We added an **Account** class use relationship to the **Customer** class as follows:
   a. Right-click the **Customer** class and select **Collection Editor** from the pop-up menu.
   b. In the Collections window, click the **Relationships** tab.
   c. Insert a relationship by clicking the down arrow on the far right and selecting the required association from the drop-down list. We selected the **Usage** relationship.
   d. In the Select Element window, navigate to and select the required element. We selected the **Account** class. Click **OK** to create the association.
   e. Click **Close** to return to the Model Explorer.

You can see our new package and classes in Figure 7-6.

![Figure 7-6 Model Explorer view of new package and classes](image-url)
7.6.2 Modeling behavior

The dynamic behavior of components can be modeled using a sequence diagram. To add a sequence diagram to the model we created in 7.6.1, “Creating an XDE model” on page 196:

1. In the Modeling perspective, Model Explorer view, right-click the destination model, Java Code Model in our case, and select Add Diagram -> Sequence: Instance from the pop-up menu.

2. Enter a name for the sequence diagram. We named our sequence diagram Diagram1. The new sequence diagram is opened in the diagram window.

3. To add classes to the diagram, use your mouse to drag the required classes from the Model Explorer to the diagram window. For our example, drag and drop first the Customer class and then the Account class from the Model Explorer to the diagram window.

4. To add a message to your interaction diagram:
   a. Click Message under the UML Sequence tab in the Toolbox view.
   b. In the diagram window, drag and drop from the requesting class to the replying class to add the message. We added a message from Customer to Account.
   c. In the diagram window, double-click the message name and select the required method. We selected Account::Operation1.

You can see our simple sequence diagram in Figure 7-7. Notice that CollaborationInstance1 and InteractionInstance1 have been created automatically. Two objects have been attached to CollaborationInstance1, and one message has been attached to InteractionInstance1.

![Sample sequence diagram](image)

Figure 7-7 Sample sequence diagram
7.6.3 Modeling structure

The static relationships between components can be modeled using a class diagram. To add a class diagram to the model we created in 7.6.1, "Creating an XDE model" on page 196:

1. In the Modeling perspective, Model Explorer view, right-click the destination model, Java Code Model in our case, and select Add Diagram -> Class from the pop-up menu.

2. Enter a name for the class diagram. We named our class diagram Diagram1. The new class diagram is opened in the diagram window.

3. To add classes to the diagram, use your mouse to drag the required classes from the Model Explorer to the diagram window. For our example, drag and drop the Customer class and the Account class from the Model Explorer to the diagram window.

You can see our sample class diagram in Figure 7-8. Notice that the use relationship from Customer to Account has been automatically added to the diagram.

![Sample class diagram](image)

Figure 7-8 Sample class diagram

7.6.4 Round-trip engineering models and code

The Rational XDE round-trip engineering process enables you to model your application, then generate the code elements of a complete Java application framework based on that model, then modify and implement the code as necessary, and then reverse engineer that code back into the model.
Round-trip engineering starting with a model

To round-trip engineer your application starting with a model:

1. Open the Java code model in the Model Explorer, right-click the model (Java Code Model in our example) and select Generate Code from the pop-up menu.

2. If the project root has not been set, you will be prompted to specify the source root directory. The Java source files that are generated are placed relative to the source root directory. This is not optional; Rational XDE requires a source root for model elements.

3. Verify that the generated source code files now exist in your project, for our example:
   a. In the Model Explorer, right-click Account.java and click Browse Code.
   b. In the Editor window, verify that the Account class has the Operation1 method.

4. In the Model Explorer add a new method named Operation2 to the Account class. To add a new method, right-click the Account class and select Add Java -> Method from the pop-up menu.

5. Right-click the model in the Model Explorer and click Generate Code once again.

6. In the Editor window verify that the Account.java source file now contains the Operation2 method.

You can see the source code for our generated class in Figure 7-9.

Figure 7-9   Generated class source code
Round-trip engineering starting with code

To round-trip engineer your application starting with Java source code:

1. Open the destination Java code model in the Model Explorer, right-click the model (Java Code Model in our example) and select More Java Actions -> Add/Remove Modeled Files from the pop-up menu.

   **Note:** If you are using our example Java Code Model project from previous steps, delete all model elements and diagrams from the model project first. If not, just create a new, empty model project.

2. If the project root has not been set, you will be prompted to specify the source root directory. The Java source files to be round-trip engineered must be placed relative to the source root directory. This is not optional; Rational XDE requires a source root for model elements.

3. In the Add/Remove Modeled Files window, click the Add Recursively button to add all the Java source files under the project folder. The Package1 package and the Account and Customer classes should appear in the Files tree.

4. Click OK to start the reverse engineering process. Model elements from the Java source files are now displayed in the Model Explorer.

5. Add a new method to the Account class:
   a. In the Model Explorer, right-click Account.java and click Browse Code.
   b. In the Editor window, add a new method named Operation2, as highlighted in Figure 7-10 on page 203.

6. Right-click the Account class in the Model Explorer and click Reverse Engineer.

7. In the Model Explorer verify that Account class now contains the Operation2 method.

   You can see the generated model elements circled in Figure 7-10 on page 203. You can now add these reverse-engineered model elements to your model diagrams, as we described in 7.6.2, “Modeling behavior” on page 199, and 7.6.3, “Modeling structure” on page 200.
Chapter 7. Application development

Figure 7-10  Generated model elements
Systems management

Systems management is an important consideration in application design and continues to be a factor in day-to-day business operations. Systems management covers many areas, typically involving:

- Application management
- Performance monitoring
- Availability management
- Security management
- Disaster recovery
- Operating system and network administration
- Asset management
- Software distribution
- Problem reporting
- Change management

Many of these are general considerations that span entire enterprise operations. We do not cover all of these topics here, but focus on system management techniques specific to Web services, J2EE Connectors and CICS, and JMS and WebSphere MQ.

Security guidelines and considerations for IBM WebSphere Application Server V5.0 can be found in the publication *IBM WebSphere V5.0 Security: WebSphere Handbook Series*, SG24-6573.
8.1 Web services system management

Planning for systems management in a Web services environment is similar to the planning needed for any other distributed system, but we face some unique challenges in Web services technology.

According to a Hurwitz Group study, security is the biggest challenge for the enterprise application of Web services. Providing a comprehensive model of security functions and components for Web services requires the integration and unification of currently available concepts and processes (policy, risk, trust) and technologies (secure messaging) with the evolving security requirements of future applications. It requires coordinated efforts by platform vendors, application developers, network and infrastructure providers, and customers.

Web services are based on plain text XML messaging that could potentially be vulnerable to interception. In addition, Web services potentially allow transactions beyond firewalls and enable external entities to invoke internal applications or access sensitive information.

Another problem is that since there is no confirmed delivery (at least not explicitly built into the Web services specification) what happens if we do not get a response to a particular request? How do we know if the Web service provider is not working, just slow, or just did not receive our request?

We have no explicit way of determining if a service on a remote server (not under our control) is working. This becomes an availability issue on our local server. At the application level, we need a way to respond to unavailable systems.

Some other areas of system management to consider with Web services include:

- Firewall considerations
  - From the perspective of the firewall, an RPC router is just another servlet that is accessed through HTTP or HTTPS.

- Load-balancing considerations
  - The same considerations and techniques as for normal servlets apply, with one exception. By default, there is no hot failover for session type Web services. If failover is a requirement, it should be handled on the next layer; for example, in a servlet acting as Web service implementation, or by using a redundantly configured EJB session facade.
  - A stateless request style Web service instance can be pooled and load balanced because it is a Java object in the JVM of the application server.
  - Failover is also supported for request-style Web services.
If session style is used, scalability is best. However, the Web service implementation class must be serializable and small.

If application style is used, load balancing is not an issue, because there is only one instance whose lifetime is identical to the one of the application server hosting it.

Other Quality of Service considerations
- Clearly define and document achievable QoS standards or requirements.
- Service providers should try to make key measurements available online.
- Use existing standards or proposed specifications as design guidelines.

8.1.1 Security considerations

Some application implementers have been slow to adapt Web services because of security concerns. As an emerging technology, it is believed that Web services have not been sufficiently hardened for production enterprise applications. While it is true that the security model for Web services is still evolving, we examine some techniques that are available today for creating secure Web services implementations. For the purposes of this discussion, we assume that the Web service is being built on top of a J2EE framework and not some other underlying technology. Based on using J2EE, existing transport-layer security technologies such as Secure Sockets Layer (SSL), Transport Layer Security (TLS), and Internet Protocol Security (IPSec), can be used with Web services. In addition, security procedures and mechanisms can be applied to the components and individual artifacts that comprise a Web service. An example would be to use the new XML digital signatures and encryption with XML messages that are part of a Web service request or response.

Some of the strengths and appeals of Web services are at the same time a challenge for application architects wanting to adhere to enterprise security standards. One of the motivations for developing the Web service model was a requirement to build a ubiquitous programming model for application-to-application communication. This means that as an application developer implementing a Web service requester, you do not know where the Web service provider application resides. It can be within your own enterprise or on a remote server anywhere on the Internet. In addition, the Web service provider can be implemented using any programming language and can reside on an application server on any platform. The Web service provider can access a new application or an existing legacy application to get the data that it will return. In addition, messaging in a Web services environment adds new dimensions to security considerations because of the more complex end-to-end round trip paths. There are also more complex transaction scenarios in the publish and bind framework of Web services as opposed to standard point-to-point messaging.
We do know, however, that any Web service provider must be built on Web service standards and communicated over standard protocols. A central tenet of Web services development is to build upon what is already there and implement by combining existing technologies. We recommend taking the same approach to Web services security.

8.1.2 Creating Secure Web services

When creating Web services based on existing applications, security cannot be compromised by the addition of the Web services.

Web services require an open security service model, based on standards, that can serve a heterogeneous “trust domain.” A Web services security model should support protocol-independent declarative security policies that Web service for J2EE providers can enforce, and descriptive security policies attached to the service definitions that clients can use in order to securely access the service.

Let us examine various security services and explore how the goals of each can be achieved when designing and deploying a Web service.

Authorization

Based on the authorization policies specified for the services, the J2EE container can enforce the authorization policies. The design and implementation of authorization mechanisms are no different with Web services than they would be with any other Web application.

In an enterprise security model, each application server and middleware element performs authorization for its resources (EJBs, servlets, queues, tables, etc.). The J2EE authentication/delegation model ensures that the user identity is available when requests are processed. On successful authentication, identity of the authenticated user is associated with the request. Based on the identity of the user, authorization decisions are made. J2EE application servers using the J2EE 1.3 security model only allow authorized access to EJBs and servlets/Gasps. Authorization to Web services implemented as JAX-RPC service endpoints will be based on the servlet/JSP security model.

Authorization can be implemented at the application level or at the server level.

Authentication

Since the Web services architecture builds on existing component technologies, authentication between enterprises is no different from current approaches to authentication across distributed systems. Authentication over Web services is the area that has received the most attention and discussion in regards to Web
services security. Several approaches have been suggested. Keep in mind that there is no accepted standard in this area and it is impossible to even identify a best practice at this time. We will present a few approaches to the topic in this section.

In order to secure access to a Web service, it may be required that the user requesting the service needs to be identified. In the J2EE security model, the login configuration policy specifies how user information is to be retrieved (for example, HTTP Basic) and the operational environment-specific policies will dictate how it gets authenticated (for example, Kerberos authentication mechanism). In the Web service security model, the login configuration should not only address authentication of immediate clients (clients submitting the request directly to the service), but also address indirect clients (an end client's identity may be part of the request, which can traverse through many intermediaries). This information is necessary for both the client (to submit the credential information in a format understood by the server) and the server (to retrieve the credential information from the transport layer or the message itself).

One approach is to place a user's identification in the header of the SOAP transmission. This identification could optionally be digitally signed by a third party. The user name could be replaced by the Kerberos credential of the user. Using this method, the J2EE container itself can perform authorization based on the user's identity. This transport-dependent approach to authentication will allow the downstream authorization of the message to be transport-neutral. In this case the user's identity information is present in the message itself.

One suggestion is for the service provider infrastructure to take care of defining the security policies. Then the Web service developers need not know how to programmatically enforce all possible authorization policies for the different servers in which the service may be deployed.

In order to maintain a level of abstraction of policies during service installation and to give the ability to specify application security policies during Web service development, Web service security policies should be expressed in terms of roles. As in J2EE, a security role is a semantic grouping of permissions that a given type of user must have in order to successfully use the Web.

It is expected that in the future, such message-level security mechanisms will become part of the supporting specifications for Web services. It is also anticipated that in future Web services, specifications methods for placing authentication policy in the service definition (WSDL) itself will be described. Based on that service definition, a client would supply appropriate credentials. If the container had policies for the service, then they must be referenced and used.
**Integrity and confidentiality**

In general, integrity and confidentiality issues in Web services today are based on existing J2EE support such as HTTPS. Message senders may also want to ensure that a message or parts of a message remain confidential and that it is not modified during transit. When a message requires confidentiality, the sender of the message may encrypt those portions of the message that are to be kept private using new XML encryption proposals. In addition, some level of integrity of a message can be achieved by using XML digital signatures to ensure that the message is not modified during transit. Both XML encryption and XML digital signatures are draft specifications to W3C as of the writing of this book.

**Auditing and non-repudiation**

Non-repudiation guarantees that the sender is authorized to send a message and cannot deny that the message was sent. An auditing capability is usually required to achieve non-repudiation. Auditing can capture a secure record of events that might affect the security of a system, such as login attempts. Currently neither the Web services nor the SOAP specifications address auditing or non-repudiation. To achieve non-repudiation in a Web services environment there are a few approaches that can be pursued:

- Use W3C SOAP Digital Signature Extension. For more information, see: http://www.w3.org/TR/SOAP-dsig/
- Place security information in the SOAP header.
- Use XML Digital Signatures as evidence of the XML message exchange.

**8.1.3 Web service security specifications**

In April 2002, IBM and Microsoft proposed a technical strategy and roadmap for “addressing security within a Web service environment.” The Web services security specifications define a comprehensive Web service security model that supports, integrates and unifies several popular security models, mechanisms, and technologies (including both symmetric and public key technologies) in a way that enables a variety of systems to securely interoperate in a platform- and language-neutral manner.

The Web services security specification provides a broad set of specifications that cover security technologies including authentication, authorization, privacy, trust, integrity, confidentiality, secure communications channels, federation, delegation, and auditing across a wide spectrum of application and business topologies. These specifications provide a framework that is extensible, flexible, and maximizes existing investments in security infrastructure. By leveraging the natural extensibility that is at the core of the Web services model, the specifications build upon foundational technologies such as SOAP, WSDL, XML
Digital Signatures, XML Encryption, and SSL/TLS. This allows Web service providers and requesters to develop solutions that meet the individual security requirements of their applications.

As shown in Figure 8-1, this set includes a message security model (WS-Security) that provides the basis for the other security specifications. Layered on this, we have a policy layer that includes a Web service endpoint policy (WS-Policy), a trust model (WS-Trust), and a privacy model (WS-Privacy). Together these initial specifications provide the foundation upon which we can work to establish secure interoperable Web services across trust domains.

![Figure 8-1 The evolving WS-Security roadmap](image)

Building on these initial specifications we will continue to work with customers, partners, and standards organizations to provide follow-on specifications for federated security, which includes secure conversations (WS-SecureConversation), federated trust (WS-Federation), and authorization (WS-Authorization).

Additionally, IBM and Microsoft are committed to working with organizations like WS-I on interoperability profiles.

The combination of security specifications, related activities, and interoperability profiles will enable customers to easily build interoperable secure Web services.

For more information, see the IBM developerWorks article Security in a Web Services World: A Proposed Architecture and Roadmap:


**WS-Security**

Web Services Security (WS-Security) Version 1.0 was jointly developed by IBM, Microsoft, and VeriSign, and was released in April 2002. It was submitted to OASIS by 18 companies, and now involves over 50 companies.

WS-Security describes enhancements to SOAP messaging to provide quality of protection through message integrity and message confidentiality. Also, this
specification defines how to attach and include security tokens within SOAP messages. Finally, a mechanism is provided for specifying binary encoded security tokens (for example, X.509 certificates). These mechanisms can be used independently or in combination to accommodate a wide variety of security models and encryption technologies.

WS-Security provides a general-purpose mechanism for associating security tokens with messages. No specific type of security token is required by WS-Security. It is designed to be extensible (such as to support multiple security token formats). For example, a requester might provide proof of identity and proof that they have a particular business certification.

Message integrity is provided by leveraging XML Signature in conjunction with security tokens (which may contain or imply key data) to ensure that messages are transmitted without modifications. The integrity mechanisms are designed to support multiple signatures, potentially by multiple actors, and to be extensible to support additional signature formats. The signatures may reference a security token.

Similarly, message confidentiality is provided by leveraging XML Encryption in conjunction with security tokens to keep portions of SOAP messages confidential. The encryption mechanisms are designed to support additional encryption technologies, processes, and operations by multiple actors. The encryption may also reference a security token.

Finally, WS-Security describes a mechanism for encoding binary security tokens. Specifically, the specification describes how to encode X.509 certificates and Kerberos tickets, as well as how to include opaque encrypted keys. It also includes extensibility mechanisms that can be used to further describe the characteristics of the security tokens that are included with a message.

For more information, see the IBM developerWorks article Web Services Security (WS-Security):


8.1.4 Web services component security

To fully examine the security implications of Web services, we must look at the individual components that make up Web services. As described in 6.2, “Design guidelines for Web services” on page 111, Web services can be composed of the following technologies:

- SOAP and HTTP
- XML (ebXML)
- WSDL
- UDDI
To fully assess the security implications for Web services, we must examine the security implications of each of these technologies.

**SOAP**
The requirements for building security into SOAP messaging are no different from the general security requirements for messaging that are already well understood. As previously mentioned, messaging in a Web services environment adds more complexity to security considerations as opposed to standard point-to-point messaging.

Because of the unique nature of Web services, some additional vulnerabilities in SOAP messaging are exposed:

- Using SSL to establish point-to-point sessions for secure SOAP messaging is vulnerable to message copying and replay.
- The tightly coupled request/response manner in which SOAP is implemented over the Web opens the door to messages being duplicated, resent, or repudiated.

**Proposed solutions to SOAP security**
SOAP can potentially be used in combination with a variety of other protocols; however, the only bindings we discuss here are SOAP in combination with HTTP and HTTP Extension Framework. The usage of a particular protocol does not change the fundamentals of the security model but may change the particular implementation. SOAP can be considered as another layer between the transport layer (for example, HTTP) and the application layer. As such, it is a convenient place for conveying message meta-information and potentially security metadata. Because of this and the fact that SOAP underlies all the other Web services layers, it has been a focus of attention in addressing Web services security. Currently, there is a submission to the W3C called SOAP-DSIG. SOAP-DSIG is an industry effort to get XML digital-signature extensions incorporated directly into the SOAP specification.

While awaiting approval of new SOAP security standards, transport security for Web services messaging remains SOAP over HTTP/SSL or TLS. Two other transport protocols that can be used to enhance Web services security are Internet Protocol Security (IPSec) and Blocks Extensible Exchange Protocol (BEEP), or more specifically, the proposed specification to the IETF for a SOAP binding to BEEP.

XML
Several new XML security specifications are working their way through three standards bodies: The World Wide Web Consortium (W3C), Internet Engineering Task Force (IETF), and Organization for the Advancement of Structured Information Standards (OASIS). A few that might have particular application for Web services include XML signature, XML encryption, XKMS, and SAML.

We take a brief look at these next. For a complete discussion, see:

http://sunonedev.sun.com/building/tech_articles/xmldata.html

XML signature
XML Signature Syntax and Processing is a specification for digitally signing electronic documents using XML syntax. Besides defining how the signature should be represented, the specification also addresses how the signature should be computed and verified. According to the W3C, “XML Signatures provide integrity, message authentication, and/or signer authentication services for data of any type, whether located within the XML that includes the signature or elsewhere.”

A key feature of the protocol is the ability to sign parts of an XML document rather than the document in its entirety. This is necessary because an XML document in Web services transactions might contain elements that will change as the document is passed along or various elements that will be signed by different parties.

XML encryption
XML encryption will allow encryption of digital content, such as Graphical Interchange Format (GIF) images, Scalable Vector Graphics (SVG) images, or XML fragments. XML Encryption allows the parts of an XML document to be encrypted while leaving other parts open, encryption of the XML itself, or the super-encryption of data (that is, encrypting an XML document when some elements have already been encrypted).

XML Key Management Specification (XKMS)
XKMS establishes a standard for XML-based applications to use Public Key Infrastructure (PKI) when handling digitally signed or encrypted XML documents. XML signature addresses message and user integrity, but not issues of trust that key cryptography ensures. XKMS attempts to relieve the incompatibility currently inherent in PKI by delegating PKI functions to systems outside of those processing XML documents. Developers can merely use a standard interface for these processes and not worry about the underlying PKI, which will also free up enterprises to use different PKI systems that will still be able to share information. This is especially important because certificate processing is expected to be too complex for lightweight XML applications.
**Security Assertion Markup Language (SAML)**

XKMS solves the problem of authenticating and authorizing users for a given application, but Web services will also require trust assertions between applications and organizations. That is where SAML comes in.

Security Assertion Markup Language is the first industry standard for secure e-commerce transactions using XML. SAML is being developed to provide a common language for sharing security services between companies engaged in business-to-business and business-to-consumer transactions. SAML standardizes the exchange of user identities and authorizations by defining how this information is to be presented in XML documents, regardless of the underlying security systems in place.

**XML security policy documents**

Security policies themselves could be identified in an XML document. These documents could have sections that are encrypted and the documents themselves could be digitally signed and distributed.

**WSDL**

WSDL has the potential to be extended to include the definition of the context needed by the business execution environment, including security. Without these extensions, users must make assumptions about security in the runtime environment of a Web service. Defining these security assertions in XML will allow us to have a common interpretation of security attributes in different implementations. It will also facilitate searching.

**UDDI**

One reason for the delay in widespread adoption of UDDI is the lack of security standards that would allow companies to restrict Web services access information to trusted partners. The third version of the UDDI standard will include security specifications.

### 8.2 J2EE Connectors and CICS system management

This section provides system management guidelines for J2EE Connector-enabled applications and the underlying CICS environment. We look at the following topics:

- Logging and tracing
- Performance monitoring and tuning
- Scalability and availability considerations
- Security considerations
8.2.1 Logging and tracing

It is often helpful to examine log and trace files when your application experiences J2EE Connector errors or problems.

Application logging
It is always important for applications to record their activity to a logging facility. When you write a log to the standard output file or standard error file, the application server will record it to the corresponding log files.

Connection factory trace
WebSphere V5.0 provides various levels of information, from high-level messages in the console to more detailed server execution logs and comprehensive tracing.

Connection factory classes can be traced using the WebSphere Application Server trace service. The trace level can also be set as a connection factory property in WebSphere Administrative Console.

Connection factory tracing is often not particularly helpful when debugging the CICS interaction. CICS TG tracing is usually the better option.

CICS TG trace
CICS Transaction Gateway (CICS TG) trace records detail activities of the CICS TG gateway daemon, such as processing of ECI requests from clients. The four levels of tracing are stack tracing, standard tracing, debug tracing, and JNI tracing. JNI tracing is usually the most useful of all traces. It is the JNI level between the Java CICS TG and the native EXCI or CICS client.

The application can enable tracing programmatically. It can also be enabled dynamically in the Gateway daemon using the TCPAdmin protocol handler, or statically as a start option. CICS TG trace is recorded in the standard output file or standard error file.

External CICS Interface trace
The External CICS Interface (EXCI) provides a programming interface for the non-CICS address space to invoke CICS programs. CICS TG utilizes EXCI to communicate with the CICS program. The CICS Transaction Gateway writes trace entries to the EXCI trace when it issues an EXCI request. The trace entries in a dump can be printed using standard z/OS utilities (GTF).
CICS trace
CICS Transaction Server provides a facility for recording CICS activity. In CICS for MVS™, there are three destinations for trace entries: Internal trace, auxiliary trace, and generalized trace facility (GTF).

8.2.2 Performance monitoring and tuning
In this section we briefly look at performance monitoring and tuning for J2EE Connectors in WebSphere V5.0.

Using Tivoli Performance Viewer
The Tivoli Performance Viewer is a graphical performance monitor for WebSphere Application Server V5.0. You can use the Performance Viewer to retrieve performance data from application servers. Data is collected continuously by application servers and retrieved as needed from within the Viewer. You can regulate the impact of data collection by using the Performance Viewer or the WebSphere Administrative Console. The Performance Viewer's graphical interface provides controls that enable you to choose the particular resources and counters to include in a view.

The JCA Connection Pools resource category provides information about J2EE Connectors, such as the number of managed connections (ManagedConnections) and the number of connection handles (Connections).

Tuning connection pooling properties
There are several parameters you can set to optimize connection pooling properties using WebSphere Administrative Console:

- Connection timeout
  This is the number of milliseconds after which a connection request is determined to have timed out and a ResourceAllocationException is thrown. The wait might be necessary if the maximum value of connections has been reached (MaxConnections). This value has no meaning if the maximum connections property has not been set.

If the connection timeout is set to a very small number such as 1, the ResourceAllocationException is thrown almost immediately after the pool manager determines that the maximum number of connections has been used. If the connection timeout is set to 0, the pool manager waits until a connection can be allocated. (In other words, it waits until the number of connections falls below the maximum connections.)
Maximum connections
This is the maximum number of managed connections that can be created by a particular ManagedConnectionFactory. After this number is reached, no new connections are created, and either the requester waits or the ResourceAllocationException is thrown. If maximum connections is set to 0, the number of connections can grow indefinitely. Maximum connections must be larger than minimum connections.

Minimum connections
The minimum number of managed connections to maintain. If this number is reached, the garbage collector will not discard any managed connections. Note that if the actual number of connections is lower than the value specified by the minimum connections settings, no attempt will be made to increase the number of connections to the minimum. Minimum connections must be less than or equal to maximum connections.

Reap time
This is the number of seconds between runs of the garbage collector. The garbage collector discards all connections that have been unused for the value specified by the unused timeout.

To disable the garbage collector, set the reap time to 0. Another way to disable the garbage collector is to set the unused timeout to 0.

Unused timeout
Number of milliseconds after which an unused connection is discarded. Setting this value to 0 disables the garbage collector.

8.2.3 Scalability and availability considerations
As shown in Figure 8-2 on page 219, there are several scalability and availability options when using J2EE Connectors to access CICS enterprise applications:

- EJB workload management, as provided by WebSphere Application Server.
- Inbound CICS TG requests (TCP or HTTP) can be workload managed using various methods.
- CICS requests (ECI or EPI) can be workload managed using CICS scalability technologies.
**WLM for EJBs**

EJBs deployed in IBM WebSphere Application Server Network Deployment V5.0 can take advantage of the WebSphere workload management (WLM) facility for EJBs. In WebSphere Application Server V5.0, workload management for EJBs is enabled automatically when clusters are created in a cell. There is no need for a special configuration to enable it. Workload management uses a WLM plug-in in the Object Request Broker (ORB) to dispatch the load.

Please refer to the publication *IBM WebSphere V5.0 Applications: Ensuring High Performance and Scalability*, SG24-6198, for details.

**WLM for CICS TG requests**

This section describes scalability and availability options between EJBs in a distributed environment (Windows or UNIX) and the CICS TG in a zSeries environment. An ECI request from a J2EE Connector resource adapter goes to the CICS TG on zSeries as an HTTP or TCP request.
Options for implementing workload managing CICS TG for z/OS requests across LPARs include:

- **IBM Load Balancer**
  
  Load Balancer is an IBM solution that provides an advanced IP level workload-balancing mechanism. The function is provided as a component of IBM WebSphere Edge Server. It can provide workload balancing for any TCP or UDP protocol, including HTTP requests to a Web server or TCP packets to an application such as the CICS Transaction Gateway. Load Balancer can be used to perform load balancing of inbound requests to CICS TG for z/OS.

- **DNS connection optimization**

  DNS connection optimization allows workloads to be distributed across multiple z/OS images. DNS connection optimization balances IP connections in a z/OS sysplex IP domain, by dynamically updating the z/OS DNS server database based on feedback from MVS WLM about the health of the registered applications. This is sometimes referred to as dynamic DNS, although this feature merely refers to the dynamic update function of the z/OS DNS server.

- **Sysplex distributor**

  Sysplex distributor is implemented in z/OS V2.10 and offers major enhancements to TCP/IP workload management in a sysplex. Balancing is enabled by using a single-cluster IP address, which routes packets onto multiple nodes. Sysplex distributor provides for close integration with the MVS WLM policy agent and service level agreements (SLAs) in making the routing decisions. This is different from Network Dispatcher, since it has to poll the WLM advisor on z/OS to update its routing information. The cluster IP address is actually a VIPA (virtual IP address) and so can be dynamically routed to another z/OS LPAR in the sysplex. This allows for failover of the cluster address.

**WLM within an LPAR**

TCP/IP port sharing provides a simple way of workload balancing HTTP requests across a group of cloned address spaces running in the same z/OS image. For our purposes these could be cloned CICS regions, CICS TG Java gateway applications, or Web servers. To enable port sharing, the address spaces are configured to listen on the same TCP/IP port number, and the SHAREPORT parameter is specified in the TCP/IP profile. As incoming client connections arrive for this port, TCP/IP will distribute them across the address spaces that are listening on the shared port. TCP/IP will select the address space with the least number of connections (both active and in the backlog) at the time that the incoming client connection request is received. This allows you to do workload balancing for incoming HTTP requests across several cloned address spaces.
The workload balancing is based entirely on the number of IP connections, and so does not take into account the individual health or capacity of any given CICS region. However, it does provide a very simple means of providing failover and workload balancing across multiple regions within an LPAR. The TCP/IP Port Sharing function is provided by the Communication Server for z/OS.

**WLM for CICS TS requests**

This section describes scalability options within a CICS world. The following functions provide workload management of requests from the CICS TG to a zSeries CICS region.

**WLM from CICS Client daemon**

When the CICS TG resides in a distributed environment, the CICS Client daemon, also sitting in the distributed environment, will kick off the CICS transaction residing on zSeries. The CICS Client daemon provides a workload management function for load balancing ECI requests. Options are provided that allow you to balance work across CICS regions using either a round-robin technique or a weighted distribution.

The CICS Client daemon also provides the ability to detect failed regions, and provides a configurable timeout period to check the status of regions that have previously failed. It does not, however, provide any form of performance agent for feedback on the status of the CICS regions. Thus, it is best viewed as a means of removing a single-point-of-failure in a listener region. Note that the CICS Client Daemon Workload Manager is not available with the CICS Client daemon on AIX and Solaris, but the exit on which it is based is provided, enabling you to implement your own customized workload manager.

**WLM from CICS TG for z/OS**

Workload management functions that are applicable when CICS Transaction Gateway resides in zSeries include:

- **External CICS Interface (EXCI)**

  EXCI provides a programming interface for non-CICS address space programs to invoke CICS programs. CICS TG utilizes EXCI to communicate with the CICS program. CICS TG provides a user-replaceable module called DFHXCURM to perform basic load balancing. It allows the destination CICS APPLID on and EXCI call to be altered and various retryable errors to be handled. This allows for basic workload balancing of EXCI calls, based on a simple availability check to be performed in this exit before the EXCI call is sent to the CICS system.
CICS multi-region operation

CICS multi-region operation (MRO) enables CICS systems that are running in the same MVS image, or in the same MVS sysplex, to communicate with each other. MRO does not support communication between a CICS system and a non-CICS system such as IMS. MRO is a widely used technique that is a central part of CICS scalability.

CICS distributed program link

CICS distributed program link (DPL) enables CICS application programs to run programs residing in other CICS regions by shipping program-control LINK requests. An application can be written without regard for the location of the requested programs; it simply uses program-control LINK commands in the usual way. Entries in the CICS program definition tables allow the system programmer to specify that the named program is not in the local region (known as the client region), but in a remote region (known as the server region).

An ECI request from CICS TG in either zSeries or a distributed environment comes to an enterprise CICS program as a DPL request. The request can be routed to a CICS program that resides in any CICS region in a sysplex.

8.2.4 Security considerations

This section provides some security guidelines for J2EE Connector-enabled applications and the underlying CICS environment. We look at the following topics:

- Signing on to the enterprise tier
- SSL encryption support
- CICS security

Signing on to the enterprise tier

As stated in “Security management” on page 143, the J2EE Connector Architecture provides security contacts for an application component. A user ID and password can be specified by the application component, or it can be specified in the deployment descriptor when the security contract is managed by the EJB container.

Authentication can be performed against the Resource Access Control Facility (RACF®) using user ID and password authentication in the CICS TG for z/OS, by setting the variable AUTH_USERID_PASSWORD=YES in the ctgstart script.
If the CICS TG runs on a distributed platform it is possible to use the ESIRequest to verify user IDs and passwords with the destination CICS region. On all platforms, it is also possible to use SSL client certificates to identify the authenticity of the CICS TG.

The authorization attributed to the server program running in CICS is always based on a user ID. The Java client program should obtain this user ID (and password) from the Web user, and flow it with every ECI or EPI call. The ATTACHSEC setting on the CONNECTION between CICS and the CICS TG will determine how CICS will use the user ID flowed on the client’s ECI and EPI calls.

As shown in Figure 8-3, steps for a typical sign-on scenario using CICS TG for z/OS are:

1. An end user enters a user ID and password from a browser.
2. The user ID and password are set in the ConnectionSpec by the application component running in the application server.
3. The CICS TG for z/OS performs an authentication using RACF DB.
4. At the enterprise CICS region, the user ID is checked using RACF DB again, and authorization is done based on the CICS resource definition.

---

**Figure 8-3  Sign-on scenario using CICS TG on a mainframe**
SSL encryption support
The client application must specify a CICS TG network protocol when it connects to the gateway daemon. There are basically two types of connections:

- TCP/IP sockets (TCP)
- HTTP sessions (HTTP)

For each of these, there is a secure version using SSL, namely SSL and HTTPS.

As an alternative to SSL, the CICS TG security exit can be used to support your own encryption/decryption procedure between the Java application and the CICS TG server, for the non-secure TCP and HTTP network protocols.

CICS security
CICS uses the z/OS System Authorization Facility (SAF) to route authorization requests to an external security manager (ESM) to perform all its security checks. Any suitable ESM could be used, but because the IBM Resource Access Control Facility (RACF) product is the most commonly used ESM, we refer to RACF when discussing CICS external security. For complete information about CICS security, refer to the CICS RACF Security Guide, SC33-1701.

To support any security checking done by CICS, the appropriate security profiles must first be defined in RACF for all the users, groups, and resources you wish to protect. Security is then enabled within a CICS region using the SEC parameter in the System Initialization Table (SIT). If SEC=YES is specified, CICS external security is enabled, and CICS then uses the SIT parameters XAPPC, XCMD, XDB2, XDCT, XFCT, XJCT, XLT, XPCT, XPPPT, XPSB, XTRAN, XTST, and XUSER to further control security. If SEC=NO is specified, no security checking is done, and users have unrestricted access.

For each CICS region that uses security, you are required to have two special-purpose user IDs, the default user ID and the region user ID. Since CICS tasks always run under a user ID, the default user ID is used when users do not explicitly sign on (and in a few other special instances). Thus the default user ID should be given very low authorization levels; it is specified on the DFLTUSER SIT parameter. The region user ID is the user ID under which the CICS job itself runs; it is by definition a powerful user ID and is also used in determining if connected CICS systems are equivalent.

Authentication of CICS users is the responsibility of RACF. Users can either be authenticated via the traditional mechanism of user ID and passwords (CICS sign-on), or more recently by the use of SSL client certificates (when using CICS Web support or CICS CORBA client support). Having authenticated the user, CICS can apply two levels of authorization to a transaction. The first is
transaction security, sometimes referred to as attach-time or transaction-attach security. The next level of security is resource security and applies to CICS-controlled resources (such as programs, files, or queues) used by the transaction.

Within an application, further authorization can be controlled by CICS. The CICS system programming command can be protected by the facilities of command security, and the issuing of requests on an interconnected CICS system can be controlled using intercommunication security. Lastly, surrogate user security can also be used to authorize an authenticated user to perform actions on behalf of a different user.

For more details on CICS security, refer to the publication Securing Web Access to CICS, SG24-5756.

8.3 Java Message Service system management

In this section we will discuss a number of system management issues concerning the Java Message Service and IBM WebSphere MQ. WebSphere MQ is a highly scalable and potentially highly distributed operational environment. For this reason, effective implementation of WebSphere MQ involves far more than a case of good message flow design. How the WebSphere MQ infrastructure is built, as well as the distribution of the WebSphere MQ components, will have a definite role in determining how the application performs.

We look at some guidelines on the following topics in this section:

- JMS administration
- JMS performance issues
- WebSphere MQ design
- WebSphere MQ administration management
- WebSphere MQ monitoring
- WebSphere MQ restart and recovery
- Managing log files
- Backing up WebSphere MQ resources
- Security considerations
8.3.1 JMS administration

To make JMS clients portable they must be isolated from the proprietary aspects of the underlying JMS Provider. This is done by defining JMS-administered objects that are created and customized by a provider's administrator and later used by clients. JMS clients access these objects through JMS interfaces that are portable. There are two types of JMS-administered objects:

- ConnectionFactory is the object a client uses to create a connection with a provider.
- Destination is the object a client uses to specify the destination of messages it is sending and the source of messages it receives.

Administered objects are placed in a JNDI name space by the JMS Provider administrator. A JMS client typically notes in its documentation the JMS administered objects it requires and how the JNDI names of these objects should be provided to it.

Two tools can be used for administrating WebSphere MQ objects WebSphere Application Server V5.0:

- WebSphere Application Server V5.0 Administrative Console
- WebSphere MQ V5.3 JMSAdmin tool

The administrative console of WebSphere Application Server can be used to administer the internal JMS Provider and the WebSphere MQ JMS Provider. (For other JMS Providers, the native tools of the provider need to be used.) The administrative console can also be used to configure and control runtime components of WebSphere JMS support (such as message-driven beans, the listener service, and listener ports).

The objects that can be managed for the internal JMS Provider are:

- WebSphere Queue Connection Factories
- WebSphere Topic Connection Factories
- WebSphere Queue Destinations
- WebSphere Topic Destinations

The objects that can be managed for the WebSphere MQ JMS Provider are:

- WebSphere MQ Queue Connection Factories
- WebSphere MQ Topic Connection Factories
- WebSphere MQ Queue Destinations
- WebSphere MQ Topic Destinations

Set the queue or topic connection factory XA Enabled attribute to indicate whether or not the JMS Provider is XA enabled.
The WebSphere MQ JMSAdmin tool allows you to administer the following types of WebSphere MQ JMS objects and store them in the JNDI name space:

- MQQueueConnectionFactory
- MQTopicConnectionFactory
- MQQueue
- MQTopic
- MQXAQueueConnectionFactory
- MQXATopicConnectionFactory
- JMSWrapXAQueueConnectionFactory
- JMSWrapXATopicConnectionFactory

The JMSWrapXA* classes are specific for the WebSphere Application Server.

### 8.3.2 JMS performance issues

Some issues that play a role in JMS messaging performance are:

- **WebSphere MQ client vs. bindings mode connection**
  
  Using a bindings mode connection to a queue manager situated on the same machine as the application server will speed up communication. As well as performance gain, it also possible to join a global transaction (which is not possible when a client connection is used).

- **Generic vs. specific message structure**
  
  Making the message structure more generic requires more translation and interpretation time at the sender and receiver ends. Making a message too specific will reduce flexibility for even small changes in the message structure. Remember to create an error queue for messages that cannot be validated.

- **Message persistent**
  
  Using persistent messages in MQ requires writing the messages to disk, which takes time, reducing performance.

- **Request/reply scenario**
  
  In a request/reply scenario, it is important that the issue of blocking calls is dealt with correctly. Essentially, EJBs should only be used with appropriate request/reply timeouts and retries.

- **Message-driven bean**
  
  Minimize the time spent in a message-driven bean processing the message. This will make message-driven bean processing faster. Let the pool of message-driven beans depend on the number of messages that arrive at the queue.
- Optimization with connection

  Start the connection when appropriate so that consumers are ready to consume messages before the producers are started. Also process messages concurrently using a server session pool for the processing of the messages. Close the connection when you are finished consuming messages.

### 8.3.3 WebSphere MQ design

The degree of complexity of WebSphere MQ system management is directly proportional to the size of the WebSphere MQ network you are trying to manage. In its simplest form, an WebSphere MQ network is comprised of a single WebSphere MQ queue manager and its resources, but can grow to hundreds or even thousands of queue managers spread throughout the enterprise.

To add to the complexity, given WebSphere MQ's middleware functionality, WebSphere MQ networks could span many platforms with different networks, hardware, operating systems, and applications. Each platform possesses its unique specific systems management needs and requirements, not to mention a varying level of support for the WebSphere MQ administration tools and interfaces.

It is no surprise, then, to see the growth of WebSphere MQ system management tools available in the marketplace today that strive in some way to deliver one or many combinations of the following WebSphere MQ system management tasks:

- **Configuration management**: The ability to deploy WebSphere MQ code and create and delete WebSphere MQ objects, including queue managers, queues, channels and processes, from a single point of control.

- **Operational management**: The ability to start and stop resources, such as queue managers, channels, trigger monitors, channel listeners, and initiators, from a single point.

- **Problem management**: The ability to detect, track, and resolve problems with WebSphere MQ objects from a single point of control.

- **Performance management**: The ability to determine performance of WebSphere MQ objects from a single point of control.

We do not focus on the growing system management tools available on the market today. Instead, we focus on introducing the reader to the basic WebSphere MQ system management guidelines along with an introduction to the WebSphere MQ facilities and tools that are available out of the box. If your
WebSphere MQ system management needs surpass those that are delivered with the product, the chosen system management tool should be an extension, or at least, a good fit with your enterprise-wide system management framework.

The following references discuss WebSphere MQ system management:

- **MQSeries SupportPac™ MS0D: Selecting MQSeries System Management tools**, a category 3 SupportPac, available at:
  

- **MessageQ.Com** at:

  [http://www.messageq.com](http://www.messageq.com)

The release of WebSphere MQ used for this discussion is IBM WebSphere MQ V5.3. In the text, when we refer to Windows, we mean Windows NT or Windows 2000.

**WebSphere MQ queue manager design**

Queue managers are the heart of the WebSphere MQ network, providing the messaging services to the applications. Think carefully about how the queue managers are defined, and their roles and their relationships to each other.

**Queue manager roles**

When designing the queue manager network, consider the distinct roles that queue managers will be expected to fulfill, together with the design priorities we may have from the application. This will enable us to define templates that can be used to deploy instances of the roles to a complete network, making scalability easier to manage.

Queue managers may be specifically designed to directly serve the calls from an application server tier, for example, an EJB container on WebSphere Application Server. An application’s queue manager may need to host local queues that are used as the input source for the application’s message flows. We must also consider the channel definitions that will be necessary when defining a new instance of the role to a network.

In practice, it may be beneficial to deploy more than one role to a single queue manager for reasons of performance or cost constraint. This is, of course possible, but by keeping the design role-focused, you will achieve flexibility in deployment.
The application may also have specific design requirements. For example, it may be vital to achieve maximum performance of message throughput between EJBs and the message consumer. Since performance is affected by network connectivity issues, hosting the roles on the same queue manager will usually mean better performance.

Alternatively, performance may be less of a priority, but achieving once-only assured delivery of a given message may be imperative. Where separate queue managers are used to host the producer and consumer roles, assured messaging is achieved.

**Use of clustering**

For ease of administration, queue managers can be configured in a cluster. To establish a clustered network of queue managers, a pair of queue managers should be configured as the cluster repository queue managers. These queue managers have a special role, in that they hold a full record of the queue managers and shared objects within the cluster. Additional queue managers can be added as members of the cluster, with a defined cluster sender channel to one of the repository queue managers.

**Administration benefits of clustering**

In a traditional WebSphere MQ network using distributed queuing, every queue manager is independent. If one queue manager needs to send messages to another queue manager, it must have defined a transmission queue, a channel to the remote queue manager, and a remote queue definition for every queue to which it wants to send messages.

A cluster is a group of queue managers set up in such a way that the queue managers can communicate directly with one another over a single network without the need for complex transmission queue, channel, and queue definitions.

Clusters can be set up easily, and typically contain queue managers that are logically related in some way and need to share data or applications. Once a cluster has been created, the queue managers within it can communicate with each other without the need for complicated channel or remote queue definitions. Even the smallest cluster will reduce system administration overheads.

Establishing a network of queue managers in a cluster involves fewer definitions than establishing a traditional distributed queuing environment. With fewer definitions to make, you can set up or change your network more quickly and easily, and the risk of making an error in your definitions is reduced.
**Clustering details**

To set up a cluster, you need to define one cluster sender (CLUSDR) definition and one cluster receiver (CLUSRCVR) definition per queue manager. You do not need to define any transmission or remote queues.

In a cluster environment you should promote one (or ideally two) queue managers as *full repository queue managers*. This means that such a queue manager knows all other queue managers in the cluster. It knows what clustered objects (local queues or any other type of object) are hosted by which queue manager and it knows how to reach those queue managers. This last point means that the full repository queue manager has a template definition of a sender and receiver channel definition that can be used to automatically create a new sender/receiver channel when needed.

To make two queue managers, QM1 and QM2, full repository queue managers for a cluster named MY_CLUSTER, you need to execute the following WebSphere MQ commands:

On QM1:

```
DEFINE CLUSCHL(TO.QM1) CHLTYPE(CLUSRCVR) +
   CONNAME(hostname1) +
   CLUSTER(MY_CLUSTER)

DEFINE CLUSCHL(TO.QM2) CHLTYPE(CLUSSDR) +
   CONNAME(hostname2) +
   CLUSTER(MY_CLUSTER)

ALTER QMGR REPOS(MY_CLUSTER)
```

On QM2:

```
DEFINE CLUSCHL(TO.QM2) CHLTYPE(CLUSRCVR) +
   CONNAME(hostname2) +
   TRPTYPE(TCP) +
   CLUSTER(MY_CLUSTER)

DEFINE CLUSCHL(TO.QM1) CHLTYPE(CLUSSDR) +
   CONNAME(hostname1) +
   TRPTYPE(TCP) +
   CLUSTER(MY_CLUSTERS)

ALTER QMGR REPOS(MY_CLUSTER)
```

The object of type CLUSRCVR, or cluster receiver, specifies how a queue manager wants other queue managers to talk to it, or more technically, how other queue managers should create a sender channel to send messages to this queue manager.
The object of type CLUSSDR, or cluster sender, should provide the queue manager with a sender channel to the other full repository queue manager in the cluster.

The ALTER QMGR command finally makes the queue manager a full repository queue manager.

As soon as these definitions are in place, you will have two-way communication between QM1 and QM2. When you now add a local queue to QM1 and you specify the cluster name MY_CLUSTER, as follows, QM1 will pass the definition of that object to the second full repository queue manager QM2:

```
DEFINE QLOCAL(CLUSTERED_QUEUE) CLUSTER(MY_CLUSTER)
```

Given that QM2 has channels to QM1 and that QM2 now knows that CLUSTERED_QUEUE exists on QM1, you do not need to define a remote queue object. Note that we have not defined transmission queues so far. WebSphere MQ cluster channels are use a common, predefined transmission queue called SYSTEM.CLUSTER.TRANSMIT.QUEUE.

The real benefit of WebSphere MQ clusters becomes clear when adding additional queue managers to the cluster. To add QM3 to the cluster, you need to define a cluster receiver channel to make clear to the cluster how other queue managers should talk to QM3:

```
DEFINE CLUSCHL(TO.QM3) CHLTYPE(CLUSRCVR) +
  CONNAME(hostname3) +
  TRPTYPE(TCP) +
  CLUSTER(MY_CLUSTER)
```

To make QM3 part of the cluster, the next and last thing you need is a cluster sender channel to one full repository queue manager. Because there are two full repository queue managers, you can choose either. Which one you select is not important. To define the cluster sender channel:

```
DEFINE CLUSCHL(TO.QM2) CHLTYPE(CLUSSDR) +
  CONNAME(hostname2) +
  TRPTYPE(TCP) +
  CLUSTER(MY_CLUSTER)
```

When this object is created, QM3 will start the channel TO.QM2 and give QM2 the definition of the channel TO.QM3. QM2 will immediately use that definition to create a sender channel from QM2 to QM3. From this point on, QM3 can use any object in the cluster MY_CLUSTER. When an application connects to QM3 and opens the queue CLUSTERED_QUEUE, QM3 will not know where that object lives. At least QM3 knows that it is not hosting this object itself. Thus, it asks its full repository queue manager QM2 about this object. QM2 replies with the definition: CLUSTERED_QUEUE is a local queue hosted by QM1. Because
QM3 does not know how to talk to QM1, QM3 sends another request to QM2 to get the communication parameters of QM1. Now, QM2 replies with the cluster receiver channel definition that QM3 uses to automatically create a sender channel from QM3 to QM1. At this point, QM3 is able to send messages to QM1 without any manual definition. Defining a cluster is even easier if you use the WebSphere MQ Explorer GUI interface.

Workload balancing and takeover
The next big advantage of clustering is the possibilities for workload balancing and takeover. Assume that we have a local queue WORKLOAD hosted on QM1 and QM2. The queue is defined into the cluster.

When an application connects to QM3 and opens the queue WORKLOAD, QM3 can now choose to which queue manager it will send messages. QM3 will select that queue manager to which it was able to set up communication. If channel TO.QM1 has gone into retry and TO.QM2 is running, QM3 will choose to send messages to QM2. If both channels are active, QM3 will send the messages to both queue managers on a round-robin basis, if the application or administrator has allowed this.

The MQOPEN now has a new option to control the spreading of workload. If you want to choose the destination at MQOPEN time, specify the option MQOO_BIND_AT_OPEN. If you want to spread the workload over the active systems, you need to use the option MQOO_BIND_NOT_FIXED. By using one of these options, an application can control whether all messages generated between MQOPEN and MQCLOSE are sent to one system or to each system that hosts the target queue.

While the above is definitely not a full coverage of WebSphere MQ clustering, we hope that you now have a basic understanding of this feature and how it can help the WebSphere MQ administrator to create a more powerful and reliable WebSphere MQ network, with fewer definitions.

8.3.4 WebSphere MQ administration management
WebSphere MQ administration tasks include creating, starting, altering, viewing, stopping, and deleting WebSphere MQ objects, including:

- WebSphere MQ queue managers
- WebSphere MQ queues
- Process definitions
- Channels
- Clusters
- Namelists
Each WebSphere MQ network has one or more instances of a queue manager known by a name within the network of interconnected queue managers. For all other object types, each object has a name associated with it and can be referenced by that name. These names must be unique within one queue manager and object type. For example, you can have a queue and a process with the same name, but you cannot have two queues with the same name.

**Administration interfaces**

These WebSphere MQ objects are managed by WebSphere MQ administration tasks that can be performed by using any of the following WebSphere MQ interfaces:

- Control commands
- WebSphere MQ commands (MQSC)
- Programmable Command Format (PCF)
- WebSphere MQ Administration Interface (MQAI)
- WebSphere MQ Explorer (available on Windows only)
- WebSphere MQ Services snap-in (available on Windows only)

**Note:** Web Administration is no longer available with WebSphere MQ V5.3.

**Control commands**

You use control commands to perform operations on queue managers, command servers, and channels. Control commands can be divided into three categories, as shown in Table 8-1.

**Table 8-1 Control command categories**

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queue manager</td>
<td>Queue manager control commands include commands for creating, starting, stopping, and deleting queue managers and command servers.</td>
</tr>
<tr>
<td>commands</td>
<td></td>
</tr>
<tr>
<td>Channel commands</td>
<td>Channel commands include commands for starting and ending channels and channel initiators.</td>
</tr>
<tr>
<td>Utility commands</td>
<td>Utility commands include commands associated with:</td>
</tr>
<tr>
<td></td>
<td>- Running MQSC commands</td>
</tr>
<tr>
<td></td>
<td>- Conversion exits</td>
</tr>
<tr>
<td></td>
<td>- Authority management</td>
</tr>
<tr>
<td></td>
<td>- Recording and recovering media images of queue manager resources</td>
</tr>
<tr>
<td></td>
<td>- Displaying and resolving transactions</td>
</tr>
<tr>
<td></td>
<td>- Trigger monitors</td>
</tr>
<tr>
<td></td>
<td>- Displaying the file names of WebSphere MQ objects</td>
</tr>
</tbody>
</table>
MQSC commands

MQSC commands are used to perform operations on queue manager objects. They are issued using the `runmqsc` command. This can be done interactively from a keyboard, or by redirecting the standard input device (stdin) to run a sequence of commands from an ASCII text file. In both cases, the format of the commands is the same.

PCF and WebSphere MQ Administration Interface (MQAI)

The purpose of WebSphere MQ Programmable Command Format (PCF) commands is to allow administration tasks to be programmed into an administration program. In this way you can create queues and process definitions, and change queue managers from a program.

PCF commands cover the same range of functions provided by the MQSC facility. Each PCF command is a data structure that is embedded in the application data part of a WebSphere MQ message. Each command is sent to the target queue manager using the MQI function, MQPUT, in the same way as any other message. The command server on the queue manager receiving the message interprets it as a command message and runs the command. To get a reply, the application issues an MQGET call and the reply data is returned in another data structure. The application can then process the reply and act accordingly.

**Note:** Unlike MQSC commands, PCF commands and their replies are not in a text format that you can read.

You can use MQAI to obtain easier programming access to PCF messages, which are considered cumbersome to work with.

WebSphere MQ Explorer

WebSphere MQ for Windows provides an administration interface called the WebSphere MQ Explorer to perform administration tasks as an alternative to using control or MQSC commands.

The WebSphere MQ Explorer allows you to perform remote administration of your network from a computer running Windows simply by pointing the WebSphere MQ Explorer at the queue managers and clusters you are interested in.

The platforms and levels of WebSphere MQ that can be administered using the WebSphere MQ Explorer, and the configuration steps you must perform on remote WebSphere MQ queue managers to allow the WebSphere MQ Explorer to administer them, are outlined in “Remote administration” on page 237.
Figure 8-4  WebSphere MQ Explorer

With the WebSphere MQ Explorer, you can:

- Start and stop a queue manager (on your local machine only).
- Define, display, and alter the definitions of WebSphere MQ objects such as queues and channels.
- Browse the messages on a queue.
- Start and stop a channel.
- View status information about a channel.
- View queue managers in a cluster.
- Create a new queue manager cluster using the Create New Cluster wizard.
- Add a queue manager to a cluster using the Add Queue Manager to Cluster wizard.
- Add an existing queue manager to a cluster using the Join Cluster wizard.

WebSphere MQ Services snap-in

The WebSphere MQ Services snap-in can be used to administer local or remote WebSphere MQ for Windows servers. It also allows you to monitor alerts created by problems in the local system.
With the WebSphere MQ Services snap-in, you can:

- Start or stop a queue manager.
- Start or stop the command servers, channel initiators, trigger monitors, and listeners.
- Create and delete queue managers, command servers, channel initiators, trigger monitors, and listeners.
- Set any of the services to start up automatically or manually during system startup.
- Modify the properties of queue managers. This function replaces the use of stanzas in configuration files (mqs.ini and qm.ini).
- Change the default queue manager.
- Modify the parameters for any service, such as the TCP port number for a listener, or a channel initiator queue name.
- Modify the behavior of WebSphere MQ if a particular service fails, for example, retry starting the service x number of times.
- Start or stop the service trace.

**Remote administration**
Both WebSphere MQ Explorer and the WebSphere MQ Services snap-in offer some form of remote administration.
The WebSphere MQ Explorer can remotely administer WebSphere MQ on the platforms listed in Table 8-2.

Table 8-2 Remote Management

<table>
<thead>
<tr>
<th>Platform</th>
<th>Minimum command level</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIX, UNIX</td>
<td>221</td>
</tr>
<tr>
<td>OS/400®</td>
<td>320</td>
</tr>
<tr>
<td>OS/2® and Windows</td>
<td>201</td>
</tr>
<tr>
<td>VMS and Tandem</td>
<td>221</td>
</tr>
<tr>
<td>MQ on zSeries</td>
<td>Indirectly; see “Points to consider when using the WebSphere MQ Explorer” on page 239</td>
</tr>
</tbody>
</table>

The platform and command level headings of Table 8-2 refer to the platform and command level queue manager attributes. Both must be used to determine which system control commands are supported. You can see the platform and command level attributes in the WebSphere MQ Explorer window, shown in Figure 8-4 on page 236.

An important point to be aware of is that the WebSphere MQ Explorer interface can only transmit commands that you would have entered in the past with runmqsc or the platform equivalent. That is, you can not use it to create or delete queue managers. Although WebSphere MQ Services snap-in allows you to start and stop remote queue managers and their associated processes, that only works when the remote system is running on Windows.

**Note:** You can do a limited number of remote operations of queue managers on non-Windows platforms, such as starting and stopping channels, but full-blown operations in a heterogeneous multi-platform configuration require more than WebSphere MQ Explorer or WebSphere MQ Services can deliver.

To remotely administer a queue manager from any of the WebSphere MQ administration interfaces, you will need the following:

- A command server running for every queue manager being administered.
- A suitable TCP/IP listener for every remote queue manager. This may be the WebSphere MQ listener or the ined daemon as appropriate.
- A server connection channel, called SYSTEM.ADMIN.SVRCONN, on every remote queue manager. This channel is mandatory for every remote queue manager being administered.
Administration interface guidelines

Deciding which of the administration interfaces or techniques are the most appropriate for a particular operation depends mainly on the platform type and the WebSphere MQ task at hand.

On Windows, you can carry out most common administration and operations tasks using the WebSphere MQ Explorer and WebSphere MQ Services snap-in tools. These tools make Windows a very convenient environment for experimentation and development. However, it is much more efficient to use one of the scripting techniques to populate and manipulate queue managers once they have been created.

If your network configuration includes a Windows server with WebSphere MQ V5.1 installed on it, you can use WebSphere MQ Explorer to carry out limited remote administration of your non-Windows queue managers. Otherwise, you will have to use the runmqsc facility interactively or with a script file, as you did with previous versions of WebSphere MQ.

WebSphere MQ script files containing control and MQSC commands are a very common way of administering WebSphere MQ. A whole WebSphere MQ configuration can be stored in such a script file and thus can be used, with minor modifications, to define new queue managers as new WebSphere MQ nodes are added to the network.

Points to consider when using the WebSphere MQ Explorer

When deciding whether to use the WebSphere MQ Explorer at your installation, bear the following points in mind:

- The WebSphere MQ Explorer works best with small queue managers. If you have a large number of objects on a single queue manager, you may experience delays while the WebSphere MQ Explorer extracts the required information to present in a view. As a rough guide as to what a “large number” is, if your queue managers have more than 200 queues or 100 channels, you may want to consider using a third-party enterprise console product instead of the WebSphere MQ Explorer.

- WebSphere MQ clusters can potentially contain hundreds or thousands of queue managers. Because the WebSphere MQ Explorer presents the queue managers in a cluster using a tree structure, the view can become cumbersome for large clusters. The physical size of a cluster does not affect the speed of the WebSphere MQ Explorer dramatically because the Explorer does not connect to the queue managers in the cluster until you select them.

- The message browser displays the first 200 messages on a queue. Only the first 1000 bytes of message data contained in a message are formatted and
displayed on your screen. Messages containing more than 1000 bytes of message data are not displayed in their entirety.

- The WebSphere MQ Explorer cannot administer a cluster whose repository queue managers are on WebSphere MQ for z/OS. To avoid this problem, nominate an additional repository queue manager on a system that the WebSphere MQ Explorer can administer. By connecting the cluster through this new repository queue manager, you can administer the queue managers in the cluster, subject to the WebSphere MQ Explorer’s usual restrictions for supported levels of WebSphere MQ.

**8.3.5 WebSphere MQ monitoring**

You can use WebSphere MQ instrumentation events to monitor the operation of queue managers. This section provides a short introduction to instrumentation events, and introduces the two Windows Service snap-ins that use WebSphere MQ instrumentation to present the user with a GUI event and monitoring tool.

**Instrumentation events**

Instrumentation events cause special messages, called event messages, to be generated whenever the queue manager detects a predefined set of conditions. For example, the following conditions give rise to a Queue Full event:

- Queue Full events are enabled for a specified queue, and
- An application issues an MQPUT call to put a message on that queue, but the call fails because the queue is full.

Other conditions that can give rise to instrumentation events include:

- A predefined limit for the number of messages on a queue being reached
- A queue not being serviced within a specified time
- A channel instance being started or stopped
- In WebSphere MQ for UNIX systems, an application attempting to open a queue and specifying a user ID that is not authorized

If you define your event queues as remote queues, you can put all the event queues on a single queue manager (for those nodes that support instrumentation events). You can then use the events generated to monitor a network of queue managers from a single node.

WebSphere MQ events are categorized as follows:

**Queue manager events** These events are related to the definitions of resources within queue managers.
Performance events  These events are notifications that a threshold condition has been reached by a resource.

Channel events  These events are reported by channels as a result of conditions detected during their operation.

When an event occurs, the queue manager puts an event message on the appropriate event queue (if such a queue has been defined). The event message contains information about the event that you can retrieve by writing a suitable MQI application program.

Each category of event has its own event queue. All events in that category result in an event message being put onto the same queue, as shown in Table 8-3.

<table>
<thead>
<tr>
<th>This event queue:</th>
<th>Contains messages from:</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYSTEM.ADMIN.QMGR.EVENT</td>
<td>Queue manager events</td>
</tr>
<tr>
<td>SYSTEM.ADMIN.PERFM.EVENT</td>
<td>Performance events</td>
</tr>
<tr>
<td>SYSTEM.ADMIN.CHANNEL.EVENT</td>
<td>Channel events</td>
</tr>
</tbody>
</table>

WebSphere MQ Alert Monitor (Windows)
The WebSphere MQ Alert Monitor is an error detection tool that identifies and records problems with WebSphere MQ on a local machine. It displays information about the current status of the local installation of a WebSphere MQ server.

From the WebSphere MQ Alert Monitor, you can:
- Access the WebSphere MQ Services snap-in directly.
- View information relating to all outstanding alerts.
- Shut down the IBM WebSphere MQ service on the local machine.
- Route alert messages over the network to a configurable user account, or to a Windows workstation or server.

If the task bar icon indicates that an alert has arisen, double-click the icon to open the Alert Monitor display. This window shows a tree view, grouped by queue manager, of all the alerts that are currently outstanding. Expand the nodes of the tree to see which services are alerted and look at the following pieces of information relating to the service:
- The date and time of the most recent alert for the service
- The command line that failed
- The error message describing why the service failed
Performance Monitor (Windows)

The Performance Monitor is a standard component of Windows. It enables you to select and display a variety of data about the performance of the Windows environment, as tabular reports or graphs. You can use it to monitor the depth of messages on WebSphere MQ queues, and the rates of message arrival and removal.

1. Access the Performance Monitor by clicking **Start -> Settings -> Control Panel -> Administrative Tools -> Performance**.

2. When you first start it, the display is empty. To add a resource that you want, click the + on the chart pane to add counters, as shown in Figure 8-6.

![Figure 8-6   Performance Monitor](image)

3. In the Add Counters window (Figure 8-7 on page 243), choose what you want to monitor.
4. Select **WebSphere MQ Queues** from the performance object drop-down.

5. Select what you want to monitor:
   - The current queue depth, that is, how many messages are in the queue.
   - The queue depths as a percentage of the maximum queue depth, that is, how full the queue is.
   - The enqueue rate in messages per second, that is, the number of messages placed in the queue. This is not necessarily the number of MQPUTs; each message segment counts as one message.
   - The dequeue rate in messages per second, that is, the number of messages removed from the queue.

6. Then select a queue from the instance list. The instance list contains only queues that have had messages inserted or removed before the Performance Monitor started.

7. Click **Add** for each selected counter.

8. Click **Close** when finished.

The graph is now displayed. Update the graph properties using the menu bar to control the graph details.
8.3.6 WebSphere MQ restart and recovery

WebSphere MQ ensures that messages are not lost by maintaining records (logs) of the activities of the queue managers that handle the receipt, transmission, and delivery of messages. It uses these logs for three types of recovery:

- Restart recovery, when you stop WebSphere MQ in a planned way
- Crash recovery, when WebSphere MQ is stopped by an unexpected failure
- Media recovery, to restore damaged objects

In all cases, the recovery restores the queue manager to the state it was in when the queue manager stopped, except that any in-flight transactions are rolled back, removing from the queues any messages that were not committed at the time the queue manager stopped. Recovery restores all persistent messages; non-persistent messages are lost during the process.

WebSphere MQ supports two types of logging:

- Circular logging
- Linear logging
Each type of logging stores the recorded data in a set of files. The differences between the two types of logging are the contents and the way that the files are linked together.

With circular logging, the log files are effectively linked together to form a ring. When data is collected, it is written sequentially into the files in such a way as to reuse the log files in the ring. You can use circular logging for crash recovery and restart recovery.

With linear logging, the log is maintained as a continuous sequence of files. When data is collected, it is written sequentially into the log files; the space in the files is not reused, so that you can always retrieve any record from the time that the queue manager was created.

Because disk space is finite, you might have to plan for some form of archiving. Also, if you are handling a high volume of persistent messages, all your log files will eventually be filled. This will result in operator messages being written to an error log file, and some action will need to be taken by the system administrator to make more log space available, or to reuse the existing space. You can use linear logging for all three types of recovery.

### 8.3.7 Managing log files

If you are using circular logging, ensure that there is sufficient space to hold the log files. You do this when you configure your system. The amount of disk space used by the log does not increase beyond the configured size, including space for secondary files to be created when required.

If you are using a linear log, the log files are added continually as data is logged, and the amount of disk space used increases with time. If the rate of data being logged is high, disk space is consumed rapidly by new log files.

Over time, the older log files for a linear log are no longer required to restart the queue manager or perform media recovery of any damaged objects.

Periodically, the queue manager issues a pair of messages to indicate which of the log files is required:

- Message AMQ7467 gives the name of the oldest log file needed to restart the queue manager. This log file and all newer log files must be available during queue manager restart.
- Message AMQ7468 gives the name of the oldest log file needed to do media recovery.
Any log files older than these do not need to be online. You can copy them to an archive medium such as tape for disaster recovery, and remove them from the active log directory. Any log files needed for media recovery but not for restart can also be offloaded to an archive.

If any log file that is needed cannot be found, operator message AMQ6767 is issued. Make the log file, and all subsequent log files, available to the queue manager and retry the operation.

Logging guidelines
Follow these guidelines when placing and managing your WebSphere MQ logs:

- When choosing a location for your log files, remember that operation is severely impacted if WebSphere MQ fails to format a new log because of lack of disk space.
- To maximize the efficiency of logging, the logging volumes should be separated from the data volumes. On small servers, this may not be practical. However, by doing so, the potential for contention between logging and writing of the queue data is reduced. This eliminates the potential of a single-point-of-failure for both the logs and the associated data.
- Whenever possible, place the log files on multiple disk drives in a mirrored arrangement. This gives protection against failure of the drive containing the log. Without mirroring, you could be forced to go back to the last backup of your WebSphere MQ system.
- Circular logging is the easiest to manage, since the logs are simply reused in a circular fashion. However, in some situations, recovery may be impossible while in this mode. Linear logs guarantee recovery but require management to prevent filling up all available disk space. Eventually, linear logs that are no longer required will need to be archived and/or deleted. This should be automated to prevent unexpected outages.
- If you are using a circular log, ensure that there is sufficient space on the drive for at least the configured primary log files. You should also leave space for at least one secondary log file, which is needed if the log has to grow.
- If you are using a linear log, you should allow considerably more space; the space consumed by the log increases continuously as data is logged.

8.3.8 Backing up WebSphere MQ resources

To take a backup of a queue manager’s data, you must:

1. Ensure that the queue manager is not running. If your queue manager is running, stop it with the `endmqm` command.
2. Locate the directories under which the queue manager places its data and its log files.

3. Take copies of all the queue manager’s data and log file directories, including all subdirectories. Make sure that you do not miss any of the files, especially the log control file and the configuration files. Some of the directories may be empty, but they will all be required if you restore the backup at a later date, so it is advisable to save them too.

4. Ensure that you preserve the ownerships of the files. For WebSphere MQ for UNIX systems, you can do this with the `tar` command.

On Windows, starting with V5.1 of WebSphere MQ, all WebSphere MQ resources are saved in the Windows registry. As a result, make sure that when backing up WebSphere MQ resources you include a backup of the Windows registry as well.

It is a good idea to save WebSphere MQ resource definitions in MQSC command files and scripts, so that these resources can be quickly redefined from scratch if required.

### 8.3.9 Security considerations

The JMS specification does not specify any features for controlling message integrity or authentication. It is expected that the JMS Provider will provide these services. Security is considered to be a JMS Provider-specific feature that is configured by an administrator rather than controlled via the JMS API by clients.

#### Message-driven bean security

Messages arriving at a listener port have no client credentials associated with them. The messages are anonymous. However, some security can be provided if the JMS listener assumes the application server process credentials when invoking the message-driven bean.

#### WebSphere MQ security

IBM WebSphere MQ queue managers handle the transfer of information that is potentially valuable, so you need the safeguard of an authority system. This ensures that the resources that a queue manager owns and manages are protected from unauthorized access, which could lead to the loss or disclosure of information.

In a secure system, it is essential that none of the following are accessed or changed by any unauthorized user or application:

- Connections to a queue manager
Access to WebSphere MQ objects such as queues, clusters, channels, and processes

- Commands for queue manager administration, including MQSC commands and PCF commands
- Access to WebSphere MQ messages
- Context information associated with messages

WebSphere MQ provides security for its administrative functions and for access to WebSphere MQ objects. This security relies on authentication being performed by the security system of the underlying operating system. Authorization of the users or groups to MQ resources is performed through the use of an access control list (ACL) maintained through the WebSphere MQ Object Authority Manager (OAM) command interface.

**The mqm group**
All queue manager processes run with these IDs:

<table>
<thead>
<tr>
<th>User ID</th>
<th>mqm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group</td>
<td>mqm</td>
</tr>
</tbody>
</table>

A user ID with the name mqm whose primary group is mqm is automatically created during installation. You can create the user ID and group yourself, but you must do this before you install WebSphere MQ.

**Administration user ID**
You must be a member of the mqm group to administer WebSphere MQ. In particular you need this authority to:

- Use the runmqsc command to run MQSC commands.
- Administer authorities using the SETMQAUT command.
- Create a queue manager using the crtmqm command.

If you are sending channel commands to remote queue managers, you must make sure that your user ID is a member of group mqm on the target system.

**OAM**
By default, access to queue manager resources is controlled through an authorization service installable component called the Object Authority Manager (OAM) for WebSphere MQ.

OAM is supplied with WebSphere MQ and is automatically installed and enabled for each queue manager you create, unless you specify otherwise.
The OAM manages users’ authorizations to manipulate WebSphere MQ objects, including queues and process definitions. It also provides a command interface through which you can grant or revoke access authority to an object for a specific group of users. The decision to allow access to a resource is made by the OAM, and the queue manager follows that decision. If the OAM cannot make a decision, the queue manager prevents access to that resource.

The OAM works by exploiting the security features of the underlying operating system. In particular, the OAM uses operating system user and group IDs. Users can access queue manager objects only if they have the required authority.

Through OAM you can control:

- Access to WebSphere MQ objects through the MQI. When an application program attempts to access an object, the OAM checks that the user ID making the request has the authorization for the operation requested. In particular, this means that queues, and the messages on queues, can be protected from unauthorized access.
- Permission to use PCF commands.

The OAM provides two control commands that allow you to manage the authorizations of users. These are:

- **SETMQAUT** set or reset authority
- **DSPMQAUT** display authority

Two of the most common authorizations assigned with the SETMQAUT command are the ability to put a message on a specific queue by issuing an MQPUT call (put authority), and the ability to retrieve a message from a queue by issuing an MQGET call (get authority).

**More information**

For more information on messaging security see:

- WebSphere InfoCentre article *Asynchronous messaging - security considerations*:
  
In this part we look at extending an existing application to the enterprise.

Chapter 9, “PDK sample overview” on page 253, provides an overview of the application.

The remaining chapters discuss the process of extending the application using three different methods:

- Chapter 10, “Web services scenario” on page 281, describes extending the application using Web services.
- Chapter 11, “J2EE Connector Architecture scenario” on page 333, describes extending the application using the J2EE Connector Architecture.
- Chapter 12, “Java Message Service scenario” on page 365, describes extending the application using the Java Messaging Service.
Chapter 9. PDK sample overview

In this chapter we introduce the Patterns Development Kit that was developed as an example of a best practice implementation of the Self-Service::Stand-Alone Single Channel application pattern and the Self-Service::Directly Integrated Single Channel application pattern. We use this application to demonstrate some of the design and development guidelines discussed in previous chapters.
9.1 The Patterns Development Kit

The Patterns Development Kit V5 (PDK) is a self-installable application that demonstrates a best practice approach to developing an application based on the following IBM Patterns for e-business:

- Self-Service::Stand-Alone Single Channel application pattern
- Self-Service::Directly Integrated Single Channel application pattern

Figure 9-1 shows the PDK welcome page.

Note: You will probably notice references to Topology 1 and Topology 2 in the PDK application and documentation (in Figure 9-1, for example). Topology 1 is a short-hand reference to the Stand-Alone Single Channel application pattern. Topology 2 refers to the Directly Integrated Single Channel application pattern.
The PDK uses an Interplanetary Weather readings theme. It demonstrates a number of technologies along with usage guidelines via an online system for transferring funds to interplanetary weather stations. From a central body's (the Guild's) annual budget, funds can be transferred to any one of a number of weather stations (or planets), for the purchase of weather monitoring equipment. An exchange rate between the Guild and each of the planets is applied to the funds transferred.

Among the technologies used in the front-end application are Struts, JavaServer Pages, the WebSphere Command Framework, and both session and entity EJBs. In Stand-Alone Single Channel mode, the exchange rate is always 1.

The PDK also implements some of the suggested design strategies for connecting Self-Service applications to the enterprise using the Directly Integrated Single Channel application pattern. In this mode, the exchange rate is retrieved from a back-end or enterprise-tier application when a funds transfer is attempted.

The back-end application takes three forms:

- An enterprise application accessed through Web services
- A CICS application
- An enterprise application accessed through WebSphere MQ

PDK V5 demonstrates the following enterprise-integration options:

- Web services to enterprise-tier Web service provider
- J2EE Connector Architecture (JCA) to CICS enterprise tier
- Java Message Service (JMS) to WebSphere MQ Server
- Stand-Alone Single Channel application with no enterprise integration

The specifics of each of these implementations is discussed in the following chapters. In this chapter, we look at the front-end application design and implementation of the PDK V5.

PDK V5 and previous versions of the PDK are available from the Patterns for e-business Web site:


### 9.2 PDK use cases

Figure 9-2 on page 256 gives a pictorial representation of these PDK use cases from Rational XDE. It shows the actors of the system, namely the user initiating the HTTP request and the enterprise systems. The user can invoke the Display Funds use case and Transfer Funds use case.
The PDK also has some internal use cases, not directly accessed by the user. The Transfer Funds use case invokes the Get Exchange Rate use case to determine the exchange rate between the Guild and the planet. In Directly Integrated Single Channel mode, the Get Exchange Rate use case can invoke the Get Exchange Rate through Web Services, through CICS, or through MQ uses cases to determine the exchange rate from the enterprise-tier.

Let us take a closer look at these two simple use cases:
- Display Funds
- Transfer Funds

### 9.2.1 Display Funds

The Display Funds use case is initiated by the user of the system. Via an HTTP request, the user invokes the Struts DisplayFundsAction, which returns a JavaServer Page displaying the current level of funding of all the planets as well as the Guild. As shown in Figure 9-3 on page 257, this JSP also includes the form that allows the user to initiate a transfer of funds.
9.2.2 Transfer Funds

The Transfer Funds use case is also initiated by the user. The user selects a planet to transfer funds to and specifies the amount to be transferred. The Struts TransferFundsAction is then invoked. If the transfer is successful, the user is returned a JSP containing the confirmation details of the transfer, as shown in Figure 9-4 on page 258. If the transfer is unsuccessful, the user is forwarded to an error page. Reasons that may cause the transfer to fail include the transfer causing the planet to be overfunded (greater than 999,999 Astros), or the Guild not having enough funds to support the transfer.
Funds have been transferred to your designated weather station.

- Guild value of transfer is : - 100
- JUPITER value of transfer is : - 300
- Using exchange rate of : - 3
- Backend used was : - WebServices
- Click here to get the latest balances.

Figure 9-4  Transfer funds result JSP

9.3 Architectural overview model

Figure 9-5 on page 259 gives an architectural overview of the entire PDK V5 application. We return to this diagram in each of the remaining chapters to discuss the design of each area in greater detail.
It this chapter, we consider the highlighted area of the diagram. We will call this area the front end of the application. It consists of the functionality used to support the PDK Lite J2EE enterprise application in Stand-Alone Single Channel mode. This application’s function is to receive and process the HTTP requests that come from the user. The PDK application uses Struts and the WebSphere Command Framework to process these requests. To support the execution of the PDK commands, another J2EE enterprise application, the PDK Command Server, is used. Both the Guild and Stations databases also exist on the front end of the application. Finally, in support of the application extension, the PDK application acts as the decision point for determining which of the available back ends to connect to during the Transfer Funds use case.

9.4 System design overview

The following sections examine in more detail the front end of the application. We start with a look at the various components that make up the front end, look at the classes used in the implementation, and then consider the interactions between them.
9.4.1 PDK component model

Figure 9-6 shows the components of the front-end application.

The PDKLiteWeb component houses the Struts actions, JSPs, commands, and command factories that are used to process the incoming HTTP requests from the user. Two Struts actions are used to parse the incoming HTTP requests from the user. The Struts actions validate the requests before calling a command factory to request an instance of a command that implements the incoming user request. The Struts action also calls a command target factory to obtain a command target instance that communicates with the command server, which in turn performs the execution of the command.

The command target servers reside in the Command Server application and take the form of an EJB and a servlet accessed through HTTP. (There is also a third dummy implementation of a command target that sits locally in the PDK Lite application itself.) The commands and targets adhere to the interface definitions as defined by the WebSphere Command Framework.
When executed, the command obtains access to the FinanceBean session EJB that acts as a facade to two entity beans and session bean. The entity beans control access to the Guild and Stations databases. The session bean is used to get the exchange rate for the station (or planet).

Once the FinanceBean has performed the required business logic, control passes back to the command that sets up its result fields. Control then passes back to the Struts action that interrogates the command’s result fields to determine the result of the command execution.

If the command executes successfully, the Struts action uses the retrieved results to create a view bean. This view bean is passed in the HTTP request to a JSP. The JSP uses the view bean to create and display a results page to the user representing the result of the command execution.

### 9.4.2 PDK object model

Our PDK object model consists of a class diagram showing the static relationships between classes and a sequence diagram showing the dynamic object interactions in time order.

#### PDK class diagram

The class diagram in Figure 9-7 on page 262 shows the classes used in the Transfer Funds use case. The various roles of these classes in the system are:

- The two Singleton factories take responsibility for determining and instantiating the command and command target implementations.
- The servlet acts as the controller using the factories and commands to carry out the necessary processing.
- The Finance bean acts as the session facade to the business model—the two entity EJBs, GuildAccounts and StationFunding.
- The View bean created by the servlet for use by the JSP to display the results of the transfer to the user.
PDK sequence diagram

The sequence diagrams in Figure 9-8 on page 263, Figure 9-9 on page 263, and Figure 9-10 on page 264 show the interaction between the classes shown in Figure 9-7.
Figure 9-8  Transfer Funds sequence diagram - part 1

Figure 9-9  Transfer Funds sequence diagram - part 2
9.5 Lower-level design considerations

In this section we look at the implementation of some of the key design points discussed in previous chapters by stepping through the Transfer Funds use case.

The key design points we will look at are:

- Validation of client input close to the client using JavaScript in the browser
- Use of Singleton classes for command and command target factories
- Implementation of MVC pattern using Struts, JSPs, and entity beans
- Use of the WebSphere Command Framework
- Use of session EJBs as a facade to entity EJBs
- Caching of home EJB references
- Use of two-phase commit (2PC)

9.5.1 Validation of client input

By validating client input as close to the client as possible we reduce the number of invalid requests that will need to be handled by the servlets, improving performance. The client-side checks should be limited to ensuring that:

- All mandatory fields are filled in.
- The data types and ranges are valid.

The client-side checks should not include business logic checks on the input. This should remain under the server's control.
In our example, the funds transfer is initiated from the results of the displayFundsRichResults JSP. This JSP is responsible for displaying the results of the Display All Balances use case and can be seen in Figure 9-3 on page 257. From the HTML form on the right-hand side of the screen, the user selects the planet to transfer funds to and enters an amount. The JSP then uses a simple JavaScript function contained in a separate file (displayBalance.js) to perform the validation of the attempted user request, ensuring that the user has indeed entered a valid transfer amount. Part of the JavaScript is shown in Example 9-1.

Example 9-1  Validating the user's request using JavaScript

```javascript
function attemptSubmit(){
...
if(isValidAmount(textEntered) == false){
   alert("Please enter a valid, positive amount to transfer");
   return false;
}
if(confirm("Are you sure you want to grant A" + textEntered + " to " + stationSelected)){
   return true;
} else {
   return false;
}
status = "Initiating funds transfer...";
return true;
}
```

9.5.2 Singleton factories

It is sometimes important or sufficient that classes have only one instance and that the Singleton pattern is designed to ensure that only one instance of a class can be created, managed, and made easily accessible. Having only one instance of a class provides memory management benefits and reduces the overhead of garbage collection.

In our example, both the CommandTarget and CommandTargetFactories have been made Singleton classes. In each case, the constructor for the class has been made private to ensure that external clients cannot create new instances. A static method named Instance() allows for the solitary instance to be accessed. The instance method checks to see if an instance of the commandTarget exists and, if it does, it is simply returned to the caller. Otherwise, the constructor is called to create the instance. This is shown in Example 9-2 on page 266.
Example 9-2  The Singleton command factory

private static CommandFactory instance = null;
...
// CommandFactory constructor - private to ensure no more than one instance
// is created.
private CommandFactory() {
...
}
// Instance() method returns a reference to the current instance, creating one
// if necessary.
public static CommandFactory Instance() {
    if (instance != null) { return instance; }
    else {
        instance = new CommandFactory();
        return instance;
    }
}

9.5.3 Model-view-controller implementation

The PDK provides a sample implementation of the MVC pattern using servlets, JSPs, and entity beans. See 6.1.1, “Model-View-Controller design pattern” on page 89, for an introduction to this pattern.

Struts action as the controller

The role of the Struts action is to act as a controller. The key requirements of the controller are to validate the incoming request; initiate the business logic to process the request; and, after execution, to pass the results to the appropriate view. We will look at each of these in turn.

First, the controller must validate the incoming request. In our case the TransferFundsAction expects the TransferFundsForm object to specify the station receiving the fund transfer and the amount to be transferred, as shown in Example 9-3. If the parameters are missing the user will be forwarded to a suitable error page.

Example 9-3  Validating the form data

// get the form data
TransferFundsForm registrationForm = (TransferFundsForm)form;
if (form == null) {
    writeToLog("perform() - [OUT]");
    return mapping.findForward("failure");
}
//...
String amount = ((TransferFundsForm)form).getAmount();
if (amount == null) {
    writeToLog("perform() - [OUT]");
    return mapping.findForward("invalid");
}
String station = ((TransferFundsForm)form).getStation();

Once the request is validated, the Struts action must process it. The PDK uses TargetableCommands to encapsulate the business logic. As stated previously, both the command factory and command target factory are implemented as Singleton instances. After accessing the instance of the command factory, the Struts action asks the factory for an object that implements the TransferFundsCommand interface. This allows the implementation of the commands to be loosely coupled to the clients that use them.

In a similar way, the Struts action accesses the Singleton instance of the command target factory to retrieve an implementation of the CommandTarget interface. Again the targets are loosely coupled. The application administrator can change the target that will be used via a properties file, without needing to alter the client.

Example 9-4 shows how the command factory and command target factory are used in the Struts action.

Example 9-4   Using the command factory and command target factory

// In TransferFundsAction
TransferFundsCommand command =
    (TransferFundsCommand)CommandFactory.Instance().
    getCommand("TransferFundsCommand");
command.setCommandTarget(CommandTargetFactory.Instance().getCommandTarget());

After receiving the command, the Struts action sets the command’s properties. In this use case, the command is passed the details of the station and transfer amount that were obtained from the HTTP request. The setHasOutputProperties call ensures that the command will be copied back from the command target after execution, bringing with it the results of the execution. The command’s execute method is then called. Control passes to the IBM Command Framework to determine the command target, and to copy the command to the target for execution. After the command executes, the Struts action checks if the command executed successfully. If not, an exception is thrown.

Example 9-5 shows how the Struts action sets and executes the command.

Example 9-5   Setting and executing the command

// Set command’s other properties...
command.setStation(station);
command.setAmount(amount.trim());
command.setHasOutputProperties(true);

// Execute it
command.execute();

// Inspect command's result field to check for error...
if (command.getMessage().startsWith("ERROR")) {
    throw new CommandException(command.getMessage());
}

If the command executes successfully, the Struts action uses the command's attributes to set the view bean, as shown in Example 9-6. The view bean contains all the data required by the JSP to display the results of the user request back to the user. The view bean is then stored in the HTTP request before the request is forwarded to the success JSP.

**Example 9-6  Setting up the view bean**

```java
// create and set view bean details
TransferFundsViewBean viewBean = new TransferFundsViewBean();
viewBean.setGuildAmount(command.getAmount());
viewBean.setExchangeRate(new Integer(command.getExchangeRate()).toString());
viewBean.setRecipient(command.getStation());
viewBean.setTransferAmount(new Integer(command.getTransferredAmount()).toString());
viewBean.setBackendUsed(command.getBackendUsed());

// add view bean to the request so that it can be accessed by JSP
request.setAttribute("transferViewBean", viewBean);

// forward to the success page
return mapping.findForward("success");
```

**JSP providing the view**

After the Struts action has controlled the processing of the HTTP request, the JSP receives the request with the additional view bean details. The JSP provides the means for generating the HTML response page for the user. It contains no business logic, just a mixture of static HTML with the necessary embedded scriptlets. These scriptlets use the view bean information supplied by the Struts action to provide the dynamic content.
The `<jsp:usebean>` tag denotes that the JSP expects to receive an object named `transferViewBean`, of type `TransferFundsViewBean`, within the scope of the HTTP request object. After retrieving the view bean, the JSP simply calls the bean’s getter methods to retrieve the required display data and places the data within the static HTML page.

**Example 9-7 Using the view bean in the JSP**

```jsp
<jsp:useBean id="transferViewBean"
type="com.ibm.pdk.view.TransferFundsViewBean" scope="request" />

Funds have been transferred to your designated weather station.
<br><br>
Guild value of transfer is :- <%= transferViewBean.getGuildAmount() %>
<br><br>
<%= transferViewBean.getRecipient() %> value of transfer is :- <%= transferViewBean.getTransferAmount() %>
<br><br>
Using exchange rate of :- <%= transferViewBean.getExchangeRate() %>
<br><br>
Backend used was :- <%= transferViewBean.getBackendUsed() %>
```

**Entity beans providing the model**
The data model for the PDK application is very simple. Two DB2 databases are used, one containing a table representing the Guild’s account details and one with a table holding the account details of the various planets. Using two databases allows us to demonstrate the use of two-phase commit for handling transactions across multiple databases.

These two tables are mapped to two entity EJBs, GuildEJB and StationsEJB. The EJBs contain the schema details of the database and includes a mapping between the database columns and the fields on the EJB, thereby producing an object view of a row in the database table.

**WebSphere Studio and Struts**
IBM WebSphere Studio Application Developer V5.0 has built-in tools that support the development of Web applications using the Struts framework. Along with wizards for creating form beans and action classes, it also features:

- Visual Web diagram editor for Struts resources
- Special editor for the Struts configuration file
**Struts Web diagram**
A Web diagram, shown in Figure 9-11, helps you visualize the flow structure of a Struts-based Web application. Because of the indirectness involved in Struts applications, being able to visually see the application's flow can help you to better understand the application. WebSphere Studio maintains synchronization between the Web diagram and the underlying application components.

![Struts Web diagram](image)

**Figure 9-11  Struts Web diagram**

**Struts configuration file editor**
This special editor view simplifies the Struts application development. Instead of editing the XML source, you can create and configure the Struts elements interactively in the editor, shown in Figure 9-12 on page 271.
9.5.4 Using the WebSphere Command Framework

The WebSphere Command Framework is encapsulated in the EJBCommandTarget.jar, which is provided with WebSphere Application Server. It consists of the interfaces that define the framework and an EJB CommandTarget implementation. This EJB (CommandServerSession) is used in the PDK CommandServer application, along with an additional example implementation (HttpServletCommandServer) written as part of the PDK, which uses an HTTP servlet as the command target.

In the PDKLite application, the EJBCommandTarget and HttpServletCommandTarget classes are both implementers of the CommandTarget interface supplied by the framework. These classes handle the communication with the command server, HttpServletCommandTarget, using HTTP to talk to the HttpServletCommandServer servlet, the EJBCommandTarget using RMI/IIOP to talk to the CommandServerSession EJB.
The TransferFunds command in our example is a trivial command. An interface, TransferFundsCommand, defines the business methods that are implemented by the TransferFundsCommandImpl class. More importantly, the class is also an extension of the generic TargetableCommandImpl class supplied in the framework. This makes methods such as setCommandTarget() available to the developer.

After setting up the command and the command target (see Example 9-4 on page 267 and Example 9-5 on page 267), the command can be executed. When the execute() method is called, the command framework determines the correct target for the execution and passes control to it. The command is serialized and transported to the target location, which could be a remote machine. The command server will then initiate the execution of the command before it is passed back to the target. Control then resumes with the object that called the execute() method.

The actual logic contained in the TransferFundsCommandImpl's execute method simply accesses a session EJB in order to perform the required business logic. In reality this command could control access to a number of session beans, coordinating the calls and hiding the implementation and access from the servlet client.

Example 9-8 shows the command's execute method performing the JNDI lookup to access the session bean, and then using the session bean to call the grantFunds method.

*Example 9-8  Finding and using the Finance session bean*

```java
// Set Context
Context context = new InitialContext();
// Lookup session bean used for transfer funds
Object obj = context.lookup("ejb/ref/Finance");
FinanceHome financeHome = (FinanceHome)javax.rmi.PortableRemoteObject.narrow(
    obj, FinanceHome.class);
Finance finance = financeHome.create();

TransferFundsRecord transfer = new TransferFundsRecord();
transfer.setRecipient(getStation());
transfer.setAmount(Integer.parseInt(getAmount().trim()));

transfer = finance.grantFunds(transfer);
```
Chapter 9. PDK sample overview

9.5.5 Session EJB facade

As stated in 6.1.8, “Best practices for EJBs” on page 109, allowing clients to access entity EJBs directly is not good practice. Instead, session EJBs are used as a facade.

In our application, the Finance session EJB provides a facade to the business model, which is encapsulated in the two entity EJBs, GuildAccounts and StationsFunding. By allowing the session bean to control the calls to the entity EJBs, we ensure that each call can take place within one transaction. Otherwise, each call from the client to the entity EJBs would require an individual transaction resulting in decreased performance due to the need to pacify and activate the entity bean on every call. It also provides the benefit of shielding the client from the business model structure.

Example 9-9 shows the session bean carrying out the first part of the funds transfer. First, we access the Guild account using the GuildAccounts EJB and then attempt to withdraw the transfer amount. If the withdrawal fails (due to insufficient funds) the session EJB initiates a rollback of the session and an exception is thrown.

Example 9-9  Withdrawing funds from the Guild database

```java
// Locate the guild entity
GuildAccountsLocal guildAccounts =
    guildAccountsHome.findByPrimaryKey(new GuildAccountsKey("GRANTS"));

// Attempt debit...
boolean result = guildAccounts.attemptWithdrawal(amount);

if (!result) {
    // Mark the current transactional context for rollback
    getSessionContext().setRollbackOnly();
    throw new EJBException(
        "Transaction rolled back - withdrawal from the Guild failed");
}
```

Before we grant the funds to our intended recipient, we must now determine the exchange rate. The code is shown in Example 9-10 on page 274. The task of controlling the communication with the back end has been passed to the ExchangeRate session EJB. The Finance session EJB calls its getExchangeRate method, passing the planet name as a parameter.

The ExchangeRate session EJB determines which back end to use to get the exchange rate. The back end is determined via the backendList environment variable defined in the EJB deployment descriptor, which associates a planet with a back end (or no back end in Stand-Alone Single Channel mode).
Again, if an exception occurs, the transaction will be rolled back and a suitable exception thrown to the command bean.

Example 9-10  Determining the exchange rate

```java
try {
    // ...
    ExchangeRate exchangeRate = exchangeRateHome.create();
    rate = exchangeRate.getExchangeRate(recipient);
    transfer.setRate(rate);
    transferAmount = amount * rate;
    transfer.setTransferAmount(transferAmount);
    transfer.setBackendUsed(exchangeRate.getBackend(recipient));
} catch (Exception e) {
    // Mark the current transactional context for rollback
    getSessionContext().setRollbackOnly();
    throw new EJBException("Transaction rolled back - failed to determine exchange rate");
}
```

The final part of the funds transfer sees the newly determined transfer amount being granted to the planet. The Finance session EJB looks up, via the StationsFunding entity EJB, the database entry for the specified planet. The attemptDeposit method is then called to complete the transfer. Again, if an error is received (because the planet is overfunded), the transfer is rolled back and an exception is thrown, as shown in Example 9-11.

Example 9-11  Depositing to the Station database

```java
// Locate correct record
StationsFundingLocal stationsFunding =
    stationsFundingHome.findByPrimaryKey(new StationsFundingKey(recipient));

// Attempt deposit
result = stationsFunding.attemptDeposit(transferAmount);

if (!result) {
    // Mark the current transactional context for rollback
    getSessionContext().setRollbackOnly();
    throw new EJBException("Transaction rolled back - Deposit to station "+ recipient + " failed");
}
```
9.5.6 Caching of EJB home references

When JNDI is used to look up or locate the EJB home interface, a significant overhead is incurred. Some of this is reduced by WebSphere providing its own caching of home interface references, but even this incurs a set amount of time. Instead, we cache EJB home interfaces using a factory class, shown in Example 9-12.

Example 9-12 EJB home factory

```java
import java.util.HashMap;
import java.util.Map;
import javax.ejb.EJBHome;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.rmi.PortableRemoteObject;

public class EJBHomeFactory {

    private static EJBHomeFactory instance = null;
    private Map homeInterfaces;
    private Context context;

    private EJBHomeFactory() throws NamingException {
        homeInterfaces = new HashMap();
        context = new InitialContext();
    }

    // Returns the single instance of the EJBHomeFactory
    public static EJBHomeFactory getInstance() throws NamingException {
        if (instance == null) { instance = new EJBHomeFactory(); }
        return instance;
    }

    // Returns the cached instance of the specified EJBHome interface class
    public EJBHome lookup(String jndiName, Class homeInterfaceClass) throws NamingException {
        // See if we already have this interface cached
        EJBHome homeInterface = (EJBHome)homeInterfaces.get(homeInterfaceClass);
        // If not, look up with the supplied JNDI name
        if (homeInterface == null) {
            Object obj = context.lookup(jndiName);
            homeInterface = (EJBHome)PortableRemoteObject.narrow(
                obj, homeInterfaceClass);
            // If this is a new ref, save for caching purposes
            homeInterfaces.put(homeInterfaceClass, homeInterface);
        }
        return homeInterface;
    }
}
```
Example 9-13 demonstrates how the EJB home factory is used to look up an EJB home. The lookup call will return the EJB home from cache if available, or look up the EJB home and cache it for the next time.

Example 9-13  EJB home lookup

```java
ExchangeRateHome exchangeRateHome = (ExchangeRateHome) EJBHomeFactory.getInstance().lookup("ejb/ref/ExchangeRate", ExchangeRateHome.class);
ExchangeRate exchangeRate = exchangeRateHome.create();
```

See the IBM developerWorks article *EJB best practices: Industrial-strength JNDI optimization* by Brett McLaughlin, available at:


9.5.7 Using two-phase commit

If we refer back to Example 9-9 on page 273, Example 9-10 on page 274, and Example 9-11 on page 274, you will notice the use of the method `getSessionContext.setRollbackOnly()`. This method is called when it becomes clear that the transaction cannot be completed successfully, either because we are unable to withdraw funds, unable to determine the exchange rate, or unable to deposit to the station's database.

The session EJB uses container-managed transactions, and so the WebSphere container is taking responsibility for starting, committing, and rolling back the transactions. The transaction support required by the EJB is defined in the EJB's deployment descriptor. By calling the `setRollbackOnly()` method, we cause WebSphere to initiate a rollback of the transaction.

To ensure that both of the external databases remain in the correct state, we use the two-phase commit DB2 driver `COM.ibm.db2.jdbc.DB2XADataSource` to enable the committing of the transaction over multiple resources. This driver is referenced by the two WebSphere data sources that provide access to the databases.

9.6 PDK development

This section covers the following topics:

- Tools and APIs used for PDK development
9.6.1 Tools and APIs

The development of PDK V5 was carried out using IBM WebSphere Studio Application Developer V5.0.

The levels of the key J2EE APIs supported by IBM WebSphere Studio Application Developer are as follows:

- Servlet 2.3
- JavaServer Page 1.2
- Enterprise JavaBean 2.0

9.6.2 PDK project structure

Figure 9-13 shows the projects that make up the PDK V5 application. The PDKLite project is a J2EE enterprise application project and as such contains a Web project (PDKLiteWeb) and an EJB project (PDKLiteEJB).

![WebSphere Studio projects view](image)

PDKLiteWeb contains all the Struts components, JSPs, images, and other Web resources required by the PDKLite application.
ExchangeRateServer is a J2EE enterprise application project that acts as the enterprise-tier application used by the PDKLite front-end application. ExchangeRateServerWeb contains the Web services SOAP server, while ExchangeRateServerEJB contains the ExchangeRateServer session EJB, which the front-end application accesses using Web services or JMS.

CommandServer is a J2EE enterprise application project that acts as a server for the command targets used by the PDK commands. CommandServerWeb contains the HTTP servlet command server, while CommandServerEJB contains the EJB command server.

In addition to the application-specific projects, a number of support projects exist, such as Build, CommandUtil, PDKCommandUtil, PDKFinanceUtil, PDKMessageUtil, and PDKSupport. Build contains Javadoc generation and PDK installation scripts. CommandUtil contains non-application specific command classes. PDKCommandUtil contains the PDK-specific command classes. PDKFinanceUtil contains lightweight finance classes that are passed around the application. PDKMessageUtil contains classes for working with XML exchange rate messages. PDKSupport contains logging, JNDI caching, and constants classes.

These support projects were developed independently from the main PDKLite, ExchangeRateServer, and CommandServer applications. They are added to the applications that require access to the classes in the support projects as Project Utility JARs.

### 9.6.3 Team development

IBM WebSphere Studio Application Developer V5.0 provides improved team support, including many usability enhancements for CVS users and better integration with Rational ClearCase® LT.

The Concurrent Versioning System (CVS) was the team development tool adopted during development of this publication. CVS allows each developer in the team to maintain his or her own workspace of WebSphere Studio projects and periodically synchronize with other team members through a common repository. The repository allows the developers to release changes and to catch up with updates made by others. The repository also provides support for creating and maintaining versions of the code.

The new VCM interface provided by Eclipse Version 2 provides support for pluggable repositories. Check with your SCM vendor for compatibility.
9.6.4 Testing

The Servers project is a server project containing an IBM WebSphere Application Server base V5.0 instance and configuration specifically for the PDKLite, ExchangeRateServer, and CommandServer applications. It contains runtime information such as the data sources and resource definitions. As well as being able to configure server parameters from the Server perspective in WebSphere Studio, the server can be configured using the Web administrative console. You can start the Web administrative console by right-clicking the server instance and selecting **Run administrative console**.

Because the IBM WebSphere Application Server base V5.0 test environment is fully integrated with the IBM WebSphere Studio Application Developer environment, it is easy to execute the application in the test environment. Simply right-click the PDKLiteWeb project and select **Run on Server** as shown in Figure 9-14.

![Figure 9-14 Running the PDKLite application in WebSphere Studio](image)

9.7 PDK runtime

The PDKLite, ExchangeRateServer, and CommandServer applications were deployed to IBM WebSphere Application Server base V5.0 running on Windows 2000, AIX, and Linux in our lab environment. For details, see Appendix A, “Lab environment scenarios” on page 413.

Through the use of environment variables associated with some of the key components in the system, the application packager/deployer can alter the runtime configuration of the PDK in a number of ways. The key environment variable is the one associated with the ExchangeRate session bean, shown in Figure 9-15 on page 280. This variable specifies the enterprise-tier connection to be used for each of the planets in the system.
The other notable environment variable is commandTargetList in PDKLiteWeb, which allows the user to choose which of the three command targets to use for command execution. See “Changing the default settings” on page 431 for further details.

9.8 More information

For more information on the topics covered in this chapter see:

- Redbook *EJB 2.0 Development with WebSphere Studio Application Developer*, SG24-6819
- Patterns for e-business downloads, including PDK Lite
  
- IBM developerWorks
  
  http://www.ibm.com/developerWorks
- WebSphere Developer Domain
  
  http://www7b.software.ibm.com/wssd
- WebSphere Best Practices Zone
  
  http://www7b.software.ibm.com/wssd/zones/bp
- Java technologies home, for servlets, JSPs, EJBs, etc.
  
  http://java.sun.com/products
Web services scenario

Our objective in this chapter is to describe how to build a Web services application, and to explore the different design options that are available with Web services. We detail how to use IBM WebSphere Studio Application Developer to build a sample application. However, the sample application we describe here is based on Java and J2EE and can therefore be built using any Java Integrated Development Environment (IDE).

We walk through how to extend the PDK Lite application described in Chapter 9, “PDK sample overview” on page 253, to include a Web services component that communicates with a Web services component on an enterprise server. This enables the PDK Lite application to query an enterprise-tier application and retrieve data using Web services technology.

This chapter describes the following:

- Creating a client to a Web service using static service binding (no UDDI).
- Integrating a Web service into a Web application.
- Creating a Web service from existing applications. This includes an example of a Web service provider and a Web service requester.
- Using the IBM WebSphere Studio Application Developer Web Service wizard to generate the files needed to create a Web service.
- An examination of the files generated by the IBM WebSphere Studio Application Developer Web Service wizard.
10.1 Architectural overview model

In this chapter, we describe how the PDK application can be extended to communicate with an external enterprise system using Web services. The architecture overview model demonstrates the entire system and shows how all of the components are involved in the overall system.

As can be seen in Figure 10-1, we have highlighted the area that we focus on in this chapter. From the J2EE Architecture point of view, this is the part of the application server business logic tier that retrieves some business data from a single application in the EIS tier. Results of the information retrieved are processed and sent back to the presentation tier.

Figure 10-1  Web services system overview model
10.2 System design overview

As system designers, the first question we should ask ourselves is "What is the business problem we are trying to solve?" In the case of our sample PDK application, we know that we have this Guild of Weather Masters who want to exchange information such as account balances and exchange rates. We know that each Station is on a different planet and its enterprise data resides on totally different platforms, databases, or applications. The challenge is in how we can provide seamless integration between these disparate systems. Web services is one choice that we have as system designers. For the sake of our Web services sample application, let us assume the following scenario.

The Guild Masters have an old mainframe-level system called the Super Saturn mainframe with a proprietary interface. They have tried to publish this interface and allow the Stations on other planets to access it but have found that there are constant problems with this approach. For one thing, the Guild Master who knows the most about the Super Saturn is 247 years old in Saturn time and is approaching retirement and they have been having problems recruiting skilled Super Saturn engineers.

As a solution, the Saturn Guild has decided to use the Web services technology to provide an interface into its own proprietary Super Saturn-based system. This Web services interface can be easily integrated with the many heterogeneous systems used by the Stations on other planets to allow the other planets’ systems to access the Super Saturn system.

To accomplish this, the Saturn Guild can simply send the other planets a WSDL file describing the Saturn Web service. (Remember that in this scenario we use a static rather than dynamic publish and discovery approach, therefore, no UDDI registry is needed.) The other planets then integrate this WSDL description into the PDK application already implemented on their planets. Using the interface and implementation information provided in the WSDL file, each planet modifies its current system to enable it to communicate with the Super Saturn application.

In this example, Saturn is the Web service provider and publisher, and the other planets are the Web service requesters.

The benefits of this approach is that neither the Saturn Guild nor the other planets require any details of how the Web service has been implemented, or which type of technology has been used.

Figure 10-2 on page 284 shows at a very high level the basic system design for Web services requesters and providers.
10.2.1 Component model

The component model shows a breakdown of both the client (Web service requester) application server and the enterprise (Web service provider) application server.

In our sample application, the PDK application servlets parse the request from the client browser and utilize the command pattern to figure out which command to call. Depending on the planet selected for transfer funds, the command bean calls on a particular EJB implementation.

In this case, a Web services session bean provides the connection to a Web services proxy, which knows how to create a SOAP request to send to the Web service provider on the enterprise application server.

The Web services proxy on the PDK application server sends a message to the enterprise application server using the SOAP and HTTP protocols.
10.2.2 Object model

In this section we provide an object model for our Web services scenario.

Class diagram
Figure 10-4 on page 286 shows a class diagram of the classes directly involved with providing Web services for the application. On the requester side, we have an EJB called ExchangeRateWSBean, which serves as the Web services requester bean. This bean serves as the interface between the PDK application and the Web service provider. As far as the application knows, it is simply calling a local method called getExchangeRate on this bean and it gets back the requested exchange rate. The FinanceBean from the PDK application requests the exchange rate using the ExchangeRateBean.

On the provider side, we have a stateless session EJB called ExchangeRateServerBean, which serves as the Web services bean for the provider. In a real application, this bean would probably act as a facade to the actual enterprise business objects. It knows which enterprise objects to call in order to retrieve the information it needs to create a response to the Web services request. In “Interaction diagram” on page 286, we describe the interaction between the classes shown in our class diagram, as well as describe the flow of control through the classes and the application.

The ExchangeRateRequest and ExchangeRateReply classes are used to manipulate the XML request and reply messages.
The interaction (sequence) diagram in Figure 10-5 on page 287 shows the flow of control through the application:

1. FinanceBean invokes the getExchangeRate method of ExchangeRateBean to get the exchange rate of a specific planet.
2. ExchangeRateBean determines that it should get the exchange rate using Web services and invokes the getExchangeRate method of ExchangeRateWSBean.
3. ExchangeRateWSBean acts as an interface to the Web service. It first calls setEndPoint() on the Proxy class, passing it to the URL of the remote Web service provider server.
4. ExchangeRateWSBean then constructs a new ExchangeRateRequest using the passed planet name.
5. ExchangeRateWSBean calls the toString() method on the ExchangeRateRequest to get an XML string representing the exchange rate request message.
6. ExchangeRateWSBean then calls the getExchangeRateXML() method on the ExchangeRateServerProxy, passing the XML exchange rate request message.
7. The ExchangeRateServerProxy sets up Call and Parameter objects, then invokes the remote Web service via SOAP and HTTP. The Web service
provider is accessed via the SOAP server on the provider’s machine, using the SOAP RPC mechanism.

8. Control now passes to the RPCRouterServlet on the Web service provider side. This router receives the Parameter object and thus knows to call the getExchangeRateXML() method on the ExchangeRateServerBean.

The ExchangeRateServerBean returns an XML exchange rate reply message back through the chain to ExchangeRateWSBean.

9. ExchangeRateWSBean then constructs a new ExchangeRateReply using the returned XML exchange rate reply message.

10. ExchangeRateWSBean calls the getRate() method on the ExchangeRateReply to get the exchange rate returned in the XML exchange rate reply message.

The exchange rate then gets passed back through the chain to the requester side and to the FinanceBean that made the request to begin with.

Figure 10-5 Sequence diagram for Web services requester and provider
10.3 Enterprise application design

In this section we describe the architecture or design decisions that went into building our sample Web services application, which we described in 10.1, “Architectural overview model” on page 282. This sample application is an extension of the PDK application, extended to use Web services to communicate between the PDK and the existing enterprise system, in our case, the fictitious Super Saturn system.

There are several architectural decisions that go into designing Web service providers and requesters. These include:

- Application pattern
- Transmission pattern
- SOAP messaging mechanism
- Static or dynamic Web services discovery
- Synchronous vs. asynchronous Web services
- Message structure

We next describe the architectural decisions that went into the design of our sample application, the PDK application using Web services.

Application pattern

In the case of our implementation of the PDK application, our first architectural decision was that we would implement a Self-Service business pattern directly connecting to the enterprise. This we call the Self-Service::Directly Integrated Single Channel application pattern, the basis of this publication.

Transmission pattern

Our next choice was to define a request-response transmission pattern. In this pattern, the Web service receives a single request and sends a single response and then the session is closed. This is an appropriate pattern for our case, where the PDK application simply requests an exchange rate from an enterprise system and receives a response via Web services.

SOAP messaging mechanism and synchronous pattern

The SOAP messaging mechanism we selected in our application is the RPC mechanism. Our choice was between using the Remote Procedure Call (RPC) mechanism or the message-oriented communication mechanism. We decided on RPC because this is the simplest and most straightforward method. It is also the most commonly used today. Also, we made the assumption that this would be a synchronous message transfer and we did not have a need for a delivery
confirmation. The message-oriented communication mechanism is most appropriate for asynchronous or confirmed delivery types of scenarios. Since our application had no need for these features, the RPC messaging mechanism was the most appropriate.

**Message structure**

This example scenario uses two XML messages (a request and a response) that are exchanged between the Web service requester and the Web service provider. These XML messages contain tags that specify the name of the planet (the request) and the resulting exchange rate (the response). See 10.4, “XML messaging design and development” on page 289, for details.

### 10.4 XML messaging design and development

In this section we look into the structure of the XML messages that we use in the PDK application. We also discuss the XML Schema, the XML DTD, and how to create the Java classes from the schema.

#### 10.4.1 XML message structure

The XML messages used in the PDK application use custom schema, rather than using any external business schema. When using an XML Schema grammar that is not specified externally, each instance document must specify the location of the grammars it uses. This can be done using an xsi:schemaLocation attribute if they use name spaces, or an xsi:noNamespaceSchemaLocation attribute otherwise.

The PDK application has two message structures; one message for the exchange rate request and one message for the exchange rate reply.

The header element for both messages consists of:

- **SourceName element**, containing the information about the application that constructed the message.
- **Version element**, containing the version number of the message structure. If a new message structure is created you can use a new version number to distinguish the new and old message structures.
- **CreateDate element**, containing the date and time when the message was created.

The body element of the exchange rate request message consists of the planet element, containing the name of the planet for the request.
The body element of the reply message consists of:

- Planet element, containing the name of the planet for the request
- Rate element, containing the exchange rate for the requested planet

Figure 10-6 shows RateRequest.xml in the WebSphere Studio XML editor’s Design view.

Example 10-1 shows a sample request message for JUPITER.

Example 10-1  Request message

```xml
<?xml version="1.0" encoding="UTF-8"?>
<RateRequest xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="RateRequest.xsd">
  <Header>
    <SourceName>PDKLite</SourceName>
    <Version>1</Version>
    <CreateDate>2003-03-04T19:21:23</CreateDate>
  </Header>
  <Body>
    <Planet>JUPITER</Planet>
  </Body>
</RateRequest>
```

Example 10-2 shows a sample reply message representing an exchange rate of “3” retrieved from the back-end system.

Example 10-2  Reply message

```xml
<?xml version="1.0" encoding="UTF-8"?>
<RateReply xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="RateReply.xsd">
  <Header>
    <SourceName>ExchangeRateServer</SourceName>
  </Header>
  <Body>
    <Planet>MARS</Planet>
    <Rate>3</Rate>
  </Body>
</RateReply>
```
10.4.2 XML DTD and XML Schema

After defining our sample XML messages using the WebSphere Studio XML editor, we can then generate a DTD and an XML Schema for each from the XML perspective in WebSphere Studio, as shown in Figure 10-7.

![Figure 10-7 Creating an XML Schema from an XML document](image)

Although we generated a DTD for the PDK XML messages, we did not actually use it. We used the XML Schema instead, for its strong typing of elements and attributes. After creating the XML Schema, additional restrictions can be applied to the elements. For example, the value of the rate element needs to be an integer. You can work on the XML Schema from WebSphere Studio’s XML perspective, using the XML Schema Outline view and XML Schema editor, as shown in Figure 10-8 on page 292.
Example 10-3 shows the XML Schema for the reply message.

Example 10-3  RateReply.xsd

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Body">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="Planet"/>
        <xsd:element ref="Rate"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="CreateDate" type="xsd:dateTime"/>
  <xsd:element name="Header">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="SourceName"/>
        <xsd:element ref="Version"/>
        <xsd:element ref="CreateDate"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="RateReply"/>
</xsd:schema>
```
10.4.3 XML and JavaBean

After generating the DTD or the XML Schema of the message it is possible to generate JavaBeans for working with instances of documents (or messages in our case). The following files are generated for you:

- A Java class for each element in the document
- A factory class for creating document instances
- Optionally, a sample application that creates an XML document instance and prints it on the screen

For the PDK application, however, the following custom-built classes were used for manipulating XML messages:

- Message, a common base class for XML messages including ExchangeRateRequest and ExchangeRateReply. Message is used to:
  - Construct an XML document from a stringified XML message.
    It validates all such messages using the message's XML Schema. Remember that validation is an expensive process, so this may not be appropriate in a production application.
  - Programatically set and get elements from the message.
  - Stringify the XML message document using the DOMWriter.
Patterns: Self-Service Application Solutions Using WebSphere V5.0

- ExchangeRateRequest, a PDK-specific class for working with XML exchange rate request messages.
- ExchangeRateReply, a PDK-specific class for working with XML exchange rate reply messages.
- MessageResolver, a class used to find the XML Schema file that the message is validated with.
- DOMWriter, a simple DOM writer used to write a DOM document to an output stream.
- RateTest, a test class for constructing ExchangeRateRequest and ExchangeRateReply XML messages.

10.5 Application development using Web services

In the following sections, we describe the building of a sample application using Web services. As previously mentioned, we extend the PDK application that we described in Chapter 9, “PDK sample overview” on page 253. We assume that the PDK application is connecting to a remote enterprise system on Saturn and that we are using Web services to send messages between the two systems.

For further coverage of Web services development using IBM WebSphere Studio Application Developer, see the redbook WebSphere Version 5 Web Services Handbook, SG24-6891.

10.5.1 Preparing for development

The complete PDK V5 application can be loaded into WebSphere Studio as described in Appendix B, “PDK sample setup” on page 421. When developing this Web services sample scenario, we started out with the projects shown in Figure 10-9 on page 295.
10.5.2 Approach to developing the sample Web services components

We are extending the PDK application to demonstrate how to use Web services to communicate between the PDK application and an existing enterprise application. This existing enterprise application may be located:

- On the same server as the PDK application
- On another server on the same LAN as the PDK application
- On another server on the same intranet, or private network, as the PDK application

The enterprise application could also be located on another server accessible on the Internet. However, our focus is on intra-enterprise applications with the Self-Service business pattern. The Extended Enterprise, or business-to-business, business pattern can be used to connect inter-enterprise applications.

In order to demonstrate how to use Web services to connect the PDK application to a remote enterprise application, we use the following steps:

- Create a sample enterprise-tier application, implemented by an enterprise bean in our case.
- Create a Web service provider for the sample enterprise-tier application.
- Create a Web service requester for the PDK application.
- Have the PDK application place a request to the remote enterprise application through the Web services interface and receive a reply via Web services.
Creating the sample enterprise-tier application
In order to create our sample Web services-enabled application, we assume that there is an application called the Super Saturn application that is running on a server somewhere on Saturn. Since we do not want to access that application directly, we build a Web services interface to this application. To do this, we start by building a stateless session EJB as a facade to the remote application. We assume that this facade EJB has access to the Super Saturn application. We then create a facade method on that EJB called getExchangeRate. We assume that this method knows how to get the real exchange rate from the Super Saturn system. In our sample case we simply have the getExchangeRate method return an XML message string that represents the exchange rate for the planet being requested.

Creating a Web service provider
Once we have created our facade stateless session EJB for the enterprise-tier application, the next step is to create the components necessary for a Web service interface to the facade session bean. To accomplish this, we demonstrate how to use the Web Service wizard supplied with IBM WebSphere Studio Application Developer. Starting with the facade session bean, the Web Service wizard automatically generates the necessary files for the Web services interface to that bean. The files generated by the Web Service wizard include:

- WSDL service implementation and interface files
- SOAP deployment descriptors
- A proxy class to access the Web services described in the WSDL

If you do not have access to WebSphere Studio these files can be generated manually.

Creating a Web services requester
Once the sample Web service provider has been created, we can now create the sample Web service requester to talk to the provider. We do this by adding a Web services interface to the existing PDK application. This is accomplished by performing the following tasks:

- Create a stateless session EJB with a method called getExchangeRate, which creates a getExchangeRate request to be sent to the Web service provider.
- Create a Web services client proxy that sends the getExchangeRate request to the Web service provider via the SOAP/HTTP protocols.

As we did on the Web service provider side, we use the Web Service wizard provided by IBM WebSphere Studio Application Developer to generate the Web services interfaces needed to turn the PDK into a Web service requester. We start with the same WSDL files generated by the Web Service Provider wizard.
We assume that the Saturn Guild gave these WSDL files to the other planets so that they could create a Web service requester for the Saturn services. This WSDL information could also be obtained dynamically through the use of the UDDI protocol as described in 6.2.2, “Web services design considerations” on page 115. However, we focus only on static Web services for our sample application.

We are now ready to begin building our Web services sample application. We begin by building the Web service provider, which is a sample application that runs on a remote server (on Saturn). Once we have built and tested our Web service provider, we extend the PDK application to be a Web service requester. We then see how these two applications talk to each other using SOAP.

### 10.5.3 Creating the enterprise-tier application

We begin by building our sample enterprise-tier application that runs on a remote server somewhere in the Guild, accessible by the intranet. We call this project ExchangeRateServer since it represents the Web service provider or “server” from a client-server prospective.

The steps involved in creating our sample Web services server application are:

1. Create a new J2EE enterprise application project to hold our Web service provider application.
2. Create a new EJB project to hold our session EJB.
3. Create a new Web project to hold the SOAP server.
4. Create the facade session bean that gets the exchange rate.
5. Test the facade session bean that gets the exchange rate.

**Create a new J2EE enterprise application**

We start by creating a new J2EE enterprise application in IBM WebSphere Studio Application Developer. To develop a J2EE enterprise application we have to create an enterprise application project (an EAR project). Our EAR project consists of one EJB module and one Web module.

To create an EAR project do the following:

1. In the J2EE Perspective, select File -> New -> Project.
2. Select J2EE in the left pane, Enterprise Application Project in the right pane, and click Next.
3. In the J2EE Specification version window, select Create J2EE 1.3 Enterprise Application project and click Next.
4. In the Enterprise Application Project window:
   a. Enter the Enterprise application project name, which is ExchangeRateServer in our case.
   b. Specify the additional modules that you want to create. Select **EJB module** and **Web module**, and uncheck **Application client module** since we do not use a client project for the sample Web services server project.
   c. The name of the EJB project should be ExchangeRateServerEJB.
   d. The name of the Web project should be ExchangeRateServerWeb.
   e. Check **Use default** for the Directory settings.
   f. Click **Finish**, as shown in Figure 10-10 on page 299.

This creates three new projects for you:
- An enterprise application (EAR) project called ExchangeRateServer
- An EJB project called ExchangeRateServerEJB
- A Web project called ExchangeRateServerWeb
5. Right-click the ExchangeRateServerWeb Web project, select Properties -> Web, and set the Context Root to ExchangeRateServer.
Creating a session EJB for the enterprise application

Creating the session EJB

Defining the getExchangeRate method
if (exchangeRateList == null) {
    rate = Integer.toString(DEFAULT_RATE);
} else {
    rate = exchangeRateList.get(planet).toString();
}

return rate;

The getExchangeRateXML method returns an stringified XML ExchangeRateReply message containing the exchange rate of the planet specified in the passed XML ExchangeRateRequest message. The code for getExchangeRate is shown in Example 10-5.

Example 10-5  Source code for getExchangeRateXML method

public String getExchangeRateXML(String reqMsg) {

    String planet = null;
    try {
        // create the XML message, set the reply queue details
        ExchangeRateRequest exchangeRateRequest =
            new ExchangeRateRequest(reqMsg);
        planet = exchangeRateRequest.getPlanet();
    } catch (Exception e) {
        // ...
    }

    String rate = getExchangeRate(planet);
    String repMsg = null;
    try {
        // create the XML message, set the reply queue details
        ExchangeRateReply exchangeRateReply =
            new ExchangeRateReply("ExchangeRateServer", planet, Integer.parseInt(rate));
        repMsg = exchangeRateReply.toString();
    } catch (Exception e) {
        // ...
    }

    return repMsg;
}

To add these methods to the new session EJB:

1. Open the ExchangeRateServerBean.java file.
2. Add the following import to the file:

   // PDK XML message classes
   import com.ibm.pdk.message.*;

3. Copy the methods from Example 10-4 on page 300 and Example 10-5 on page 301 into the file.

4. Save the file.

**Promoting the new methods to the EJB remote interface**

   In order for the SOAP RPCRouter servlet to call the getExchangeRate method in the session EJB, we must first promote it to the remote interface:

   1. With ExchangeRateServerBean.java still open in the editor, select the `getExchangeRate` method in the Outline view.
   2. Right-click the `getExchangeRate` method and select **Enterprise Bean -> Promote to Remote Interface** from the pop-up menu.
   3. Repeat for the `getExchangeRateXML` method.

**Generating the deployed code**

   To generate the deployed code to support execution in WebSphere V5.0 for the ExchangeRateServer session EJB:

   1. In the J2EE Navigator view, select the **ExchangeRateServerEJB** EJB module.
   2. Before generating the deployed code, we recommend that you validate the EJB project first.

      The validation process for an EJB project tests that the code is compliant with the Sun Enterprise JavaBeans specification. Completing this step greatly enhances the chances of the deployed code generation being successful.

      Right-click the **ExchangeRateServerEJB** EJB module and select **Run Validation** from the pop-up menu.

   3. To generate the deployed code, right-click the **ExchangeRateServerEJB** EJB module and select **Generate -> Deploy and RMIC Code** from the pop-up menu.

   4. After selecting the session bean, click **Finish** in the Generate Deploy and RMIC Code window.

In the Navigator view you should see the deployed code for the ExchangeRateServer EJB in the com.ibm.pdk.server.ejb.exchange package in the ejbModule folder.
We have now completed developing the session EJB and generating the deployed code. The next step is to test the session bean in the local WebSphere instance.

Testing the session EJB for the enterprise application
Before we continue generating a Web service from our newly created session bean, we should test the bean. This allows us to identify any potential problems early in the development phase. The following section describes deploying and testing the session bean we have just created, by performing the following:

► Associating the project with a server configuration
► Running the universal test client
► Invoking an instance of the session bean and testing its methods

Associating the project with a server configuration
We use a single application server instance and configuration, PDKServer, to deploy all of the enterprise application samples developed in this book. To add the ExchangeRateServer project to the WebSphere v5.0 Test Environment server configuration:

1. Switch to the Server perspective and expand Servers in the Server Configuration view.
2. Right-click WebSphere v5.0 Test Environment and select Add -> ExchangeRateServer, as shown in Figure 10-11.

Figure 10-11   Adding the project to the server configuration

Executing the universal test client
IBM WebSphere Studio Application Developer provides a browser-based universal test client enabling you to connect to an EJB and test its methods. There are two approaches to launching the test client:

► The first is to right-click the EJB module and select Run on Server. This starts the server in debug mode and opens the test client.
► The second approach is to start the server manually from the Server perspective, and then launch the test client.
To start the server manually and then launch the test client:

1. In the Server perspective, Server Configuration view, double-click the **WebSphere v5.0 Test Environment** server configuration to start the configuration editor.

2. Click the **Configuration** tab and check that the Enable universal test client checkbox is enabled. Close the editor.

3. In the Servers view, select the **WebSphere v5.0 Test Environment** server instance and start the server using the Start icon, as shown in Figure 10-12, or click **Start** from its pop-up menu.

   ![Figure 10-12  Starting the WebSphere server instance](image)

4. Wait until the server has fully started (indicated either by the **Server server1 open for e-business message** in the Console view or a Started status in the Servers view).

5. To start the universal test client, select the **ExchangeRateServerEJB** project in the Navigator view, right-click and select **Run on Server** from the pop-up menu.

   Alternatively, you can click the Open Web Browser icon in the toolbar and enter the following URL in the location field:

   **http://localhost:9080/UTC**

   The browser opens with the universal test client home page (Figure 10-13 on page 305).
Chapter 10. Web services scenario

Figure 10-13 Home page for the Universal Test Client

**Testing the EJB**

Most of the testing is done using the JNDI Explorer to locate the EJB and the bean page to instantiate EJBs and run their methods.

The first test should be to see if your EJB has been added to the JNDI name space:

1. Click the JNDI Explorer link.
2. Locate the EJB in the JNDI Explorer, `ejb -> ExchangeRateServer` in this case.

To test the ExchangeRateServer session bean:

1. In the JNDI Explorer, click the link for your EJB, `ejb -> ExchangeRateServer` in this case, to open the EJB page.
2. Expand the EJB References tree to see the methods available on the EJB's home interface.
3. Select the EJB's create() link.
4. In the Parameters pane, click the Invoke button to call the create() method. In the result section you should see an instance of the EJB, ExchangeRateServer, returned from the server.
5. Click the Work with Object button. This action adds the remote EJB object, ExchangeRateServer, to the EJB references tree. Expanding the tree for this object displays the methods available on the EJB's remote interface.
6. Click the required remote method link, `getExchangeRate` in this case.
7. In the Parameters pane, enter a Value for each parameter. In this case enter a planet name (MARS, JUPITER, NEPTUNE, or PLUTO).

8. Click the **Invoke** button. The result section displays the exchange rate returned for that planet, as shown in Figure 10.5.4.

![IBM Universal Test Client](image)

**Figure 10-14** Results from the `getExchangeRate` method call on the remote interface

### 10.5.4 Creating the Web service

The following sections describe how to turn the enterprise session EJB we just created into a self-describing Web service using the WebSphere Studio Web Service wizard. The assets generated by the wizard are shown in Figure 10-15.

![Web Services Exchange Rate Application](image)

**Figure 10-15** Assets generated from Web Service wizard during this section

The Web service has a simple interface consisting of a method that returns an exchange rate for the requested planet.
There are a number of steps we perform when executing the wizard:

- Define the Web service type.
- Configure the Web service identity.
- Define the Web service scope.
- Define encoding styles and server mappings.
- Create a service proxy and client mappings.
- Create a sample test client.

Once completed, we have developed the Web service and are able to investigate its content.

10.5.5 Using the Web Service wizard

To launch the Web Service wizard take these steps:

1. Select and right-click the ExchangeRateServerWeb module, and select **New** -> **Other** -> **Web Services** -> **Web Service**.

2. In the first panel of the wizard, shown in Figure 10-16 on page 308:
   a. Select the Web service type of **EJB Web service**.
   b. Select **Start Web service in Web project** to start the server.
   c. Select **Generate a proxy** and **Generate a sample** (for testing).
   d. Select **Overwrite files without warning** if you have to redo the wizard.
   e. Select **Create folders where necessary**.

   Click **Next**.
3. In the Web Service Deployment Settings window, shown in Figure 10-17 on page 309:
   a. Select **Use defaults** for the Web service deployment environment selection.
   b. Select the **ExchangeRateServerWeb** Web project.
   Click **Next**.
4. In the Web Service EJB Configuration window:
   a. Click the **Browse EJB Beans** button. In the Browse EJB Beans window, select the **ExchangeRateServerBean** session bean and click **OK**.
   b. Back in the Web Services EJB configuration window, the wizard has now filled in all the information about the session EJB for us, as shown in Figure 10-18 on page 310. Accept all the defaults and click **Next**.
5. In the Web Service JavaBean Identity window, shown in Figure 10-19 on page 311:
   a. Change the Web Service URI to urn:ExchangeRateServer.
   b. We also abbreviated the ISD and WSDL file paths, as shown in the figure.
   Click **Next**.
6. In the Web Service JavaBean Methods window, shown in Figure 10-20 on page 312, select which methods of the bean you want to expose to the Web service interface, and the encoding style you want to use:

   a. Make sure the getExchangeRate and getExchangeRateXML methods are selected.

   b. Select SOAP encoding for Input and Output encoding for getExchangeRate and for getExchangeRateXML.

   Click Next.
7. In the Web Service Binding Proxy Generation window, shown in Figure 10-21 on page 313:
   a. The WSDL Binding is pre-selected.
   b. Change the Project ExchangeRateServerWeb.
   c. Change the proxy Class to `com.ibm.pdk.server.ws.proxy.ExchangeRateServerProxy`.
   d. Check **Show mappings**.
      Click **Next**.
8. In the Web Service XML to Java Mappings window, the string entry defines the mechanism to map the input XML data type to the Java type expected by our bean. Leave the default mapping (string) for the input parameter.

   Click **Next**.

9. The Web Service SOAP Binding Mapping Configuration window displays the encodings specified for SOAP encoding and Literal XML encoding. These are set based on our previous definitions in the wizard and cannot be changed in our example.

   Click **Next**.

10. In the Web Service Test window, shown in Figure 10-22 on page 314, we have the option to generate a Web application that uses the proxy to test the completed service.

    a. Click **Test the generated proxy**.

    b. Leave the default JSP Folder location.
c. Uncheck **Run test on server** (we will be examining the generated files before executing the application).

Click **Next**.

![Web Service Test window](image)

**Figure 10-22  Web Service Test window**

11. In the Web Service Publication window, leave both Launch the Web Services Explorer options unchecked. We will not be publishing the Web service in this example.

Click **Finish** to complete the wizard.

WebSphere Studio now generates the Web service and associated files. We investigate the generated file next.
10.5.6 Examining the generated files

As we have seen in the previous sections, the Web Service wizard is extremely comprehensive and powerful, and is responsible for automating a large portion of the artifacts based on its input. Before we test the Web service, let us look at the generated files, which are highlighted in Figure 10-23.

![J2EE Navigator view after the generation of the Web service](image)

Based on the selections made when we ran the Web Service wizard, the following files are created in the ExchangeRateServerWeb project:

- The Java Source folder contains the generated `com.ibm.pdk.server.ws.proxy.ExchangeRateServerProxy` proxy class that can be locally invoked by a Web service client.
- The Web Content/admin folder contains SOAP administration client files (see 10.5.8, “Viewing the deployed Web services” on page 317).
- The Web Content/sample/ExchangeRateServer folder contains the generated test client (see 10.5.10, “Web service test client” on page 322).
The Web Content/WEB-INF/isd folder contains the Web service deployment descriptor, ExchangeRateServer.isd. This information about the services that should be made available to clients is used by the SOAP run-time.

The Web Content/WEB-INF/lib folder has three added library files that are part of the SOAP run-time:
- soapcfg.jar
- webservice-runtime.jar
- xsd.bean.runtime.jar

The Web Content/wsdl folder contains the WSDL files:
- ExchangeRateServer.wsdl, service interface
- ExchangeRateServerBinding.wsdl, service binding
- ExchangeRateServerEJB.wsdl, service binding for EJB
- ExchangeRateServerService.wsdl, service implementation

Finally, the Web Content folder itself contains the files:
- dds.xml, the deployment descriptor for SOAP, a concatenation of all the ISD files (only one for now)
- soap.xml, the Apache SOAP server configuration file

### 10.5.7 SOAP router servlets

The final element required to complete our server-side Web service implementation is the definition of the SOAP router servlets in the deployment descriptor of our Web project.

1. Open the ExchangeRateServerWeb project's web.xml file.
2. Switch to the Servlets tab in the editor to see the SOAP servlets, as shown in Figure 10-24 on page 317.
   - rpcrouter
     - This is the SOAP router servlet invoked for all RPC style calls to the Web service. This is used by our application.
   - messagerouter
     - This router is used for all message style calls to the Web service.
Chapter 10. Web services scenario

317

Figure 10-24   SOAP router servlet definitions

The correct router to invoke is defined in the service implementation WSDL file with the <soap:address> tag. SOAP routers are defined for each Web application module deployed on the application server, so multiple routers on different URLs can exist on the same server configuration.

The implementation of these SOAP routers is provided in a file called soapcfg.jar, which is added to a Web Content/WEB-INF/lib folder by the Web Service wizard when it is invoked for the first time in the project.

Close the web.xml file and ignore any changes made.

10.5.8 Viewing the deployed Web services

Included in our Web project is a Web application that can be used to view the deployed Web services in the Web module and to modify their configuration. These files have been added to the Web Content/admin folder of the Web project.

To invoke the XML-SOAP Admin client:

1. Right-click the Web Content/admin/index.html file and select Run on Server from the pop-up menu.

2. Click the List all services link on the left-hand side of the window. The resulting service listing, generated from the dds.xml file, appears as shown in Figure 10-25 on page 318. As you can see, we have successfully deployed one Web service, urn:ExchangeRateServer.
3. Click the `urn:ExchangeRateServer` link to see the deployed service information, as shown in Figure 10-26 on page 319.
This allows us to view all of the important information about the Web service we have developed, including the provider class, its exposed methods, and the provider type—in this case, a stateless session EJB.

Note that the deployment process is not an explicit step. The Web Service wizard generates an entry in dds.xml, which is picked up by the router servlet when it is initialized.

It is also possible to start and stop specific services from the XML-SOAP Admin client using the Start a service and Stop a service links on the left in the window shown in Figure 10-25 on page 318.

This can be performed while an application that uses the Web service is executing. This may be useful when testing how a client application handles a scenario where the Web service is unavailable.

4. Close the browser.
10.5.9 Web service client proxy

An optional step in the Web Service wizard generates a client proxy to the Web service. This was called ExchangeRateServerProxy.java and can be found in the com.ibm.pdk.server.ws.proxy package in the Java Source folder in the ExchangeRateServerWeb project.

This class can be completely regenerated with access only to the generated WSDL files. It is worth investigating this class in more detail because it demonstrates how to programmatically invoke the Web service from a client.

One of the key instance variables in the class is defined as:

```java
private Call call;
```

The Call class (from the org.apache.soap.rpc package) is the main component responsible for completing the SOAP call between the client and the server. Note that the API shields you from the implementation details of the SOAP message contents and significantly simplifies the calling mechanism.

The proxy class sets up the URL of the SOAP router servlet in the declaration of the stringURL variable. This is then used in the getURL method to determine the URL to be invoked from the Web service.

```java
private String stringURL =
    "http://localhost:9080/ExchangeRateServer/servlet/rpcrouter";
```

The proxy has a method to invoke the service, getExchangeRateXML, which has the same method signature as the method included in the Web service implementation bean on the server side.

The Call object is set up before it is invoked by:

- Setting the method name and encoding style:
  ```java
call.setMethodName("getExchangeRateXML");
call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
```

- Setting the target URN to invoke the service:
  ```java
  String targetObjectURI = "urn:ExchangeRateServer"
call.setTargetObjectURI(targetObjectURI);
  ```

- Setting up the call parameters:
  ```java
  Vector params = new Vector();
  Parameter reqMsgParam = new Parameter("reqMsg", java.lang.String.class, reqMsg, Constants.NS_URI_SOAP_ENC);
  params.addElement(reqMsgParam);
  call.setParams(params);
  ```
The Web service is invoked:

Response resp = call.invoke(getURL(), SOAPActionURI);

Finally, the return value is obtained from the response:

Parameter refValue = resp.getReturnValue();

This code defines the URN required to invoke the Web service, the default encoding style for the proxy (SOAP encoding), and the URL of the target SOAP router. Again, these values can be obtained from the generated WSDL files.

See Example 10-6 for the full listing of the getExchangeRateXML method.

Example 10-6  Source code for getExchangeRateXML method

```java
public synchronized java.lang.String getExchangeRateXML(java.lang.String reqMsg) throws Exception {
    String targetObjectURI = "urn:ExchangeRateServer";
    String SOAPActionURI = "";
    
    if (getURL() == null) {
        throw new SOAPException(Constants.FAULT_CODE_CLIENT, "A URL must be specified via ExchangeRateServerProxy.setEndPoint(URL).".);
    }
    
    call.setMethodName("getExchangeRateXML");
    call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
    call.setTargetObjectURI(targetObjectURI);
    Vector params = new Vector();
    Parameter reqMsgParam = new Parameter("reqMsg", java.lang.String.class, reqMsg, Constants.NS_URI_SOAP_ENC);
    params.addElement(reqMsgParam);
    call.setParams(params);
    Response resp = call.invoke(getURL(), SOAPActionURI);
    
    //Check the response.
    if (resp.generatedFault()) {
        Fault fault = resp.getFault();
        call.setFullTargetObjectURI(targetObjectURI);
        throw new SOAPException(fault.getFaultCode(), fault.getFaultString());
    } else {
        Parameter refValue = resp.getReturnValue();
        return ((java.lang.String)refValue.getValue());
    }
}
```
10.5.10 Web service test client

Because we selected the option to generate a client proxy for the example Web service, the wizard also provided the option to build a browser-based client for testing the proxy and Web service. Like the universal test client, this is a great way of performing unit and connectivity tests on the Web service.

Four files are generated in the Web Content/sample/ExchangeRateServer folder of the Web project for the test client:

- **TestClient.jsp**
  
  This is the frame set for the test client and should be the URL you use to launch the browser.

- **Method.jsp**
  
  This page provides a list of the available Web service methods. When a link is selected, Input.jsp is invoked.

- **Input.jsp**
  
  This provides a form for the input parameters for each method defined in Method.jsp. On submitting the form, Result.jsp is invoked.

- **Result.jsp**
  
  Contains an instance of the client proxy, and invokes it using the input parameters from the Input.jsp. On return it invokes a domWriter method (in the JSP) that writes the returned XML document to the output stream.

To start the sample application:

1. Right-click **TestClient.jsp** and select **Run on Server**.

   The embedded browser should be launched, and after a short delay as the JSPs are compiled, the test client is displayed.

2. In the Methods pane, click the **getExchangeRate** link.

3. In the Inputs pane, enter a valid planet name and click **Invoke**. The Result frame should display the string returned from the Web service, as shown in Figure 10-27 on page 323.
Your Web service provider implementation is now complete and tested.

10.5.11 Creating the Web service requester

Now that we have successfully created and tested our exchange rate Web service provider, the next step is to create a client in the PDK Web application, which can query the Web services exchange rate service provider to get the current exchange rate. This demonstrates how to build a Web service client from a given set of WSDL documents.

For the first phase of this implementation, we assume that the Saturn Guild sends the WSDL files to the other planets that have deployed the PDK application, and that they integrate the Web service into the PDK application at development time using static bindings. This is a more realistic scenario than directly copying the client proxy between projects. The more advanced topic of dynamic binding to a Web service and searching for implementations in a UDDI registry is not covered.

The scenario is illustrated in Figure 10-28 on page 324.
To create the PDK Web service requester, you must perform the following steps:

1. Create/select the Web project that will hold the Web service test client. We will use the existing PDKLiteWeb project.

2. Create/select the EJB project that will hold the EJB called by the PDK Lite application to gain access via Web services to the remote enterprise application. We will use the existing PDKLiteEJB project.

3. Copy the Web service provider WSDL from the ExchangeRateServer project to the PDK project.

4. Use the Web service client wizard to generate a client proxy from the WSDL.

5. Test the Web service using a new Web service test client.

6. Create a new stateless session EJB for the PDK application that serves as a Web service requester. This EJB implements a method called getExchangeRate, which takes a String input of a planet name and returns an exchange rate as an int.

7. Test the complete application.

**Copying the WSDL service implementation**

The Web service client proxy can be created from a WSDL service implementation file contained in the same Web project:

1. Create a wsdl folder in the Web Content folder of the PDKLiteWeb project to contain the copied WSDL service implementation file.

2. Create a static folder in the Web Content/wsdl folder.

3. Copy the following files from the Web Content/wsdl folder in the ExchangeRateServerWeb project to the new Web Content/wsdl/static folder in the PDKLiteWeb project:
   - ExchangeRateServer.wsdl
   - ExchangeRateServerBinding.wsdl
Creating the Web service requester client proxy

We can now use the Web service client wizard to create the client proxy. The generated proxy and test client should be similar to the one we created on the Web service provider in 10.5.5, “Using the Web Service wizard” on page 307, but in the PDKLiteWeb application.

To launch the Web service client wizard, follow these steps:

1. Select the `ExchangeRateServerService.wsdl` file in PDKLiteWeb/Web Content/wsdl/static to prefill the wizard fields.
2. Right-click the file and select New -> Other -> Web Services -> Web Service Client, then click Next.
3. In the first window of the wizard:
   a. Leave Client proxy type as Java proxy.
   b. Select Test the generated proxy.
   c. Select Overwrite files without warning if you have to redo the wizard.
   d. Select Create folders where necessary.
   Click Next.
4. In the Web Service WSDL File Selection window, the WSDL file is preselected:
   `/PDKLiteWeb/Web Content/wsdl/static/ExchangeRateServerService.wsdl`
   Click Next.
5. In the Web Service Binding Proxy Generation window, shown in Figure 10-29 on page 326:
   a. Select the PDKLiteWeb project.
   b. Enter a proxy class name of:
      `com.ibm.pdk.ejb.exchangeWS.proxy.ExchangeRateServerProxy`
   Click Next.
6. In the Web Service Test window we have the option to generate a Web application that uses the proxy to test the completed service.
   a. Click **Test the generated proxy**.
   b. Leave the default JSP Folder location.
   c. Select **Run test on server**.
   Click **Finish**.

7. The Web Service Test Client window provides the option of automatically launching the Universal Test Client. We first investigate the generated code and therefore do not select this option.
   Click **Next**.

8. In the Web Service Sample Generation window, we have the option to generate a Web application that uses the proxy to test the completed service.
   a. Select **Generate a sample**.
   b. Leave Launch the sample unchecked (we will be examining the generated files before executing the application).
Testing the Web service requester
To start the Web service requester test client:
1. Navigate to PDKLiteWeb -> Web Content -> sample -> ExchangeRateServer.
2. Right-click TestClient.jsp and select Run on Server.
   The embedded browser should be launched and, after a short delay as the JSPs are compiled, the test client is displayed.
3. In the Methods pane, click the getExchangeRate link.
4. In the Inputs pane, enter a valid planet name and click Invoke. The Result pane should display the string returned from the Web service, as shown in Figure 10-30.

Your Web service requester client proxy is now complete and tested.

Create a session EJB for the Web service requester
The next step is to create a stateless session bean that will call a Web service to access the remote enterprise application. This bean contains a method that returns an exchange rate when passed the planet’s name. This exchange rate is obtained by calling the ExchangeRateServer Web service provider running on the remote enterprise application server. To create this session EJB:

1. Create the session EJB.
2. Create the methods we want to make available on this EJB.
3. Promote these methods to the Remote interface of the EJB.
4. Deploy the EJB.

**Creating the session EJB**
To create a session EJB in the PDKLiteEJB project:

1. Select and right-click the PDKLiteEJB module, and select New -> Other -> EJB -> Enterprise Bean, then click Next.
2. In the Create an Enterprise Bean wizard, set the EJB project to PDKLiteEJB and click Next.
3. In the Create a 2.0 Enterprise Bean window, complete the fields as follows:
   a. Set the enterprise bean type to Session bean.
   b. Set the Bean name to ExchangeRateWS.
   c. Set the Source folder to /ejbModule.
   d. Set the Default package to com.ibm.pdk.ejb.exchangeWS.
   Click Next.
4. In the Enterprise Bean Details window:
   a. Set the EJB binding name to ejb/ExchangeRateWS.
   b. Leave the defaults for the remaining settings.
   Click Finish to create the session EJB.

**Defining the getExchangeRate method**
Next we have to define the getExchangeRate method to return the exchange rate for the requested planet. The getExchangeRate method returns an int value for the exchange rate of the planet requested. The getExchangeRate method calls the getExchangeRate method on the Web service proxy. The proxy in turn creates a Web service request, which is sent to the Web service provider. The Web service provider returns a message containing the exchange rate for the planet requested. This is returned as a string as part of the SOAP message. Thus, when we call the getExchangeRate method on the Web service proxy, we get back a String corresponding to the exchange rate requested. We then simply have to convert that String to an int. The code for getExchangeRate for the ExchangeRateWS EJB is shown in Example 10-4 on page 300.

**Example 10-7  Source code for getExchangeRate method**

```java
public int getExchangeRate(String planet) throws EJBException {
    int rate = 0;
    try {
        com.ibm.pdk.ejb.exchangeWS.proxy.ExchangeRateServerProxy proxy =
        new com.ibm.pdk.ejb.exchangeWS.proxy.ExchangeRateServerProxy();
```
// create the XML message, set the reply queue details
ExchangeRateRequest exchangeRateRequest =
    new ExchangeRateRequest("PDKLite", planet);
String reqMsg = exchangeRateRequest.toString();

String repMsg = proxy.getExchangeRateXML(reqMsg);

// create an XML reply object from the reply message and get the rate
ExchangeRateReply exchangeRateReply = new ExchangeRateReply(repMsg);
rate = exchangeRateReply.getRate();

} catch (Exception e) {
    // ...
}

return rate;

---

To add this method to the new session EJB:

1. Open the ExchangeRateWSBean.java file.

2. Add the following imports to the file:
   // Classes for SOAP
   import java.net.*;
   import org.apache.soap.*;
   // PDK XML message classes
   import com.ibm.pdk.message.*;

3. Copy the method from Example 10-7 on page 328 into the file.

4. Save the file.

Promoting the new methods to the EJB remote interface

In order for the PDK Lite application to call the getExchangeRate method in the session EJB, we must first promote it to the remote interface:

1. With ExchangeRateWSBean.java still open in the editor, select the getExchangeRate method in the Outline view.

2. Right-click the getExchangeRate method and select Enterprise Bean -> Promote to Remote Interface from the pop-up menu.

3. Close the ExchangeRateWSBean.java editor.
Generating the deployed code

To generate the deployed code to support execution in WebSphere V5.0 for the ExchangeRateWS session EJB:

1. In the J2EE Navigator view, select the PDKLiteEJB EJB module.

2. Before generating the deployed code, we recommend that you validate the EJB project first.

   The validation process for an EJB project tests that the code is compliant with the Sun Enterprise JavaBeans specification. Completing this step greatly enhances the chances of the deployed code generation being successful.

   Right-click the PDKLiteEJB EJB module and select Run Validation from the pop-up menu.

3. To generate the deployed code, right-click the PDKLiteEJB EJB module and select Generate -> Deploy and RMIC Code from the pop-up menu.

4. After selecting the session bean, click Finish in the Generate Deploy and RMIC Code window.

In the Navigator view you should see the deployed code for the ExchangeRateWS EJB in the com.ibm.pdk.ejb.exchangeWS package in the ejbModule folder.

We have now completed developing the session EJB and generated the deployed code. The next step is to test the session bean in the local WebSphere instance.

Testing the session EJB for the Web service requester

The first test should be to see if your EJB has been added to the JNDI name space:

1. Copy the com.ibm.pdk.ejb.exchangeWS.proxy package (including the Web service proxy class, ExchangeRateServerProxy.java) from PDKLiteWeb/Java Source to PDKLiteEJB.ejbModule.

2. Start the universal test client by selecting the PDKLiteEJB project in the Navigator view, right-click, and select Run on Server from the pop-up menu.

3. Click the JNDI Explorer link.

4. Locate the EJB in the JNDI Explorer, ejb/ExchangeRateWS in this case.

To test the ExchangeRateWS session bean:

1. In the JNDI Explorer, click the link for your EJB, ejb -> ExchangeRateWS in this case, to open the EJB Page.

2. Expand the EJB References tree to see the methods available on the EJB’s home interface.
3. Select the EJB’s create() link.

4. In the Parameters pane, click the **Invoke** button to call the create() method. In the result section you should see an instance of the EJB, ExchangeRateWS, returned from the server.

5. Click the **Work with Object** button. This action adds the remote EJB object, ExchangeRateWS, to the EJB references tree. Expanding the tree for this object displays the methods available on the EJB’s remote interface.

6. Click the required remote method link, **getExchangeRate** in this case.

7. In the Parameters pane, enter a value for each parameter. In this case enter a planet name (MARS, JUPITER, NEPTUNE, or PLUTO).

   Click the **Invoke** button. The result section displays the exchange rate returned for that planet.

   See “Testing the session EJB for the enterprise application” on page 303 for a detailed example of using the universal test client.

   Your Web service requester implementation is now complete and tested.

### Removing the Web service requester test client

Once you have finished testing the Web service requester, you can delete the Web service requester test client from the PDKLiteWeb module. Delete the following folders/packages from PDKLiteWeb:

- Java Source/com.ibm.pdk.ejb.exchangeWS.proxy
- Web Content/sample/ExchangeRateServer
- Web Content/wsdl

Restart the WebSphere v5.0 Test Environment server and check that ExchangeRateWS EJB still works.
This chapter describes lower-level design recommendations, installation, and configuration specifics for using the J2EE Connector Architecture to connect to CICS on zSeries. We discuss component, class, and interaction diagrams with code snippets related to the Self-Service::Directly Integrated Single Channel application pattern.

It is assumed that the reader is familiar with the PDK scenario and the Transfer Funds use case discussed in Chapter 9, “PDK sample overview” on page 253.
11.1 Architectural overview model

In this chapter, we describe how the PDK application was extended to communicate with an external enterprise system using the J2EE Connector Architecture. The architecture overview model demonstrates the entire system and shows how all of the stakeholders are involved in the overall system.

As can be seen in Figure 11-1, we have highlighted the area that we are focusing on in this chapter. From the J2EE architecture point of view, this is the part of the application server business logic tier that retrieves some business data from a single application in the EIS tier. Results of the information retrieved are processed and sent back to the presentation tier.

![Figure 11-1 Architectural overview model](image)

11.2 System design overview

In the J2EE Connector scenario, we connect the PDK application to a back-end CICS application using the CICS ECI resource adapter. The ECI resource adapter is one of the resource adapters that allows Java applications to connect to CICS. It uses the External Call Interface (ECI) of the CICS Transaction Gateway (CICS TG) to communicate with CICS. CICS TG can link to a CICS
enterprise-tier, or back-end, application passing data in a buffer called the COMMAREA. The CICS application receives messages from the PDK application and it sends back the exchange rate via the COMMAREA. This exchange rate information is used during the transfer funds business function.

The following assumptions apply to the PDK J2EE Connector scenario:

- Of the several resource adapters supported for CICS, we are using a CICS ECI resource adapter for the PDK system.
- Using the resource adapter interface, we can invoke an enterprise-tier application in both a managed or non-managed environment. For our PDK scenario, our application is under a managed environment and it is managed by WebSphere Application Server.
- Our enterprise-tier application for PDK is a COMMAREA-based CICS program, which resides in a zSeries environment.

### 11.2.1 Component model

The following section describes the PDK J2EE Connector scenario components and their relationships. The component model is described from an application developer's point of view rather than a user's point of view. We invoke the enterprise application using CCI calls to perform a service.

Figure 11-2 shows the components of the PDK system and relationships between them for the J2EE Connector scenario.
The J2EE Connector scenario components are:

- **Exchange Rate EJB**
  
  The Exchange Rate session bean component is a custom-developed EJB. It receives a request from the application, as described in Chapter 9, “PDK sample overview” on page 253, to get an exchange rate for a specific planet. Exchange Rate EJB uses the Common Client Interface (CCI) to send a request to a CICS back-end application via a CICS ECI resource adapter. The reply is sent back from the resource adapter with the value of the exchange rate, which is eventually sent back to the session bean. All the interactions between the components are performed in a synchronous manner, that is, the session bean is blocked until the reply is sent. The application resides in a J2EE application server, which means that runtime support for the EJB component is supported by an application server implicitly.

- **CICS ECI resource adapter**
  
  The CICS ECI resource adapter provides the Common Client Interface for the Exchange Rate EJB. It receives a request from the Exchange Rate EJB and passes it to the CICS application using the CICS ECI-specific protocol. The reply from the CICS application is passed back to the Exchange Rate EJB via the resource adapter synchronously. The CICS ECI resource adapter has an interface to an application server that provides the system contacts defined in the J2EE Connector Architecture.

- **Application server**
  
  The application server provides system and component contracts to both the Exchange Rate EJB and the CICS ECI resource adapter.

- **CICS application**
  
  The CICS application receives a request from the CICS ECI resource adapter containing the name of a planet. It returns the exchange rate for the specified planet using the reply COMMAREA. The reply COMMAREA is passed to the ExchangeRateJ2CBean synchronously.

### 11.2.2 Object model

In this section, we provide an object model for our J2EE Connector scenario.

**Class diagram**

Figure 11-3 on page 337 shows a class diagram for the J2EE Connector scenario. FinanceBean, ExchangeRateBean, ExchangeRateJ2CBean, and GenericRecord are the classes that were custom developed for the PDK application. The rest of the classes are provided by the ECI resource adapter or the J2EE packages.
Figure 11-3  Class diagram for J2EE Connector scenario

The diagram shows the static relationships between the classes. FinanceBean uses ExchangeRateBean, which then uses ExchangeRateJ2CBean to retrieve an exchange rate from the enterprise tier. ExchangeRateJ2CBean uses ConnectionFactory, Connection, Interaction, GenericRecord, and ECIInteractionSpec. The GenericRecord class provides a byte array for the CICS COMMAREA, which is used to pass the exchange rate record to the back-end application. The GenericRecord class is an implementation of the Record class provided by J2EE Connector class library.

The ECIInteractionSpec class is the specification of an interaction only for CICS ECI. The superclass of ECIInteractionSpec is InteractionSpec. The various kinds of resource adapters extend InteractionSpec.

InteractionSpec and Record are the only classes that need to be specific to a particular resource adapter. If an application needs to interact with another resource adapter, the only changes that need to be made are to use the new Record class and InteractionSpec class. The use of the other classes, such as ConnectionFactory, Connection, and Interaction, remains the same. For example, if the application needs to have an interface with the IMS Resource adapter, the application needs a new Record for IMS and it needs to create and set an IMS InteractionSpec, which is passed to the generic interaction object.
Interaction diagram

Figure 11-4 on page 339 shows the message interactions of the classes. The sequence of the message flow is as follows:

1. FinanceBean invokes the getExchangeRate method of ExchangeRateBean to get the exchange rate of a specific planet.
2. ExchangeRateBean determines that it should get the exchange rate using CICS and invokes the getExchangeRate method of ExchangeRateJ2CBean.
3. ExchangeRateJ2CBean gets a connection using a connection factory. The runtime connection attributes are configured from the application server connection factory properties. This may not mean a physical connection to the enterprise has been opened. A physical connection may have been opened by another connection instance and pooled by an application server. An application program only needs to be concerned with the logical connection given by the ConnectionFactory, no matter if the physical connection is opened or not.
4. ExchangeRateJ2CBean invokes the connection createInteraction method.
5. Connection creates the interaction.
6. ExchangeRateJ2CBean creates a CICS ECI interaction spec.
7. ExchangeRateJ2CBean sets the COMMAREA length in CICS ECI interaction spec.
8. ExchangeRateJ2CBean sets the reply length in CICS ECI interaction spec.
9. ExchangeRateJ2CBean sets the CICS program name in CICS ECI interaction spec.
10. ExchangeRateJ2CBean sets the interaction verb in CICS ECI interaction spec, for synchronous calling in this scenario.
11. ExchangeRateJ2CBean creates a GenericRecord and sets it with the planet name.
12. ExchangeRateJ2CBean invokes the execute method of the interaction passing the interaction spec and the input and output record created in steps 6 through 11. The input and output record is the same object in this scenario. The output record is returned synchronously to ExchangeRateJ2CBean.
13. ExchangeRateJ2CBean retrieves the exchange rate from the record object.
14. ExchangeRateJ2CBean closes the interaction.
15. ExchangeRateJ2CBean closes the connection. This may not mean that a physical connection to the enterprise is closed, but an application program should close the logical connection every time an interaction is completed.
Finally the exchange rate is converted into ASCII format and returned to the ExchangeRateBean, which returns it to the FinanceBean.

11.3 Low-level design considerations

In this section, we discuss some of the low-level design considerations made while adding J2EE Connector support to the PDK application.
11.3.1 Connection management

This section describes the things need to be done by the application program to connect to CICS using a J2EE Connector. According to the class diagram shown in Figure 11-3 on page 337, the application program uses ConnectionFactory and Connection to manage a connection. A reference of ConnectionFactory is obtained from the JNDI name space using InitialContext, and an instance of Connection is obtained from ConnectionFactory. The application program only uses these two classes to handle connections. The physical connection management is done by the following components:

- **Application server and resource adapter**
  Connections from the application server to the CICS Transaction Gateway via the ECI resource adapter can utilize the application server pool manager to reuse free connections as they become available. This applies both to network connections when using a remote Gateway daemon and local connections when using a local CICS Transaction Gateway. Similarly, terminals in use via the EPI resource adapter are also pooled, although connections are not pooled for the EPI resource adapter.

- **Inside the resource adapter**
  Some resource managers have their own connection management functions. The CICS Transaction Gateway internally manages the connections from the Client daemon to the attached CICS regions, and this is not visible to the application server or resource adapter.

**JNDI lookup**

First of all, the PDK application looks up a connection factory instance using the JNDI interface. Example 11-1 shows a sample of this lookup.

**Example 11-1  Acquiring a connection factory using JNDI lookup**

```java
// get JNDI context
javax.naming.InitialContext ctx = new javax.naming.InitialContext();
// get local JNDI environment
javax.naming.Context env = (javax.naming.Context)ctx.lookup("java:comp/env");
cf = (javax.resource.cci.ConnectionFactory) env.lookup("eis/ref/PDKCICS");
```

Prior to running this sample, you need to define a resource reference and binding for eis/ref/PDKCICS. See “Adding the resource reference” on page 351 and “Adding the resource reference binding” on page 351 for details.
Dealing with a connection

The application component invokes the getConnection method on the connection factory to get a CICS connection, as shown in Example 11-2. The returned connection instance represents an application-level handle to an underlying physical connection.

Example 11-2   Acquiring a connection

```java
// use connection factory to get a connection handle
conn = cf.getConnection();
```

You can specify a user ID and password in a connection spec, as shown in Example 11-3.

Example 11-3   Acquiring a connection specifying user ID and password

```java
// create a connection spec
ECIConnectionSpec connspec = new ECIConnectionSpec(userid, password);
// use connection factory to get a connection handle
conn = cf.getConnection(connspec);
```

The enterprise system uses the user ID and password for authentication and authorization. A connection spec is specific to a resource adapter, so the application needs to use the required connection spec, such as ECIConnectionSpec. Creating ConnectionSpec is optional because the authentication details can be specified when deploying the application. See 11.3.6, “Security” on page 346, for details.

After the component finishes with the connection, it closes the connection using the close method, as shown in Example 11-4.

Example 11-4   Closing a connection

```java
// close a connection
conn.close();
```

If an application component fails to close an allocated connection after its use, that connection is considered an unused connection. The application server manages the clean up of unused connections. When a container terminates a component instance, the container cleans up all connections used by that component instance.
11.3.2 Creating the input and output record

After a connection to the enterprise tier is established, you need to create an input and an output record to communicate with the enterprise application. This section describes how to create these records with two different approaches:

- Implement the javax.resource.cci.Record interface to make a custom record.
- Use a tool such as IBM WebSphere Studio Application Developer Integration Edition to import the COMMAREA structure from an enterprise application written in C or COBOL.

Creating a custom record

A record is the Java representation of a data structure used as input or output to an EIS function. You can create a custom record that is specific to an enterprise application by extending the Record interface. In Example 11-5, the class GenericRecord supports a simple getter-setter design pattern for its field values for a generic ECI application. If you are not going to use a development tool for the record generation, this generic record provides a simple interface to communicate with COMMAREA-based enterprise applications.

Example 11-5  Implementation of Record

```java
package com.ibm.pdk.ejb.exchangeJ2C.record;

public class GenericRecord implements javax.resource.cci.Record,
    javax.resource.cci.Streamable {

    private byte commarea[] = null;

    public GenericRecord() {
        super();
    }

    public GenericRecord(byte[] comm) {
        setCommarea(comm);
    }

    public Object clone() throws CloneNotSupportedException{
        return super.clone();
    }

    public void setCommarea(byte[] comm) {
        try {
            read(new java.io.ByteArrayInputStream(comm));
        } catch (java.io.IOException ioe) {
        }
    }

    // Other methods...
}
```
public byte[] getCommarea() {
    return commarea;
}

public void read(java.io.InputStream in) throws java.io.IOException {
    commarea = new byte[in.available()];
    in.read(commarea);
}

public void setRecordName(String arg1) {}

public String getRecordName() {
    return null;
}

public void setRecordShortDescription(String arg1) {}

public String getRecordShortDescription() {
    return null;
}

public void write(java.io.OutputStream out) throws java.io.IOException {
    out.write(commarea);
    out.flush();
}

Generating a record using development tools

The IBM VisualAge for Java - Enterprise Access Builder (EAB) tool provides the capability to generate CCI custom records from COBOL structures. On the other hand, IBM WebSphere Studio Application Developer does not have a capability to generate a CCI record. However, the records generated by EAB can be used to build and deploy the application in the WebSphere Studio environment.

IBM WebSphere Studio Application Developer Integration Edition introduces a new dynamic record generation mechanism using WSDL. Service messages are described by the XML schema language and stored in a WSDL file. At runtime, the input and output messages are constructed dynamically using this meta-information. The generated code and WSDL files are specific to the WebSphere Studio Integration Edition environment so it cannot be used in the standard WebSphere Studio or VisualAge for Java environments.
11.3.3 Data conversion

An enterprise tier might use a different code set from the client application. Typically the enterprise tier resides in a mainframe using the EBCDIC code set, and the client application is written in Java using Unicode. Likewise, encoding of an integer in the PC world is different from that in the UNIX and mainframe worlds. There are various options to performing data conversion between different systems. Please refer to 7.3.4, “Data conversion” on page 190, for details.

If you are using a record generation tool such as EAB with VisualAge for Java or WebSphere Studio Integration Edition to create records, data conversion is automatically done by the tool when you specify relevant information.

If you create a record manually using a generic record, you need to perform code conversion yourself.

For the PDK scenario, we use standard javax.io classes to perform a data conversion. The following samples show how to perform the code conversion using these classes.

Example 11-6 shows how the String planet name is converted to EBCDIC using the String getBytes method, and then used to set the input record.

```java
Example 11-6  Data conversion (Unicode to EBCDIC)
rateRecord = new GenericRecord(planet.getBytes("IBM037"));
```

Example 11-7 shows how the whole output COMMAREA returned from the enterprise tier is converted from EBCDIC to a String format using a String constructor.

```java
Example 11-7  Data conversion (EBCDIC to Unicode)
String commarea_rate = new String(rateRecord.getCommarea(),"IBM037");
```

11.3.4 Executing the enterprise application

An interaction enables an application program to execute functions provided by the enterprise tier. The execute method takes an input record, output record, and an InteractionSpec. This method executes the enterprise tier function represented by the InteractionSpec and updates the output record.
An Interaction instance is created from a Connection and is required to maintain its association with the Connection instance. The close method releases all resources maintained by the resource adapter for the interaction. Closing an Interaction instance should not close the associated connection instance.

Example 11-8 shows how to prepare the InteractionSpec and how to execute the interaction on a CICS ECI resource adapter.

Example 11-8 InteractionSpec and interaction

```java
// use connection to create an interaction object
Interaction interaction = connection.createInteraction();
// Create and setup the CICS ECI interation spec
ECIInteractionSpec iSpec = new ECIInteractionSpec();
iSpec.setCommareaLength(12);
iSpec.setReplyLength(12);
iSpec.setFunctionName("CALCRATE");
iSpec.setInteractionVerb(ECIInteractionSpec.SYNC_SEND_RECEIVE);
// Create rateRecord
...
// Invoke the program in synchronous manner
interaction.execute(iSpec, rateRecord, rateRecord);
```

In the PDK scenario, we are using a synchronous ECI call because the scenario does not require a non-blocking asynchronous call since we are only accessing one EIS at a time. Please refer to 6.3, “Design guidelines for J2EE Connectors” on page 135, for further details on synchronous and asynchronous CCI calls.

### 11.3.5 Transaction management

The two-phase commit capability of CICS ECI resource adapters is only supported with WebSphere Application Server for z/OS. When the application server resides in a distributed environment, like our PDK test environment, the transaction-type attribute of the EJB should be Bean so that the CICS resource is out of scope of global transactions. Instead, all the syncpoint control should be performed as a local transaction in the application.

If multiple CICS programs are needed to accomplish one transaction for a resource adapter, then local transaction calls should be used in ExchangeRateJ2CBean.

In our PDK scenario we have a session bean called FinanceBean, which talks to several resource managers. If needed, the transaction manipulation should be done by this component using local transaction calls. The PDK application does not issue local transaction calls, so syncpoint control is done within the resource manager.
Please refer to 6.3, “Design guidelines for J2EE Connectors” on page 135, for further details on transaction management.

11.3.6 Security

The PDK application does not need to sign on to EIS because the user ID and the password can be specified in the resource adapter properties in the WebSphere Application Server runtime environment or the WebSphere Studio Integration Edition test environment. Refer to “Adding a J2C adapter to the server configuration” on page 353 for the detailed setup.

The user ID and password can also be specified using a component-managed authentication alias, a container-managed authentication alias, or programatically using the ConnectionSpec.

11.4 Application development using J2EE Connectors

This section describes the implementation of a sample application using J2EE Connectors and IBM WebSphere Studio Application Developer V5.0. We extend the PDK application so it can access an enterprise-tier CICS application using the J2EE Connector scenario described in 11.2, “System design overview” on page 334.

11.4.1 Installing the J2EE Connector resource adapter

To import the CICS J2C resource adapter into your WebSphere Studio workspace:

1. Open the J2EE perspective, J2EE Navigator view.
2. Select File -> Import from the Studio main menu.
3. In the Import window, select RAR file as the import source and click Next.
4. In the next window:
   a. Click Browse, navigate to the RAR file to import (cicseci.rar), and click OK.
   b. Set the New project name to PDKConnector.
   Click Finish to import the J2C resource adapter.

Note: The CICS ECI RAR file, cicseci.rar, is packaged with the IBM CICS Transaction Gateway for Windows Version 5.0.
11.4.2 Create a session EJB

In this step we create a stateless session bean named ExchangeRateJ2C. The purpose of this bean is to access the remote CICS enterprise application using the J2EE Connector. This bean contains a method that returns an exchange rate when passed the planet's name. This exchange rate is obtained by calling the CICS application running on the remote enterprise application server. To create this session EJB:

1. Set the project build path.
2. Create the record class.
3. Create the session EJB.
4. Create the methods we want to make available on this EJB.
5. Promote these methods to the Remote interface of the EJB.
7. Deploy the EJB.

Setting the project build path
To add the CICS EIC resource adapter to the EJB project build path:

1. Right-click PDKLiteEJB and select Properties -> Java Build Path -> Projects.
2. Check the PDKConnector project in the list of required projects on the build path, then click OK.

Create the record class
We need to create a custom record class that implements javax.resource.cci.Record. To create the GenericRecord class described in “Creating a custom record” on page 342:

1. Select the ejbModule folder in the PDKLiteEJB project.
2. Right-click ejbModule, select New -> Other -> Java -> Class, then click Next.
3. In the Java Class window, shown in Figure 11-5 on page 348:
   b. Set Name to GenericRecord.
   Click Finish.
4. Use the source code editor to input the GenericRecord source. The source is provided in Example 11-5 on page 342.

Creating the session EJB

To create a session EJB in the PDKLiteEJB project:

1. Select and right-click the PDKLiteEJB module, and select **New -> Other -> EJB -> Enterprise Bean**, then click **Next**.

2. In the Create an Enterprise Bean wizard, set the EJB project to PDKLiteEJB and click **Next**.

3. In the Create a 2.0 Enterprise Bean window, complete the fields as follows:
   a. Set the enterprise bean type to Session bean.
   b. Set the Bean name to ExchangeRateJ2C.
   c. Set the Source folder to /ejbModule.
   d. Set the Default package to com.ibm.pdk.ejb.exchangeJ2C.

   Click **Next**.

4. In the Enterprise Bean Details window:
   a. Set the Transaction type to Bean.
   b. Set the EJB binding name to ejb/ExchangeRateJ2C.
   c. Leave the defaults for the remaining settings.

   Click **Finish** to create the session EJB.

Define the getExchangeRate method

Next, we have to define the getExchangeRate method to return the exchange rate for the requested planet. The getExchangeRate method returns an int value for the exchange rate of the planet requested. The getExchangeRate method calls the CALCRATE function on the CICS enterprise tier. The CALCRATE
function returns a message containing the exchange rate for the planet requested. This is returned in the COMMAREA field in GenericRecord. Thus, when we execute the J2EE Connector interaction with CICS, we need to convert the EBCDIC commarea to a Unicode Java String. We then simply have to convert that String to an int. The code for getExchangeRate for the ExchangeRateJ2CBean is shown in Example 11-9 on page 350.

**Note:** For simplicity, we have not including logging or connection factory caching in Example 11-9. Please see ExchangeRateJ2CBean in the PDK sample for a more complete code example. The PDK V5 sample is available for download from the Patterns for e-business Web site:

public int getExchangeRate(String planet) {
    try {
        // get JNDI context
        javax.naming.InitialContext ctx = new javax.naming.InitialContext();
        // get local JNDI environment
        javax.naming.Context env = (javax.naming.Context) ctx.lookup("java:comp/env");
        ConnectionFactory cf = (javax.resource.cci.ConnectionFactory) env.lookup("eis/ref/PDKCICS");
        // use connection factory to get a connection handle
        Connection conn = cf.getConnection();
        // use connection to create an interaction object
        Interaction interaction = conn.createInteraction();
        // Set the CICS program name to the function name
        ECIInteractionSpec iSpec = new ECIInteractionSpec();
        iSpec.setFunctionName("CALCRATE");
        iSpec.setCommareaLength(12);
        iSpec.setReplyLength(12);
        iSpec.setInteractionVerb(ECIInteractionSpec.SYNC_SEND_RECEIVE);
        // Create rateRecord
        planet = (planet.trim().toUpperCase() + "            ").substring(0, 11);
        GenericRecord rateRecord = new GenericRecord(planet.getBytes("IBM037");
        // Invoke the program in synchronous manner
        interaction.execute(iSpec, rateRecord, rateRecord);
        interaction.close();
        conn.close();
        String commarea_rate = new String(rateRecord.getCommarea(),"IBM037");
        int rate = new Integer(commarea_rate.substring(8,9)).intValue();
        return(rate);
    } catch (Exception e) {
        //...
        return(2); // default exchange rate
    }
}
Promoting the new methods to the EJB remote interface

In order for the PDK Lite application to call the getExchangeRate method in the session EJB, we must first promote it to the remote interface:

1. With ExchangeRateJ2CBean.java still open in the editor, select the getExchangeRate method in the Outline view.
2. Right-click the getExchangeRate method and select Enterprise Bean -> Promote to Remote Interface from the pop-up menu.
3. Close the ExchangeRateJ2CBean.java editor.

Adding the resource reference

We must edit the deployment descriptor in the PDKLiteEJB project, and define a resource reference for the J2EE Connector resource adapter:

1. Open EJB Deployment Descriptor under the PDKLiteEJB module.
2. Select the References tab.
3. Select ExchangeRateJ2C, then click Add.
4. In the Add Reference wizard, select EJB resource reference, and click Next.
5. In the Add EJB Resource Reference window:
   a. Set the Name to eis/ref/PDKCICS.
   b. Set the Type to javax.resource.cci.ConnectionFactory.
   c. Set Authentication to Application.
   Click Finish to add the resource reference.

Adding the resource reference binding

To complete our session bean, we must define a JNDI binding for the J2EE Connector resource reference:

1. Still in the EJB Deployment Descriptor, References tab, select ExchangeRateJ2C -> ResourceRef eis/ref/PDKCICS.
2. Set the JNDI name to eis/ECICICS, as shown in Figure 11-6 on page 352.
   The resource reference is shown in Figure 11-6 on page 352.
3. Save and close the EJB Deployment Descriptor.

Generating the deployed code
To generate the deployed code to support execution in WebSphere V5.0 for the ExchangeRateJ2C session EJB:

1. In the J2EE Navigator view, select the PDKLiteEJB EJB module.
2. Before generating the deployed code, we recommend that you validate the EJB project.
   The validation process for an EJB project tests that the code is compliant with the Sun Enterprise JavaBeans specification. Completing this step greatly enhances the chances of the deployed code generation being successful.
   Right-click the PDKLiteEJB EJB module and select Run Validation from the pop-up menu.
3. To generate the deployed code, right-click the PDKLiteEJB EJB module and select Generate -> Deploy and RMIC Code from the pop-up menu.
4. After selecting the session bean, click Finish in the Generate Deploy and RMIC Code window.

In the Navigator view you should see the deployed code for the ExchangeRateJ2C EJB in the com.ibm.pdk.ejb.exchangeJ2C package in the ejbModule folder.

We have now completed developing the session EJB and generated the deployed code. The next step is to test the session bean in the local WebSphere instance.
11.4.3 Testing the session EJB

This section describes deploying and testing the ExchangeRateJ2C session bean we have just created, by performing the following:

- Adding a J2C adapter to the server configuration
- Invoking an instance of the session bean and testing its methods

**Adding a J2C adapter to the server configuration**

To add a J2C resource adapter and connection factory to your server configuration:

1. Switch to the Server perspective and expand **Server Configurations** in the Server Configuration view.
2. Click the **J2C** tab, then click the **Add** button on the right of the J2C Resource Adapters list under Node Settings.
3. In the Create Resource Adapter window, select **PDKConnector** in the Resource Adapter Name field, then click **OK**.
4. Select the new **PDKConnector** in the J2C Resource Adapters list, then click the **Add** button on the right of the J2C Connection Factories.
5. In the Create Connection Factory window:
   - Set the **Name** to **ECICICS**.
   - Set the JNDI name to **eis/ECICICS**.
   - Accept the default values for the remaining fields.
   - Click **OK**.
6. In the Resource Properties list, shown in the lower part of Figure 11-7 on page 354, complete the required connection properties. For CICS:
   - Set the **ServerName** to your CICS region name.
   - Set the **ConnectionURL** and **PortNumber** for your CICS Transaction Gateway.
   - Set your CICS user name and password.
Figure 11-7   Configuring the J2C connection factory

7. Save your changes and close the editor.

Testing the EJB
The first test should be to see if your EJB has been added to the JNDI name space:

1. Start the universal test client by selecting the PDKLiteEJB project in the Navigator view, right-click, and select Run on Server from the pop-up menu.

2. Click the JNDI Explorer link.

3. Locate the EJB in the JNDI Explorer, ejb/ExchangeRateJ2C in this case.

To test the ExchangeRateJ2C session bean:

1. In the JNDI Explorer, click the link for your EJB, ejb -> ExchangeRateJ2C in this case, to open the EJB Page.

2. Expand the EJB References tree to see the methods available on the EJB’s home interface.

3. Select the EJB’s create() link.

4. In the Parameters pane, click the Invoke button to call the create() method. In the result section you should see an instance of the EJB, ExchangeRateJ2C, returned from the server.
5. Click the **Work with Object** button. This action adds the remote EJB object, ExchangeRateJ2C, to the EJB references tree. Expanding the tree for this object displays the methods available on the EJB’s remote interface.

6. Click the required remote method link, **getExchangeRate** in this case.

7. In the Parameters pane, enter a value for each parameter. In this case enter a planet name (MARS, JUPITER, NEPTUNE, or PLUTO).

8. Click the **Invoke** button. The result section displays the exchange rate returned for that planet, as shown in Figure 11-8.

![Figure 11-8](image)

**Figure 11-8**  Results from the getExchangeRate method call on the remote interface

See “Testing the session EJB for the enterprise application” on page 303 for a detailed example of using the Universal Test Client.

Your ExchangeRateJ2C EJB implementation is now complete and tested.

### 11.5 Runtime configuration

In this section we discuss some of the more pertinent points about runtime configuration of IBM WebSphere Application Server V5.0 and IBM CICS to satisfy the needs of the PDK application.

Please see Appendix A, “Lab environment scenarios” on page 413, for a description of the various lab environments where we deployed the PDK sample application.
11.5.1 Runtime configuration design

There are two typical runtime configurations for the PDK application:

- Local CICS Transaction Gateway
- Remote CICS Transaction Gateway on z/OS

In the PDK scenario we use the remote CICS TG. The CICS TG resides in z/OS and a transaction is invoked from a distributed environment remotely using TCP protocol.

The remote CICS TG on z/OS option is recommended in most situations for the following reasons:

- According to client/server architecture, the CICS client-interface components should be provided by the z/OS enterprise tier.
- Management of the CICS client interface can be performed at the enterprise tier using standard tools and procedures.
- Availability of the CICS client interface depends only on the enterprise tier.
- The CICS TG on zSeries can be shared by multiple client applications, including WebSphere Application Server for z/OS client applications.
- CICS resources and skills can remain focused on z/OS, rather than needing to cover other distributed platforms and locations.

In some circumstances it may not be possible locate the CICS TG on zSeries, due to limited system capacity or when the enterprise system is not under your control. The local CICS TG provides an alternative in such situations.

Let us take a brief look at system design with these configurations.

Local CICS Transaction Gateway

Figure 11-9 on page 357 shows a system design with the application server and CICS TG in the distributed environment. A request from a CICS resource adapter can be passed directly to the CICS TG Client daemon using a local Gateway. The Client daemon then invokes an enterprise-tier CICS program using ECI calls. The underlying protocols that the ECI call uses to invoke z/OS CICS applications are TCP/IP, APPC, or TCP62 (APPC over TCP/IP protocol). The ECI call can be routed to any CICS server, including z/OS CICS regions. TCP/IP connectivity is only possible when using CICS TS V2.2.
Remote CICS Transaction Gateway on z/OS

Figure 11-10 on page 358 shows a system design having a remote CICS TG on z/OS. The protocol between an application server and a CICS TG can be either TCP or SSL, or HTTP or HTTPS. The CICS TG invokes a CICS enterprise-tier program using the External CICS Interface (EXCI). EXCI is an interface that allows a z/OS address space, such as a native Java process running under the z/OS UNIX service, to invoke a CICS transaction. The CICS program is invoked using a mirror transaction that resides in the same CICS region as the CICS application. EXCI calls can be routed to any CICS region within a sysplex, which means that the CICS application can reside in a different z/OS system from the CICS TG.
11.5.2 WebSphere Application Server configuration

This section describes the runtime setup for the CICS ECI resource adapter in the WebSphere Application Server V5.0. If you install the PDK application as described in Appendix B, “PDK sample setup” on page 421, this setup is performed automatically.

Adding a CICS ECI resource adapter

To install the CICS ECI J2C resource adapter in WebSphere V5.0:

1. Locate your CICS ECI J2C resource adapter archive (RAR) file.

   The CICS ECI adapter we used is available with the IBM CICS Transaction Gateway V5.0 for Windows, AIX, and Linux. On Windows, the default location for the adapter is:

   `C:\Program Files\IBM\IBM CICS Transaction Gateway\deployable\cicseci.rar`

   We suggest that you copy this file to the `<WAS_HOME>\installableConnectors` folder. This makes it easy to relocate the .rar file when you want to reinstall it on the same or another WebSphere node.

2. Open the WebSphere Administrative Console.

4. In the Resource Adapters form, set the scope to **Server**, then click **Apply**.
5. Click the **Install RAR** button.
6. In the Install RAR File form, shown in Figure 11-11, browse to the RAR file, cicseci.rar in our case, then click **Next**.

![Install RAR File](image)

**Figure 11-11  Installing the RAR file**

7. In the Resource Adapters Configuration form, leave all the fields blank so that WebSphere uses the values from the RAR file deployment descriptor. Click **OK**.
9. In the Resource Adapters Configuration form:
   a. Set the Name to PDKConnector.
   b. Set the Archive Path to `${CONNECTOR_INSTALL_ROOT}/cicseci.rar` from the drop-down list.
      Click **OK**.

**Adding a connection factory**

To add a connection factory for the PDK application:

1. In the Resource Adapters form, click the new **PDKConnector** in the Resource Adapters list.
2. In the PDKConnector Configuration form, scroll down to Additional Properties and click **J2C Connection Factories**.
3. In the J2C Connection Factories form click **New**.
4. In the J2C Connection Factories General Properties form, shown in Figure 11-12:
   a. Set the Name to ECICICS.
   b. Set the JNDI name to eis/ECICICS.
   c. Accept the default values for the remaining fields.
   Click OK.

![J2C Connection Factories General Properties](image)

5. Back in the J2C Connection Factories form, click the new ECICICS in the J2C Connection Factories list.
6. In the ECICICS Configuration form, scroll down to Additional Properties and click Custom Properties.
7. In the Custom Properties form, shown in the lower part of Figure 11-7 on page 354, complete the required connection properties. For CICS:
   a. Set the ServerName to your CICS region name.
   b. Set the ConnectionURL and PortNumber for your CICS Transaction Gateway.
   c. Set your CICS user name and password.
8. Save your changes.

### 11.5.3 CICS Transaction Server configuration

This section describes steps needed to configure the CICS Transaction Server for Z/OS for the PDK application.
Defining a connection and session from CICS TG

To define a connection using a CICS terminal:

1. Sign on to the CICS terminal and enter the following command:

   `CEDA DEFINE CONN(PDKC) G(PDK)`

2. In the Define Connection form, shown in Figure 11-13, enter the following:

   **NETNAME**  Your CICS TG netname
   **ACCESSMETHOD**  Must be IRc
   **PROTOCOL**  Must be Exci
   **CONNTYPE**  Generic or Specific

   The CONNTYPE can be defined as Specific or Generic. Both Generic and Specific EXCI connections can be shared by multiple users. The only difference is that the pipe is named in the Specific case.

   ![Figure 11-13 Defining a connection to the CICS TG](image)

To define a session using a CICS terminal:

1. At a CICS terminal enter the following:

   `CEDA DEFINE SESS(PDKS) G(PDK)`

2. In the Define Sessions form, shown in Figure 11-14 on page 362, enter the following:

   **CONNECTION**  PDKC
   **PROTOCOL**  Must be Exci
RECEIVEPFX  Two characters, such as PK
RECEIVECOUNT  More than 1

You must leave SENDPFX and SENDCOUNT blank.

\begin{verbatim}
CEDA DEFINE Sessions( PDKS )
  Sessions  ==> PDKS
  Group    ==> PDK
  Description ==>
  SESSION IDENTIFIERS
  Connection ==> PDKC
  SESSName ==> 
  NETnameq ==> 
  M0dename ==> 
  SESSION PROPERTIES
  Protocol  ==> Exci | Lu61 | Exci
  Maximum  ==> 000, 000          0-999
  RECEIVEPfx ==> PD
  RECEIVECOUNT ==> 010          1-999
  SENDPfx ==> 
  SENDCount ==> 1-999
  SENDSize  ==> 04096            1-30720
  RECEIVESize  ==> 04096           1-30720

OVERTYPE TO MODIFY
CEDA DEFINE Sessions( PDKS )
  Sessions  ==> PDKS
  Group    ==> PDK
  Description ==> SESSION IDENTIFIERS
  Connection ==> PDKC
  SESSName ==> 
  NETnameq ==> 
  M0dename ==> 
  SESSION PROPERTIES
  Protocol  ==> Exci | Lu61 | Exci
  Maximum  ==> 000, 000          0-999
  RECEIVEPfx ==> PD
  RECEIVECOUNT ==> 010          1-999
  SENDPfx ==> 
  SENDCount ==> 1-999
  SENDSize  ==> 04096            1-30720
  RECEIVESize  ==> 04096           1-30720

SYSID=PAME APPLID=SCSCPAME
\end{verbatim}

Figure 11-14  Defining a session for the CICS TG

3. To install the connection and the session, enter the following command:

\begin{verbatim}
CEDA I G(PDK)
\end{verbatim}

**Setting up the PDK CICS application**

To set up the PDK CICS application:
1. Invoke a batch job to compile and link the PDK application with the following steps:
   a. CICS translate.
   b. C compile.
   c. Prelink and link with Language Environment® of z/OS.
   d. Place the load module called CALCRATE in one of the CICS RPL libraries.
2. Add a program definition using the CICS terminal:
   a. Sign on to the CICS terminal and type the following:

\begin{verbatim}
CEDA DEFINE PROG(CALCRATE) GROUP(PDK)
\end{verbatim}

   b. Install the program definition using a CICS terminal.

\begin{verbatim}
CEDA INSTALL GROUP(PDK)
\end{verbatim}
You can find the source code for the CALCRATE program in the file
PDK\Lite\TestDrive\Resources\workspacePDK\PDKLiteEJB\ejbModule\xcom\lib
m\pdk\ejb\exchangeJ2C\cics\calcrate.c, included in the PDK V5 sample.

PDK V5 is available from the Patterns for e-business Web site:

Java Message Service scenario

This chapter describes lower-level design recommendations, installation, and configuration specifics for using the Java Message Service (JMS) and IBM WebSphere MQ to connect to an enterprise-tier application. We discuss component, class, and interaction diagrams with code snippets related to the Self-Service:Directly Integrated Single Channel application pattern.

It is assumed that the reader is familiar with the PDK scenario and the Transfer Funds use case discussed in Chapter 9, “PDK sample overview” on page 253.

Throughout this chapter, references to queues are made when referring to IBM WebSphere MQ and to destinations when talking about JMS.
12.1 Architectural overview model

In this chapter, we describe how the PDK application was extended to communicate with an external enterprise system using JMS and IBM WebSphere MQ. The architecture overview model demonstrates the entire system and shows how all of the stakeholders are involved in the overall system.

As can be seen in Figure 12-1, we have highlighted the area that we are focusing on in this chapter. From the J2EE architecture point of view, this is the part of the application server business logic tier that retrieves some business data from a single application in the EIS tier. Results of the information retrieved are processed and sent back to the presentation tier.

![Figure 12-1  Architectural overview model](image)

12.2 System design overview

In the JMS scenario, we connect the PDK application to an enterprise-tier application using the IBM WebSphere MQ JMS Provider. The IBM WebSphere MQ JMS Provider allows Java applications to send and receive messages using WebSphere MQ messaging. The PDK application sends request messages to
the enterprise-tier application using a request queue. The enterprise-tier application receives messages from the PDK application and it sends back the exchange rate using a reply queue. This exchange rate information is used during the transfer funds business function.

12.2.1 Component model

The component model shows a breakdown of both the PDK application server and the enterprise-tier application.

In our sample application, the PDK application servlets parse the request from the client browser and utilize the command pattern to determine which command is called. The back-end connection that is used depends on the planet selected for transfer funds.

In this case, a JMS session bean provides the connection to a pair of WebSphere MQ queues. The JMS session bean binds to the request and reply destinations to send and receive messages for exchange rates.

12.2.2 Object model

In this section we provide an object model for our JMS scenario.

Class diagram

Figure 12-3 on page 368 shows both of the classes developed for this scenario and classes used from the JMS specification. The FinanceBean, ExchangeRateBean, and ExchangeRateJMSBean were custom developed for the front-end PDK application. The ExchangeRateRequestJMSBean,
ExchangeRateReplyJMSBean, and ExchangeRateServerBean were custom developed for the back-end application. The ExchangeRateRequest and ExchangeRateReply classes are used to manipulate the XML request and reply messages.

**Interaction diagrams**

Figure 12-4 on page 369 shows how the FinanceBean uses ExchangeRateBean, which then uses ExchangeRateJMSBean to retrieve an exchange rate from the enterprise tier. The ExchangeRateJMSBean encapsulates the JMS function, only exposing a method called getExchangeRate.
Figure 12-4  Sequence diagram for invoking ExchangeRateJMSBean

Figure 12-5 shows the ExchangeRateJMSBean interacting with JMS objects to send the XML exchange rate request message to the enterprise tier.

ExchangeRateJMSBean constructs a new ExchangeRateRequest using the passed planet name. It calls the toString() method on the ExchangeRateRequest to get an XML string representing the exchange rate request message. ExchangeRateJMSBean sends the stringified XML message to the enterprise tier using the QueueSender.

Figure 12-5  Sequence diagram for sending request message
Figure 12-6 on page 370 shows the second half of the interaction of the ExchangeRateJMSBean with JMS classes.

An XML exchange rate reply message is received from the enterprise tier. ExchangeRateJMSBean constructs a new ExchangeRateReply using the XML reply message. It calls the getRate() method on the ExchangeRateReply to get the exchange rate returned in the XML exchange rate reply message.

12.3 Low-level design considerations

In this section we discuss some of the low-level design considerations made while adding JMS support to the PDK application, including:

- Messaging model
- Looking up JMS resources from JNDI
- Managing connections, sessions, and queue receivers and senders
- Message settings such as message time-to-live, reply-to, and message persistence
Enterprise simulation application design

We conclude with an alternate design that we considered for implementing the JMS scenario.

### 12.3.1 Point-to-point messaging model

The messaging model used for this example is the JMS point-to-point model. The point-to-point model specifies how to work with specific queues of messages. A client typically sends a message to a specific queue and/or receives a message from a specific queue.

Our PDK application is only connecting to a single WebSphere MQ server on the back end and therefore point-to-point is the most suitable model. If there had been the need to listen for multiple exchange rates and display the changing values in real time, then a publish/subscribe model may be more applicable.

The front-end PDK application uses the request/reply messaging pattern. The back-end application uses the message consumer pattern to process requests and the send-and-forget pattern to deliver the reply.

For further reference on JMS point-to-point and publish/subscribe, take a look at the JMS Specification available at:

http://java.sun.com/products/jms/docs.html

### 12.3.2 Message structure

This example scenario uses two XML messages (a request and a response) that are exchanged between the front-end application and the back-end application. These XML messages contain tags that specify the name of the planet (the request) and the resulting exchange rate (the response). See 10.4, "XML messaging design and development" on page 289, for details.

### 12.3.3 JMS resource lookups using JNDI

Looking up JMS resources is not too different from looking up other types of resources in the JNDI directory, as shown in Example 12-1.

**Example 12-1  JNDI lookup of JMS connection factory**

```java
Context ctx = (Context) getInitialContext().lookup("java:comp/env");
factory = (QueueConnectionFactory) ctx.lookup("jms/ref/PDKQCF");
```

The JMS resource reference, jms/ref/PDKQCF, is bound to the JMS resource JNDI name during application assembly or deployment.
12.3.4 Managing JMS resources

In our scenario, the JMS connection to the underlying messaging provider (IBM WebSphere MQ or the internal JMS Provider) is created on the first use of the connection. After that, the connection is cached. The key point, shown in Example 12-2, is that we do not start the connection until we are actually ready to use it.

Example 12-2 Code showing how JMS resources are managed

```java
public int getExchangeRate(String planet) throws EJBException {
    try {
        // establish a connection to the Queue Connection Factory and start it
        getConnection().start();
        // ...
    } finally {
        // close the JMS connections
        try {
            closeConnection();
        } catch (JMSException je) {
            // ...
        }
    }
    // ...
}

private QueueConnection getConnection() throws JMSException, NamingException {
    if (connection == null) {
        // create connection to queue manager
        Context ctx = (Context) getInitialContext().lookup("java:comp/env");
        QueueConnectionFactory factory =
            (QueueConnectionFactory) ctx.lookup("jms/ref/PDKQCF");
        connection = factory.createQueueConnection();
    }
    return connection;
}

private void closeConnection() throws JMSException {
    if (session != null) {
        session.close();
        session = null;
    }
    if (connection != null) {
        connection.close();
        connection = null;
    }
}
```
After we are finished using the connection, it is closed.

JMS sessions are designed for synchronous access only. A session can only be used by a single client and not shared among other clients. Similarly, an instance of a MessageProducer or MessageConsumer can only be used by a single client. JMS sessions are opened for the duration of message sending or receiving. The JMS session is then closed after use.

On completion of interaction with the MessageProducer or MessageConsumer (sender or receiver), the session needs to be closed. In the case of our stateless ExchangeRateJMSBean, shown in Example 12-2 on page 372, after the completion of the getExchangeRate method we call the closeConnection() method to close both the Session and the Connection. Closing the Connection automatically closes any associated Session. We call the closeConnection() method immediately after we are finished because we know this stateless bean is put back into the pool of ExchangeRateJMSBean objects. The next client is most likely a completely different caller needing to get its own Session from a Connection.

### 12.3.5 Synchronous vs. asynchronous

The nature and simplicity of the rate information in this example and the nature of the front-end Web interface clearly dictates a synchronous (request/reply) messaging style.

We could have implemented asynchronous (send-and-forget) messaging to the enterprise tier. Asynchronous messaging, however, would have required a change to the client interface and navigation. The change would have to provide a mechanism such that the exchange rate and subsequent movement of money could have been queried at a later stage with a separate Web page.

For the enterprise tier application, a message-driven bean is used to asynchronously receive the request message from the front-end application using the message consumer pattern. The exchange rate reply message is returned using the send-and-forget pattern.

**Note:** Always implement appropriate timeout and retry conditions when using a blocking request/reply in a stateless session EJB.

The EJB 2.0 specification does point out that only one client will have access to an instance of a stateless EJB while it is servicing a client-invoked method. If, however, a blocking wait occurs for an indefinite period, the container may run short of available instances of the specific EJB to service other clients and thus slow down the overall performance of the application.
12.3.6 Message selectors

The JMS API provides the ability to select a specific message from a destination by the act of browsing. Browsing can be achieved in a number of ways. In Example 12-3, we show how we created a QueueReceiver from the front-end application of our JMS session with a message selector mask.

Example 12-3  QueueReceiver with a message selector

```java
// store the message id to test that the reply message is the correct one
String messageId = reqMsg.getJMSMessageID();

// retrieve a queue connection factory session and use it to
// establish a QueueReceiver session for the reply queue with mask for specific
// JMSCorrelationID
QueueReceiver receiver = getSession().
    createReceiver(replyQ, "JMSCorrelationID =" + messageId + ";

// receive a message from the reply queue
TextMessage repMsg = (TextMessage) receiver.receive(10000);
```

An alternate way is use a JMS QueueBrowser with a message selector, as shown in Example 12-4. Once the browser is created, it is possible to iterate messages in a queue.

Example 12-4  Message browsing using QueueBrowser

```java
QueueBrowser browser = getSession().createBrowser(replyQ);
java.util.Enumeration messages = null;
TextMessage tempTextMessage = null;
for (messages = browser.getEnumeration();messages.hasMoreElements(); ) {
    tempTextMessage = (TextMessage)messages.nextElement();
    if (tempTextMessage.getJMSCorrelationID.equals(messageID)) {
        System.out.println("Message found on queue!!!!");
    }
}
browser.close();
```

Adding a message selector, as shown in Example 12-5, filters all but the messages that match the selection criteria. This results in a much smaller enumeration.
Example 12-5   QueueBrowser with message selector

QueueBrowser browser = getSession().createBrowser(replyQ, "propertyx = 100");

The QueueReceiver and CorrelationID works well for our request/reply scenario. The QueueBrowser alternative might be considered where you have an application routing messages and their CorrelationIDs to other applications for processing.

12.3.7 Message time to live

In a similar way to message selectors, WebSphere MQ (or the internal WebSphere JMS Provider) provides a mechanism through JMS to set the time-to-live of a message. The code in Example 12-6 shows how we set time-to-live within this scenario to 20 seconds.

Example 12-6   Setting the JMS message time-to-live

sender.send(reqMsg,DeliveryMode.NON_PERSISTENT,0,20000);

Setting the message time-to-live is an important best practice to avoid large numbers of unconsumed messages on queues.

JMSExpiration

JMSExpiration is the sum of the time-to-live and current GMT. If, however, time-to-live is set to zero, then JMSExpiration is set to zero.

If it is determined that GMT is now greater than the JMSExpiration value, then the message should be destroyed by the messaging provider. JMS, however, does not automatically notify the messaging provider. It is always good to check the JMSExpiration manually in the code.

For further reference, please see the JMS Specification at:

http://java.sun.com/products/jms/docs.html

12.3.8 Persistent vs. non-persistent messages

A persistent or durable message is an obvious performance issue. However, if the operation is to transfer funds, then it becomes a necessary choice to ensure or guarantee delivery.
PERSISTENT is the default setting. In our scenario, we chose to explicitly set the use of non-persistent messages. Explicitly setting the delivery mode to NON_PERSISTENT requires setting the correct parameter on the message, as shown in Example 12-7 on page 376. Alternatively, it can be set within the send method provided by the QueueSender, as shown in Example 12-6.

Example 12-7  Setting JMS message delivery mode

sender.send(reqMsg,DeliveryMode.NON_PERSISTENT,0,20000);

12.3.9 Enterprise application design

Figure 12-7 shows the design of the simulated enterprise-tier server. IBM WebSphere MQ server and WebSphere Application Server V5.0 are deployed on the enterprise server. The enterprise-tier application reads messages from the request queue and replies with an exchange rate on the reply queue.

The request message consumer is implemented using a message-driven bean. The message-driven bean onMessage method is invoked by the application server when a message arrives on the PDK request queue. The onMessage method invokes the session bean to process the request. The session bean uses the ExchangeRateServerBean (as described in 10.5.3, “Creating the enterprise-tier application” on page 297) to determine the exchange rate for the requested planet, then places the exchange rate on the PDK reply queue using the send-and-forget pattern.

12.3.10 Alternate publish/subscribe design

In this design, shown in Figure 12-8 on page 377, we remove the need for a blocking wait in the stateless EJB. This is achieved by making the application use a locally stored replica of the exchange rate data. This local replica is periodically refreshed from the enterprise application.
This design uses a message-driven bean in the front-end application to listen and store rate information to a local database. The enterprise tier periodically publishes a new rate to all listeners.

### 12.4 Development environment configuration

In this section, we cover how IBM WebSphere Studio Application Developer V5.0 can be configured to develop and test a JMS application. We describe how to:

- Configure the WebSphere JMS Provider to start when you start your Studio application server instance.
- Add message queues to the WebSphere JMS Provider.
- Add a JMS queue connection factory and JMS queues for accessing the message queues configured in the JMS Provider.
- Add a listener port to pass messages to a message-driven bean.

**Note:** Instead of using the WebSphere Studio server configuration editor as described here, you can enable the administrative console for your server configuration. You can then use the console, similar to the method described in 12.5.5, “IBM WebSphere Application Server V5.0 JMS configuration” on page 401.
12.4.1 Adding JMS resources to the server configuration

To set up the PDK JMS resources in your WebSphere Studio server configuration:

1. Switch to the Server perspective and expand **Server Configurations** in the Server Configuration view.
2. Click the **JMS** tab.
3. Navigate to JMS Server Properties under Server Settings:
   a. Set Initial Start to START.
   b. Click the **Add** button on the right of the Queue Names to add the following queues:
      - PDKRequestQueue
      - PDKReplyQueue
4. Navigate to JMS Connection Factories under Server Settings:
   a. Click the **Add** button on the right of the WASQueueConnectionFactory entries.
   b. In the Add WASQueueConnectionFactory window:
      - Set Name to PDKConnManager.
      - Set JNDI Name to jms/PDKConnManager.
      - Set Description to PDK JMS Queue Connection Factory.
      Click **OK**.
5. Navigate to JMS Destinations under Server Settings:
   a. Click the **Add** button on the right of the WASQueue entries.
   b. In the Add WASQueue window:
      - Set Name to PDKReplyQueue.
      - Set JNDI Name to jms/PDKReplyQueue.
      - Set Description to PDK Queue for Exchange Rate replys.
      Click **OK**.
   c. Click the **Add** button on the right of the WASQueue entries.
   d. In the Add WASQueue window:
      - Set Name to PDKRequestQueue.
      - Set JNDI Name to jms/PDKRequestQueue.
      - Set Description to PDK Queue for Exchange Rate requests.
      Click **OK**.

Figure 12-9 shows the resulting WebSphere Studio WebSphere JMS Provider configuration.
6. Save your changes.

In the Studio test environment, we run the PDKLite application and the enterprise tier application (ExchangeRateServer) in the same application server and both use the same JMS Provider. No additional JMS configuration is necessary, as both applications have access to the JMS resources we have just configured.
12.4.2 Adding a Listener Port to the server configuration

We are using a message-driven bean as our message consumer in the enterprise tier application, so a Listener Port also needs to be configured. To configure the Listener Port for the message-driven bean:

1. Switch to the Server perspective and expand Server Configurations in the Server Configuration view.
2. Click the EJB tab.
3. Navigate to Listener Ports under Server Settings:
   a. Click the Add button on the right of the WASQueue entries.
   b. In the Add Listener Port window:
      • Set the Name to PDKRequestPort.
      • Set the Connection Factory JNDI name to jms/PDKConnManager.
      • Set the Destination JNDI name to jms/PDKRequestQueue.
      • Set the Initial State to START.
      • Accept the defaults for the other fields.

   Click OK.

   Figure 12-10 shows the resulting WebSphere Studio Listener Port configuration.

   ![WebSphere v5.0 Server Configuration](image)

   **Figure 12-10** WebSphere Studio Listener Port configuration

4. Save your changes and close the editor.
5. Restart the application server to activate the new listener port.
The WebSphere Studio JMS configuration and listener port configuration is now ready.

12.4.3 Co-located WebSphere Application Server

If you have the WebSphere Application Server and its Embedded Messaging feature installed on the same machine as WebSphere Studio, then the Studio test environment can use the WebSphere Application Server embedded messaging server. This way you have only one set of messaging server binaries installed.

To check that Studio and your server configuration within Studio are pointing to the WebSphere Application Server embedded messaging server:

1. Make sure the `<STUDIO_INSTALL>/runtimes/base_v5/properties/implfactory.properties` file is pointing to the Embedded JMS Provider class, as follows:

   #Embedded JMS Provider  
   com.ibm.ws.messaging.JMSProvider=com.ibm.ws.messaging.JMSEmbeddedProviderImpl
   
   #MQ Simulator for Java Developers JMS Provider for WebSphere Studio Unit Test Environment  
   com.ibm.ws.messaging.JMSProvider=com.ibm.ws.messaging.JMSMQJDProviderImpl

2. Open your server configuration and go to the Variables tab. Under the Node section, make sure the `MQ_INSTALL_ROOT` and `WAS_PUBSUB_ROOT` variables are set to the correct locations, for example:

   MQ_INSTALL_ROOT=c:/websphere mq
   WAS_PUBSUB_ROOT=c:/websphere mq/wemps

   **Note:** If you created a server configuration before installing embedded messaging, the server configuration will point to the MQJD messaging and not the embedded messaging. To use the embedded messaging, create a new server configuration or change the variables listed above.

12.5 Runtime configuration

In this section, we discuss some of the more pertinent points around runtime configuration of WebSphere Application Server V5.0 and IBM WebSphere MQ to satisfy the needs of the PDK application.

Please see Appendix A, “Lab environment scenarios” on page 413, for a description of the various lab environments where we deployed the PDK sample application.
12.5.1 JNDI name spaces

This section examines the various JNDI names of the queues and connection factories from the code, the development environment, and the final deployment in WebSphere Application Server V5.0. We are only going to use the connection factory example in this section, but the same is true for other JMS entities, such as Queues.

The mapping shown in Table 12-1 works on IBM WebSphere Studio Application Developer V5.0 and on WebSphere Application Server V5.0.

<table>
<thead>
<tr>
<th>Name space</th>
<th>PDK sample name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source code resource reference</td>
<td>jms/ref/PDKQCF</td>
<td>Provides direction to JNDI name to allow maximum name flexibility during deployment</td>
</tr>
<tr>
<td>(java:comp/env/...)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Studio resource reference</td>
<td>jms/ref/PDKQCF</td>
<td>Created as required by EJB specification</td>
</tr>
<tr>
<td>(ejb-jar.xml)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Studio bindings</td>
<td>jms/PDKConnManager</td>
<td>Created if deploy time connection factories or destinations are already known; links resource references with the WebSphere JNDI name</td>
</tr>
<tr>
<td>(ibm-ejb-jar-bnd.xmi)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WebSphere identifier name</td>
<td>PDKConnManager</td>
<td>Required by WebSphere Application Server as a unique identifier for connection factories and destinations</td>
</tr>
<tr>
<td>WebSphere JNDI name</td>
<td>jms/PDKConnManager</td>
<td>PDKQueueConnection factory</td>
</tr>
</tbody>
</table>

Starting in the code, the JMS resource JNDI name is looked up as jms/ref/PDKQCF, as shown in Example 12-8.

**Example 12-8  Looking up the JMS resource JNDI name**

```
Context ctx = (Context) getInitialContext().lookup("java:comp/env");
factory = (QueueConnectionFactory) ctx.lookup("jms/ref/PDKQCF");
```

When getting ready to deploy this code or to work correctly in the test environment in WebSphere Studio, the ejb-jar.xml deployment descriptor needs to specify the EJB’s resources requirements, as shown in Figure 12-11. This is so that the deployer is able to select the available resources from within WebSphere Application Server.
The WebSphere Bindings setting allows you to bind a resource reference to the JNDI name of the JMS resource. Deployment is much smoother if these JNDI names have been created ahead of time.

![Image of WebSphere Bindings setting](image)

Figure 12-11 Setting up JMS resource references and bindings

The connection factory name can be any name you care to choose. In our scenario we chose to name the factory PDKConnManager. The more important aspect within the WebSphere Studio server configuration is to get the JNDI name correct. The JNDI name shown in Figure 12-12 must match the name specified in the WebSphere Bindings, which is shown in Figure 12-11 on page 383.
Similarly, when deploying the application to the WebSphere Application Server runtime environment, the JNDI name specified in the WebSphere Bindings (shown in Figure 12-11 on page 383) must match the JDNI name of the queue connection factory configured in WebSphere. See 12.5.5, “IBM WebSphere Application Server V5.0 JMS configuration” on page 401.

### 12.5.2 IBM WebSphere MQ logical configuration

As shown in Appendix A, “Lab environment scenarios” on page 413, we deployed the PDK application across two tiers: The application server tier and the enterprise tier. We need IBM WebSphere MQ to be configured and communicating between these two tiers.

As shown in Figure 12-13 on page 385, the configuration of IBM WebSphere MQ was achieved with the help of creating a WebSphere MQ cluster. Once both machines are participating in a cluster, a cluster queue defined on the enterprise tier is immediately visible on the application server tier and vice versa.
As shown in Figure 12-13, the PDK.REQUEST queue is defined locally on the enterprise tier and PDK.REPLY is defined locally on the application server tier. The PDK application reads from the PDK.REPLY queue. The enterprise-tier application reads from PDK.REQUEST queue.

**Note:** It does matter where the queues are defined. When using IBM WebSphere MQ queues, it is important to note that it is not possible to read messages from a remote queue. Messages can only be read from local queues. Messages can be put on remote queues, using a local queue automatically provided by the cluster, as shown in Figure 12-13.

### 12.5.3 IBM WebSphere MQ configuration for Windows 2000

In this section we describe the steps needed to configure the WebSphere MQ queue managers, queues, and the cluster used in our JMS scenario on Windows 2000.

**Note:** When installing IBM WebSphere MQ V5.3.1 for this scenario, we selected the following WebSphere MQ features:

- Server
- Java Messaging
IBM WebSphere MQ enterprise tier configuration
We perform the following steps on our enterprise-tier IBM WebSphere MQ Server for Windows 2000:

- Define a local queue manager.
- Create a cluster receiver channel.
- Set the cluster repository queue manager.
- Create the request queue.

Defining a local queue manager
To define a local queue manager on the enterprise-tier WebSphere MQ server:

1. Right-click the Queue Managers folder in the IBM WebSphere MQ Explorer and select New -> Queue Manager from the pop-up menu.

2. In Step 1 of the wizard, shown in Figure 12-14:
   - Set the queue manager name to PDK.QUEUE.MANAGER.
   - Check Make this the default queue manager.
   - Click Next.

3. In Step 2 of the wizard, accept the defaults and click Next.

4. In Step 3 of the wizard, check Create Server Connection Channel to allow remote administration of the queue manager over TCP/IP. Click Next.
5. In Step 4 of the wizard, set Listen on port number to a free TCP/IP port. The default is port 1414. Click Finish to create the queue manager.

Creating a cluster receiver channel

To add a cluster receiver channel to the enterprise-tier WebSphere MQ server:

1. Navigate to PDK.QUEUE.MANAGER -> Advanced -> Channels in the WebSphere MQ Explorer tree view.

2. Right-click Channels and select New -> Cluster Receiver Channel from the pop-up menu, as shown in Figure 12-15.

3. In the Create Cluster Receiver Channel window, shown in Figure 12-16 on page 388:
   - Set the Channel Name field to TO.PDK.QUEUE.MANAGER.
   - Set the Connection Name field to <hostname>(<port_number>), entsrv1w(1414) in our case.
Figure 12-16  Create Cluster Receiver Channel

4. Click the **Cluster** tab.

5. In the Sharing In Clusters pane, shown in Figure 12-17 on page 389:
   - Select **Shared in cluster**.
   - Set the Cluster field to PDK_CLUSTER.

Click **OK** to create the cluster receiver channel.
Chapter 12. Java Message Service scenario

Figure 12-17  Create Cluster Receiver Channel - Sharing In Clusters

Setting the cluster repository queue manager
In each cluster, at least one queue manager needs to be assigned as a full repository manager. To make PDK.QUEUE.MANAGER the repository manager of the PDK_CLUSTER cluster:

1. Right-click PDK.QUEUE.MANAGER and select Properties from the pop-up menu.
2. In the PDK.QUEUE.MANAGER Properties window, click the Repository tab.
3. In the Cluster Repository pane, shown in Figure 12-18 on page 390:
   - Select Repository for a cluster.
   - Set the Cluster field to PDK_CLUSTER.
   - Click OK to set the queue manager for the cluster repository.
Creating the request queue

Finally, using the Queue Creation wizard, create the queue for incoming requests to the enterprise-tier application:

1. Expand the PDK.QUEUE.MANAGER folder in IBM WebSphere MQ Explorer.
2. Right-click the Queues folder and select **New -> Local Queue**.
3. In the Create Local Queue window, shown in Figure 12-19 on page 391, set the Queue Name field to PDK.REQUEST.
4. Click the **Cluster** tab.

5. In the Sharing In Clusters pane, shown in Figure 12-20 on page 392:
   - Select **Shared in cluster**.
   - Set the Cluster field to PDK_CLUSTER.

   Click **OK** to create the local queue.
Figure 12-20  Adding queues to clusters

**IBM WebSphere MQ application server tier configuration**

We perform the following steps on our application server tier IBM WebSphere MQ server for Windows 2000:

1. Define a local queue manager.
2. Create a cluster receiver channel.
3. Create a cluster sender channel.
4. Create the reply queue.

**Defining a local queue manager**

To define a local queue manager on the enterprise-tier WebSphere MQ server:

1. Right-click the Queue Managers folder in the IBM WebSphere MQ Explorer and select **New -> Queue Manager** from the pop-up menu.

2. In Step 1 of the wizard:
   - Set the Queue Manager Name field to PDK.CONN.MANAGER.
   - Check **Make this the default queue manager**.

   Click **Next**.

3. In Step 2 of the wizard, accept the defaults and click **Next**.
4. In Step 3 of the wizard, check Create Server Connection Channel to allow remote administration of the queue manager over TCP/IP. Click Next.

5. In Step 4 of the wizard, set the Listen on port number field to a free TCP/IP port. The default is port 1414. Click Finish to create the queue manager.

Creating a cluster receiver channel
To add a cluster receiver channel to the application server tier WebSphere MQ server:

1. Navigate to PDK.CONN.MANAGER -> Advanced -> Channels in the WebSphere MQ Explorer tree view.
2. Right-click Channels and select New -> Cluster Receiver Channel from the pop-up menu.
3. In the Create Cluster Receiver Channel window:
   - Set the Channel Name field to TO.PDK.CONN.MANAGER.
   - Set the Connection Name field to <hostname>(<port_number>).
     Use the host name of this machine and the port number from “Defining a local queue manager” on page 392, which is appsrv1w(1414) in our case.
4. Click the Cluster tab.
5. In the Sharing In Clusters pane:
   - Select Shared in cluster.
   - Set the cluster name to PDK_CLUSTER.
   Click OK to create the cluster receiver channel.

Creating a cluster sender channel
To add a cluster sender channel to the application server tier WebSphere MQ server:

1. Navigate to PDK.CONN.MANAGER -> Advanced -> Channels in the WebSphere MQ Explorer tree view.
2. Right-click Channels and select New -> Cluster Sender Channel from the pop-up menu.
3. In the Create Cluster Sender Channel window, shown in Figure 12-16 on page 388:
   - Set the Channel Name field to TO.PDK.QUEUE.MANAGER.
   - Set the Connection Name field to <hostname>(<port_number>), entsrv1w(1414) in our case.
4. Click the **Cluster** tab.
5. In the Sharing In Clusters pane:
   - Select **Shared in cluster**.
   - Set the Cluster field to PDK_CLUSTERS.
   - Click **OK** to create the cluster sender channel.

**Create the reply queue**
Finally, using the Queue Creation wizard, create the queue for incoming requests to the enterprise-tier application:

1. Expand the PDK.CONN.MANAGER folder in IBM WebSphere MQ Explorer.
2. Right-click the Queues folder and select **New -> Local Queue**.
3. In the Create Local Queue window, set the Queue Name field to PDK.REPLY.
4. Click the **Cluster** tab.
5. In the Sharing In Clusters pane:
   - Select **Shared in cluster**.
   - Set the Cluster field to PDK_CLUSTERS.
   - Click **OK** to create the local queue.

### 12.5.4 IBM WebSphere MQ configuration for AIX and Linux

In this section we describe the steps needed to configure the WebSphere MQ queue managers, queues, and the cluster used in our JMS scenario on IBM AIX V5.1 and Red Hat Linux Advanced Server 2.1.

We use the WebSphere MQ Commands command-line interface in this section. You can also use IBM WebSphere MQ Explorer on Windows to manage AIX and Linux WebSphere MQ resources. See “Using IBM WebSphere MQ Explorer with AIX and Linux” on page 399 for details.

We identify steps where there are differences in the AIX and Linux setup.

**Note:** On Linux, IBM WebSphere MQ is installed under the /opt directory by default, not the /usr as it is on AIX. The properties and error reports are located on /var/mqm for both operating systems.
IBM WebSphere MQ enterprise tier configuration

We perform the following steps on our enterprise-tier IBM WebSphere MQ Server for AIX and Linux:

- Define a local queue manager.
- Create a cluster receiver channel.
- Set the cluster repository queue manager.
- Create the request queue.

**Defining a local queue manager**

To define a local queue manager on the enterprise-tier WebSphere MQ server:

1. Log in as a user in the mqm group:
   ```
   su - mqm
   ```
2. Create the queue manager:
   ```
   cd /var/mqm
crtmqm -q PDK.QUEUE.MANAGER
   ```
   We use the -q option to specify PDK.QUEUE.MANAGER as the default queue manager.
3. Update /etc/services to add the following line (you may need to switch to root user to allow changes to this file):
   ```
   MQSeries        1414/tcp                        # MQSeries port
   ```
4. For AIX, start a listener as follows:
   a. Update /etc/inetd.conf to add the following line (you may need to switch to root user to allow changes to this file):
      ```
      MQSeries stream tcp nowait mqm /usr/mqm/bin/amqcrsta amqcrsta -m PDK.QUEUE.MANAGER
      ```
   b. Restart inetd to pick up the changes:
      ```
      refresh -s inetd
      ```
   For Linux, start a listener using one of two ways:
   
   - Start the listener manually:
     ```
     runmqlsr -m PDK.QUEUE.MANAGER -t tcp -p 1414
     ```
     **Note:** You will have to restart the listener every time you reboot the machine.
   - Or configure xinetd to start it:
     i. Log in as root.
ii. Create a file called MQSeries in the /etc/xinetd.d directory and add the content shown in Example 12-9 on page 396, where the `<queue_manager_name>` is PDK.QUEUE.MANAGER in this case.

iii. Restart xinetd:

```
kill -1 <process id of xinetd daemon>
```

**Example 12-9  Service definition file under xinet.d**

```
# description: MQSeries Queue Manager Service

service MQSeries
{
    socket_type = stream
    protocol    = tcp
    user        = mqm
    wait        = no
    server      = /opt/mqm/bin/amqcrsta
    server_args = -m <queue_manager_name>
}
```

5. Start the queue manager:

```
strmqm PDK.QUEUE.MANAGER
```

**Creating a cluster receiver channel**

To add a cluster receiver channel to the enterprise-tier WebSphere MQ server:

1. Log in as a user in the mqm group and start WebSphere MQ Commands:

```
su - mqm
runmqsc PDK.QUEUE.MANAGER
```

2. Create the cluster receiver channel:

```
define channel(TO.PDK.QUEUE.MANAGER) +
    chltype(CLUSRCVR) +
    conname('<hostname>(<port_number>)') +
    cluster(PDK_CLUSTER)
end
```

Where `<hostname>(<port_number>)` is entsrv1a(1414) for AIX or entsrv1l(1414) for Linux in our Linux lab environment.

**Setting the cluster repository queue manager**

To set PDK.QUEUE.MANAGER as the queue manager for the PDK_CLUSTER cluster repository:

1. Log in as a user in the mqm group and start WebSphere MQ Commands:

```
su - mqm
runmqsc PDK.QUEUE.MANAGER
```
2. Set the queue manager for the cluster repository:

   alter qmgr repos(PDK_CLUSTERS)
   end

Creating the request queue
Finally, create the queue for incoming requests to the enterprise-tier application:

1. Log in as a user in the mqm group and start WebSphere MQ Commands:

   su - mqm
   runmqsc PDK.QUEUE.MANAGER

2. Create the local request queue:

   define qlocal('PDK.REQUEST') cluster(PDK_CLUSTERS)
   end

IBM WebSphere MQ application server tier configuration
We perform the following steps on our application server tier IBM WebSphere MQ server for AIX and Linux:

- Define a local queue manager.
- Create a cluster receiver channel.
- Create a cluster sender channel.
- Create the reply queue.

Defining a local queue manager
To define a local queue manager on the application server tier WebSphere MQ server:

1. Log in as a user in the mqm group:

   su - mqm

2. Create the queue manager:

   cd /var/mqm
   crtmqm -q PDK.CONN.MANAGER

3. Update /etc/services to add the following line (you may need to switch to root user to allow changes to this file):

   MQSeries 1414/tcp

4. For AIX, start a listener as follows:

   a. Update /etc/inetd.conf to add the following line (you may need to switch to root user to allow changes to this file):

      MQSeries stream tcp nowait mqm /usr/mqm/bin/amqcrsta amqcrsta -m
      PDK.CONN.MANAGER
b. Restart inetd to pick up the changes:
   
   refresh -s inetd

   For Linux, start a listener using one of two ways:
   
   – Start the listener manually:
     
     runmqslsr -m PDK.CONN.MANAGER -t tcp -p 1414
   – Or configure xinetd to start it:
     
     i. Log in as root.
     
     ii. Create a file called MQSeries in the /etc/xinetd.d directory and add the
         content shown in Example 12-9 on page 396, where the
         <queue_manager_name> is PDK.CONN.MANAGER in this case.
     
     iii. Restart xinetd:
         
         kill -1 <process id of xinetd daemon>

   5. Start the queue manager:

   strmqm PDK.CONN.MANAGER

   **Creating a cluster receiver channel**
   
   To add a cluster receiver channel to the application server tier WebSphere MQ
   server:

   1. Log in as a user in the mqm group and start WebSphere MQ Commands:

      su - mqm
      runmqsc PDK.CONN.MANAGER

   2. Create the cluster receiver channel:

      define channel(TO.PDK.CONN.MANAGER) +
      chltype(CLUSRCVR) +
      conname('<hostname>({port_number})') +
      cluster(PDK_CLUSTER)
      end

      Where <hostname>({port_number}) is appsrv1a(1414) for AIX or
      appsrv1l(1414) for Linux in our Linux lab environment.

   **Creating a cluster sender channel**
   
   To add a cluster sender channel to the application server tier WebSphere MQ
   server:

   1. Log in as a user in the mqm group and start WebSphere MQ Commands:

      su - mqm
      runmqsc PDK.CONN.MANAGER

   2. Create the cluster sender channel:

      define channel(TO.PDK.QUEUE.MANAGER) +
chltype(CLUSSDR) +
conname('<hostname>(<port_number>)') +
cluster(PDK_CLUSTER)
end

Where <hostname>(<port_number>) is entsrv1a(1414) for AIX or
entsrv1l(1414) for Linux in our Linux lab environment.

Note: The cluster sender channel uses the queue manager name and
<hostname>(<port_number>) of the cluster repository queue manager on
the enterprise tier.

Create the reply queue
Finally, using the queue creation wizard, create the queue for incoming requests
to the enterprise-tier application:
1. Log in as a user in the mqm group and start WebSphere MQ Commands:
   su - mqm
   runmqsc PDK.CONN.MANAGER
2. Create the local reply queue:
   define qlocal('PDK.REPLY') cluster(PDK_CLUSTER)
   end

Using IBM WebSphere MQ Explorer with AIX and Linux
Here are some quick directions for using IBM WebSphere MQ Explorer on
Windows to manage AIX and Linux WebSphere MQ resources:
1. Start WebSphere MQ Commands on AIX or Linux:
   runmqsc <queue_manager_name>
2. Make sure you have a SYSTEM.ADMIN.SVRCONN channel:
   dis chl(SYSTEM.ADMIN*) all
   If not, then define the channel as follows:
   define channel(SYSTEM.ADMIN.SVRCONN) chltype(SVRCONN)
3. Exit WebSphere MQ Commands:
   end
4. Start the command server to allow WebSphere MQ to process remote
commands from WebSphere MQ Explorer:
   strmqcsv PDK.CONN.MANAGER
5. Enable the remote Windows user to connect to the system. You can do this in one of two ways:
   – Add the user ID you are logged on to Windows with as a user ID on AIX or Linux and add it to the mqm group.
   – Turn off the security on the AIX or Linux side:
     ```
     runmqsc <queue_manager_name>
     alter channel(SYSTEM.ADMIN.SVRCONN) chltype(SVRCONN) mcauser('mqm')
     refresh security(*)
     end
     ```

6. Before starting WebSphere MQ Explorer on Windows, make sure that the Microsoft Management Console file used is not read-only. The default file is `<MQ_HOME>/MQSeries.msc`.

7. In WebSphere MQ Explorer, highlight **Queue Managers**, right-click, and select **Show Queue Manager**.

8. In the Show Queue Manager window, shown in Figure 12-21 on page 400:
   a. Select **Show a remote queue manager**.
   b. Enter the queue manager name and connection name.
   Click **OK** and repeat the procedure for the other queue manager.

![Figure 12-21 Connecting a remote queue manager](image)

9. Save the changes when you exit the IBM WebSphere MQ Explorer.
Once the queue managers are visible through the Windows IBM WebSphere MQ Explorer, you can configure WebSphere MQ on AIX or Linux in the same way as the Windows setup we described in 12.5.3, “IBM WebSphere MQ configuration for Windows 2000” on page 385. The difference is that during the cluster creation, none of the queue managers are on the local Windows machine; they are both remote.

12.5.5 IBM WebSphere Application Server V5.0 JMS configuration

Use the WebSphere Administrative Console to administer JMS resources. As shown in Figure 12-22, you can work with JMS Providers, connection factories, and destinations under Resources in the console navigation tree.

![Figure 12-22   Configuring the WebSphere MQ JMS Provider](image)

**Application server tier JMS configuration**

To configure the WebSphere MQ JMS Provider for the application server tier:

1. Open the WebSphere Administrative Console.
2. Select Resources -> WebSphere MQ JMS Provider in the navigation tree, as seen in Figure 12-22.
3. In the WebSphere MQ JMS Provider form, set the scope to Server, then click Apply.
4. In the WebSphere MQ JMS Provider form, scroll down to Additional Properties and click WebSphere MQ Queue Connection Factories.
5. In the WebSphere MQ Queue Connection Factories form click **New**.

6. In the WebSphere MQ Queue Connection Factories General Properties form, shown in Figure 12-23 on page 403:
   a. Set the Name to PDKConnManager.
   b. Set the JNDI name to jms/PDKConnManager.
   c. Enter the Description, for example, PDK Queue Connection Factory.
   d. Set the Queue Manager to PDK.CONN.MANAGER, as defined in WebSphere MQ.
   e. Set Transport Type to Bindings.
   f. Check Message Retention.
   g. Check XA enabled.
   h. Accept the defaults for the other fields.

Click **OK**.
7. Select Resources -> WebSphere MQ JMS Provider in the navigation tree.

8. In the WebSphere MQ JMS Provider form, scroll down to Additional Properties and click WebSphere MQ Queue Destinations.

9. In the WebSphere MQ Queue Destinations form click New.

10. In the WebSphere MQ Queue Destinations General Properties form, shown in Figure 12-24 on page 405:
   a. Set the Name to PDKRequestQueue.
   b. Set the JNDI name to jms/PDKRequestQueue.
   c. Enter the Description, for example, PDK Queue for Exchange Rate requests.
d. Set the Base Queue Name to PDK.REQUEST, as defined in Websphere MQ.
e. Accept the defaults for the other fields.
Click **OK**.
11. Back in the WebSphere MQ Queue Destinations form click **New**.
12. In the WebSphere MQ Queue Destinations General Properties form:
   a. Set the Name to PDKReplyQueue.
   b. Set the JNDI name to jms/PDKReplyQueue.
   c. Enter the Description, for example, PDK Queue for Exchange Rate replies.
   d. Set the Base Queue Name to PDK.REPLY, as defined in Websphere MQ.
   e. Accept the defaults for the other fields.
   Click OK.

13. Save your changes.

The JMS Provider configuration for the application tier is finished.

**Enterprise server tier JMS configuration**
To configure the WebSphere MQ JMS Provider for the enterprise server tier:
1. Open the WebSphere Administrative Console.
2. Select Resources -> WebSphere MQ JMS Provider in the navigation tree.
3. In the WebSphere MQ JMS Provider form, set the scope to Server, then click Apply.
4. In the WebSphere MQ JMS Provider form, scroll down to Additional Properties and click WebSphere MQ Queue Connection Factories.
5. In the WebSphere MQ Queue Connection Factories form click New.
6. In the WebSphere MQ Queue Connection Factories General Properties form:
   a. Set the Name to PDKConnManager.
   b. Set the JNDI name to jms/PDKConnManager.
   c. Enter the Description, for example, PDK Queue Connection Factory.
   d. Set the Queue Manager to PDK.QUEUE.MANAGER, as defined in Websphere MQ.
   e. Set Transport Type to Bindings.
   f. Check Message Retention.
   g. Check XA enabled.
   h. Accept the defaults for the other fields.
   Click OK.
7. Select Resources -> WebSphere MQ JMS Provider in the navigation tree.
8. In the WebSphere MQ JMS Provider form, scroll down to Additional Properties and click WebSphere MQ Queue Destinations.
9. In the WebSphere MQ Queue Destinations form click **New**.

10. In the WebSphere MQ Queue Destinations General Properties form:
   a. Set the Name to PDKRequestQueue.
   b. Set the JNDI name to jms/PDKRequestQueue.
   c. Enter the Description, for example, PDK Queue for Exchange Rate requests.
   d. Set the Base Queue Name to PDK.REQUEST, as defined in Websphere MQ.
   e. Accept the defaults for the other fields.
   Click **OK**.

11. Back in the WebSphere MQ Queue Destinations form click **New**.

12. In the WebSphere MQ Queue Destinations General Properties form:
   a. Set the Name to PDKReplyQueue.
   b. Set the JNDI name to jms/PDKReplyQueue.
   c. Enter the Description, for example, PDK Queue for Exchange Rate replies.
   d. Set the Base Queue Name to PDK.REPLY, as defined in Websphere MQ.
   e. Accept the defaults for the other fields.
   Click **OK**.

13. Save your changes.

The JMS Provider configuration for the enterprise tier is finished.

**Enterprise tier Listener Port configuration**

We are using a message-driven bean as our message consumer in the enterprise tier application, so a Listener Port also needs to be configured. To configure the Listener Port for the message-driven bean:

1. Open the WebSphere Administrative Console.

2. Select **Servers -> Application Servers** in the navigation tree.

3. In the Application Servers form click the required server, **server1** in our case.

4. In the server1 form, scroll down to Additional Properties and click **Message Listener Service**.

5. In the Message Listener Service Additional Properties form click **Listener Ports**.

6. In the Listener Ports form click **New**.
7. In the new listener port General Properties form, as shown in Figure 12-25 on page 408:
   a. Set the Name to PDKRequestPort.
   b. Set the Initial State to Started.
   c. Set the Connection factory JNDI name to jms/PDKConnManager.
   d. Set the Destination JNDI name to jms/PDKRequestQueue.
   e. Accept the defaults for the other fields.
   
   Click OK.

8. Save your changes.

9. Restart the application server to activate the new listener port.

The enterprise tier JMS configuration and listener port configuration is now ready.
12.5.6 Enterprise simulation application setup

On the enterprise tier, we have a J2EE application that simulates the enterprise application. This application is packaged in a EAR file named ExchangeRateServer.ear. You can find ExchangeRateServer.ear in the PDK_Lite\TestDrive\Install\09InstallApps folder of the PDK sample.

Once you have completed the WebSphere JMS configuration, install ExchangeRateServer.ear from Applications -> Install New Application in the WebSphere Administrative Console navigation tree.

PDK V5 and previous versions of the PDK are available from the Patterns for e-business Web site:

Included in this part are the following appendixes:

► Appendix A, “Lab environment scenarios” on page 413

In Part 3, “Technical scenarios” on page 251, we described deploying the PDK sample application in various lab environments. This appendix describes the lab environments we used:

– Microsoft Windows 2000 lab environment
– IBM AIX 5.1 lab environment
– Red Hat Linux Advanced Server 2.1 lab environment

► Appendix B, “PDK sample setup” on page 421

This appendix describes how to set up the Pattern Development Kit Lite V5 sample in the following environments:

– IBM WebSphere Application Server base V5.0
– IBM WebSphere Studio Application Developer V5.0

It also describes how to change the default PDK application settings.
Lab environment scenarios

In Part 3, “Technical scenarios” on page 251, we described deploying the PDK sample application in various lab environments. This appendix describes the lab environments we used:

- Microsoft Windows 2000 lab environment
- IBM AIX 5.1 lab environment
- Red Hat Linux Advanced Server 2.1 lab environment
Windows lab environment

Figure A-1 shows the Windows lab environment we used when deploying the PDK sample application, as described in Part 3, “Technical scenarios” on page 251. The node configurations are:

- **DMZ Web server**
  - **Host name**: websrv1w
  - **IP address**: 192.168.10.7
  - **Operating system**: Windows 2000 Server, Service Pack 3
  - **Software**: IBM HTTP Server V1.3.26

- **Domain firewall**
  - **Host name**: fw2w
  - **IP address**: 192.168.10.2 (DMZ)
  - **Operating system**: Windows NT Server V4.0, Service Pack 6a
  - **Software**: IBM SecureWay Firewall V4.2
Appendix A. Lab environment scenarios

Front-end application server
- **Host name**: apsprv1w
- **IP address**: 10.2.105.122
  - 192.168.10.102 (from DMZ)
- **Operating system**: Windows 2000 Server, Service Pack 3
- **Software**:
  - IBM HTTP Server V1.3.26
  - IBM WebSphere Application Server V5.0
  - IBM WebSphere MQ Server V5.3

Back-end application server
- **Host name**: entsrv1w
- **IP address**: 10.2.105.111
- **Operating system**: Windows 2000 Server, Service Pack 3
- **Software**:
  - IBM HTTP Server V1.3.26
  - IBM WebSphere Application Server V5.0
  - IBM WebSphere MQ Server V5.3

CICS server
- **Host name**: wtsrsc66oe
- **IP address**: 10.1.6.29
- **Operating system**: z/OS 1.3
- **Software**:
  - CICS Transaction Server V2.2
  - CICS Transaction Gateway V5.0

IBM AIX lab environment

Figure A-2 shows the IBM AIX lab environment we used when deploying the PDK sample application, as described in Part 3, “Technical scenarios” on page 251. The node configurations are:

DMZ Web server
- **Host name**: webserv1a
- **IP address**: 192.168.10.12
- **Operating system**: IBM AIX 5.1
**Domain firewall**

**Host name**  fw2w  
**IP address**  192.168.10.2 (DMZ)  
10.2.104.14 (intranet)

**Operating system**  Windows NT Server V4.0, Service Pack 6a

**Software**  IBM SecureWay Firewall V4.2

**Front-end application server**

**Host name**  appsrv1a  
**IP address**  10.2.105.42  
192.168.10.101 (from DMZ)

**Operating system**  IBM AIX 5.1

**Software**  IBM HTTP Server V1.3.26  
IBM WebSphere Application Server V5.0  
IBM WebSphere MQ Server V5.3

**Back-end application server**

**Host name**  entsrv1a
Appendix A. Lab environment scenarios

<table>
<thead>
<tr>
<th>IP address</th>
<th>10.2.105.33</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating system</td>
<td>IBM AIX 5.1</td>
</tr>
<tr>
<td>Software</td>
<td>IBM HTTP Server V1.3.26</td>
</tr>
<tr>
<td></td>
<td>IBM WebSphere Application Server V5.0</td>
</tr>
<tr>
<td></td>
<td>IBM WebSphere MQ Server V5.3</td>
</tr>
</tbody>
</table>

- CICS server
  | Host name | wtsc66oe |
  | IP address | 10.1.6.29 |
  | Operating system | z/OS 1.3 |
  | Software | CICS Transaction Server V2.2 |
  | | CICS Transaction Gateway V5.0 |

Linux lab environment

Figure A-3 shows the Linux lab environment we used when deploying the PDK sample application, as described in Part 3, “Technical scenarios” on page 251. The node configurations are:

- DMZ Web server
  | Host name | websrv1l |
  | IP address | 192.168.10.6 |
  | Operating system | Red Hat Linux Advanced Server 2.1 |
  | Software | IBM HTTP Server V1.3.26 |
  | | IBM WebSphere Application Server V5.0 Web server plug-in |
Figure A-3  Linux lab environment

- **Domain firewall**
  
  **Host name**  fw2w  
  **IP address**  192.168.10.2 (DMZ)  
  10.2.104.14 (intranet)  
  
  **Operating system**  Windows NT Server V4.0, Service Pack 6a  
  **Software**  IBM SecureWay Firewall V4.2  

- **Front-end application server**
  
  **Host name**  appsrv1l  
  **IP address**  10.2.105.28  
  192.168.10.103 (from DMZ)  
  
  **Operating system**  Red Hat Linux Advanced Server 2.1  
  **Software**  IBM HTTP Server V1.3.26  
  IBM WebSphere Application Server V5.0  
  IBM WebSphere MQ Server V5.3  

- **Back-end application server**
  
  **Host name**  entsrv1l  
  **IP address**  10.2.105.11  
  
  **Operating system**  Red Hat Linux Advanced Server 2.1
Software
IBM HTTP Server V1.3.26
IBM WebSphere Application Server V5.0
IBM WebSphere MQ Server V5.3

- CICS server
  Host name  wtsc66oe
  IP address  10.1.6.29
  Operating system  z/OS 1.3
  Software  CICS Transaction Server V2.2
            CICS Transaction Gateway V5.0
PDK sample setup

In this appendix we first describe the Patterns for e-business downloads that are available, including the Patterns Development Kit Lite V5.

We then explain in detail how to set up the PDK Lite V5 sample in the following environments:

- IBM WebSphere Application Server base V5.0
- IBM WebSphere Studio Application Developer V5.0

We also describe how to change the default PDK Lite V5 application settings.
Patterns for e-business downloads

Move your e-business to market fast with proven, reusable solution designs from IBM's Patterns for e-business. This section lists digital resources from the Patterns project, designed to first familiarize you with the theories and application of the Patterns for e-business, and then help solve your e-business problems using these proven implementations. The Patterns Development Kit Lite (PDK Lite), described below, is available for immediate download.

Download the Patterns Development Kit Lite (PDK Lite) from:

Patterns Development Kit Lite (PDK Lite)

The Patterns for e-business Development Kit Lite (PDK Lite) is a complete, self-configuring, end-to-end skeleton Web application in one self-extracting package. PDK Lite is currently offered in four versions for implementing various Self-Service solution designs:

- PDK Lite Version 3.0 uses Websphere Application Server Advanced Edition Version 4.0 to implement the Stand-Alone Single Channel application pattern, and does support the use of EJBs.
- Version 4.1 implements both a Stand-Alone Single Channel solution, and a Directly-Integrated Single Channel solution, and features the use of Web services, J2EE Connectors, and JMS technology to implement the Directly Integrated Single Channel application pattern.
- PDK Lite Version 5.0 uses WebSphere Application Server V5.0 to implement the Stand-Alone Single Channel application pattern and the Directly Integrated Single Channel application pattern. It demonstrates use of Web services, J2EE Connectors, and JMS integration technologies.

How to download and install the PDK Lite

To download and install the PDK Lite:

1. Register with the Patterns for e-business download site by opening the Download the PDK Lite link at the end of this section.
2. Download the self-extracting zip file for the PDK Lite version you want.
3. Extract the PDK Lite file to the root of your hard drive, for example, C:\.
4. Before proceeding with the install, you must view the PDK README.html file for further instructions. The README file in the PDK download's root directory is always the most up-to-date copy. Follow the directions in the README file to install the PDK Lite. You can also preview the README file now for each of the PDK Lite versions:
   - PDK Lite Version 2.0 README
   - PDK Lite Version 3.0 README
   - PDK Lite Version 4.1 README
   - PDK Lite Version 5.0 README

Download the Patterns Development Kit Lite (PDK Lite) from:

**IBM WebSphere Application Server base V5.0**

This section provides details on installing and running the PDK Lite V5.0 sample application in IBM WebSphere Application Server base V5.0.

**Before you start**

Before you start installing PDK Lite, check the following prerequisites:

▶ System requirements
▶ Products required
▶ Information required

**System requirements**

PDK Lite has the following system requirements:

▶ Operating system:
  PDK Lite has been deployed on the following operating systems:
  - Windows 2000 Server Service Pack 3
  - AIX 5.1
  - Red Hat Linux Advanced Server 2.1
Hardware:
- Intel platform: 500 MHz Pentium (Windows or Linux)
- AIX platform: RS/6000 at 375 MHz
- 384 MB RAM minimum, 512 MB recommended

Products required
The following product levels were used to develop and test PDK Lite. Please install these products at the specified version levels before attempting to install and run PDK Lite:
- IBM WebSphere Application Server base V5.0 (including IBM HTTP Server V1.3.26). To get a free trial copy, click the Downloads link at:
  http://www.ibm.com/software/webservers/appserv
- IBM DB2 Universal Database V8.1. To get a free trial copy, click the Downloads link at:
  http://www.ibm.com/software/data/db2
- IBM CICS J2EE Connector resource adapter. The CICS ECI J2EE Connector adapter we used is available with the IBM CICS Transaction Gateway V5.0 for Windows, AIX, and Linux.
  The PDK sample will install without the CICS resource adapter, but J2EE Connector CICS back-end integration will not be available.
- Internet Explorer 5.0 or Netscape Navigator 4.7 (or higher).

Information required
Before starting the PDK installation, make sure you have the following information available:
- PDK Lite installation directory.
  On Windows, if you plan to extract the PDK sample to C:\, for example, then the PDK Lite installation directory will be C:\PDK_Lite.
  On AIX or Linux, if you plan to extract the PDK sample to /usr, for example, then the PDK Lite installation directory will be /usr/PDK_Lite.
- IBM WebSphere Application Server installation directory. For example, C:\WebSphere\AppServer on Windows, /usr/WebSphere/AppServer on AIX, or /opt/WebSphere/AppServer on Linux.
- IBM HTTP Server installation directory. For example, C:\IBMHttpServer on Windows, /usr/IBMHttpServer on AIX, or /opt/IBMHttpServer on Linux.
- IBM DB2 installation directory. For example, C:\SQLLIB on Windows, or /home/db2inst1/sqllib on AIX or Linux.
J2EE Connector resource adapter archive file name and path. For example, C:\Program Files\IBM\IBM CICS Transaction Gateway\deployable\cicseci.rar on Windows.

Node or machine name. To check your node name, log in to the WebSphere Administrative Console. The node name appears at the top of the navigation tree on the left, under the user ID. Note that the WebSphere node name is case sensitive.

IBM DB2 user ID and password for database access. For example, db2admin on Windows, or db2inst1 on AIX or Linux.

Installing the PDK sample

To install the PDK sample application in IBM WebSphere Application Server base V5.0:

1. Download the PDK sample.
2. Extract the PDK sample to the required location, for example, C:\ or /usr.
3. Make sure you are logged in as a user with administrative privileges:
   - In Windows, make sure you are a member of the Administrators group and the mqm group.
   - On AIX or Linux, make sure you log in as root and root is a member of the mqm group and the mqbrkrs group.
4. On AIX or Linux, make sure DB2 has a TCP/IP node entry in the node directory for accessing the database node:
   a. At the command line:
      
      su - db2inst1
      
      db2 list node directory
      
   b. If the local node does not appear, use the following command to add it:
      
      db2 catalog tcpip node <nodename> remote <hostname> server 50000
      
      Set <nodename> to the host name of your machine, provided it conforms to DB2 naming conventions (starts with a letter, no dashes, up to 8 characters, etc.).
      
      Set <hostname> to the host name of your machine.
      
      Use the hostname command to find out the host name of your machine.
   c. Exit the db2inst1 shell back to your root login.
5. Open a command window and change to the PDK_Lite folder:
   - On Windows, for example:
      
      cd \PDK_Lite
– On AIX or Linux, for example:
  
  ```bash
  cd /usr/PDK_Lite
  ```

6. Edit the PDK environment variables setup script to match your environment:
   – On Windows, edit environment.cmd.
   – On AIX or Linux, edit environment.sh.

   Check the environment settings carefully, including case, as any errors will cause the PDK installation to fail. When entering product paths, do not add quotes, even if the path includes spaces.

7. Set the script file permissions on AIX and Linux systems, from the PDK_Lite folder:
   
   ```bash
   chmod 755 *.sh
   chmod 755 TestDrive/Install/*/*.sh
   ```

8. Start the run_all script from the PDK_Lite folder:
   – On Windows:
     ```bash
     run_all.cmd
     ```
   – On AIX or Linux:
     ```bash
     ./run_all.sh
     ```

   When the script starts, you will get another chance to confirm your environment variables settings. On Windows, close Notepad to return to the script after you have rechecked the environment variables. The script will pause at each stage so you can review what is happening before continuing. On Windows, you will see the DB2 command window open several times. Make sure you wait for the DB2 command window to finish before continuing the main script.

**Running the PDK sample**

When the run_all script completes, you should be able to access the sample Web site with the URL [http://localhost/selfservice/index.html](http://localhost/selfservice/index.html). Click Continue to enter the site. You can then start the PDK application by clicking Run Application in the side menu.

You are presented with a start window showing the Guild's coat of arms. Clicking Click here to get the latest balances displays the balances for all weather stations and the Guild of Weather Masters. From there, you can then enter an amount to transfer from the Guild to one of the stations using a text entry field and a drop-down menu. Once you have submitted your transfer request, you are told whether it has succeeded or failed. You can then return to the balance display page to review the balances.
You can set up the PDK sample in IBM WebSphere Studio Application Developer. WebSphere Studio can be used to view or modify the application source code, and to run the application in the Studio test environment.

**Note:** This procedure assumes that you have already set up the required application databases. These are set up during the procedure described in “IBM WebSphere Application Server base V5.0” on page 423.

To install the PDK sample application in the IBM WebSphere Studio Application Developer V5.0 for Windows/Linux workspace:

1. Log in as a user with database and JMS server access:
   - On Windows, the user should be a member of the Administrators and mqm groups.
   - On Linux, the user should be a member of the mqm and mqbrkrs groups. Also run the db2profile script to allow DB2 database access. For example, add the following to your ~/.bashrc script:
     ```
     if [ -f /home/db2inst1/sqllib/db2profile ]; then
       /home/db2inst1/sqllib/db2profile
     fi
     ```
2. Download the PDK sample.
3. Extract the PDK sample to the required location, for example:
   - C:\ on Windows
   - Your home directory (~) on Linux
4. Stop WebSphere Application Server if it is running locally.
5. Start Application Developer using the -data option to specify the PDK workspace folder, for example:
   - On Windows:
     ```
     wsappdev -data C:\PDK_Lite\TestDrive\Resources\workspacePDK
     ```
   - On Linux:
     ```
     wsappdev50 -data "~/PDK_Lite/TestDrive/Resources/workspacePDK"
     ```
6. Import the CICS J2C resource adapter into your WebSphere Studio workspace. (Skip this step if you do not have access to the CICS J2C resource adapter. You will need to remove the ExchangeRateJ2C EJB later.)
   a. Open the Server perspective.
   b. Select **File -> Import** from the Studio main menu.
c. In the Import window, select RAR file as the import source and click Next.

d. In the next window, click Browse, navigate to the RAR file to import (cicseci.rar, for example), and click OK. Set the New project name to PDKConnector, and click Finish to import the J2C resource adapter.

7. Import the PDK projects into your WebSphere Studio workspace:

a. Select File -> Import from the Studio main menu.
b. In the Import window, select Existing Project into Workspace as the import source and click Next.
c. In the next window, click Browse, then navigate to the Build project folder, for example:
   - On Windows:
     C:\PDK_Lite\TestDrive\Resources\workspacePDK\Build
   - On Linux:
     ~/PDK_Lite/TestDrive/Resources/workspacePDK/Build

   Click OK, then click Finish to import the project.
d. Repeat steps a to c for each of the remaining PDK projects:
   - PDKSupport
   - PDKMessageUtil
   - PDKFinanceUtil
   - PDKFinanceEJB
   - CommandUtil
   - ExchangeRateServerEJB
   - ExchangeRateServerWeb
   - ExchangeRateServer
   - PDKLiteEJB
   - PDKLiteWeb
   - PDKLite
   - PDKCommandUtil
   - CommandServerEJB
   - CommandServerWeb
   - CommandServer
   - Servers

e. Now that you have imported all of the PDK projects, open the Tasks view. There may be three warnings for the CommandServerEJB project that can be ignored. If you have any errors, check that you imported all projects, including the CICS RAR file.
If you do not have access to the CICS RAR file, it is probably easiest just to disable the ExchangeRateJ2C EJB as follows:

i. Navigate to PDKLiteEJB -> ejbModule -> ejb-jar.xml in the Navigator view.

ii. Open ejb-jar.xml in the EJB Deployment Descriptor editor and select the Source view.

iii. Comment out the ExchangeRateJ2C session bean, for example:

```
<!--
<session id="ExchangeRateJ2C">
...
</session>-->
```

iv. Save your changes.

v. Select PDKLiteEJB -> Properties -> Java Build Path -> Projects in the Navigator view. Deselect the PDKConnector project and click OK.

You will still have a number of Connector-related errors appearing in the Task view, but the application will run (without CICS integration).

8. Next, we need to generate the EJB deployment code:

a. Select then right-click the PDKLiteEJB project, and select Generate -> Deploy and RMIC Code from the pop-up menu.

b. In the Generate Deploy and RMIC Code window, click Select all to select all the EJBs, then click Finish.

c. Repeat for the ExchangeRateServerEJB project.

Note: Do not generate EJB deployment code for the CommandServerEJB project. The CommandServerSession EJB is implemented in the WebSphere command package, provided in the WebSphere runtime.

9. Copy the Finance EJB remote classes to the PDKFinanceEJB project:

a. Switch to the J2EE perspective.

b. Select the J2EE Navigator view.

c. Copy the remote classes from PDKLiteEJB -> ejbModule -> com.ibm.pdk.ejb.finance to PDKFinanceEJB -> com.ibm.pdk.ejb.finance:

- Finance.java
- _Finance_Stub.java
- FinanceHome.java
- EJSSStatelessFinanceHomeBean_xxxxxx.java
- EJSSRemoteStatelessFinanceHome_xxxxxx.java
- _FinanceHome_Stub.java
- _EJSRemote StatelessFinanceHome_xxxxxx_Tie.java

You should overwrite any existing files in PDKFinanceEJB.

10. Next, we need to update the test server configuration:

a. Open the test server configuration in the editor. In the J2EE Hierarchy view, right-click **WebSphere v5.0 Test Environment** under the **Servers** folder, and select Open.

b. Install the J2C resource adapter in the test server. (Skip this step if you do not have access to the CICS J2C resource adapter.)

i. Click the **J2C** tab, then click the **Add** button on the right of the J2C Resource Adapters list under Node Settings.

ii. In the Create Resource Adapter window, select **PDKConnector** in the Resource Adapter Name field, then click **OK**.

iii. Select the new **PDKConnector** in the J2C Resource Adapters list, then click the **Add** button on the right of the J2C Connection Factories.

iv. In the Create Connection Factory window, set Name to **ECICICS**, set JNDI name to **eis/ECICICS**, then click **OK**.

v. In the Resource Properties list, set the properties required for your environment (for example, ServerName, ConnectionURL, PortNumber, UserName, Password).

c. Check that the PDK database authentication alias is correct. Click the **Security** tab and set your database access user ID and password in the PdkDbAuthAlias JAAS Authentication entry. We used db2admin on Windows or db2inst1 on Linux.

d. Check that the JMS variables are correct for your JMS Provider:

i. Click the **Variables** tab and check the MQ_INSTALL_ROOT and WAS_PUBSUB_ROOT variables.

ii. Also make sure the com.ibm.ws.messaging.JMSPublisher entry in the `<STUDIO_INSTALL>/runtimes/base_v5/properties/implfactories.properties` file is pointing to the correct class.

For the WebSphere Application Server embedded JMS Provider (with WebSphere Application Server installed locally), we used:

- MQ_INSTALL_ROOT to C:/WebSphere MQ on Windows, or /opt/mqm on Linux.
- WAS_PUBSUB_ROOT to C:/WebSphere MQ/WEMPS on Windows, or /opt/wemps on Linux.

- In the implfactories.properties file, set com.ibm.ws.messaging.JMSPublisher to com.ibm.ws.messaging.JMSEmbeddedProviderImpl.
For the WebSphere Studio unit test JMS Provider (with WebSphere Application Server not installed locally):

- MQ_INSTALL_ROOT to `${WAS_INSTALL_ROOT}/mqjms`.
- WAS_PUBSUB_ROOT is left blank.
- In the implfactories.properties file, set
  com.ibm.ws.messaging.JMSProvider to
  com.ibm.ws.messaging.JMSMQJDProviderImpl.
- Save the server configuration if required, then close the editor.

11. Start the test server.

12. Open the PDK home page:

   In the Navigator view, select **PDKLiteWeb**, then right-click and select **Run on Server** from the pop-up menu.

13. Try a funds transfer using Web services:

   a. Click the link to get the latest balances. Balances for each planet should be displayed.

   b. Enter an amount greater than zero, select **To JUPITER** (Jupiter is configured to use Web services by default), and click **Transfer**.

   c. You should get a message that funds have been transferred to the designated weather station.

14. Repeat for the other back-end connection technologies that are available in your environment. The default technologies for each planet are:

   - MARS uses no back-end (Stand-Alone Single Channel application pattern).
   - JUPITER uses Web services.
   - NEPTUNE uses the J2EE Connector and CICS.
   - PLUTO uses JMS and WebSphere MQ.

### Changing the default settings

Once you have installed the PDK, it is ready to run without any alteration. However, there are options you can specify to produce different effects, or to change the underlying technologies used, without having to change any source code.
Selecting different topologies

The ExchangeRate session bean controls which topology is used for each planet in order to retrieve the exchange rate during a funds transfer. By default, the topologies that are executed are:

- MARS uses no back-end (Stand-Alone Single Channel application pattern).
- JUPITER uses Web services.
- NEPTUNE uses the J2EE Connector and CICS.
- PLUTO uses JMS and WebSphere MQ.

To change the topology, edit the backendList environment entry defined in the PDKLiteEJB module deployment descriptor for the ExchangeRate session EJB.

You can edit the deployment descriptor in IBM WebSphere Studio Application Developer V5.0, or in the Application Assembly Tool provided with WebSphere Application Server (type `assembly` from a command prompt).

Configuring access to a CICS server

The PDK sample application can be configured to access a CICS server using the WebSphere Administrative Console to set the ECICICS J2EE Connector adapter properties.

Source code for our CICS application is available in `com/ibm/pdk.ejb/exchangeJ2C/cics/calcrate.c`, included in the PDKLiteEJB module.

Configuring access to a remote Web services server

The PDK sample application can be configured to access the back-end Web service running in a remote application server.

To access a remote server, edit the exchRateServerUrl environment entry defined in the PDKLiteEJB module deployment descriptor for the ExchangeRateWS session EJB.

You can edit the deployment descriptor in IBM WebSphere Studio Application Developer V5.0, or in the Application Assembly Tool provided with WebSphere Application Server (type `assembly` from a command prompt).

Deploy the updated PDK applications on the front-end application server as normal. Then deploy just ExchangeRateServer.ear on the remote application server.
Changing the results page

By default, the weather stations are displayed in a rich graphical page, using special view bean methods to calculate the number of different pieces of weather-reading equipment a station can afford.

If you would like to use a simpler (and faster) display, you can change the Struts action mapping for the display funds action in the WEB-INF/struts-config.xml file.

Change the “success” forward for the /displayFunds action path from:

displayFundsRichResults.jsp

To:

displayFundsSimpleResults.jsp.

Using the command factory

Servlets in the Web application determine which type of command to use by making a call to the command factory Singleton class’s getCommand(String type) method, passing the class name of the interface the command must implement (for example, GetAllBalancesCommand). This factory reads from the environment entries defined in the PDKLiteWeb module deployment descriptor. These environment entries map interfaces to implementing classes.

If you decide to write a new implementation, say of the GetAllBalancesCommand interface, edit the GetAllBalancesCommand environment entry to specify your new class as that to which the factory should return.

You can edit the deployment descriptor in IBM WebSphere Studio Application Developer V5.0, or in the Application Assembly Tool provided with WebSphere Application Server (type assembly from a command prompt).

Using the command target factory

Similarly, the type of command target to use is determined by making a call to the CommandTargetFactory class’s getTarget() method. This returns an instance of the first valid class found using the commandTargetList environment entry defined in the PDKLiteWeb module deployment descriptor. To change the default command target, you must edit this environment entry. The PDKLiteWeb module also has an environment entry for each of the three provided command targets:

- commandTargetLocal for com.ibm.pdk.commandTargetUtil.LocalCommandTarget
- commandTargetEJB for com.ibm.websphere.commandUtil.EJBCommandTarget
- `commandTargetServlet` for `com.ibm.pdk.commandTargetUtil.HttpServletCommandTarget`

List only the command targets you want to use in the `commandTargetList` environment entry, colon (:) separated. For example, listing only “`commandTargetLocal`” in `commandTargetList` will leave only the `com.ibm.pdk.commandTargetUtil.LocalCommandTarget` class. This ensures that all servlets use the command targeting policy best suited to running your Web and EJB containers on the same machine.

You can edit the deployment descriptor in IBM WebSphere Studio Application Developer V5.0, or in the Application Assembly Tool provided with WebSphere Application Server (type `assembly` from a command prompt).

### Adding new weather stations

You can add your own weather stations to the PDK Lite, by following these steps:

1. Edit the file `stations.txt` under the `PDK_Lite\TestDrive\Install\07ImportStationData` directory and add in a new station name and a starting funds balance, for example, "MERCURY", 575000.

2. Run the `runImportStationsData` script to import your new station data into the database.

3. Add your new weather station to the `stationsList` environment entry in the `PDKLiteEJB` module, for example:

   MARS:JUPITER:NEPTUNE:PLUTO:MERCURY

   You can edit the deployment descriptor in IBM WebSphere Studio Application Developer V5.0, or in the Application Assembly Tool provided with WebSphere Application Server (type `assembly` from a command prompt).

4. To supply an image for your new station, you must use the file-naming convention of `<STATION_NAME>.gif`, and copy your file to the `images/stations` folder in the `PDKLiteWeb` module.

5. Re-deploy the `PDKLite.ear` application and your new weather station should appear with the others when you click the link to get the latest balances. Its name is also added to the drop-down menu for you.
Improving performance

You can make the PDK Lite application run faster by disabling the comments written to the <WAS_HOME>/logs/server1/SystemOut.log file by the Java components in the PDK (servlets, command beans, and so on).

By default, debug mode is set to true. To set it to false, edit the file Logger.properties, found in the PDKSupport module that is packaged in each of the PDK enterprise applications (PDKLite.ear, ExchangeRateServer.ear, and CommandServer.ear). Change the line debug=true to debug=false.
Additional material

This publication refers to additional material that can be downloaded from the Internet as described below.

**Locating the Web material**

The Web material associated with this publication is available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser to:

ftp://www.redbooks.ibm.com/redbooks/SG246591

Alternatively, you can go to the IBM Redbooks Web site at:

ibm.com/redbooks

Select the Additional materials and open the directory that corresponds with the redbook form number, SG246591.

The PDK V5.0 sample (along with previous versions of the PDK) is also available from the Patterns for e-business Web site:

Using the Web material

The additional Web material that accompanies this publication includes the following files:

<table>
<thead>
<tr>
<th>File name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDK_Lite.zip</td>
<td>Zipped PDK V5.0 sample</td>
</tr>
</tbody>
</table>

System requirements for downloading the Web material

The following system configuration is recommended:

- **Hard disk space:** 20 MB
- **Operating System:** Windows, AIX, Solaris, Linux
- **Processor:** 500 MHz Pentium, pSeries
- **Memory:** 384 MB RAM minimum

How to use the Web material

Create a subdirectory (folder) on your workstation, and unzip the contents of the Web material zip file into this folder.
Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

IBM Redbooks

For information on ordering these publications, see “How to get IBM Redbooks” on page 442.

- Design and Implement Servlets, JSPs, and EJBs for IBM WebSphere Application Server, SG24-5754
- EJB 2.0 Development with WebSphere Studio Application Developer, SG24-6819
- IBM WebSphere V5.0 Security: WebSphere Handbook Series, SG24-6573
- Java Connectors for CICS: Featuring the J2EE Connector Architecture, SG24-6401
- Legacy Modernization with WebSphere Studio Enterprise Developer, SG24-6806
- MQSeries Programming Patterns, SG24-6506
- MQSeries Publish/Subscribe Applications, SG24-6282
- Patterns for the Edge of Network, SG24-6822
- Revealed! Architecting Web Access to CICS, SG24-5466
- Securing Web Access to CICS, SG24-5756
- Self-Service Applications using IBM WebSphere V4.0 and IBM MQSeries Integrator, SG24-6160
- Self-Service Patterns using WebSphere Application Server V4.0, SG24-6175
- Web Services Wizardry with WebSphere Studio Application Developer, SG24-6292
- WebSphere Version 5 Web Services Handbook, SG24-6891
Other resources

These publications are also relevant as further information sources:

- *WebSphere MQ Using Java*, SC34-6066.

Referenced Web sites

These Web sites are also relevant as further information sources:

- IBM Patterns for e-business
  
- IBM aphaWorks
  
IBM CICS
http://www.ibm.com/software/ts/cics

IBM developerWorks
http://www.ibm.com/developerworks

IBM WebSphere MQ
http://www.ibm.com/software/ts/mqseries

IBM Pervasive computing software
http://www.ibm.com/pvc

IBM Web services
http://www.ibm.com/software/solutions/webservices

IBM Web Services and UDDI
http://www.ibm.com/services/uddi

IBM WebSphere software platform
http://www.ibm.com/software/webservers/appserv

IBM WebSphere Developer Domain
http://www7b.boulder.ibm.com/wsdd/

Apache Jakarta Project
http://jakarta.apache.org/

Apache XML Project
http://xml.apache.org/

Bluetooth Web site
http://www.bluetooth.com

ECMAScript Language Specification
http://www.ecma-international.org/publications/standards/ECMA-262.HTM

MessageQ
http://www.messageq.com/

Rational XDE site
http://www.rational.com/products/xde/

Sun Java 2 Platform, Enterprise Edition
http://java.sun.com/j2ee

Sun Java Technology Products and APIs
http://java.sun.com/products/
How to get IBM Redbooks

You can order hardcopy Redbooks, as well as view, download, or search for Redbooks at the following Web site:

ibm.com/redbooks

You can also download additional materials (code samples or diskette/CD-ROM images) from that site.

IBM Redbooks collections

Redbooks are also available on CD-ROMs. Click the CD-ROMs button on the Redbooks Web site for information about all the CD-ROMs offered, as well as updates and formats.
## Abbreviations and acronyms

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>BMP</td>
<td>Bean Managed Persistence</td>
</tr>
<tr>
<td>CCI</td>
<td>Common Client Interface</td>
</tr>
<tr>
<td>CICS</td>
<td>Customer Information Control System</td>
</tr>
<tr>
<td>CORBA</td>
<td>Common Object Request Broker Architecture</td>
</tr>
<tr>
<td>CSS</td>
<td>Cascading Style Sheets</td>
</tr>
<tr>
<td>CICS TG</td>
<td>CICS Transaction Gateway</td>
</tr>
<tr>
<td>CVS</td>
<td>Concurrent Versioning System</td>
</tr>
<tr>
<td>DMZ</td>
<td>Demilitarized zone</td>
</tr>
<tr>
<td>DNS</td>
<td>Domain Name System</td>
</tr>
<tr>
<td>DOM</td>
<td>Document Object Model</td>
</tr>
<tr>
<td>EAD4J</td>
<td>Enterprise Application Development frameworks for Java</td>
</tr>
<tr>
<td>EAI</td>
<td>Enterprise Application Integration</td>
</tr>
<tr>
<td>EAR</td>
<td>Enterprise Archive</td>
</tr>
<tr>
<td>ECI</td>
<td>External Call Interface</td>
</tr>
<tr>
<td>EIS</td>
<td>Enterprise Information System</td>
</tr>
<tr>
<td>EJB</td>
<td>Enterprise JavaBean</td>
</tr>
<tr>
<td>EPI</td>
<td>External Presentation Interface</td>
</tr>
<tr>
<td>ERP</td>
<td>Enterprise Resource Planning</td>
</tr>
<tr>
<td>HTML</td>
<td>HyperText Markup Language</td>
</tr>
<tr>
<td>HTTP</td>
<td>HyperText Transfer Protocol</td>
</tr>
<tr>
<td>HTTPS</td>
<td>HyperText Transfer Protocol Secure</td>
</tr>
<tr>
<td>IBM</td>
<td>International Business Machines Corporation</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environments</td>
</tr>
<tr>
<td>IIO</td>
<td>Internet Inter-ORB Protocol</td>
</tr>
<tr>
<td>IPSec</td>
<td>Internet Protocol Security</td>
</tr>
<tr>
<td>ITSO</td>
<td>International Technical Support Organization</td>
</tr>
<tr>
<td>J2C</td>
<td>J2EE Connector</td>
</tr>
<tr>
<td>J2EE</td>
<td>Java 2 Platform, Enterprise Edition</td>
</tr>
<tr>
<td>JAF</td>
<td>JavaBeans Activation Framework</td>
</tr>
<tr>
<td>JAR</td>
<td>Java archive</td>
</tr>
<tr>
<td>JDBC</td>
<td>Java database connectivity</td>
</tr>
<tr>
<td>JMS</td>
<td>Java Message Service</td>
</tr>
<tr>
<td>JNDI</td>
<td>Java Naming and Directory Interface</td>
</tr>
<tr>
<td>JRE</td>
<td>Java Runtime Environment</td>
</tr>
<tr>
<td>JSP</td>
<td>JavaServer Pages</td>
</tr>
<tr>
<td>JTA</td>
<td>Java Transaction API</td>
</tr>
<tr>
<td>JVM</td>
<td>Java Virtual Machine</td>
</tr>
<tr>
<td>LDAP</td>
<td>Lightweight Directory Access Protocol</td>
</tr>
<tr>
<td>MQAI</td>
<td>WebSphere MQ Administration Interface</td>
</tr>
<tr>
<td>MQSC</td>
<td>WebSphere MQ Commands</td>
</tr>
<tr>
<td>OAM</td>
<td>Object Authority Manager</td>
</tr>
<tr>
<td>OLTP</td>
<td>online transaction processing</td>
</tr>
<tr>
<td>ORB</td>
<td>Object Request Broker</td>
</tr>
<tr>
<td>PCF</td>
<td>Programmable Command Format</td>
</tr>
<tr>
<td>PDA</td>
<td>Personal Digital Assistant</td>
</tr>
<tr>
<td>PDK</td>
<td>Patterns Development Kit</td>
</tr>
<tr>
<td>PKI</td>
<td>Public-Key Infrastructure</td>
</tr>
<tr>
<td>Acronym</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>RACF</td>
<td>Resource Access Control Facility</td>
</tr>
<tr>
<td>RAR</td>
<td>Resource Adapter Archive</td>
</tr>
<tr>
<td>RMI</td>
<td>Remote Method Invocation</td>
</tr>
<tr>
<td>SOA</td>
<td>Service oriented architecture</td>
</tr>
<tr>
<td>SSL</td>
<td>Secure Sockets Layer</td>
</tr>
<tr>
<td>UDDI</td>
<td>Universal Description Discovery and Integration</td>
</tr>
<tr>
<td>URL</td>
<td>Uniform Resource Locator</td>
</tr>
<tr>
<td>VRU</td>
<td>Voice Response Unit</td>
</tr>
<tr>
<td>WAP</td>
<td>Wireless Application Protocol</td>
</tr>
<tr>
<td>WAR</td>
<td>Web Archive</td>
</tr>
<tr>
<td>WAS</td>
<td>WebSphere Application Server</td>
</tr>
<tr>
<td>WLM</td>
<td>Workload Management</td>
</tr>
<tr>
<td>WML</td>
<td>Wireless Markup Language</td>
</tr>
<tr>
<td>WS-I</td>
<td>Web Services Interoperability Organization</td>
</tr>
<tr>
<td>WSDK</td>
<td>WebSphere SDK for Web Services</td>
</tr>
<tr>
<td>WSDL</td>
<td>Web Services Description Language</td>
</tr>
<tr>
<td>WSIF</td>
<td>Web Services Invocation Framework</td>
</tr>
<tr>
<td>WSTK</td>
<td>Web Services Toolkit</td>
</tr>
<tr>
<td>XDE</td>
<td>(Rational) Extended Development Environment</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
<tr>
<td>XSL</td>
<td>Extensible Stylesheet Language</td>
</tr>
<tr>
<td>XSLT</td>
<td>Extensible Stylesheet Language Transformations</td>
</tr>
</tbody>
</table>
Index

Symbols
.NET 129

Numerics
80/20 situation 3

A
Agent application pattern 23
Agent-oriented service definition 107
AIX 43, 48, 394, 415, 423
ALTER QMGR 232
AMQ6767 246
AMQ7467 245
AMQ7468 245
Apache SOAP 183
APPLET tag 60
Application design 87
Application development 176
Web services 180
Application gateways 27
Application patterns 5, 12, 20
Application server 29
ASCII 190
As-is Host application pattern 22
ASR See Automatic Speech Recognition
Asynchronous CCI calls 148
Asynchronous messaging 155
Asynchronous Web services 121, 123
Auditing 210
Authorization 208
Automatic Speech Recognition 63
Axis 185

B
Backups 246
Bandwidth 29
Bean Managed Persistence 73, 110
Bean Scripting Framework 183
Best practices 5, 16
Enterprise JavaBeans 109
J2EE Connector Architecture 149
Java Message Service 172
Web services 129
WebSphere MQ 172
Bindings mode 168
Bluetooth 64
BMP See Bean Managed Persistence
Business Logic 94
Business patterns 5, 7
Self-Service 19
BytesMessage 154

C
C programming language 48, 190
Cache 108
EJB Home interface 110, 275
calcrate.c 363
Cascading Style Sheets 57, 95
Validator tools 58
Catcher servlet 106
CCI See Common Client Interface
CESN transaction 146
Channel events 241
cHTML 67
CICS Gateway for Java 41
CICS See Customer Information Control System
CICS TG See CICS Transaction Gateway
CICS Transaction Gateway 41
Local 356
Remote on z/OS 357
TCP protocol 50
Tracing 216
CICS Transaction Server 41
Configuration 360
Circular logging 244–245
Class diagrams 261
J2EE Connectors 336
JMS 367
Web services 285
ClearCase LT 278
Client container 56
Client mode 169
CLUSRCVR 231
CLUSSDR 231
Cluster receiver channel 231
Cluster sender channel 230–231
Clustering 171, 230, 232, 240
CMP See Container Managed Persistence
COBOL 190
Collection object 154
Command
  execute() 106
Command beans 100
Command caching 107
  Advantages 108
  Disadvantages 108
Command classes 108
Command interface 108
Command model 105
Command package 106
Command shipping 106
  Advantages 106
  Disadvantages 106
COMMAREA 149, 190, 342
Common Client Interface 135, 137, 186
  Asynchronous calls 148
  Framework classes 138
Common Object Request Broker Architecture See CORBA
Component managed sign-on 144
Composite patterns 5, 10
Concurrent Versioning System 278
Confidentiality 210
Connection 345
Connection management 140
Connection pooling 110, 140
  Tuning 217
ConnectionFactory 166
  Tracing 216
Container Managed Persistence 73, 110
Container managed sign-on 144
Control commands 234
Controller 66, 90
Cookies 98
CORBA 74, 84
Correlation ID 122, 155
crtmqm 248
CSS See Cascading Style Sheets
Customer Information Control System 41
  See also IBM CICS
  Data conversion 191
  ECI resource adapter 145–146
  Failover 220
Resource adapters 80, 145
Scalability 218
System management 215
Workload management 218
CVS See Concurrent Versioning System

D
Data conversion 190, 344
Database server node 29
DB2 stored procedures 183
dds.xml 316
Decomposition application pattern 23
DEFINE CLUSCHL 232
DEFINE QLOCAL 232
Demilitarized Zone 29–30, 33
Design
  Application 87
  J2EE Connector Architecture 135
  Java Message Service 150
  System 87
  Web services 111
Destination 166
Development
  J2EE Connector Architecture 186
  Java Message Service 192, 194
  Web services 177
DFHCNV 191
DFHMIRS, CICS mirror program 191
DHTML 67
Directly Integrated Single Channel
  Product mappings 46
  Runtime pattern variations 36
  Runtime patterns 32
Directly Integrated Single Channel application pattern 22, 253
Directory and security services node 30
Directory services 28
Display funds 256
Distributed Program Link 149
DMZ See Demilitarized Zone
DNS connection optimization 220
DNS See Domain Name Server
Domain firewall 28, 30
Domain Name Server 27
DPL See Distributed Program Link
DSPMQAUT 249
Dynamic behavior 199
Dynamic delegation 107
Dynamic HTML
   DHTML 55, 57
Dynamic Web services 78, 119

E
EAD4J 104
EAI See Enterprise Application Integration
EBCDIC 190, 344
ebXML 132
ECI resource adapter 145–146
ECI See External Call Interface
ECMA-262 59
ECMAScript 59
EJB container 72, 110, 229
EJB modules 74
EJB remote interface 302, 329
EJB See Enterprise JavaBeans
EMBED tag 60
dmqm 246
Enterprise Access Builder 343
Enterprise application 74, 297
Enterprise Application Development frameworks for Java See EAD4J
Enterprise Application Integration 75
Enterprise JavaBeans 72
   Bean Managed Persistence
      Best practices 109
   Cache the EJB Home interface 110
   Container Managed Persistence
      Entity beans 72, 105, 110, 269
      Message-driven beans 72, 159, 161, 376
      Session beans 72, 109, 300, 347
      Stateless session beans 110
Enterprise Resource Planning 79
Enterprise Services 187
Enterprise Services toolkit 187
Entity EJBs 72, 105, 110, 269
EPI See External Presentation Interface
ERP See Enterprise Resource Planning
execute() 106
Existing applications 28
Extensible Stylesheet Language Transformations 69
External Call Interface 145
External Presentation Interface 146

F
Failover

CICS 220
Web services 206
WebSphere MQ 233
Firewall 43, 47, 206
Domain 28
Protocol 27
Formatter beans 99
   Advantages 99
   Disadvantages 99

G
GenericRecord 342
Get authority 249
getExchangeRate 273, 300, 350, 372
getProperty 98
Global Services Method 176
Global transactions 142
GSMethod 176
Guidelines 5, 16

H
Helper objects 110
HTML 55, 57
   Validator tools 57
HTTP 45, 49
HTTP server 26
HTTP tunneling 61
HTTPS 45, 49
HttpServletRequest 106
HttpServletResponse 106

I
IBM HTTP Server 43, 47
IBM UDDI Business Registry 113
IDE See Integrated Development Environments
IMAP 75
i-mode 63
IMS J2C resource adapter 80, 192
inetc 238, 395
Instrumentation 240
Integrated Development Environments 91
Integration patterns 5, 8
Integrity 210
Interaction 344
Interaction Controller 66, 90
Interaction diagrams 262
   J2EE Connectors 338
JMS 368
Web services 286
Internet Protocol Security 207
Interoperability
Web services 127, 133
IPSec See Internet Protocol Security

J
J2C See J2EE Connector Architecture
J2C tool plug-in 189
J2EE 53, 56, 66, 80
J2EE 1.3 82
J2EE Connector Architecture 38, 79, 135, 186, 215, 333
Advantages 81
Availability 218
Best practices 149
CICS resource adapters 145
Installation 346
Class diagrams 336
Design guidelines 135
Development guidelines 186
Disadvantages 81
Enterprise Services 187
IMS resource adapter 80, 192
Input record 342
Interaction diagrams 338
JNDI lookup 340
Logging 216
Managed environment 136
Non-managed environment 136
Output record 342
Sample scenario 333
Scalability 218
Security 143, 222
System management 215
Tracing 216
Workload management 218
JAF See JavaBeans Activation Framework
Java 2 Platform, Enterprise Edition See J2EE
Java applets 59
Disadvantages 60
Java Database Connectivity 45, 49, 137
Java Message Service 38, 82, 150, 192, 194, 225, 365
Advantages 83
Asynchronous 155
Best practices 172

Class diagrams 367
Consumers 161, 172
Design guidelines 150
Development guidelines 192, 194
Disadvantages 84
Error handling 173
Interaction diagrams 368
Java bindings mode 168
Java client mode 169
JNDI 165
JNDI lookup 371
JNDI name spaces 382
Managing resources 372
Message models 150
Message selectors 374
Message time to live 375
Message types 154
Non-persistent messages 375
Point-to-point messaging 152
Producers 161, 172
Publish /subscribe messaging 153
Publish/subscribe 376
Request/reply
Synchronous 373
Sample scenario 365
Synchronous 155
System management 225
WebSphere MQ support 83
Java Naming and Directory Interface 74, 165
Java Runtime Environment 60
Java Transaction API 74
JavaBeans 67
JavaBeans Activation Framework 74
JavaMail 75
JavaScript 56, 59
JavaServer Pages 67, 268
JAX-RPC 185
JBDC See Java Database Connectivity
JMS See Java Message Service
JMSException 173
JNDI See Java Naming and Directory Interface
JRE See Java Runtime Environment
JScript 58–59
JSR 101 185
JTA See Java Transaction API

L
Layered design 130
LDAP  74
Linear logging  244
Linux  44, 48, 394, 417, 423
Load Balancer  220
Load balancing  206
Local queue manager  397
Local queues  229
Local transaction optimization  143
Local transactions  141
Logging  216
Loose coupling  130

M
Macromedia Flash  56
Managed environment  136
MapMessage  155
Message models  150
Message selector  172
Message structure  120, 289
Message timeout  172
MessageConsumer  165
Message-driven beans  72, 159, 161, 376
Message-oriented Web services  117
MessageProducer  165
messagerouter  316
Messaging  83
XML  289
Messaging mechanisms  117, 179
Microbrowser  64
MIME  75
Mirroring  246
Mobile clients  63
Model  66, 94
  Class diagram  200
  Dynamic behavior  199
  Interaction diagram  199
  Sequence diagram  199
  Static behavior  200
Model II  102
Model-View-Controller  66, 89, 161, 266
  Advantages  94
  Disadvantages  94
MQAI See WebSphere MQ Administration Interface
MQCLOSE  233
MQGET  235, 249
mqm group  248
MQOO_BIND_AT_OPEN  233
MQOO_BIND_NOT_FIXED  233
MQOPEN  233
MQPUT  235, 240, 249
MQSC See WebSphere MQ Commands
MQSeries See WebSphere MQ
MS0D WebSphere MQ SupportPac  229
MVC See Model-View-Controller

N
Node types  26
Non-managed environment  136
Non-persistent messages  172
Non-repudiation  210
Notification operation  117

O
OAM See Object Authority Manager
Object Authority Manager  248
Object Management Group  84
Object Request Broker  74
ObjectMessage  154
OTMA See Online transaction processing
OMG See Object Management Group
One-way transmission  116
Online transaction processing  41

P
Page Constructor  93
Palm-OS  63
Patterns Development Kit  253
CICS application  362
Class diagrams  261
Download  437
Interaction diagrams  262
Sample scenario  253
Use cases  255
Patterns for e-business  3
  Application patterns  5, 12
  Best practices  5, 16
  Business patterns  5, 7
  Self-Service  19
  Composite patterns  5, 10
  Guidelines  5, 16
  Integration patterns  5, 8
  Node types  26
  Product mappings  5, 16, 37, 42, 46
  Runtime patterns  5, 13, 25, 29, 32
Web site 6
PCF See Programmable Command Format
PDA See personal digital assistant
PDK See Patterns Development Kit
Performance events 241
Performance Monitor 242
Performance Viewer 217
Persistent messages 172
personal digital assistant 26, 88, 95
Pervasive computing 26
PKI See Public Key Infrastructure
Point-to-point messaging 152, 371
POP3 75
Product mappings 5, 16, 37, 42, 46
Programmable Command Format 235
Protocol firewall 27
Provider-dynamic 119
Public Key Infrastructure 27, 70, 214
Publish/subscribe 153, 159
Put authority 249

Q
Quality of Service 134, 149
Queue Full event 240
Queue manager
   Events 240
      Full repository 231

R
RACF See Resource Access Control Facility
RAR See Resource Adapter Archive
Rational ClearCase LT 278
Rational Unified Process 176
Rational XDE 195, 200
   Model projects 196
RealPlayer 56
Redbooks Web site 442
   Contact us xvi
Remote administration 237
Remote Method Invocation 74
Remote queues 230, 240
Reply queue 394
Repository queue managers 230–231, 240
Request queue 390, 397
Request/reply 156
Request-response transmission 116
Resource Access Control Facility 222
Resource Adapter Archive 135

Resource adapters
   CICS 80, 145
Result beans 96, 98, 100
   Advantages 97
Reverse engineering 200
RMI See Remote Method Invocation
RMI/IIOP 74, 84
Round-trip engineering 200
Router application pattern 22
RPC-based Web services 117
rpcrouter 316
runmqslr 395
runmqsc 235, 239, 399
Runtime patterns 5, 13, 25, 29, 32, 38

S
S/390 190
Sample scenario
   J2EE Connector Architecture 333
      Java Message Service 365
         Patterns Development Kit 253
   Web services 281
Scalability
   J2EE Connectors 218
   Web services 132
   WebSphere MQ 229
Screening routers 27
SCRIPT tag 60
Secure Sockets Layer 207, 224
Security
   Component managed sign-on 144
      Container managed sign-on 144
   J2EE Connector Architecture 143, 222
   Web services 207
   WebSphere MQ 247
Security services 28
Self-Service business pattern 19–20
Send-and-forget 157
Service broker 113
Service oriented architecture 78
Service provider 113
Service requester 114
Servlets 66
Session EJBs 72, 109, 273, 300, 347
setInteractionVerb 148
SETMQAUT 248–249
Simple Object Access Protocol 49
   Admin client 317

450 Patterns: Self-Service Application Solutions Using WebSphere V5.0
Apache SOAP 183
Messaging mechanisms 117, 179, 288
Security 213
Singleton pattern 265
SMTP 75
SOA See Service oriented architecture
SOAP See Simple Object Access Protocol
SoapEarEnabler 184
Solicit-response 116
Stand-Alone Single Channel
Product mappings 42
Runtime pattern variations 32
Runtime patterns 29
Stand-alone Single Channel application pattern 21, 253
Stateless session beans 110
Static relationships 200
Static Web services 78, 119
StreamMessage 154
Struts 102, 266, 269
Sun ONE 129
SVRCONN 399–400
Swing 59
Synchronous messaging 155
Synchronous Web services 121
Sysplex distributor 220
System contracts 135, 140
System design 87
System management
J2EE Connector Architecture 215
Java Message Service 225
Web services 206
SYSTEM.ADMIN.CHANNEL.EVENT 241
SYSTEM.ADMIN.PERFM.EVENT 241
SYSTEM.ADMIN.QMGR.EVENT 241
SYSTEM.ADMIN.SVRCONN 238

Transaction ID 122
Transaction management 140, 345
Transaction Server 41
Transactions
Global 142
Local 141
Local optimization 143
Transfer funds 257
Transmission patterns 115, 177
Transports 122
TTS See Text-to-Speech
two-phase commit 276, 345
Type-dynamic 120

U
UDDI See Universal Description Discovery and Integration
UDDI4J 183
Unicode 190, 344
Universal Description Discovery and Integration 113
Universal test client 303, 354
Use cases 255
useBean 98

V
Validating input 264
Validator tools
CSS 58
HTML 57
VBScript 58
View 66, 93
View beans 97
Advantages 98
Disadvantages 98
VisualAge for Java 91, 343
Voice-enabled applications 63
VoiceXML 67

W
WAP See Wireless Application Protocol
Web application server 26, 65
Web application server node 30
Web browser 56
Web client 55
Web container 66
Web modules 74
Patterns: Self-Service Application Solutions Using WebSphere V5.0

Web server  26
Web server plug-in  43, 47
Web server redirector  29, 43, 47
Web services  38, 111, 177, 206, 281
Advantages  79
Application development  180
Application pattern  288
Asynchronous  121, 123
Best practices  129
Class diagrams  285
Client proxy  320
Creation  306
Design guidelines  111
Development guidelines  177
Disadvantages  79
Dynamic  78, 119
Exchange Rate sample  306
Failover  206
Interaction diagrams  286
Interoperability  127, 133
Maintainability  133
Message structure  120
Message-oriented  117
Reliability  134
Requesters  323
Reuseability  133
RPC-based  117
Sample scenario  281
Scalability  132
Security  207
Service broker  113
Service provider  113
Service requester  114
Static  78, 119
Synchronous  121, 288
System management  206
Transmission patterns  115, 177, 288
Transports  122
Workload management  206
Web Services Description Language  112, 187
Web Services Interoperability Organization  127
Web Services Invocation Framework  189
Web Services Toolkit  185
WebSphere Application Server  39
J2C configuration  358
JMS configuration  401
SOAP EAR enabler  184
SOAP support  183
UDDI support  184
Web server plug-in  43, 47
Web services support  183
XML support  184
WebSphere Command Framework  105, 271
WebSphere MQ  42, 172
Backups  246
Best practices  172
Client  193
Cluster receiver channel  231, 387, 393, 396, 398
Cluster repository queue manager  389, 396
Cluster sender channel  230–231, 393
Clustering  171–172, 230
Configuration for AIX  394
Configuration for Linux  394
Configuration for Windows  385
Control commands  234
Events  240
Failover  233
Instrumentation  240
JMS support  83
Local queue manager  386, 392, 395
Local queues  229
Log files  245
Recovery  244
Remote administration  237
Reply queue  399
Restart  244
Scalability  229
Security  247
Server  193
System management  225
Workload management  233
WebSphere MQ Administration Interface  235
WebSphere MQ Alert Monitor  241
WebSphere MQ classes for Java  167
WebSphere MQ classes for JMS  167
WebSphere MQ Commands  235
WebSphere MQ Explorer  235, 239, 400
WebSphere MQ Services  236, 239, 241
WebSphere MQ SupportPac MS0D  229
WebSphere SDK for Web Services  184
WebSphere Studio Application Developer  91
J2C development  186
JMS configuration  377
JMS development  193
Rational XDE  196
Web Service wizard  307
Web services development 182
WebSphere Studio Application Developer Integration Edition
  Enterprise Services development 187
  J2C development 187
WebSphere Web Services for J2EE Technology Preview 185
Windows 42, 46, 414, 423
Windows CE 63
Wireless Application Protocol 56, 63–64
  Microbrowser 64
Wireless Markup Language 56, 64, 67
WML See Wireless Markup Language
WMLScript 64
Workload management 42
  CICS 218
  Web services 206
  WebSphere MQ 233
WS-Authorization 211
WS-Coordination 128
WSDL See Web Services Description Language
WS-Federation 211
WS-I See Web Services Interoperability Organization
WSIF See Web Services Invocation Framework
WS-Security 211
WS-Transaction 128

X
XA resource coordinator 169
XAResource 142
XCF 80
XDE See Rational XDE
XForms 63
XHTML
  Extended HTML 62
XML 55, 64, 67, 131
  Encryption 70, 214
  Messaging 289
  Performance 132
  Security 214
XML-SOAP Admin client 317
XSLT See Extensible Stylesheet Language Transformations

Z
z/OS 415
zSeries 48, 190
Patterns: Self-Service Application Solutions Using WebSphere V5.0
Patterns: Self-Service Application Solutions Using WebSphere V5.0

Integrate Web applications with the enterprise tier
Explore Web services, J2EE Connectors, and JMS solutions
Learn by example with sample scenarios

The Patterns for e-business are a group of proven, reusable assets that can be used to increase the speed of developing and deploying Web applications. This IBM Redbook focuses on the Self-Service::Stand-Alone Single Channel application pattern for facilitating user access to business sites, and the Self-Service::Directly Integrated Single Channel application pattern for including one or more point-to-point connections with back-end applications.

Part 1 of this publication guides you through the process of selecting an Application and Runtime pattern. Next, the platform-specific Product mappings are identified based upon the selected Runtime pattern.

Part 2 of this publication provides a set of guidelines for building your Web application and for integrating it with the enterprise tier using Web services, J2EE Connectors, and JMS. These guidelines include technology options, application design, application development, and systems management.

Part 3 of this publication teaches you by example how to design and build sample solutions using IBM WebSphere Application Server V5.0 with Web services, J2EE Connectors, and IBM CICS, and JMS and IBM WebSphere MQ.

SG24-6591-00     ISBN 073842787X