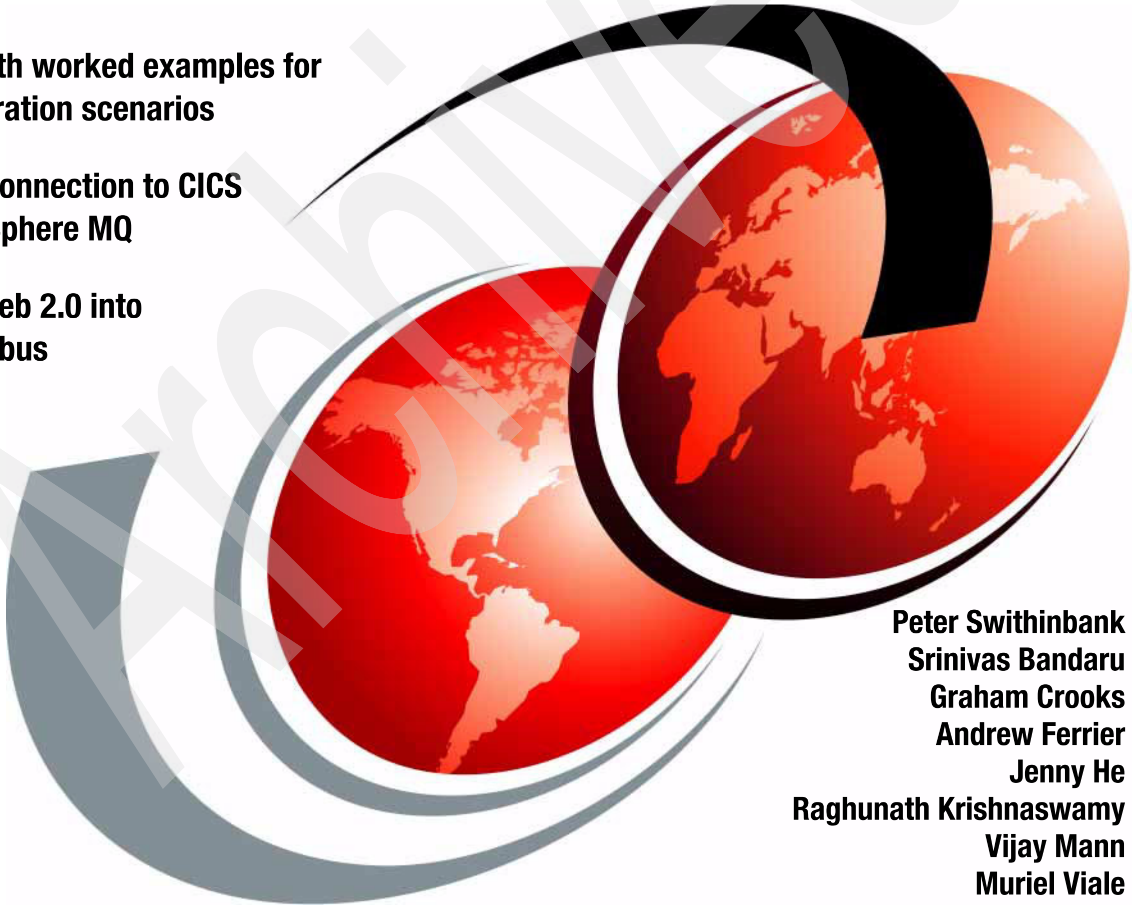
Connecting Enterprise Applications to WebSphere Enterprise Service Bus

- Patterns with worked examples for many integration scenarios
- Code-free connection to CICS using WebSphere MQ
- Integrate Web 2.0 into the service bus



Peter Swithinbank
Srinivas Bandaru
Graham Crooks
Andrew Ferrier
Jenny He
Raghunath Krishnaswamy
Vijay Mann
Muriel Viale

Redbooks

ibm.com/redbooks

International Technical Support Organization

Connecting Enterprise Applications to WebSphere Enterprise Service Bus

September 2007

SG24-7406-00

Note: Before using this information and the product it supports, read the information in "Notices" on page xi.

First Edition (September 2007)

This edition applies to Version 6.0.2 of WebSphere Enterprise Service Bus.

Contents

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 illustrate 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.

Trademarks

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

Redbooks (logo) ®	CICS®	MVS™
developerWorks®	DataPower®	Rational®
iSeries®	DB2®	Redbooks®
pSeries®	Informix®	S/360™
z/OS®	IBM®	WebSphere®
zSeries®	IMS™	
Cloudscape™	MQSeries®	

The following terms are trademarks of other companies:

BAPI, SAP, and SAP logos are trademarks or registered trademarks of SAP AG in Germany and in several other countries.

Oracle, JD Edwards, PeopleSoft, Siebel, and TopLink are registered trademarks of Oracle Corporation and/or its affiliates.

DataStage, are trademarks or registered trademarks of Ascential Software Corporation in the United States, other countries, or both.

EJB, Java, Java Naming and Directory Interface, Javadoc, JavaScript, JDBC, JSP, J2EE, Sun, and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

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

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

Preface

Is this an IBM® Redbooks® publication about connectors and adapters, a book about the WebSphere® Enterprise Service Bus (or WebSphere ESB, as it is commonly called)? There are Redbooks about using adapters to connect applications to WebSphere, and Redbooks about the WebSphere Enterprise Service Bus. Do they not cover everything one needs to know about how to connect applications to the WebSphere Enterprise Service Bus?

In a sense they do. If you need to build a WebSphere Adapter, the IBM Redbooks publication *WebSphere Adapter Development*, SG24-6387, tells you how to use the prepackaged WebSphere Adapters and how to build a new custom WebSphere Adapter using the WebSphere Adapter Toolkit. If you want to connect to SAP®, then *Connect WebSphere Service-Oriented Middleware to SAP*, SG24-7220, tells you all you want to know. There is an IBM Redbooks publication called *Getting Started with WebSphere Enterprise Service Bus V6,* SG24-7212, that goes a lot deeper into how to build solutions using an Enterprise Service Bus than its modest title suggests. And there are Redbooks on solution patterns for using an ESB, for example, *Patterns: Integrating Enterprise Service Buses in a Service-Oriented Architecture*, SG24-6773, and the seminal *Patterns: Implementing an SOA using an Enterprise Service Bus*, SG24-6346. There are also Redbooks and Redpapers on particular adapters, such as *A Simple Example: Using the WebSphere Adapter for Flat File*, REDP-4235, and *"WebSphere Business Integration Adapters: An Adapter Development and WebSphere Business Integration Solution*, SG24-6345. There is also an IBM Redbooks publication in preparation that describes how to migrate from these WebSphere Business Integration Adapters to the new generation of WebSphere Adapters, *WebSphere Adapter Development*, SG24-6387.

What is missing from this weighty bookshelf? What is missing is a book that looks at the problem of *choosing how* to integrate applications into a service-oriented architecture solution using an Enterprise Service Bus.

This book takes as its starting point that at the heart of any WebSphere-based service-oriented solution is an Enterprise Service Bus architecture that connects all the components in the solution together as software services. The are different products you can use to implement the Enterprise Service Bus architecture. The WebSphere Enterprise Service Bus *product* is a J2EE™ implementation of the Enterprise Service Bus *architecture* based upon the Service Component Architecture. As a product, it is not the only WebSphere product to implement an Enterprise Service bus, but it is the only WebSphere product[1] (at least today) to implement the Enterprise Service Bus using the

Service Component Architecture. If you have decided to make Service Component Architecture the platform for your service-oriented architecture, then the WebSphere Enterprise Service Bus will be at the center of your solution, connecting applications together and integrating them as software services.

The WebSphere Enterprise Service Bus should be seen then, not only as a means to connect services together using the Service Component Architecture, but also a means to connect applications to services using the Service Component Architecture — the center, as IBM marketeers like to say, of the WebSphere Connect Ecosystem. The WebSphere Enterprise Service Bus, in addition to providing functionality to help manage connecting services together, also provides functionality to build the *on-ramp* and *off-ramp* to a service-based architecture for a variety of different types of application connectivity, from standard Web services to complex packaged applications.

At the heart of this book are descriptions of different patterns to follow to connect applications to the SCA using WebSphere ESB, and advice on how to choose which pattern, and which implementation to use.

You learn in this book that your WebSphere ESB connection options are far from exhausted even when you have searched an application in vain for its Web service interface to use to connect to WebSphere ESB. WebSphere Enterprise Service Bus can be thought of as an services adapter platform that you extend to connect applications that do not have a service interface or a ready-built adapter to use. WebSphere ESB provides other options to connect applications without building a custom WebSphere Adapter. But you need to decide which implementation route to take. Sometimes building a custom WebSphere Adapter is the correct course to pursue, and sometimes extending WebSphere ESB interfaces is a quicker and cheaper option. We provide theoretical and practical information to guide you to make a good choice.

Our goal in this book is to introduce you to the possibilities of using the WebSphere Enterprise Service Bus beyond its comfort-zone of connecting services together, to describe a set of business and connection patterns to help you select which connectivity solution to pursue, and to provide you with working examples of solutions that extend the connectivity capabilities of WebSphere ESB beyond services and prepackaged adapters.

[1] WebSphere Process Server includes WebSphere Enterprise Service Bus.

Choice of examples

Our choices of examples in this book are driven partly by the field experience of the residents who contributed to its writing, and partly by the desire to describe connectivity options that are not covered in other books and articles. We hope that two sets of examples should prove particularly interesting to you:

- Using the CICS/MQSeries® gateway as a loose coupling between a WebSphere Application and a CICS® application. The gateway uses WebSphere MQ messaging to connect to CICS. Compare it with the CICS Transaction Gateway, which offers a synchronous interface to CICS, and is used by the WebSphere Java™ Connector Architecture (JCA) adapters to CICS.

  Both solutions have their place, but using the asynchronous solution has, until now, only been available to enterprises that have invested in the WebSphere Message Broker to perform the necessary mapping of the CICS COMMAREA in the MQ message. We describe how the same popular solution can easily be down-scaled to work with the WebSphere ESB product.

  We are indebted to the work of some of our colleagues in Hursley to describe an extension of the same solution, which is available to download into WebSphere Integration Developer using the standard Rational® Product Updater tool. The extension uses an Eclipse plug-in to add a customizable mediation primitive that removes the need to write any code to implement the solution.

- The second noteworthy set of examples is based on using WebSphere ESB as the platform to connect the WebSphere family to the rapidly growing number of loosely termed Web 2.0 applications on the Internet using Service Component Architecture. These examples demonstrate alternate patterns of connection to and from WebSphere ESB to http: applications.

Audience

The primary audience for this book is two-fold. Firstly, it has been written for Infrastructure or Enterprise architects who want to understand what value WebSphere ESB has to offer and what role it plays in integrating existing applications and new non-J2EE applications into a service-oriented architecture. Part 1, “Background” on page 1, and Part 2, “Scenarios and patterns” on page 125, which review Enterprise Application Integration, Service Component Architecture, the WebSphere Enterprise Service Bus and adapters, and then define a number of business and connection integration patterns, will be of particular interest to this audience.

Part 3, “Working examples” on page 171, will be of interest to our other primary audience, which is integration specialists who are implementing some of these solutions. Part 3, “Working examples” on page 171, is a set of practical examples and is provided with additional supporting material you can download to help you model your solution on what the residents have already implemented in simple examples.

In writing this book, we were particularly asked to consider the needs of IBM Business Partners who are frequently engaged in connecting applications, sometimes their own applications, to their clients’ IT infrastructure. The selection of business patterns was influenced by answering the question, “what kinds of scenarios are typical of Business Partner engagements, and how can using ESB architecture simplify the implementation of solutions that partners are asked for?” So, for example, we base one of the business patterns on a Business Partner who comes to engagements with existing applications that they then tailor to their client’s needs. By incorporating WebSphere ESB into their architecture they may be able to perform much of the connectivity customization using WebSphere ESB rather than making specialized modifications to their own applications, or writing their own *mini-ESB* customization component themselves.

The team that wrote this book

This book was produced by a team of specialists from around the world working at the International Technical Support Organization, Raleigh Center.

Figure 1 Peter, Andrew, Sreenivas, Raghu, Muriel, and Vijay

Peter Swithinbank is a Project Leader at the ITSO, Hursley Center. He writes IBM Redbooks and teaches IBM classes worldwide on Web services and building business integration solutions. Peter has worked for IBM for 28 years and has been with the ITSO for two years. He has a diploma in software engineering from Oxford University and an M.A. in Geography from the University of Cambridge.

Srinivas Bandaru has 12 years of experience in the software industry and is currently a Process Server and Message Broker Specialist at Sarasu IT Solutions, Ltd, an IBM Business Partner. Sreenivas has a B.S. in Electrical Engineering from Jawaharlal Nehru Technological University, Hyderabad, India, and an M.S. in Computer Science and Engineering from the University of South Carolina.

Graham Crooks is an e-Architect covering the Asia Pacific region and is based in Melbourne, Australia. He has 14 years of experience in the IT field and five years in Web development. He has worked at IBM for seven years. He has written and taught extensively on WebSphere Application Server application development with the Rational toolset.

Andrew Ferrier is a Staff Software Engineer at IBM Hursley Laboratories in the United Kingdom. He has almost five years of experience in middleware and messaging, including WebSphere MQ, WebSphere Message Broker, and WebSphere ESB. He holds a degree in Computing from Imperial College, London. His particular area of expertise is WebSphere ESB. He currently works in the Customer SWAT Team for that product. He writes externally on the SOA tips 'n' tricks blog at: `http://soatipsntricks.wordpress.com/`.

Jenny He is a Solution Test Specialist in the United Kingdom. She has five years of experience with various software and platforms. She holds a Ph.D. in the field of Optical Networks from the University of Essex in the United Kingdom. She has been actively involved in modeling and monitoring business processes since she joined IBM five years ago. Her areas of expertise include WebSphere Message Broker, WebSphere Business Modeler, and WebSphere Integration Developer. Jenny is a member of IEEE.

Raghunath Krishnaswamy is an IT Architect for IBM Global Business Services India. He has nine years of IT experience. He is an IBM Certified Solution Designer for service-oriented architectures, specializing in process modeling, enabling, and messaging and brokering architectures using WebSphere product suites. He holds a Bachelor of Engineering degree in Electronics and communication from the University of Madras (Chennai).

Vijay Mann is a Technical Staff Member at the IBM India Research Laboratory (IRL) in New Delhi, India. He is currently a part of the high-performance computing group at IRL. Prior to this, he was a part of the WebSphere Application Server eXtended Deployment (XD) development team. He has been with IBM since 2003 and has worked extensively with various WebSphere Application Server offerings in the areas of BPEL4WS-based business processes, Web services, performance tuning, and problem determination. He holds a bachelor’s degree in Electronics and Telecommunications from Malaviya National Institute of Technology, Jaipur, India, and a master’s degree in Computer Engineering from Rutgers, The State University of New Jersey, USA.

Muriel Viale is a Certified IT Specialist working in SW IOT WebSphere Technical Sales support, located in La Gaude, France. Muriel has worked for IBM for 20 years, focusing on distributed object technologies since 1997. She is a WebSphere Application Server product specialist, J2EE architecture and deployment consultant, and has 10 years of customer experience in Europe, with a special focus on Application Integration.

We would particularly like to thank the following people for their help with this project.

Sue Bayliss is a Software Engineer in the Scenario Analysis Lab and Federated Integration Test in IBM Hursley, and gave us access to their CICS system to test the IBM WebSphere Integration Developer - WebSphere MQ-CICS Bridge Plug-in.

Stephen Cocks is a WebSphere ESB Architect in IBM Hursley and provided valuable information about the administration of WebSphere ESB.

Simon Holdsworth is a WebSphere ESB Architect in IBM Hursley and provided us with useful background about using WebSphere ESB to connect to http.

Chris Markes is a WebSphere ESB Developer in IBM Hursley and helped us with information about mediation flows.

Phil Norton is a WebSphere Messaging Developer in IBM Hursley and helped us from his extensive experience with WebSphere ESB.

Ian Parkinson is a WebSphere ESB Software Engineer in IBM Hursley and provided us with additional information about WebSphere ESB data bindings.

Chris Vavra is the WebSphere ESB Product Manager and is based at IBM Burlingame. Chris sponsored this IBM Redbooks publication, and helped to define its scope and contents.

I would also like to thank **Deanna Polm** and **Sangam Racherla** of the ITSO location team in San Jose, California, who hosted our residency so efficiently.

Become a published author

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

Your efforts will help increase product acceptance and customer satisfaction. As a bonus, you will 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

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 book form found at:

  ibm.com/redbooks

- Send your comments in an email to:

  redbooks@us.ibm.com

- Mail your comments to:

  IBM Corporation, International Technical Support Organization
  Dept. HYTD Mail Station P099
  2455 South Road
  Poughkeepsie, NY 12601-5400

Part 1

Background

In Part 1 we discuss a number of topics as background to considering how to use the WebSphere Enterprise Service Bus to connect enterprise applications.

Chapter 1, “Connecting enterprise applications” on page 3, is a brief history of Enterprise Application Integration. In it we discuss different patterns of integration and how they have been influenced by technology and the requirements placed on IT by the business environment. The On Demand business is fuelled by the pace of globalization and the competitive opportunities presented by the Internet. Service-oriented architecture (SOA) has evolved from the need for an IT architecture that responds to the rapidly changing global business environment and integrates companies’ existing investment in IT with services available over the Internet. The Enterprise Service Bus (ESB) is a fundamental component of the SOA reference architecture linking existing applications and new services together to appear as a seamless service bus.

Chapter 2, “Service Component Architecture” on page 51, describes the Service Component Architecture (SCA) and its realization in the WebSphere Enterprise Service Bus. SCA is an open IT SOA architecture supported by a number of vendors, including IBM. SCA differs from earlier interoperability architectures such as CORBA or Web services by describing how interfaces are wired together in an abstract way in terms of Quality of Service and interaction style. Using the SCA tooling provided in WebSphere Integration Developer, an integration specialist constructs an integration architecture by wiring SCA components together and describing the business objects that flow between application.

Chapter 3, “Connecting to the WebSphere Enterprise Service Bus” on page 63, describes the architecture of the WebSphere Enterprise Service Bus, how it implements the Service Component Architecture, and the mediation component that is special to the WebSphere Enterprise Service Bus.

Chapter 4, “Adapters” on page 107, defines the concept of an adapter, describes the different styles of adapter that are provided for the WebSphere family of products, and introduces the new WebSphere Adapters that are based on Java Connector Architecture 1.5 and SCA.

1

Connecting enterprise applications

This chapter provides an overview of the various options available for connecting enterprise applications. It also provides a brief progression of Enterprise Application Integration (EAI) technologies.

1.1 Introduction to integration

At its highest level, integration can be divided into two essentially different approaches: process-focused integration and data-focused integration. For a broad discussion of different application integration patterns, see the Redpaper *Application Integration Patterns,* REDP3837*.*

1.1.1 Process-focused integration

Process-focused integration is the integration of the functional flow of processing between applications. The objective of process-focused integration is to produce a consolidated view of a business entity, such as a customer, by integrating multiple business systems together into a single functional whole. Integrating applications using the Enterprise Service Bus (ESB), which is the focus of this book, is one of the integration patterns that falls into this category. This is what is traditionally referred to as Enterprise Application Integration (EAI).

EAI refers to message-based, transaction-oriented, application-to-application integration using either a point- to-point or a point-to-hub (brokering or bus) integration pattern. The core benefits offered by Enterprise Application Integration are a focus on:

- Integrating both business processes and data
- Reusing and distributing business processes and data
- Simplifying application consolidation by integrating above the level of individual applications and so reducing the amount of detailed application-specific knowledge required by integration specialists

IBM product portfolio for EAI includes WebSphere MQ, WebSphere Adapters, WebSphere Message Broker, WebSphere Process Server, and WebSphere Enterprise Service Bus (WebSphere ESB), amongst others. A complete list of the twenty to thirty middleware products in the IBM Business integration portfolio can be found at:

http://www-306.ibm.com/software/info1/websphere/index.jsp?tab=products/businessint

1.1.2 Data-focused integration

Data-focused integration is the sharing of information or data that is used by applications. Data integration can be further subdivided into expose, transform, and load (ETL) and Enterprise Information Integration (EII).

Expose, transform, and load (ETL)

Expose-transform-load refers to the three-step process of extracting data from various outside sources (*extract*), transforming it to fit the business needs of the enterprise (*transform*), and finally loading it into the data warehouse (*load*). The last step might only consist of loading it into any database rather than a full-blown data warehouse. It is designed to process very large amounts of data.

The first step of extraction of data may include extracting data from various sources. These typically include *flat files* and *relational databases* but may also include non relational data structures such as IBM Information Management System (IMS™), or other data structures such as Indexed Sequential Access Method (ISAM), a method for storing data for fast retrieval that forms the basic data store of almost all relational and non-relational databases), or Virtual Storage Access Method (VSAM), a record-oriented disk file storage scheme that was first used in the OS/VS2 operating system, and later used throughout the Multiple Virtual Storage (MVS™) architecture and now in z/OS®.

The transformation phase consists of applying a set of rules to the extracted data so that it fits the business needs. Some examples of these rules include applying mathematical data transformations (converting hours in a column to minutes, for example) or loading only a subset of the available data.

Enterprise Information Integration

Enterprise Information Integration refers to an optimized and transparent data access and a transformation layer providing a single relational interface across all enterprise data. It enables the integration of structured and unstructured data to:

- Provide real-time read and write access.
- Transform data for business analysis and data interchange.
- Manage data placement for performance, currency, and availability.

IBM WebSphere Federation Server is one product that fits into this category. It enables applications to access and integrate diverse data and content sources as though they were a single source (regardless of where the information resides) while retaining the autonomy and integrity of the data and content sources. WebSphere Federation Server offers an integrated view of a wide range of heterogeneous data and content sources.

IBM WebSphere Transformation Extender (WTX, formerly known as WebSphere DataStage™ TX) is another product for enterprise information integration. Originally targeted at ETL scenarios providing batch load and transform, it is now also integrated as a data dictionary, data mapping product with process focussed integration products such as IBM DataPower®, WebSphere Message Broker, and WebSphere ESB. It enables an enterprise-wide data dictionary and data mapping capability.

Primarily, WTX provides a unique graphical user environment that allows integration designers to visualize complex data types in graphical form and provide powerful data processing and manipulation capabilities. It helps consistent data transformation and validation across any infrastructure and results in faster application deployment with reduced development and maintenance costs.

Figure 1-1 compares the three integration approaches.

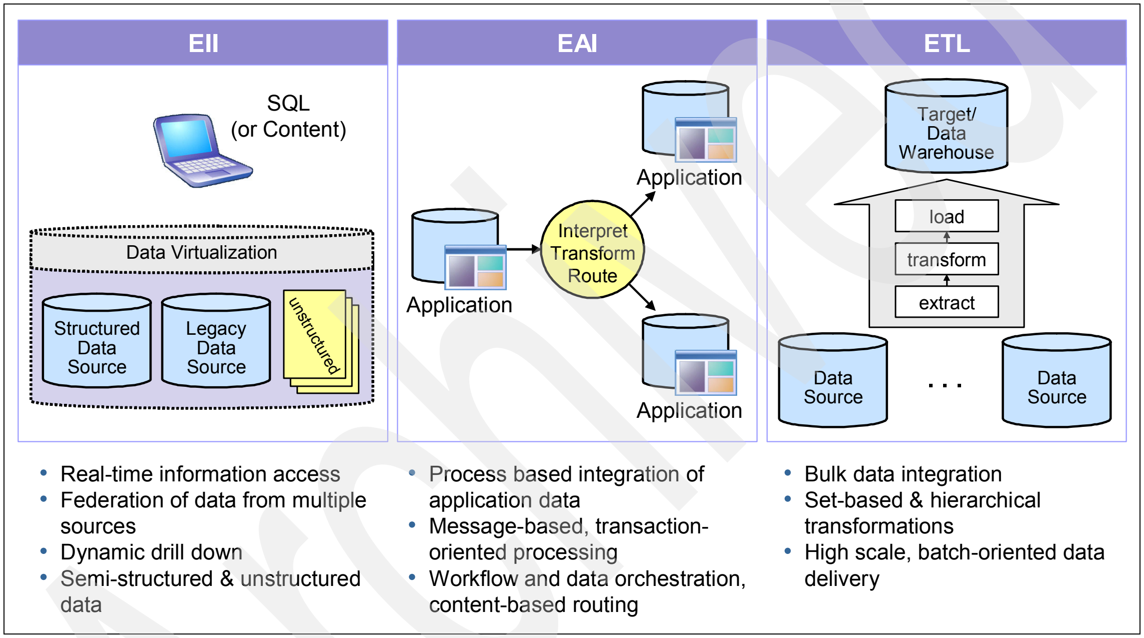


Figure 1-1 Comparison of various integration approaches

1.1.3 Which pattern of application integration

When should you use EII over EAI, or ETL over EII? Here are some basic guidelines.

EII - Enterprise Information Integration

EII is indicated when:

- Connecting a large repository with selected data from other sources
- Extending existing well-designed Enterprise Data Warehousing solutions
- Source data exhibits the following properties:
  - Volatility is high.
  - Selectivity is granular.
  - Connectivity is reliable.
  - Service levels are compatible.
  - Transformations are minimal and can be expressed as SQL.

Enterprise Application Integration (EAI)

EAI is the architecture of choice for solutions involving:

- Integration of transactions and not large data sets
- Simple joins of small amounts of data
- Data sources that are not directly accessible as relational tables

Expose, transform, and load (ETL)

ETL is optimal for integration when you require:

- Data consolidation
- Complex transformations

Typically a combination of the above three approaches is used.

1.2 EAI patterns

The remainder of this chapter and this book focus only on process-based integration. The Patterns for e-Business list four styles of EAI interaction, which are shown in Figure 1-2.

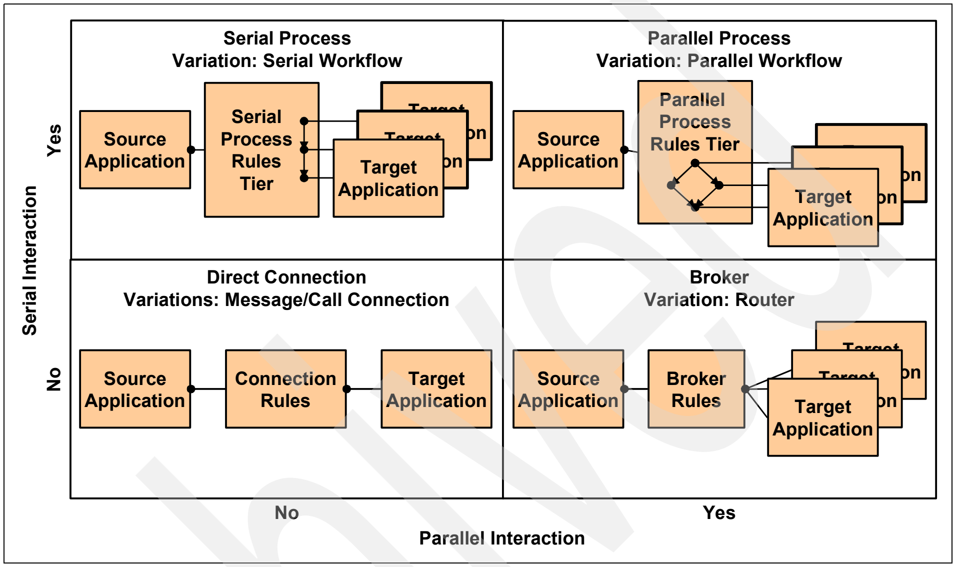


Figure 1-2 Patterns for process-focused integration

The interaction patterns are described below.

1.2.1 Direct connection or point-to-point integration

This pattern is the simplest interaction style that is based on a one-to-one topology, enabling a pair of applications to talk directly to each other. The variations of this pattern are *message connection* and *call connection*, which apply to *one-way request* and *request/reply* interactions, respectively. Both variations may be used either with *synchronous* or *asynchronous* communication protocols. However, there are preferences for a specific protocol type depending on the variation. For example, the call connection variation has a more natural fit with synchronous protocols, while the message connection variation favors asynchronous protocols.

1.2.2 Broker or hub and spoke integration

This pattern is based on a one-to-N topology. The topology separates distribution rules from the applications and enables a single interaction from the source application to be distributed to multiple target applications concurrently.

The router variation applies to solutions where the source application initiates an interaction that is forwarded to, at most, one of multiple target applications, and the publish/subscribe variation describes one or more sources and targets, typically with a broker forwarding publications to targets based on the publication category or content.

1.2.3 Serial Process

This pattern extends the one-to-N topology of the Broker pattern by facilitating sequential execution of business services hosted by target applications based on a defined set of process rules. A workflow variation extends these capabilities by supporting human interaction to complete specific process steps.

1.2.4 Parallel Process

This pattern extends the one-to-N topology of the Broker pattern by facilitating parallel (concurrent) execution of processes. A Workflow variation extends these capabilities by supporting human interaction to complete specific process steps.

You can find more details about these patterns at:

http://www.ibm.com/developerworks/patterns

You can also find more information in the IBM Redbooks publication *Patterns: SOA with an Enterprise Service Bus in WebSphere Application Server V6,* SG24-6494, found at:

http://www.redbooks.ibm.com/abstracts/sg246494.html

Advantages of a hub and spoke (or broker) integration pattern over a point-to-point integration pattern can be seen in the simplification of the patterns of interactions shown Figure 1-3.

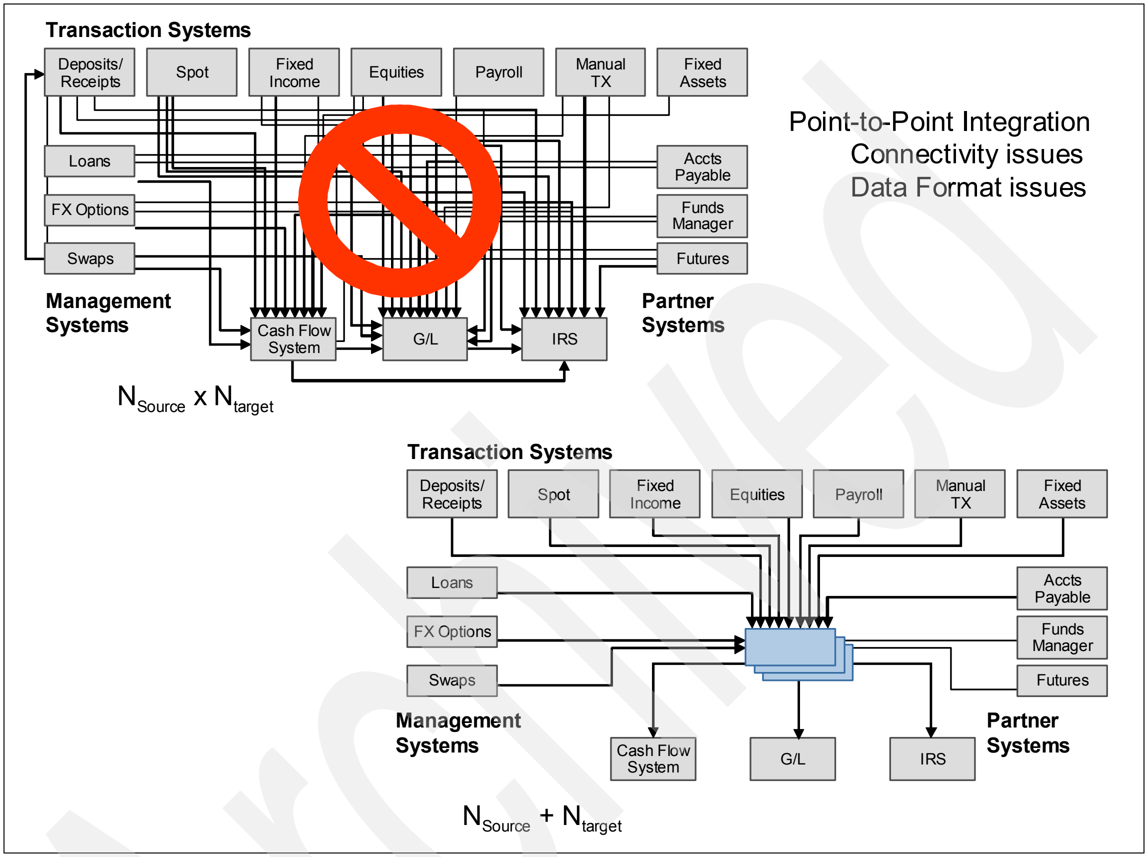


Figure 1-3 Advantages of a hub and spoke (broker) pattern over point-to-point integration

1.3 Evolution of Enterprise Application Integration (EAI)

This section describes the evolution of Enterprise Application Integration from the tight affinity of software and hardware in the earliest computers to the world of virtualized hardware today.

1.3.1 Incompatible architectures

It seems inconceivable to new students of computer science that before the mid 1960s, with every new generation of computers all applications had to be rewritten or were condemned to run in emulation mode, where available.

It was IBM S/360™ computing architecture that — for the first time — gave customers the realistic expectation that at least future hardware from the same manufacturer would continue to run today's applications. This was achieved by separating the hardware architecture, essentially the instruction set, which was enshrined, for S/360, in the Principles of Operation (POP), from the hardware implementing the instructions.

Of course, with multiple manufacturers, and with multiple layers of software with complex interdependencies, the picture is a lot more complex, and the industry, while understanding *write once, run anywhere*, is far from achieving it. The scale of the problem of incompatible architectures is vastly larger today than it was in the 1950s and 1960s, simply because of the scale of the industry.

The industry has set its sights, and is getting closer, to a second, more attainable goal, of *write once, use anywhere*.

1.3.2 Batch-oriented applications written in silos

Historically, applications were written to solve specific, well-defined problems, such as payroll or inventory management, with a clearly defined return on investment compared to the preceding manual system the computer often replaced. People were excited and satisfied enough that a computer solved the problem. Each investment justified massive expenditure and made plentiful returns. The results were single monolithic applications that solved one problem at a time and solved them well.

There was no vision at that time of how information technology (IT) could be applied to much broader business problems. After all, the computer network had scarcely been invented, and input and output was carried by hand to and from computers as boxes of cards, or reels of computer tape. As a result, solutions evolved on a great variety of platforms safely separated by physical distance and connected by the physical media carried between them.

With the advent of computer networks, batch applications could be made to work more efficiently by substituting file transfer over networks for physical media. For example, a sales tax calculation application in an enterprise would get developed and used in a variety of departments, as shown in Figure 1-4.

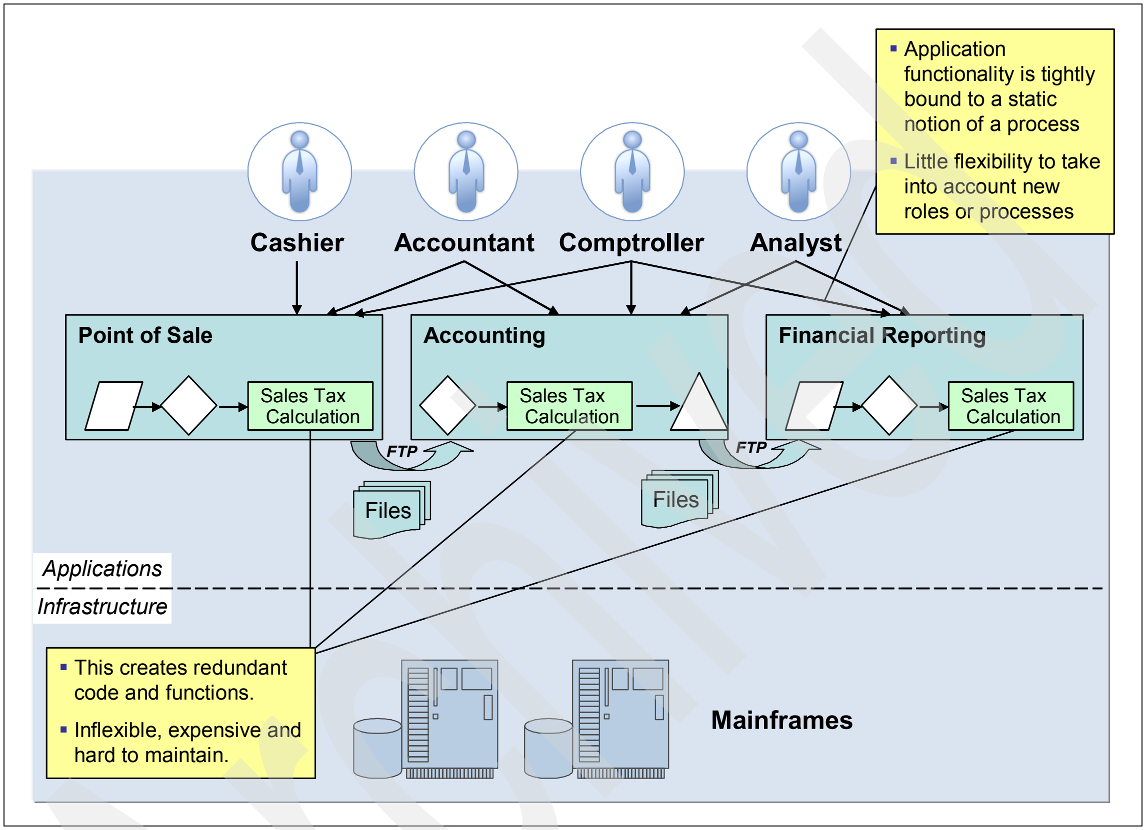


Figure 1-4 Application development in silos and integration through file sharing

Over time, business requirements grew and new functionalities were added to existing applications, making them even bigger monolithic applications. This continued till the point it started affecting their performance and they were no longer able to solve the single problem that they were supposed to solve in an efficient way. At that point, IT personnel started considering peeling layers of functionality from the monolithic application to make them separate applications in themselves and then connecting them together using Job Control Language or data pipes of some kind. This gave rise to the early days of application integration.

If and when high performance was needed, it was achieved by hosting the applications on the same system. This was not a big restriction, as most applications at that time were batch oriented and large central computers (mainframes) were the accepted technology standard.

1.3.3 Data sharing through files

Data was moved between systems to the applications that required it. This included physical movement of data such as delivery of tapes or using file transfer protocols over the network such as FTP.

Whenever a business identified a need for information to be shared across its computing platforms, it had to use the networking capabilities of the day, which were anything but user friendly. Protocols on all levels were proprietary, often complex, and usually not well understood, especially when it came to cross-platform implementations. Files remained the favorite entities to share, for a number of reasons:

- They were well understood and had worked well between applications on the same system.
- Support was available for cross-platform file transfers and file sharing on network servers.
- Above all, most applications were still batch oriented.

Where online processing had been introduced, businesses found it more acceptable from a risk and system capacity perspective to just collect data during the online day (in files) and do the actual processing during nightly batch runs. This mode of operation is still quite prevalent in businesses today.

1.3.4 Near real-time communication - Client Server Systems

With the development of departmental computers (a response to both the falling cost of hardware and the increasing cost of changing those ever-growing monolithic applications in the data center), a need for real-time (or near real-time) communication between two applications on disparate platforms emerged. This led to the development of initial client server systems, where clients would access applications hosted on the server over a network. This brought a number of problems with it:

- Technology choices had to be made on all levels of network protocols — on both platforms. There was no widely accepted networking architecture.
- Communications functions had to be added to the applications. This involved highly specialized programming skills, often quite different on each of the platforms.
- The APIs were complex at best, and lots of exception handling and recovery logic was required in the code if there was to be any degree of success.

In a nutshell, network applications were exceedingly expensive to build, and so client server solutions tended to use proprietary client-server frameworks that solved the network problems in only a limited way, but delivered immediate business value. By not solving the network problem, such applications could only deliver an additive, rather than multiplicative (or network effect) rate of return as more solutions were developed.

Initial client server systems were two-tier systems that gave way to multi-tier or 3-tier client server systems (refer to Figure 1-5 on page 15).

1.3.5 Point-to-point integration with Remote Procedure Calls (RPCs)

A Remote Procedure Call is a technology that allows one program to call or invoke a function in a program located in another computer in a network without the application developer having to understand network details. (A procedure call is also sometimes known as a method innovation or function call or a subroutine call.)

RPC as a general concept dates as far back as 1976 and it still forms the underlying principle behind most EAI technologies. RPC is a popular paradigm for implementing the *client/server model* of *distributed computing*. The requesting program is a client and the service-providing program is the server.

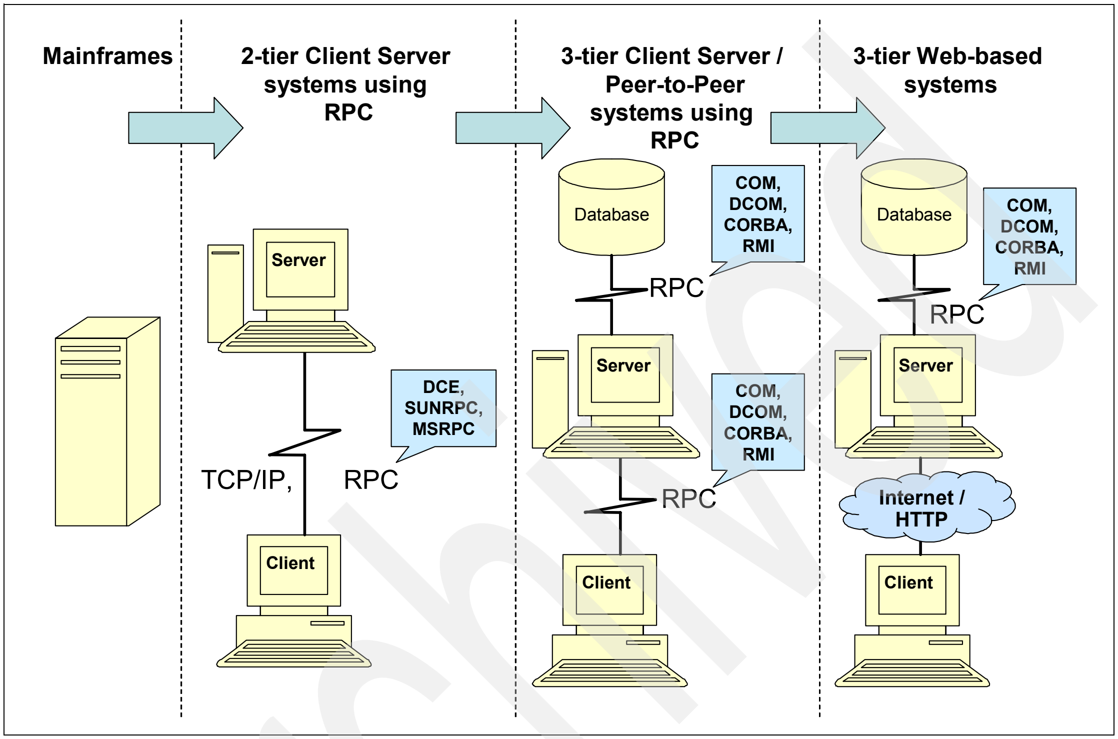


Figure 1-5 Evolution of client server systems using RPC mechanisms

Like a regular or local procedure call, an RPC is a synchronous operation requiring the requesting program to be suspended until the results of the remote procedure are returned.

When program statements that use RPC are compiled into an executable program, a stub is included in the compiled code that acts as the representative of the remote procedure code. The stub is frequently known as the *proxy*.

Early implementation of RPC included *Sun™ RPC* and *DCE/RPC* in Distributed Computing Environment (DCE) (in the 1980s). In the mid 1990s Microsoft® used DCE/RPC as the basis of their Microsoft RPC (*MSRPC*), and implemented *DCOM* on top of it. At the same time, Object Management Group came up with CORBA. Remote Method Invocation (RMI) is another RPC mechanism for connecting two remote Java objects together.

When RPC is used in an object-oriented world, the objects are referred to as distributed components. A progression of client server systems using various RPC mechanisms is shown in Figure 1-5 on page 15.

All RPC approaches (sometimes, confusingly, referred to as Message Passing — a procedure call is essentially sending a message) are essentially point-to-point integration approaches. This brings us to Message Oriented Middleware (MOM).

1.3.6 Message Oriented Middleware

Most RPC approaches require the program on the destination computer to be up and running for the remote procedure call to succeed. This results in very tightly coupled systems. As the number of connected IT systems participating in an integration solution system grows, the requirement that each destination program (or even the destination computer) is available to respond to the remote procedure becomes increasingly problematic (Figure 1-6). For the integrated solution to complete successfully, one either has to increase the availability of component applications to 99.9% (downtime ~ 10 minutes a week), or else be able to tolerate the unavailability of component applications.

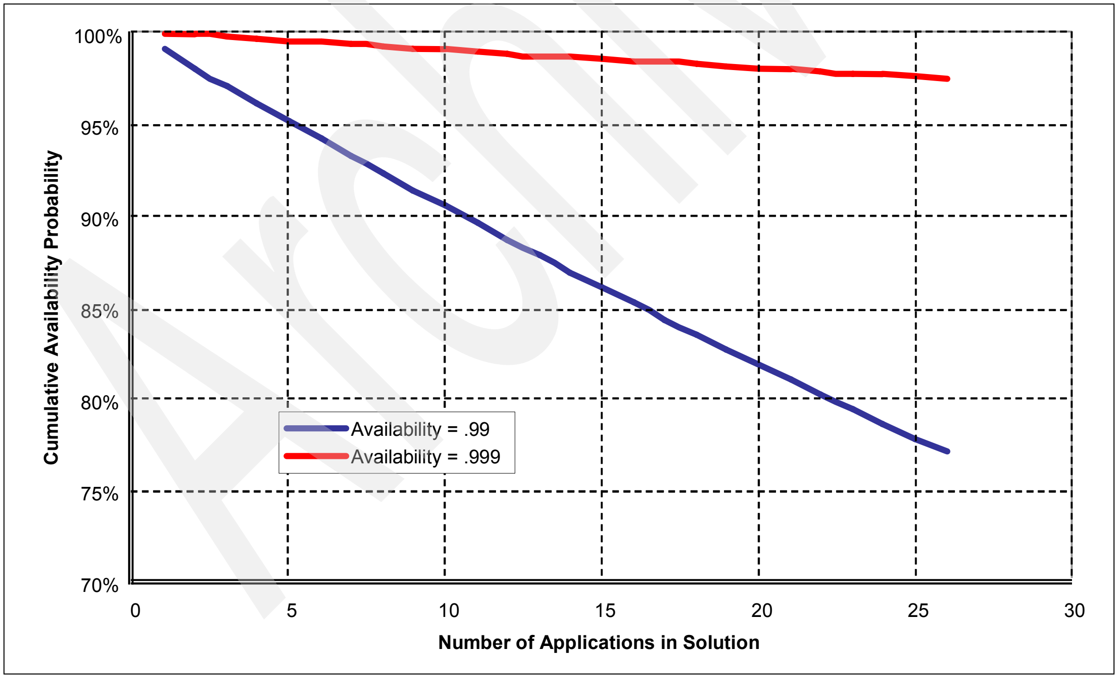


Figure 1-6 Probability solution is available given .99 and .999 availability of component applications

This is where messaging middleware comes in, notably MQSeries (now known as WebSphere MQ). Asynchronous communication via queues allows for loose coupling of systems and toleration of unavailability (if queues are kept safe on disks) without compromising the ability to complete the solution or the performance of the solution[1] (Figure 1-7).

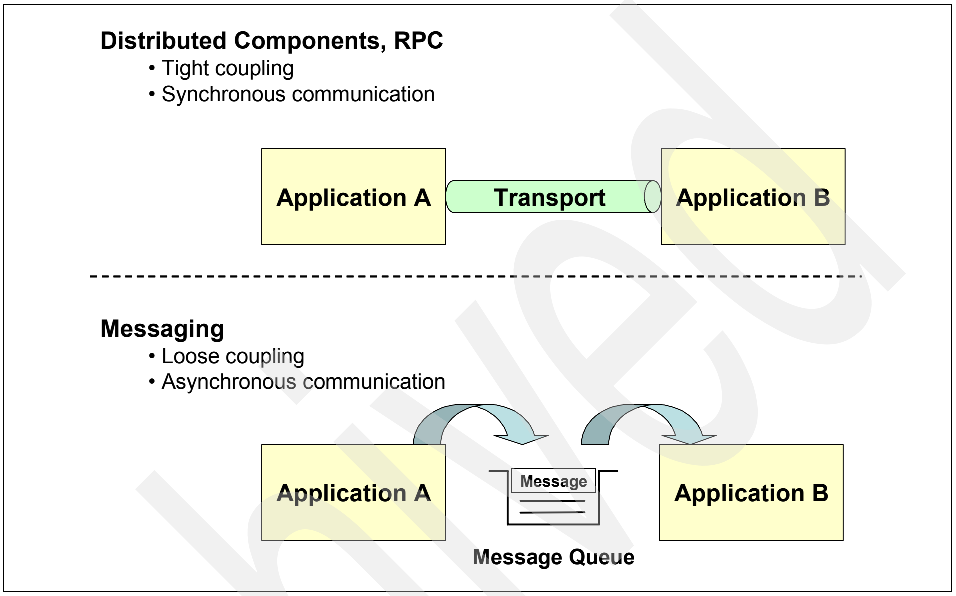


Figure 1-7 Loose coupling in messaging systems

By providing a simple, easy-to-use API that hides network error handling, in addition to hiding network complexity, from application programmers, MQSeries makes it much easier and faster to write communication functions into application programs. Furthermore, since it is available on so many different platforms, it makes cross-platform communications much easier as well.

Transactional capabilities allow users to build robust messaging applications that ensure data integrity across systems. For many classes of application it removes the necessity to lock resources simultaneously on multiple systems, with significant performance and capacity gains. In addition, message queuing allows for easy transition between the batch-oriented processing model of the past and event-driven, real-time processing, which is very much a requirement in today's world of e-business. This is because the queues serve as buffers between new online systems and old batch systems that are eventually replaced by online back-end systems sooner or later. Well-defined message interfaces also facilitate this.

[1] Comparing the performance of MOM versus RPC systems is like comparing apples and oranges. Good or bad performance, however you define it, is possible with either paradigm.

1.3.7 Adapters

Point-to-point integration techniques involve development of components that convert one data format (or protocol) to another data format (or protocol). This clearly underlines the need for cleanly defined interfaces and data formats. The requirement is not really new, since the use of databases and flat files in batch processing also requires agreement between applications on what data is placed where with what meaning.

A messaging system increases the magnitude of the demand for clearly defined interfaces. The feasibility of connecting pairs of applications using the messaging platform leads to more requirements for interoperability as the benefits of integration come within reach. To integrate disparate applications with different interfaces using the message transport, work needs to be done to map from one interface to the other. The demand for cheaper ways to integrate disparate applications led to the development of adapters.

The use of messaging encouraged the development of common messaging formats to allow disparate systems to talk to each other. This requires all the disparate systems to communicate either in the same data format (and protocol) as the messaging middleware or through some component that would convert the native data format (and protocol) of a system to the messaging middleware data format (and protocol). These components are known as half adapters (shown in Figure 1-8). They convert native data format (or protocol) to that of the messaging system and not of the target destination system. The advantage of half adapters is that they reduce the number of unique adapters in the solution to the sum (source + target), rather than the product (source x target).

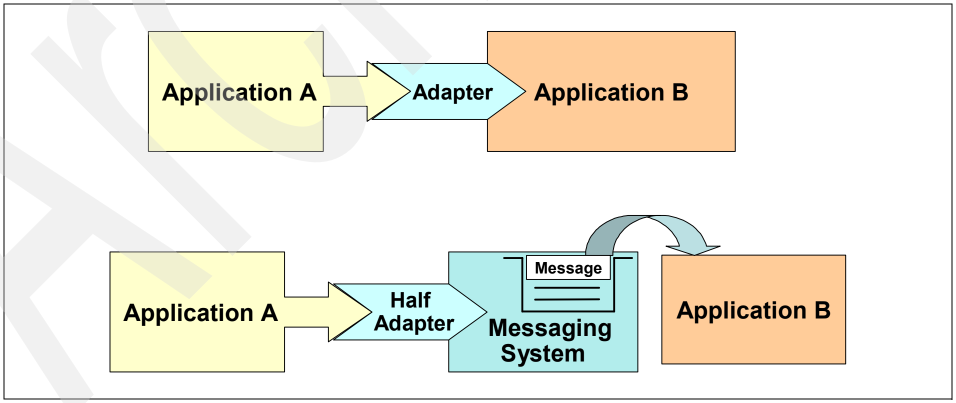


Figure 1-8 Adapters and half adapters

Originally, any applications that wanted to use WebSphere MQ would need to do so via the provided API, the *MQI*. In many instances, however, this was not possible, as in the case of off-the-shelf packages, or it was difficult, as with some legacy applications. To help in these situations, a series of components were developed that are now called bridges, adapters, or connectors. They all provide a WebSphere MQ interface for specific applications or groups of applications in a noninvasive fashion. The bridges for CICS and IMS are examples of this, which allow an WebSphere MQ enabled application to start transactions in these systems and receive results back.

Adapters use a number of techniques to interface with packages, such as database or flat file interfaces, terminal emulation, or package-specific APIs. They typically provide and expect messages in specific formats based on the application's capabilities and requirements, although some implementations provide a degree of message transformation.

Adapters can be categorized as application adapters or technology adapters. Application adapters are used for connecting to specific applications such as PeopleSoft®, SAP, or Siebel®, whereas technology adapters are used for a specific technology for interfacing with a backend system. For example, a JDBC™ adapter could be used for interacting with DB2® or with Oracle® databases.

1.3.8 Message Brokering

At this point of the evolution, there is still a need for application programming, if only to route messages between packaged applications and to transform them according to the applications' requirements. The promise of EAI is that ultimately any application should be able to communicate with any other application, as long as there is some meaning in such a communication. It should then also be possible to re-configure such *networks of applications* to form innovative solutions without changing the applications themselves.

Message brokering technology is a response to the challenge to build new integration applications without reprogramming existing applications. Brokers sit on top of the common messaging protocol enabled by the messaging half adapters. The common protocol exposes data traffic to and from applications at a common presentation layer. It is possible to build integration applications based on the data traffic exposed in the presentation layer. Message brokers are integration servers that are located centrally between applications, for example, in a hub-and-spoke topology. This is shown in Figure 1-9.

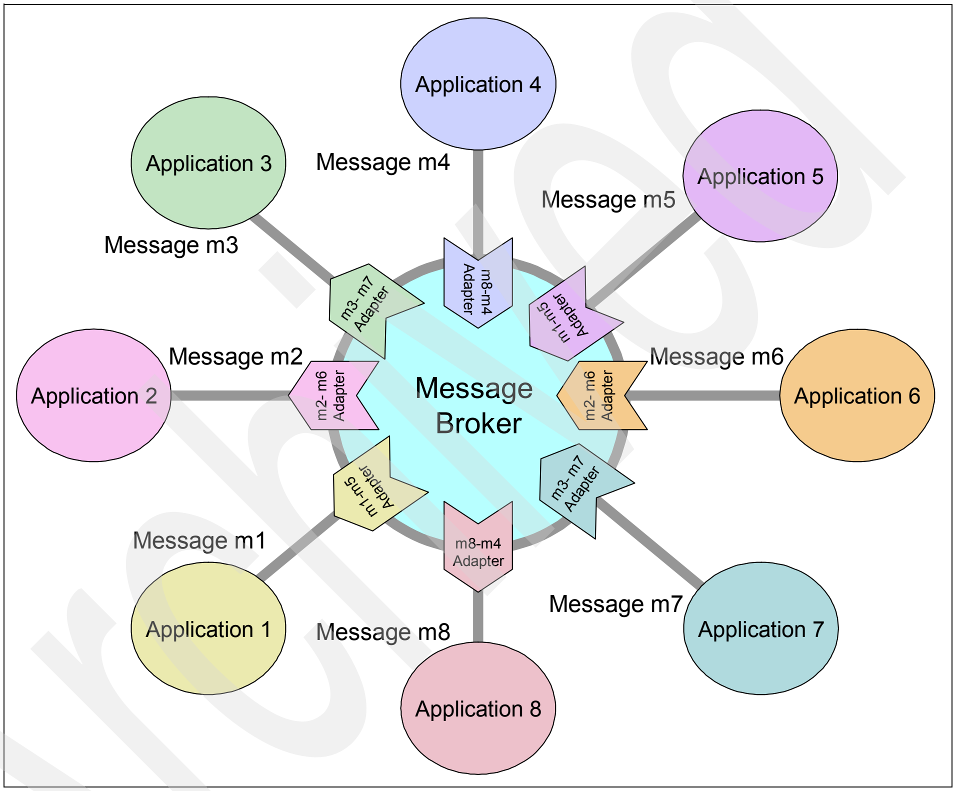


Figure 1-9 Message brokering between various message formats using adapters

Let us look at an example. Two applications (A and B) have been communicating via MQSeries for a while. They used a particular message interface that was agreed upon between their developers. A new packaged application (C) is introduced and it is determined that the messages exchanged by A and B could be used as data feed into C, but the message formats are not quite correct. Since C is a package and it cannot be modified, we now face modifying both A and B. If either of them cannot be changed for whatever reason (it could be a package itself, or be an older application with no maintenance skills available, or the owners feel a change would be too risky), the project stops right there. Even if there are no inhibitors, changes like this can be very costly and time consuming. Our example only features three applications, but imagine a scenario where something as central as a general ledger needs to be replaced. You might be looking at hundreds of interfaces.

So it would be very advantageous to have a central hub where messages can be reformatted and rerouted according to easy-to-set-up rules that are quickly readjusted to changing requirements. The hub is called a message broker. IBM premier message broker product is IBM WebSphere Message Broker.

1.3.9 Data synchronization

Message Oriented Middleware led to the development of another set of products patterned on processing important messages, referred to as *events* in near real time. Detecting events and using them to drive sequences of related business activities is the core function of IBM WebSphere InterChange Server. It integrates business applications by means of collaborations, which contain the business logic and rules for processing the business events. Interfacing to the applications is achieved by means of connectors, which are very similar in function to, and often interchangeable with, the adapters developed initially for WebSphere MQ.

The centerpiece of the IBM WebSphere InterChange Server data model is a set of generic business objects (GBOs) that are used to carry event-related information between the applications. Standard collaborations are to do tasks such as synchronizing customer name and address files use GBOs, and are provided with IBM WebSphere InterChange Server. Mapping facilities are provided between these generic business objects and application-specific data objects (ASBOs).

Again, the driving force is the half adapter pattern: reduce an S x T sized problem to an S + T size problem, but this time the savings could be even greater: if C is the number of different collaborations, then the problem, in theory, should scale down from C x S x T to C + S + T. In practice the effort will not scale down that greatly, as mapping to a GBO is rarely perfect, and testing and some tweaks may be needed for each collaboration implemented for each S x T combination.

Much of the functionality provided by the IBM WebSphere InterChange Server, and more, is now found in WebSphere Process Server.

1.3.10 Business process management

With a comprehensive set of adapters to help integrate all applications of the enterprise, a message broker that makes it easy to model the information flow between applications, and event driven collaborations to control the data flows that take place, Enterprise Application Integration is still not complete. The flow of data through an enterprise (or between separate business entities, when it comes to B2B) needs to be managed according to certain business process rules for a complete Enterprise Application Integration solution.

A classic example is a customer service situation. A customer service request can arrive via a number of different channels, such as a telephone call, voice message, fax, e-mail, or even a traditional letter. Any such triggering event must be captured, categorized, and sent on its way through the organization along a predetermined path, while being monitored by a management system that ensures that a resolution (and customer satisfaction) is achieved within certain time frames and that any slippage is detected, escalated, and remedied according to specific service level parameters.

This is called Business Process Management (BPM), and IBM offers WebSphere MQ Workflow and WebSphere Process Server for this function. It allows you to model a business process step-by-step by integrating applications that can either perform their functions unattended in the background or interactively in a dialog with a human operator. Both WebSphere MQ Workflow and WebSphere Process Server are equipped to drive and manage the process flow between steps, to interface with the applications, and to assign work to the staff members according to an organizational profile.

One common denominator of a message-oriented processing environment — and a differentiator from older batch-oriented systems — is that it supports event-driven processing in real time (or near real time, where appropriate). There are many business situations where that capability adds great value (straight-through processing, same day value, online banking, or ordering over the Internet, for example).

1.3.11 Impact of the Internet and emergence of Web services

With the emergence of the Internet as a business channel, many applications need to be made available over the Internet in some form or the other. This has led to the development of Web services. Web services are self-contained, applications that can be described, published, located, and invoked over networks. Web services encapsulate business functions, ranging from simple queries to full business process interactions. The services can be new or wrap around existing applications.

The following are the core technologies used for Web services:

- Extensible Markup Language (XML) is the markup language that underlies most of the specifications used for Web services. XML is a generic language that can be used to describe any kind of content in a structured way, separated from its presentation to a specific device.

- SOAP (originally an acronym for Simple Object Access Protocol) is an application-to-application protocol that isolates network, transport, programming language, and platform differences to allow a client to call a remote service without knowledge of how the service is coded or where the service is hosted. The message format is XML.
- Web Services Description Language (WSDL) is an XML-based interface and interface description language. The service provider uses a WSDL document in order to specify the operations a Web service provides, the parameters and data types of these operations and the location and transport bindings to be used to access the service.
- Web Services Inspection Language (WSIL) is an XML-based specification about how to locate Web services without the necessity of using UDDI. However, WSIL can be also used together with UDDI, that is, it is orthogonal to UDDI and does not replace it.
- 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.

The evolution and wide acceptance of Web services on the back of the Internet has taken Enterprise Application Integration to a new level. It has led to new architectures like service-oriented architecture (SOA) and patterns like Enterprise Service Bus (ESB). These have helped to overcome some of the practical problems associated with using Web services. We discuss SOA and ESB in more detail in the next subsection.

1.3.12 Web 2.0

Web 2.0 is too new to characterize confidently, and its impact on Enterprise Application Integration is yet to be seen. However, it does have features that impinge directly on application integration (see 1.6, “Web 2.0 - next generation Web services” on page 43).

What is Web 2.0? Web 2.0 is a loose constellation of innovations that have emerged from the existence of the Internet. Web 1.0 might be described as harnessing the Internet to perform tasks that were already being performed without the Internet, but perhaps performed less well, less often, or less cheaply. Web 2.0 will perhaps be described as exploiting the Internet to perform tasks that could not be conceived without the Internet. Par excellence, Web 2.0 applications exploit the *network effect* of the Internet — doing things that become more valuable the more people do them together. The best Web 2.0 applications will be those that get better the more that people use them.

The key to exploiting the network effects of the Web is to expose data and services in standardized ways that can easily be exploited, and those exploits, in turn, being exposed to be exploited even further.

One trend we observe is that data and services are being exposed on the Web using the http protocol in different ways — not just exposing applications as WS-I basic profile 1.1 compliant Web services, but using other protocols such as REST and JSON.

For applications connected to WebSphere ESB to participate in the Web 2.0 world, we need the ESB to connect using the simple http-based protocols. We include a description of the architecture and a sample in this Book. It is another example of the power of the half adapter pattern, that having written a half adapter between the ESB and another http protocol, that adapter can be used again and again.

Figure 1-10 summarizes the evolution of Enterprise Application Integration. The rest of this chapter describes the two leaf nodes (SOA and ESB) and Web 2.0 technologies.

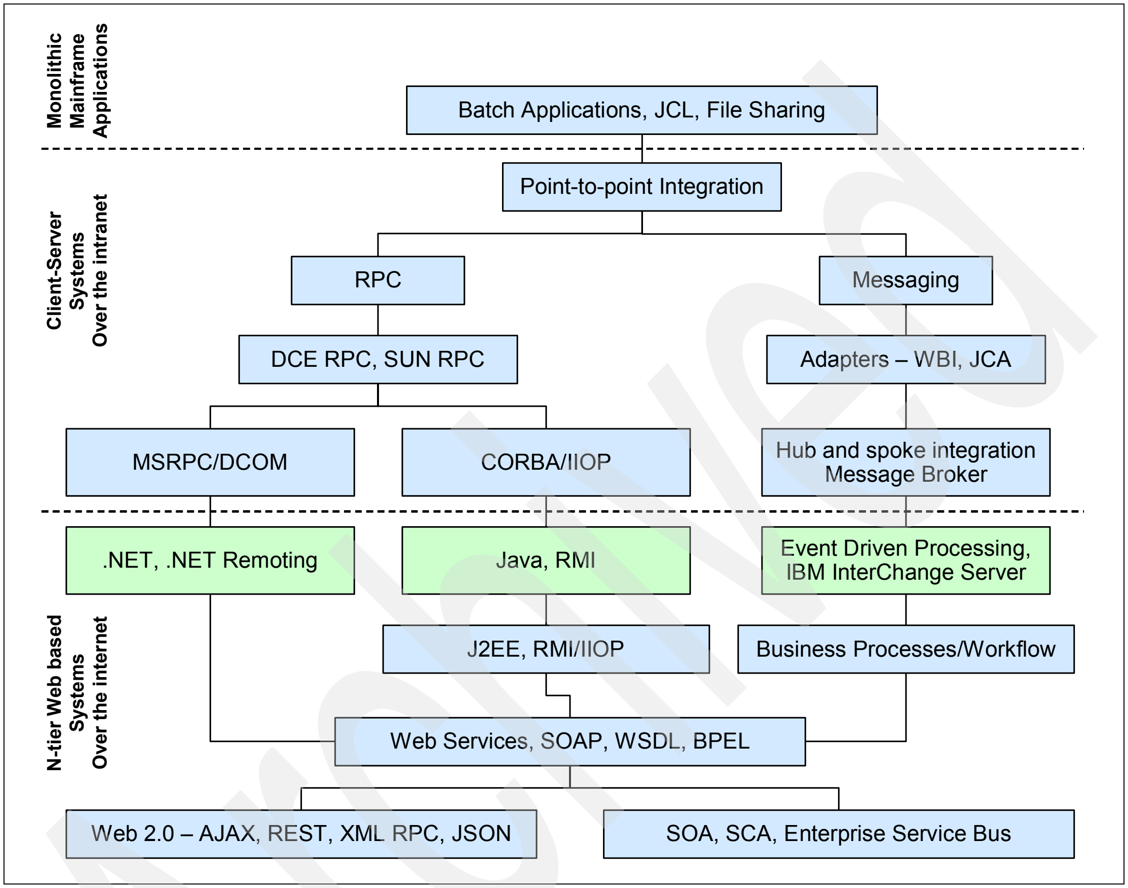


Figure 1-10 Evolution of Enterprise Application Integration

1.4 Service-oriented architecture (SOA)

SOA is an integration architecture approach based on the concept of a service. The business and infrastructure functions that are required to build distributed systems are provided as services that collectively, or individually, deliver application functionality to either end-user applications or other services. Applications collaborate by invoking each other's services. Services are composed into larger sequences to implement business processes. SOA requires a consistent mechanism for services to communicate. That mechanism should be loosely coupled and should support the use of explicit interfaces.

SOA brings the benefits of loose coupling from the messaging world and encapsulation from the object world to integration at an enterprise level. It applies successful concepts proved by Object Oriented development, Component Based Design, and Enterprise Application Integration technology to an architectural approach for IT system integration.

Services are the building blocks for SOA, providing function out of which distributed systems can be built.

1.4.1 Definition of a service

A service is defined as a discrete function that is offered to an external consumer. The consumer may be a Web page, another business function, or a collection of functions that together form a process.

There are many additional aspects to a service that must also be considered in the definition of a service within a SOA. The most commonly agreed-on aspects are:

- Services encapsulate reusable business function.
- Services are defined by explicit, implementation-independent interfaces.
- Services are defined in a way that stress location and transport transparency.

1.4.2 SOA and Web services

SOA is a concept that describes a service-based applications integration architecture. Web services are a specific set of standards and specifications that can be thought of as one method of implementing a limited SOA specifically for the Web. However, it is better to think of SOA and Web services as complementary, with Web services describing a standardized way to describe and invoke services, and SOA providing architectural patterns to address business, life cycle, and management concerns.

- Web Services provide:
  - A means to create well-documented interfaces for services using standards such as Web services Definition Language (WSDL) and document-style SOAP
  - A means to compose Web services using Business Process Execution Language for Web Services (WS-BPEL)
  - Simple communication mechanisms that are location-transparent and interoperable

- SOA provides:
  - Architectural patterns for realizing a business design through the implementation and composition of software services
  - A reference architecture model to decompose the functional components of an integration solution (Figure 1-16 on page 33)
  - A model for the life cycle and governance of services to help make reusability of services a reality

Working together, Web services and SOA have the potential to address many of the technical issues that are faced when trying to build an on demand environment.

Web services as an SOA on its own can at best be described at point-to-point SOA. You need much more than Web services to realize business functions as services in an enterprise.

1.4.3 SOA and EAI

SOA begins with the concept of a business as a mechanism that works to a design, even if the design is poorly documented. The information system design takes aspects of the business design and turns them into information processes to make a business work more efficiently and flexibly. The service reference model is the framework for SOA solutions and includes aspects such as:

- Exposing integrated applications as services
- Mixing and matching services to provide business processes
- Including human as well as computer tasks as services
- Modeling and managing business processes
- Monitoring business process performance

In contrast, EAI begins with the history of IT investment in applications and infrastructure and provides patterns and products to integrate the applications together to be able to deliver required new business function by reusing existing applications as far as possible.

In a sense SOA takes EAI to the new level where the integration patterns are a given, based on services, and the focus has shifted to aligning IT services more closely with the need to deliver value to fundamental business design by using IT. This is shown in Figure 1-11.

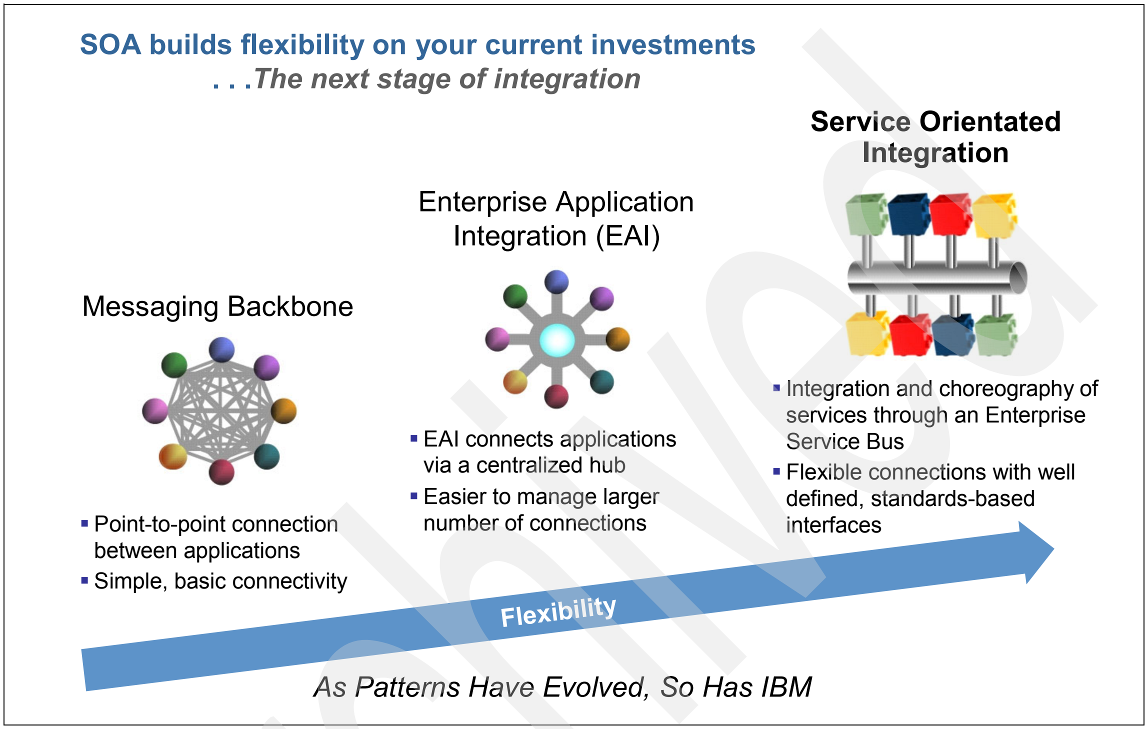


Figure 1-11 Evolution of application integration patterns

1.4.4 Enterprise Service Bus (ESB)

An Enterprise Service Bus is the latest stage in the evolution of EAI technologies. The Enterprise Service Bus is emerging as a middleware infrastructure component that supports the connectivity of SOA within an enterprise. An ESB enables all applications to participate in a SOA. The need for an ESB can be seen by considering how it can support the concepts of SOA connectivity by:

- Providing a consumer view of a service decoupled from the deployed implementation of the underlying application or service
- Decoupling technical aspects of service interactions
- Unifying access to applications using a common service model
- Providing dynamic access to services described in a service repository

This is achieved by replacing direct connections between service consumers and providers, with a bus architecture (Figure 1-12), and integrating this with the WebSphere Services Registry and Repository for service lookup.

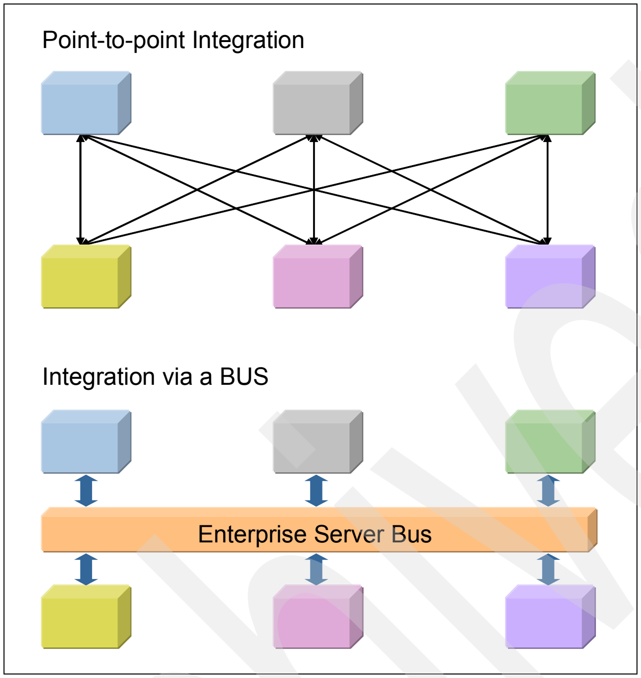


Figure 1-12 Overcoming the problems of point-to-point integration with ESB

An ESB brings several benefits to an SOA environment. These are highlighted in Figure 1-13 and Figure 1-14 on page 31.

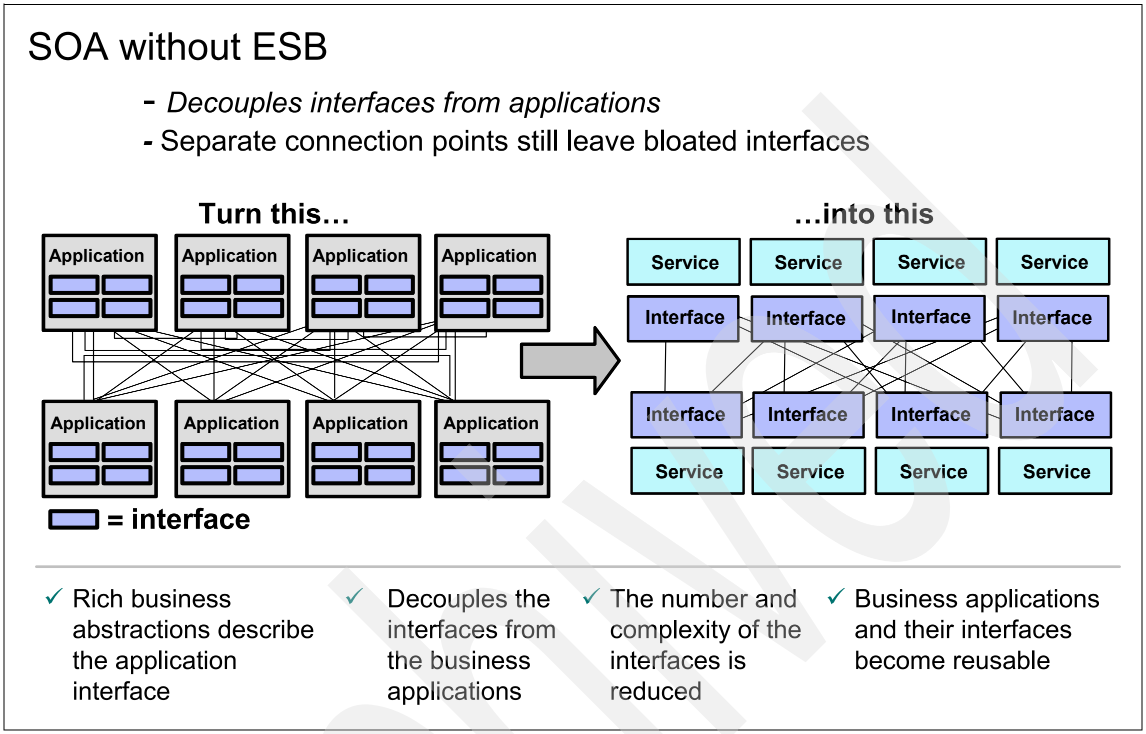


Figure 1-13 SOA without ESB

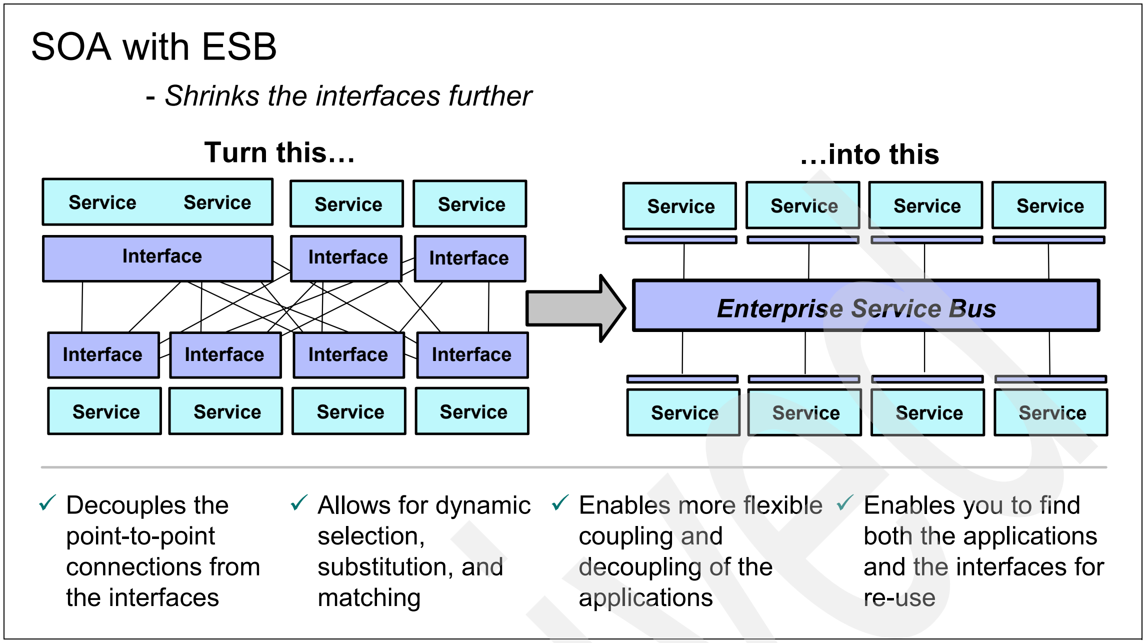


Figure 1-14 SOA with ESB

An ESB can be used to perform some of the following middleware functions:

- Map service requests from one protocol and address to another.
- Transform data formats.
- Support a variety of security and transactional models between service consumers and service providers and recognize that consumers and providers may support or require different models.
- Aggregate or disaggregate service requests and responses.
- Support communication protocols between multiple platforms with appropriate qualities of service.
- Provide messaging capabilities such as message correlation and publish/subscribe to support different messaging models such as events and asynchronous request/response.

In addition an ESB should:

- Support high volumes of individual interactions.
- Support more established integration styles, such as message-oriented and event-driven integration, to extend the reach of the SOA. The ESB should allow applications to be SOA enabled either directly or through the use of adapters.

- Support centralization of enterprise-level qualities of service and manageability requirements into the hub.

Figure 1-15 shows an enterprise service bus at a conceptual level.

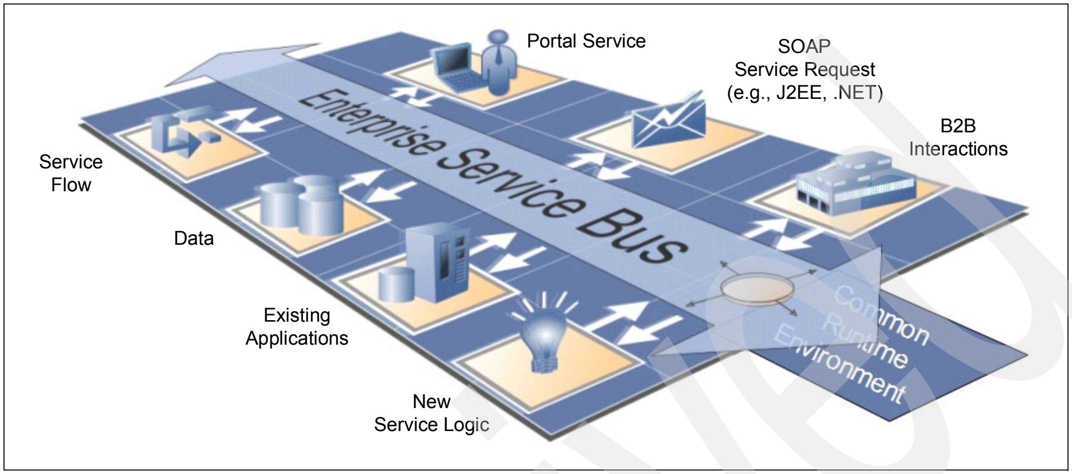


Figure 1-15 High-level view of an enterprise service bus

A detailed description of the WebSphere Enterprise Service Bus implementation of an ESB is given in Chapter 2, “Service Component Architecture” on page 51.

1.4.5 SOA reference architecture

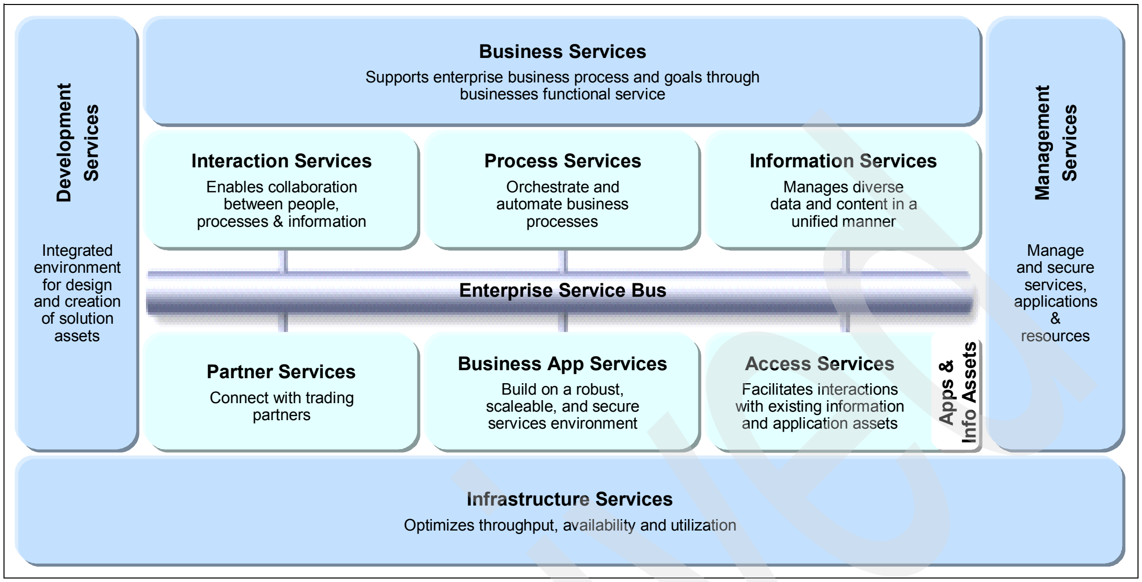


Figure 1-16 SOA reference architecture

The SOA reference architecture, shown in Figure 1-16, outlines the key capabilities that are required for comprehensive, enterprise-wide SOA solutions. These capabilities can be implemented on a build-as-you-go basis, allowing capabilities and project-level solutions to be easily added as new requirements are addressed over time. The SOA reference architecture consists of the following service categories:

- Development Services: Tools are an essential component of any comprehensive integration architecture. The SOA Architecture includes both Development Services, which are used to implement custom artifacts that leverage the infrastructure capabilities.
- Business Innovation and Optimization Services are used to monitor and manage the runtime implementations at both the IT and business process levels.
- Enterprise Service Bus: At the core of the SOA reference architecture is the Enterprise Service Bus. This delivers all of the connectivity capabilities required to leverage the services implemented across the entire architecture. Mediation services are all provided through the ESB.
- Interaction Services provide the capabilities required to deliver IT functions and data to end users, meeting the user's specific usage preferences.

- Process Services provide the control services required to manage the flow and interactions of multiple services in ways that implement business processes.
- Information Services provide the capabilities required to federate, replicate, and transform data sources that may be implemented in a variety of ways.
- Access Services: Existing enterprise applications and enterprise data are accessible from the ESB through a set of Access Services that provide the bridging capabilities between legacy applications and pre-packaged applications.
- Partner Services provide the document, protocol, and partner management capabilities required for business processes that involve interactions with outside partners and suppliers.
- Business Application Services provide runtime services required for new application components to be included in the integrated system.
- Infrastructure Services: Underlying all these capabilities of the SOA reference architecture is a set of Infrastructure Services that are used to optimize throughput, availability, and performance.
- IT Services Management Services include capabilities that relate to scale and performance. For example, edge services, clustering services, and virtualization capabilities allow efficient use of computing resources based on load patterns.

The SOA reference architecture is a comprehensive architecture that covers the integration needs of an enterprise. Its services are well integrated and are delivered in a modular way, allowing SOA implementations to start at a small project level. As each additional project is tackled, new functions can be added, incrementally enhancing the scope of integration across the enterprise.

1.4.6 SOA programming model

The programming model used for creating a service-oriented architecture involves building business possesses by composing a series of invocations that act on data. The key entities in this programming model are business objects, Service Data Objects (SDOs), Service Component Architecture (SCA), and Business Process Execution Language (BPEL). Figure 1-17 shows an overview of the programming model for the WebSphere Enterprise Service Bus.

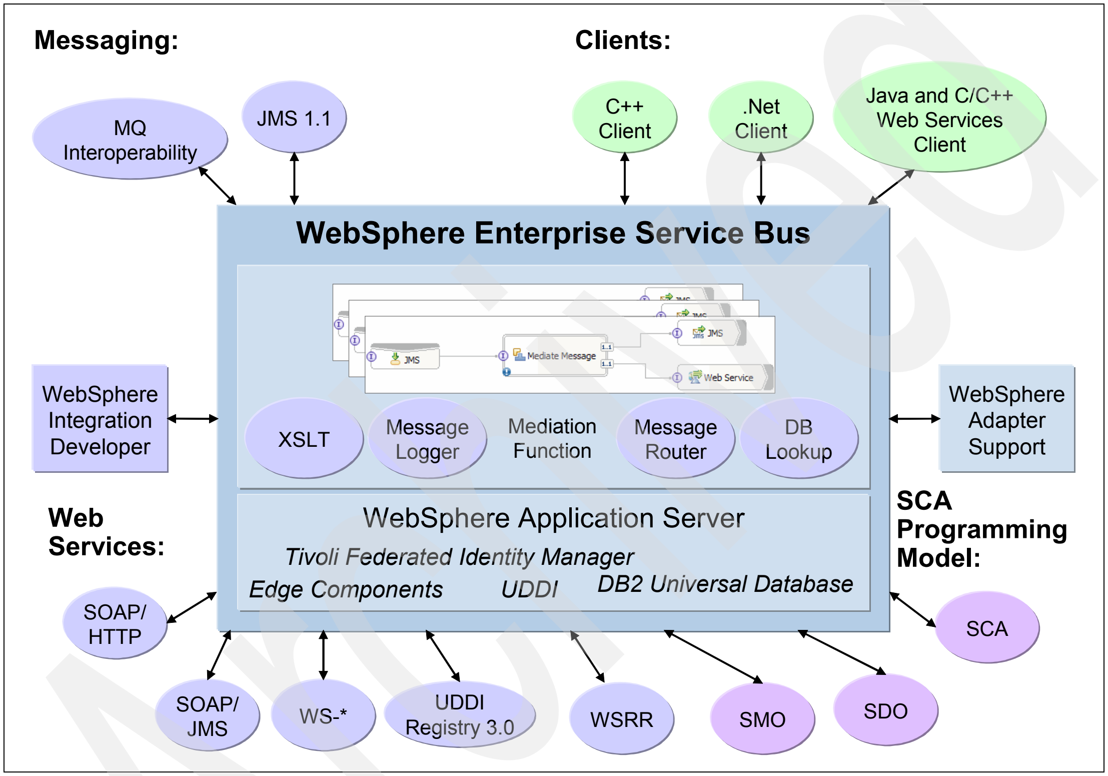


Figure 1-17 Programming model for the WebSphere Enterprise Service Bus

Business objects

Business data that is exchanged between integrated applications within an Enterprise Service Bus is represented by business objects. Business objects in the WebSphere Enterprise Service Bus or WebSphere Process Server world specify a schema for a data object. Business objectss associated with an application are called application-specific business objects (ASBOs). ASBOs are sometimes also referred to as service specific business objects (SSBOs). Generic business objects (GBOs) are business objects that exist within IBM WebSphere InterChange Server (ICS), formerly known as IBM Crossworlds

InterChange Server). An ASBO can also be thought of as a specialization of GBO for a particular application. The terms ASBOs and GBOs are more common in the ICS world, although they are equally valid elsewhere as general concepts. Every business object that is transferred between applications typically needs an associated map.

Service Data Objects (SDOs)

Business objects are based on Service Data Objects. SDOs provide an abstraction that can be used over various types of data, providing a common mechanism for accessing data. The fundamental concept in the SDO architecture is the data object. In fact, the term SDO is often used interchangeably with the term data object. A data object is a data structure that holds primitive data, multi-valued fields (other data objects), or both. The data object also has references to metadata that provide information about the data found in the data object. In the SDO programming model, data objects are represented by the commonj.sdo.DataObject Java interface definition. This interface includes method definitions that allow clients to obtain and set the properties associated with DataObject.

Another important concept in the SDO architecture is the data graph. A data graph is a structure that encapsulates a set of data objects. From the top level data object in the graph, all other data objects can be reached by traversing the references from the root data object. In the SDO programming model, data graphs are represented by the commonj.sdo.DataGraph Java interface definition.

Service Component Architecture (SCA)

SCA is defined as a programming model for implementing the SOA architecture. SCA provides a standardized way to define and invoke services.

SCA separates application business logic and the implementation details. It helps an organization build a SOA by organizing code that implements business logic into entities called components. Components offer their capabilities through a service-oriented interface and make use of existing capabilities by invoking other components through their service-oriented interfaces, called references.

It provides a model that defines interfaces, implementations, and references in a technology-neutral way, letting you then bind these elements to any technology-specific implementation. SCA provides an abstraction that covers stateless session EJBs, Web services, Plain Old Java Objects (POJOs), WS-BPEL processes, database access, enterprise information system (EIS) access, and other component types.

Business Process Execution Language (BPEL)

Composition is done using Business Process Execution Language, which is used to define the overall business process. When the business process accesses data, it does so by making calls to SCA services passing business objects.

We describe SDOs and SCA in more detail in Chapter 2, “Service Component Architecture” on page 51.

1.5 Building an SOA - IBM software offerings

IBM has a large product portfolio to cater to the various application integration needs of its customers.

Application integration for your SOA might start with WebSphere MQ for reliable service connectivity. In order to build a complete SOA, the Enterprise Service Bus connecting applications to each other might be WebSphere Enterprise Service Bus or WebSphere Message Broker. WebSphere Enterprise Service Bus would be your choice if your solution is mainly focussed around products in the WebSphere Application Server family, and WebSphere Message Broker if you are connecting applications in a more heterogeneous environment. You might also use WebSphere DataPower for specialized bridging tasks or implementing an XML firewall between your intranet and the Internet. These are compared in Figure 1-18.

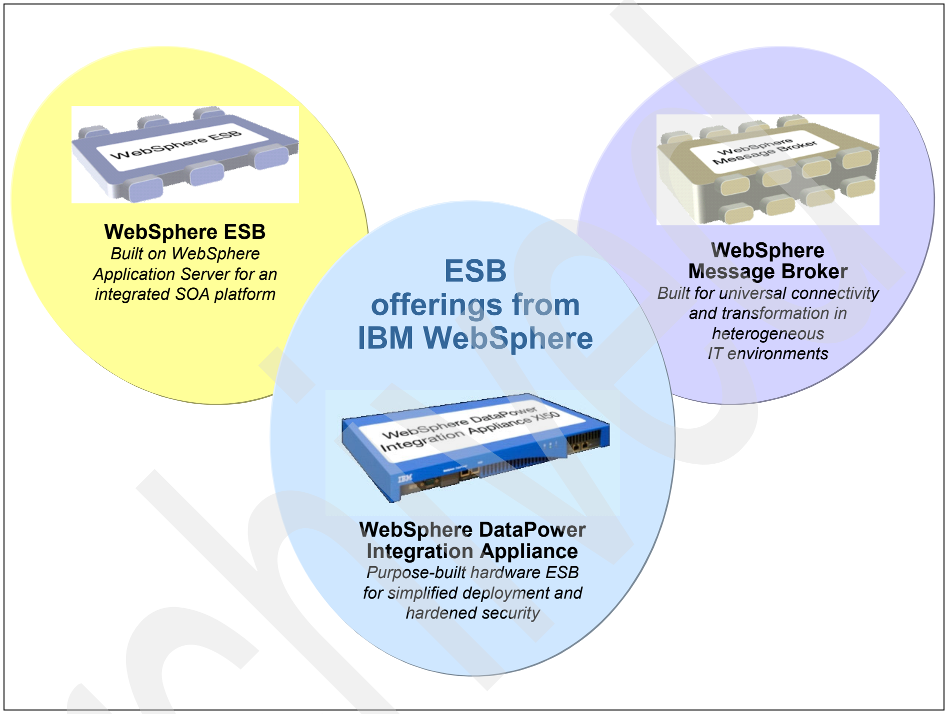


Figure 1-18 WebSphere ESB products

IBM Enterprise Service Bus without limits (Figure 1-19) extends beyond pure Enterprise Service Bus functions to include capabilities such as:

- SOA appliances (DataPower) to enable specialized connectivity needs such as an XML firewall, in alternative hardware
- Adapters (WebSphere Adapters and WebSphere Business Integration Adapters) to accelerate integration with configurable qualities of service
- More powerful data transformations through IBM WebSphere Transformation Extender (WTX, formerly known as WebSphere DataStage TX)
- WebSphere Service Registry and Repository (WSRR), for dynamic lookup of services and policy-based SOA governance

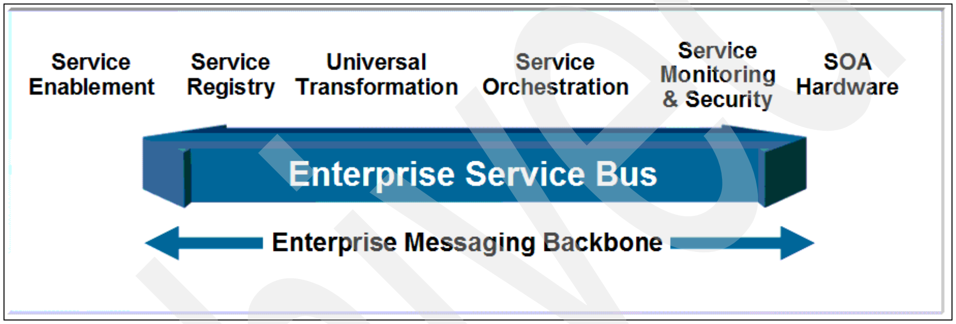


Figure 1-19 IBM Enterprise Service Bus without limits

Figure 1-20 shows some of the offerings from IBM and how they fit into the SOA reference architecture. In the following sections we describe the salient features of some of the offerings from the IBM software portfolio.

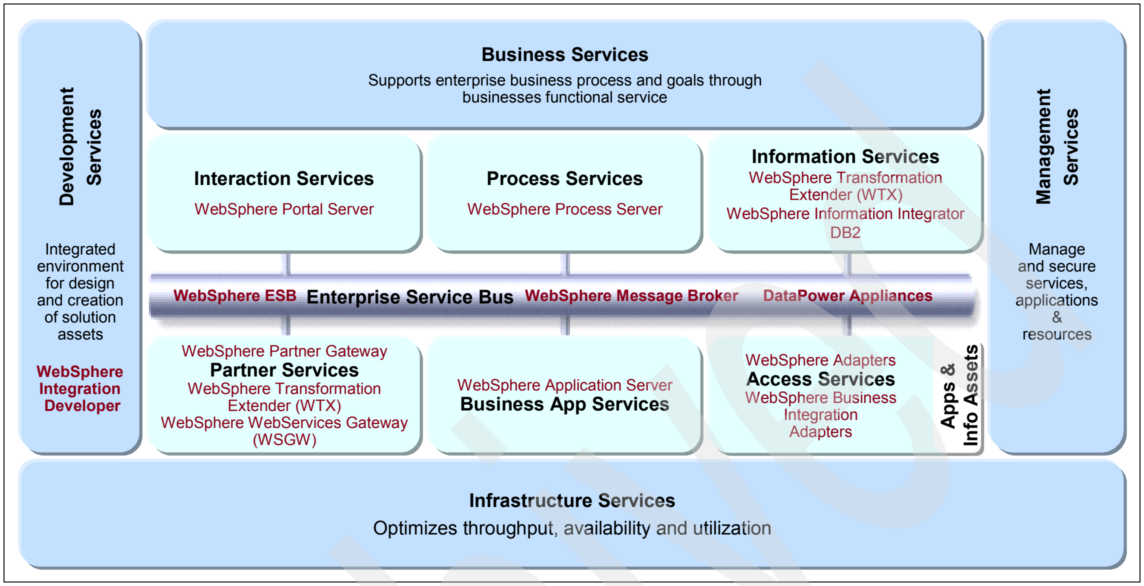


Figure 1-20 IBM software offerings for building service-oriented architecture

1.5.1 WebSphere MQ and Extended Security Edition

WebSphere MQ is the messaging backbone of many businesses today. It:

- Enables application integration by helping business applications to exchange information across different platforms, sending and receiving data as messages
- Is the proven, reliable messaging backbone for SOA connectivity
- Supports virtually any commercial IT system (over 80 platform configurations)
- Helps preserve data integrity end-to-end, even when a single transaction updates multiple IT systems
- Supports for C, C++, Java JMS messaging, and C#/.NET applications

1.5.2 WebSphere Message Broker

WebSphere Message Broker is a high-function message hub, connecting WebSphere MQ, Http, JMS, WebSphere Platform Messaging, and WebSphere Business Integration Adapters. Since v6.0 it has supported service integration as well as message integration. It is sometimes known as the Enterprise service bus built for universal connectivity and transformation in heterogeneous IT environments.

- It delivers a universal Enterprise Service Bus providing connectivity and data transformation for both standard and nonstandard based applications and services.
- Integrates heterogeneous systems, platforms, devices, and APIs.
- Rich transformation engine, high performance, and integrated complex event processing.
- IBM z/OS support.
- It exposes non-services applications as services.
- WebSphere Transformation Extender (WTX, formerly known as WebSphere DataStage TX) plug-in.
- Additional support for 64 bit platforms.

1.5.3 WebSphere Enterprise Service Bus

The WebSphere Enterprise Service Bus is packaged both as part of WebSphere Process Server and as a standalone product. It is implemented as multiple J2EE applications running on the WebSphere Application Server Network Deployment edition.

- It integrates Web services connectivity, MQ and MQ/JMS messaging, and service-oriented integration using WebSphere Adapters.
- It connects to the WebSphere Application Server.
- It connects to the WebSphere Transformation Extender (WTX, formerly WebSphere DataStage TX)[2].

1.5.4 WebSphere adapters

Adapters allow you to quickly and easily create integrated processes that exchange information between widely used packaged applications.

- They service-enable applications to participate in a service bus.
- They include integration tools to simplify the customization of adapters and services.

There are two major types or styles of adapters.

[2] Currently configured as a service offering

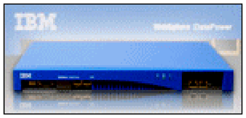
WebSphere Business Integration Adapters

These are the existing adapters that are used today with the WebSphere Interchange Server and other brokers. They are *JMS* based.

WebSphere Adapters

There are new versions of the WebSphere Adapters that use Java Connector Architecture JCA 1.5. They are bi-directional adapters and are integrated with the Service Component Architecture and include Enterprise Metadata Discovery as part of their tooling in WebSphere Integration Developer.

1.5.5 WebSphere DataPower Integration Appliances



The DataPower appliances are dedicated hardware units with custom software to improve performance of XML-based applications, and provide security in the form of an XML firewall, encryption/decryption for WS-Security, and crash protection for servers against denial of service attacks using Web services as a delivery vector. The integration capabilities of DataPower devices go beyond XML-based transformations, and the device is integrated with WebSphere Transformation Extender (WTX, formally WebSphere DataStage TX) to provide a very powerful, easily managed transformation appliance.

- These are 1U (1.75" thick) rack-mountable, purpose-built network devices that help secure and accelerate XML and Web services deployments.
- Higher levels of security assurance come with using dedicated hardware.

1.5.6 WebSphere Process Server

WebSphere Process Server is an advanced integration server catering for Business Process Management applications designed around the Service Component Architecture. It incorporates WebSphere Application Server Network Deployment and WebSphere ESB, as well as providing a run time for other types of SCA components including BPEL process, business rules, and human tasks. WebSphere Integration Developer provides the tooling for both WebSphere Process Server and WebSphere Enterprise Service Bus.

1.5.7 WebSphere Partner Gateway

WebSphere Partner Gateway provides a B2B management hub to support trading partners with different IT connectivity capabilities, and also supports standards-based EDI transports widely used by online trading partners. It integrates with the service-oriented architecture WebSphere integration products providing a bridge between the different technologies. It allows users to connect to their global trading partners and other businesses via electronic data interchange (EDI). The product enables business communications by transforming data between ROD, XML, and EDI formats.

1.5.8 WebSphere Service Registry and Repository

The WebSphere Service Registry and Repository provides a runtime Web services registry and repository to manage the description, dependencies, life cycle, and auditability of Web services to promote practical reusability in an enterprise environment.

- Integrated service metadata repository to govern services and manage service life cycle
- Seamless publish and find capabilities across all phases of SOA
- Reuse with dynamic and efficient interactions between services, promoting visibility, consistency, and reducing redundancy in your SOA

1.5.9 IBM WebSphere Business Integration for Financial Networks

This product gives banks and other financial institutions an integration platform for their mission-critical, on demand financial payment messaging needs. WebSphere Business Integration for Financial Networks offers integration for financial applications that require access to networks such as the SWIFT Secure IP Network (SIPN) and provides a single window to SWIFTNet services such as FIN, InterAct, and FileAct.

1.5.10 WebSphere Business Services Fabric

This is an end-to-end SOA platform for modeling, assembly, and deployment of business services that is integrated with WebSphere Process Server and WebSphere Integration developer. The focus is on building solutions from pre-built service components.

1.6 Web 2.0 - next generation Web services

Even though Web services as a standard implies use of SOAP as a message format, the phrase *Web services* is loosely used to refer to services on the Internet that accept messages in a form other than SOAP.

These new *Web services* or *XML Web services* are just as much part of the SOA concept as are SOAP services. Neither Web services nor SOA, despite common misconception, mandate SOAP. A large group of people advocate the idea of simply exchanging raw XML documents or some other format (plain text, JSON) directly over HTTP, either through a RPC mechanism (XML-RPC, JSON RPC) or through an approach loosely advocated under the banner of Representational State Transfer (REST). Advocates of these Web services complain that SOAP is complex, stunts its XML payload, and does not take enough advantage of the fundamental strengths of the Web. SOAP advocates have worked to address these matters by shifting emphasis from SOAP's remote procedure call (RPC) roots to what is called the document-literal style of SOAP. In the RPC style, the data to be transmitted is marshalled into discrete data types in a special XML payload format (called the SOAP encoding). In the document-literal style, the XML payload consists of more natural XML formats that generally tend to be more descriptive and human-readable.

1.6.1 Communicating over HTTP using non SOAP data

For the purpose of this discussion, by a lightweight service on the Internet we mean any service that communicates using non-SOAP protocol over HTTP. The data format could be plain XML (for XML-RPC services) or JSON (JavaScript™ Object Notification, for JSON RPC services) or just plain text (for HTTP services exposing their interface in a RESTful way). While JSON RPC services and XML-RPC services are tied to a data format (JSON and XML, respectively), RESTFul services just make use of REST as an architecture guideline and can make use of any data format (plain text, XML, or JSON being the commonly used data formats). The original description and rationale for REST was described by Roy Thomas Fielding in his Ph.D. thesis "*Architectural Styles and the Design of Network-based Software Architectures.*"

A large number of services make use of XML-RPC, JSON RPC, and REST to expose their interfaces for consumption over the Internet. These services have gained popularity with the emergence of Asynchronous JavaScript and XML (AJAX) based Web clients that can invoke these services through standard Web browsers. One of the reasons for their growing popularity is that they are more lightweight and quicker to parse in a Web browser as compared to the heavier SOAP messages used by standard Web services.

The following sections give an overview of these lightweight services along with examples.

1.6.2 An example SOAP message

In order to understand the issues involved, let us take an example stock quote service that accepts a stock quote request and returns the price in response.

A SOAP request message for such a service is shown in Figure 1-21.

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
  xmlns:tns="http://itso.ibm.com/stock">
  <soap:Header>
</soap:Header>
  <soap:Body>
    <tns:GetStockPrice>
      <tns:Stock>IBM</tns:Stock>
    </tns:GetStockPrice>
  </soap:Body>
</soap:Envelope>
```

Figure 1-21 A stock quote service request message in SOAP

1.6.3 Web service expecting non standard SOAP headers

A Web service expecting nonstandard SOAP headers requires its clients to put additional parameters in the SOAP header other than what is permitted by the available tooling. We investigate the use of mediation facilities provided by the WebSphere Enterprise Service Bus to insert these additional SOAP headers so that a standard Web service client generated by the tooling can be used as is, with the additional header requirements dealt with by the service bus infrastructure, rather than *polluting* the application layer with transport-specific considerations.

In the SOAP message above, the header is empty. A Web service expecting nonstandard SOAP headers might require a CustomerId to be present in the SOAP header (as shown in Figure 1-22).

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
  xmlns:tns="http://itso.ibm.com/stock">
  <soap:Header>
    <tns:CustomerId>45</tns:CustomerId>
  </soap:Header>
  <soap:Body>
    <tns:GetStockPrice>
      <tns:Stock>IBM</tns:Stock>
    </tns:GetStockPrice>
  </soap:Body>
</soap:Envelope>
```

Figure 1-22 A stock quote service request message using additional SOAP headers

1.6.4 XML-RPC

One antecedent of SOAP that is still in fairly wide use is XML Remote Procedure Calls (XML-RPC). XML-RPC defines procedure calls encoded in XML and communicated over HTTP. It retains some popularity because of its simplicity and the fact that most languages and many application frameworks now have standard or readily available XML-RPC implementations. It does have some notable weaknesses, including primitive data typing and lack of support for character encodings (an astonishing flaw given its use of XML).

XML-RPC service

A XML-RPC service expects its clients to communicate over plain XML. XML-RPC is not really a new technology. Rather, it is a new way of using existing technologies (XML over HTTP). XML-RPC does not use the standard XML Schema Definition (XSD) data types, and introduces its own eight data types:

- int
- double
- string
- boolean
- base64
- dateTime.iso8601
- array
- struct

XML-RPC messages tend to be less verbose. For the example SOAP message given above, the XML-RPC message is shown in Figure 1-23.

```
<?xml version="1.0"?>
<methodCall>
   <methodName>GetStockPrice</methodName>
   <params>
      <param>
         <value><string>IBM</string></value>
      </param>
   </params>
</methodCall>
```

Figure 1-23 A stock quote service request message in XML-RPC

1.6.5 REST

Representational State Transfer (REST) is broad term that defines an architectural style for interaction between systems. It is not a standard with precise rules or well-defined APIs. Nevertheless, interaction using REST are now widespread in the Internet world.

RESTful interactions are based on HTTP as the communication transport, with URLs and XML documents as the syntax for interactions. The HTTP request uses the *logical* URL, the document sent for information transfer from the requester to the provider, and the document received for information transfer from the provider to the requester. There is a URL for each resource and the URL is simply a *noun*. The *verb* for the interaction is defined by an HTTP method (GET, POST, PUT, or DELETE).

REST by itself does not recommend any data format and can use either plain text, xml, or JSON.

RESTful service over HTTP

A RESTful service exposes its interfaces through distinct URL patterns that support the four CRUD operations. It makes use of all the four HTTP operations:

- POST or PUT for create
- GET for read
- PUT for update
- DELETE for delete

For example, in order to support a RESTful shopping cart service, various operations must be implemented, as shown in Table 1-1.

Table 1-1 Designing a RESTful shopping cart service

Operations	URL	HTTP Method	Response
Create a cart.	/cart	PUT	cart URL
Delete a cart.	/cart/cartID	DELETE	status code
Retrieve the cart.	/cart/cartID	GET	cart content
Add an item to the cart.	/cart/cartID/	PUT	item URL
Remove an item from the cart.	/cart/cartID/itemID	DELETE	status code

Note: Even though CRUD is often used to explain REST, REST is not limited to a one-to-one mapping of CRUD operations to HTTP methods. In order to understand REST, one has to think in terms of resources and the semantics of each HTTP operation. For example, one can use either PUT to create a shopping cart (by using the URL /cart) or one can use POST to create a shopping cart by posting to a cartHandler (by using the URL /cartHandler/cart). PUT is used to create or update a resource on the server, whereas POST can be used to POST to an existing entity (or a handler), which in turn acts on your request (and can create, update, or delete a resource in response to your request).

A RESTful service is not tied to a particular data format, although plain text, JSON, and XML are commonly used. For the stock quote example, above a RESTful call can be just a HTTP GET with the stock name as part of the URL:

```
GET /stock?StockName=IBM HTTP/1.1
```

The above example (creating a shopping cart in a RESTful way using the WebSphere Enterprise Service Bus) is discussed in detail in the following article:

http://www-128.ibm.com/developerworks/websphere/library/techarticles/0611_flurry/0611_flurry.html

1.6.6 JSON RPC

JSON is a subset of the object literal notation of JavaScript and is a common way to represent data structures in JavaScript. JSON is a text format that is completely language independent, but uses conventions that are familiar to programmers of the C family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others. JSON is built on two structures:

- A collection of name/value pairs. In various languages, this is realized as an object, record, struct, dictionary, hash table, keyed list, or associative array.
- An ordered list of values. In most languages, this is realized as an array, vector, list, or sequence.

For more information see:

http://json.org/

JSON-RPC is a lightweight remote procedure call protocol in which JSON serializes requests and responses. Sending a request to a remote service invokes a remote method. The request is a single object with three properties:

- method - A string containing the name of the method to be invoked.
- params - An array of objects to pass as arguments to the method.
- id - The request ID. This can be of any type. It is used to match the response with the request that it is replying to.

When the method invocation completes, the service must reply with a response. The response is a single object with three properties:

- result - The object that was returned by the invoked method. This must be null in case there was an error invoking the method.
- error - An error object if there was an error invoking the method. It must be null, if there was no error.
- id - This must be the same ID as the request it is responding to.

JSON RPC Service

JSON RPC services make use of JSON as the data format. It uses HTTP POST and GET for making the remote procedure call. A remote procedure call in JSON-RPC consists of four properties:

- Version of the JSON RPC protocol
- Method name
- Optional array of arguments to be passed to the method
- Optional request ID that is used to match responses with requests

The stock quote request message in JSON RPC is shown in Figure 1-24.

```
{
    "version" : "1.1",
    "method"  : "GetStockPrice",
    "params"  : [ IBM ]
}
```

Figure 1-24 A stock quote service request message in JSON RPC

JSON RPC internally uses the XMLHttpRequest object that is at the core of AJAX concepts. It consists of server-side classes (there are server-side libraries available in several languages including Java) as well as client-side Javascript libraries, and these two in conjunction provide a RPC layer.

1.7 Summary

This chapter reviewed some of the history of application integration leading to service-oriented architecture and introduced the Service Component Architecture. Service-oriented architecture is a broad concept with its roots in understanding how IT should assist businesses to make the transition to an on demand world. Service Component Architecture is the subject of the next chapter. It is an new open framework for creating and managing service-oriented solutions, and is the programming model for WebSphere ESB.

2

Service Component Architecture

In Chapter 1, “Connecting enterprise applications” on page 3, we explained the principles of a service-oriented architecture (SOA) and its benefits. We mentioned Service Component Architecture (SCA) as a programming model and architecture that provides a framework for implementing a SOA. In this chapter, we provide a brief introduction to SCA, along with the related technology of Service Data Objects (SDOs).

This chapter is not a comprehensive description of SCA. We discuss the terms and concepts that are of most use in order to get a good understanding of how enterprise applications can be connected to WebSphere Enterprise Service Bus using adapters and other types of connection.

SCA is used by the Business Integration set IBM WebSphere products, such as WebSphere Enterprise Service Bus and WebSphere Process Server, as well by many other vendors and products. It is not yet implemented across the entire family.

2.1 Service Component Architecture

SCA is designed around *components*, which encapsulate services that can be invoked. SCA components expose interfaces that define the information that must be supplied to invoke a service. Each *interface* is defined by either a Java interface or a Web services Description Language (WSDL) Port Type. As a corollary, these interfaces are expressed in terms of data types that are either Java types (primitive or class-based) or that are defined by an XML schema.

SCA components can also invoke services that are exposed by other SCA components. We say that one SCA component *references* the other. Service invocation is an entirely black-box affair. It does not matter to one component how another is implemented.

An interface can have one or more *operations* (also sometimes called functions). When a component is invoked, an operation name is specified. These operations can be one-way or two-way. Figure 2-1 shows a simplified UML model of the SCA architecture.

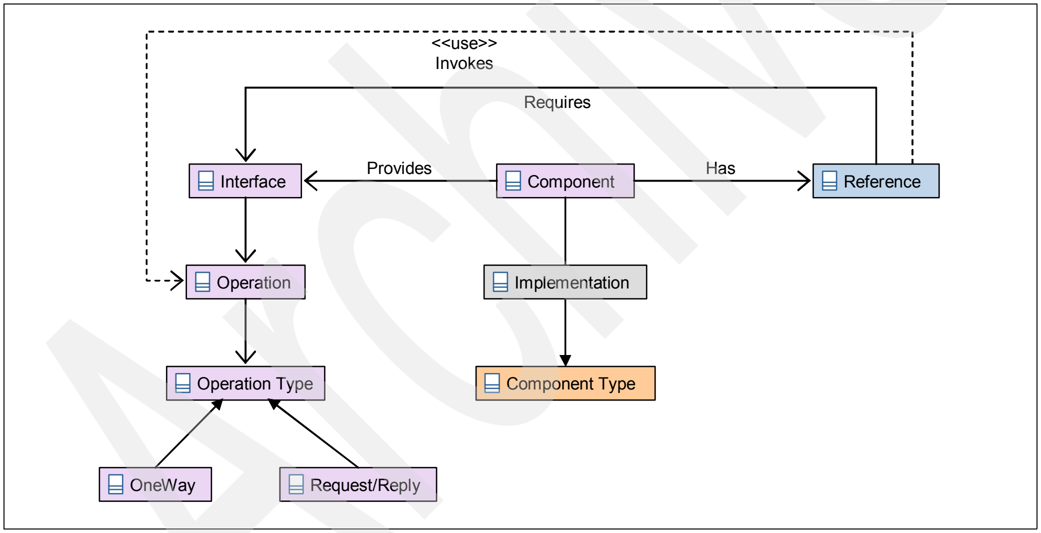


Figure 2-1 Simplified SCA component UML model

The Open Service Oriented Architecture collaboration is documenting the SCA. You will find a more complete UML model of SCA in “SCA Service Component Architecture” at the Open Service Oriented Architecture Web site:

`http://www.osoa.org/display/Main/Home`

One-way operations (also called fire-and-forget) have data that flows into the SCA component from the caller, but the SCA component sends no data back to the caller. Two-way operations (which are often also called request/reply) have data flowing into the component (request), and then back to the caller (reply). In between the request and reply phases, the component can invoke other components if it wishes, as well as performing other functions.

Attention: Some architectural models call one-way operations *asynchronous* and two-way operations *synchronous* (for example, the Unified Modelling Language (UML) uses these terms). SCA uses these terms in a different, and orthogonal, way.

SCA components can be have several types of *implementation*, such as Java, BPEL, or C++. WebSphere ESB introduces a particular component type, a mediation flow, which is discussed in greater depth in 3.4.2, “Mediation flow components” on page 73. Figure 2-2 illustrates some of these principles.

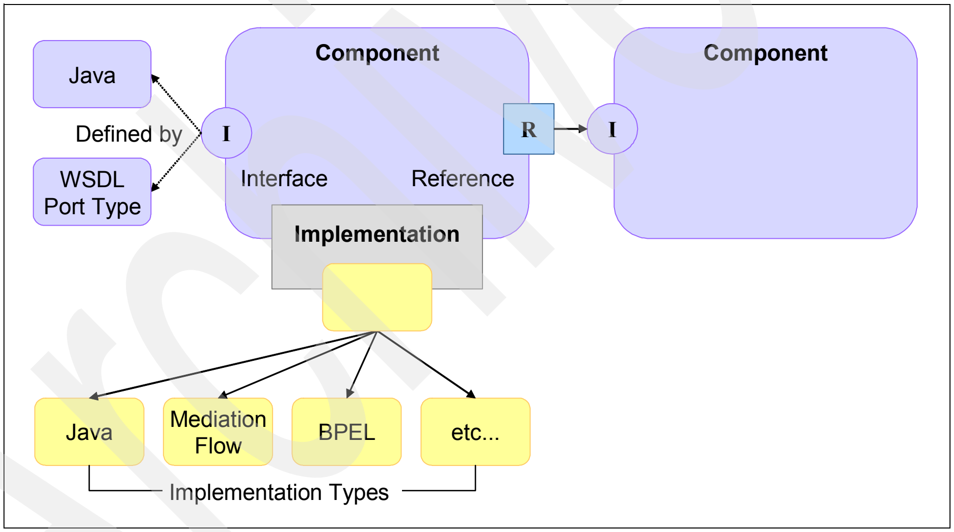


Figure 2-2 SCA component principles

Restriction: Only components that are implemented in Java can have interfaces defined by Java.

SCA components are typically useful when they are connected together with *wires*. These wires connect references to interfaces. A set of interconnected components is packaged together in an SCA *module*, which is typically the unit of deployment.

Implementing SCA components is a conceptually separate task from assembling them together in this way (also known as composing a module).

Modules expose *exports*, which can be invoked either from other SCA modules via *imports*, or via other means (for example, a client external to the SCA model). SCA imports can also call out to services outside the SCA model. Modules are designed such that they are tightly coupled within (all components are specified at design time, together with their wirings), and are loosely coupled without (different modules could connect together in different ways at run time.
Figure 2-3 shows a simplified UML model of an SCA module, with the concept of a wire connecting references to interfaces, either in the same module or through exports and imports in different SCA modules.

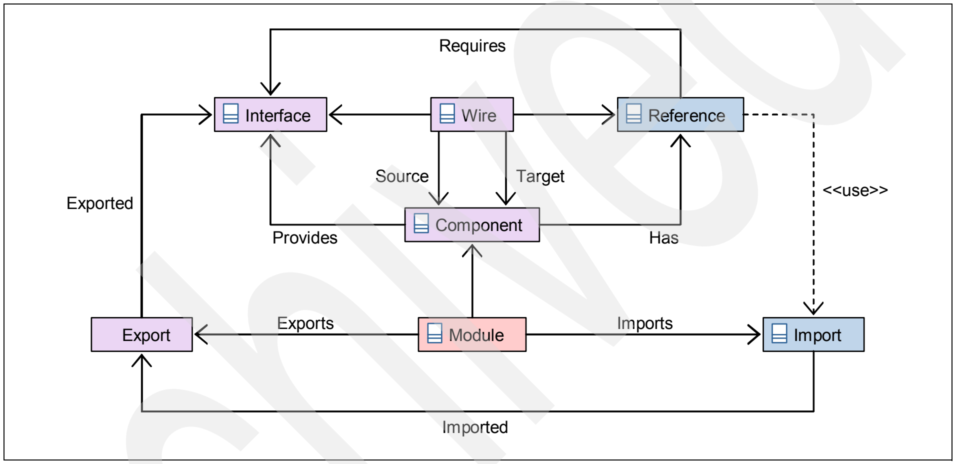


Figure 2-3 Simplified SCA module UML model

Tip: It is worth knowing that these concepts are not always strictly adhered to. For example, the dynamic endpoint support in WebSphere Enterprise Service Bus 6.0.2 effectively allows the wiring between a mediation flow component and an import to be dynamically selected. See Chapter 3, “Connecting to the WebSphere Enterprise Service Bus” on page 63, for more information, in particular “Dynamic endpoints” on page 81.

You can see an example of how these concepts fit together in Figure 2-4.

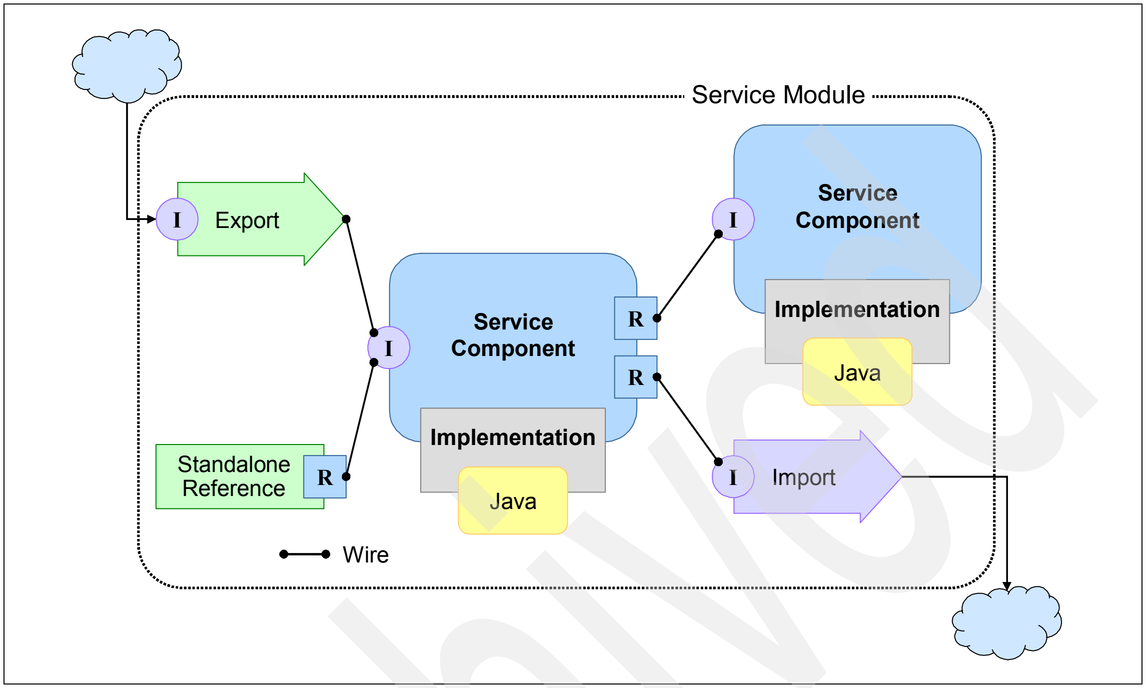


Figure 2-4 Example of an SCA module

As shown in Figure 2-4, it is possible for non-SCA components (typically implemented in Java) to invoke SCA components. In order for this to be possible, a stand-alone reference has to be created to this component. A stand-alone reference performs a similar role to an export.

Both imports and exports have *bindings*, which can be of a variety of types, such as Web services or the SCA default binding. These bindings define the communication mechanism used to invoke an SCA module (exports) or call out from one (imports). Stand-alone references do not have bindings. Figure 2-5 is a simplified UML model of SCA transport bindings.

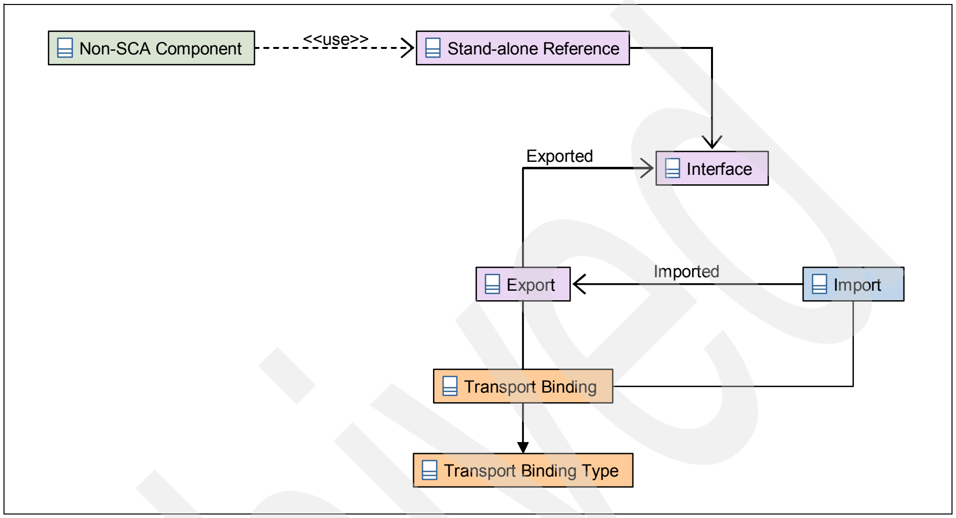


Figure 2-5 Simplified model of SCA transport bindings

You can see how an import interacts with the rest of an SCA module in Figure 2-6. This illustration mentions a JMS Binding. This is explained in more detail in 3.5.3, “JMS bindings” on page 83.

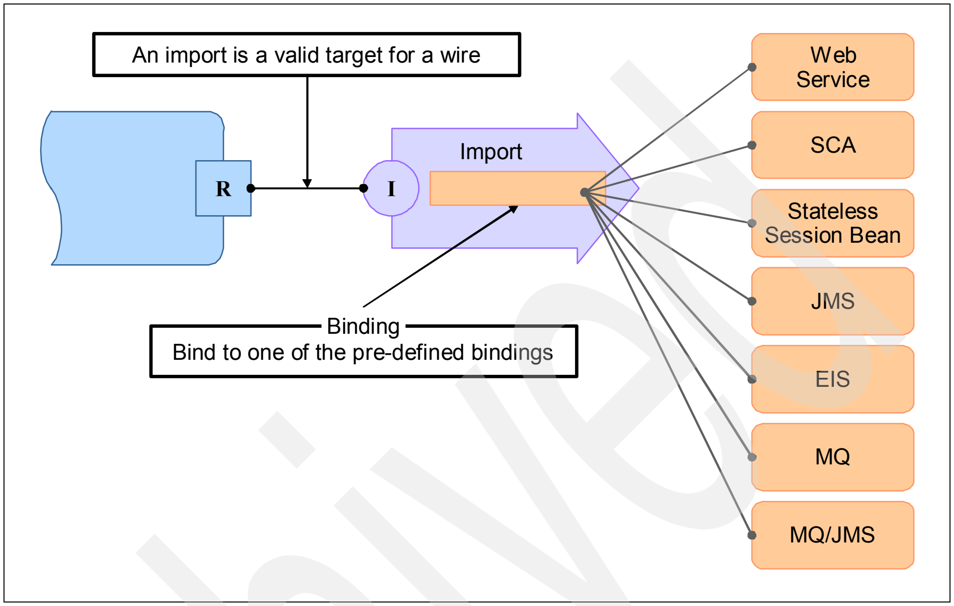


Figure 2-6 How an import fits into an SCA module

You can see how an export interacts with the rest of an SCA module in Figure 2-7.

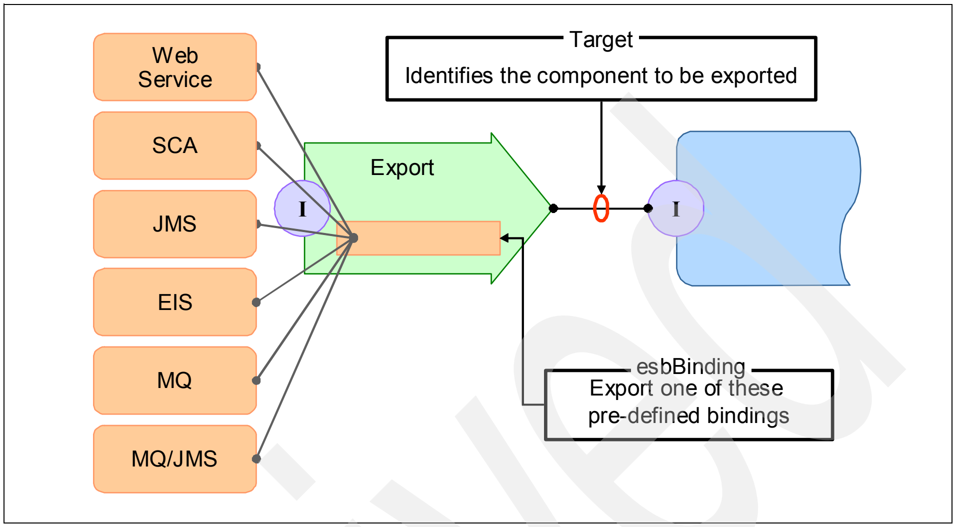


Figure 2-7 How an export fits into an SCA module

Attention: Standard SCA terminology calls import and export bindings simply *bindings*. However, to be precise, in this section, we refer to what we shall call transport bindings. Data and header bindings, which are discussed later, are related to transport bindings but are not the same concept.

SCA modules are primarily defined by Service Component Definition Language (SCDL) files. Typically there is one for each import, export, and component in the module, as well as one for the module itself. It is not normally necessary to understand SCDL directly, as it is typically machine-generated (for example, WebSphere Integration Developer performs this task when you visually compose SCA modules). Also, Rational Software Architect v7.0.0.2 generates SCDL from a UML model and from a BPEL model imported from WebSphere Business Modeler. WebSphere Business Modeler Advanced generates SCDL and BPEL from a business model.

SCA makes a distinction between synchronous and asynchronous invocation patterns. Some transport bindings can only operate in one of these ways, but some (such as the SCA default binding) can operate in either.

Attention: SCA as we have described it above is based on Version 0.9 of the specification. This is also the version used in WebSphere Enterprise Service Bus (as explored in Chapter 3, "Connecting to the WebSphere Enterprise Service Bus" on page 63), which is why we have discussed it here. Newer versions of the specification have made some changes, such as making the SCA model recursive. Modules are now called composites and can contain other composites. For more information see the following Web site:

`http://www.osoa.org/display/Main/SCA+changes+since+0.9+specification`

2.2 Service Data Objects

Service Data Objects (SDOs) are the primary method of data exchange within SCA. They are passed from one SCA component to another during invocation as part of an *SCA message* (which also contains other details, such as the operation being invoked). The interface that SDOs expose to SCA components and other applications is defined by the commonj.sdo.DataObject Java interface. This interface includes method definitions that allow clients to obtain and set the data associated with a data object.

A data object contains a number of properties. These are either atoms of data (with an associated type) or references to other data objects. In this way, data objects can form a hierarchy (Figure 2-8).

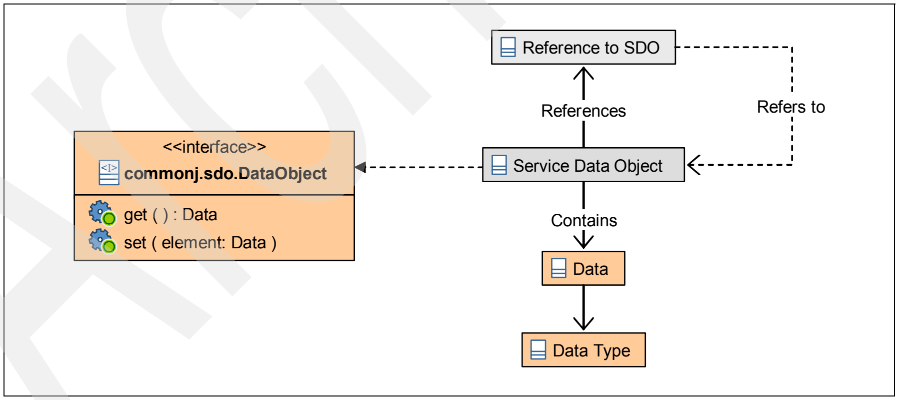


Figure 2-8 Simplified SDO UML model

As an example, consider modeling customer data with an SDO. The properties associated with the customer might be firstName (String), lastName (String), and customerID (long). The following pseudo code shows how you might use the DataObject API to obtain and set properties for the customer data object:

```
DataObject customer = ...
DataObject order = ...
customer.setString("firstName","John");
customer.setString("lastName","Doe");
customer.setInt("customerID", 123);
order.setDataObject(“customer”, customer);
// later on...
int id = order.getDataObject(“customer”).getInt(“customerID”);
```

Tip: Often SDOs are referred to simply as data objects (DOs), which is the primary term we use in this book. In practice, these terms mean the same thing.

The structure of these data objects is represented in Figure 2-9. In fact, this figure shows business objects and how they appear in WebSphere Integration Developer, but the distinction between business objects and data objects is explained in 3.2, “Business objects” on page 67. For the time being, you can assume that they are the same.

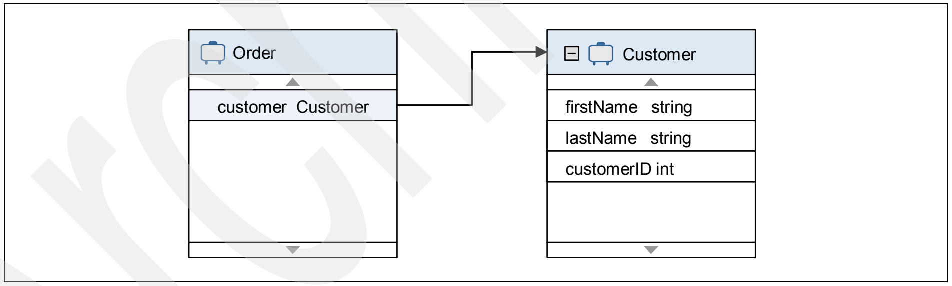


Figure 2-9 A representation of two data objects

Data graphs are related to data objects. They contain a root object and a change summary. They are typically used when SDO is being used in a disconnected manner. For example, a database might return an SDO containing some data in the root object and an empty change summary. As an application changes the data object, the changes are written to the change summary. When the data object is passed back to the database to update it, it will use the original data object and the change summary to calculate what needs to be updated. In the SDO programming model, data graphs are represented by the commonj.sdo.DataGraph Java interface definition.

Figure 2-10 shows a UML model of data graphs. The Data Mediator is responsible for performing create, read, update, and delete (CRUD) operations on the data source, and for creating data graphs from data sources. By using the change summary, updates to the data source are optimized, and common update patterns, such as optimistic locking,[1] are implemented where the data source is left unlocked between read and update operations.

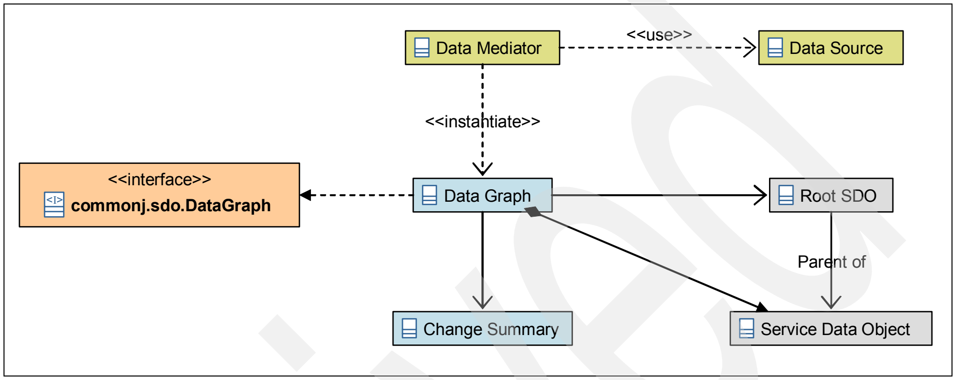


Figure 2-10 Simplified UML model of data graphs

[1] At the time of update if the data in the data source hasn't changed since it was read, it is updated. If the data has changed, then the update conflict needs to be sorted out by a context specific mechanism.

3

Connecting to the WebSphere Enterprise Service Bus

In the Chapter 2, “Service Component Architecture” on page 51, we introduced some of the terminology involved in the Service Component Architecture. In this chapter, we go on to discuss how WebSphere Enterprise Service Bus provides a manifestation of an Enterprise Service Bus (ESB) based on this technology.

Given that the focus of this book is on the connection of enterprise applications to WebSphere Enterprise Service Bus rather than its mediation capabilities, we focus the majority of this chapter on connection facilities (although the mediation capabilities are briefly covered). This chapter does not provide a detailed reference to all of the facilities of WebSphere ESB. For that refer to the WebSphere ESB InfoCenter:

`http://publib.boulder.ibm.com/infocenter/dmndhelp/v6rxmx/index.jsp?topic=/com.ibm.websphere.wesb.doc/info/welcome.html`

You might also find the book *Getting Started with WebSphere ESB*, SG24-7212, useful:

`http://www.redbooks.ibm.com/abstracts/sg247212.html`

This book is written for WebSphere Integration Developer and WebSphere Enterprise Service Bus versions 6.0.1, rather than the latest version (which, at the time of writing, are 6.0.2). However, it is still useful, as it covers some topics that are not discussed in this book.

3.1 Overview of WebSphere Enterprise Service Bus

In 2.1, “Service Component Architecture” on page 52, we discussed SCA generally, explaining how it provides a software architecture to implement an SOA.

WebSphere Enterprise Service Bus is based on SCA. It contains an *SCA run time*, which contains a Java API and other infrastructure required to support certain SCA component and binding types.

SCA Modules are called *mediation modules* in WebSphere Enterprise Service Bus. It also supports two types of SCA component, Java components and mediation flow components. These are described in more detail in 3.4, “WebSphere Enterprise Service Bus SCA Components” on page 69. Figure 3-1 is a UML model showing the relationship between Mediation and generic SCA classifiers.

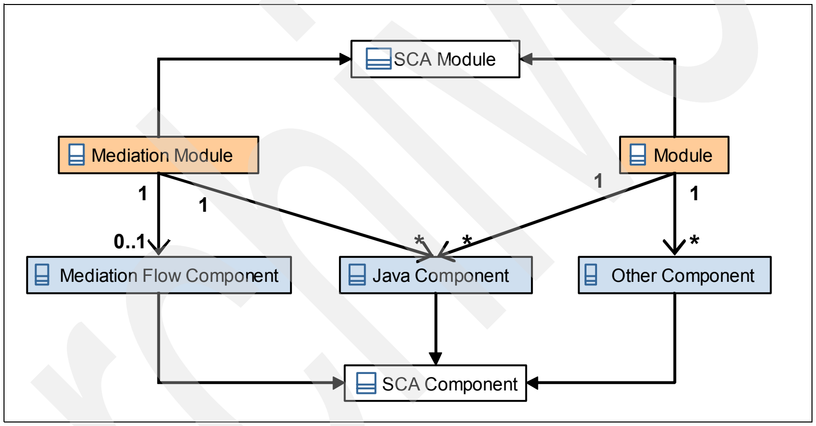


Figure 3-1 Simplified mediation component UML model

WebSphere Integration Developer (the development environment for WebSphere Enterprise Service Bus) provides the ability to compose mediation modules that can be deployed onto WebSphere Enterprise Service Bus. These modules are composed in the Assembly Diagram view in WebSphere Integration Developer. This is where SCA components are joined together, interfaces defined on components, and so on. All of this is possible without writing any code. Figure 3-2 is an example of how a typical SCA module might appear.

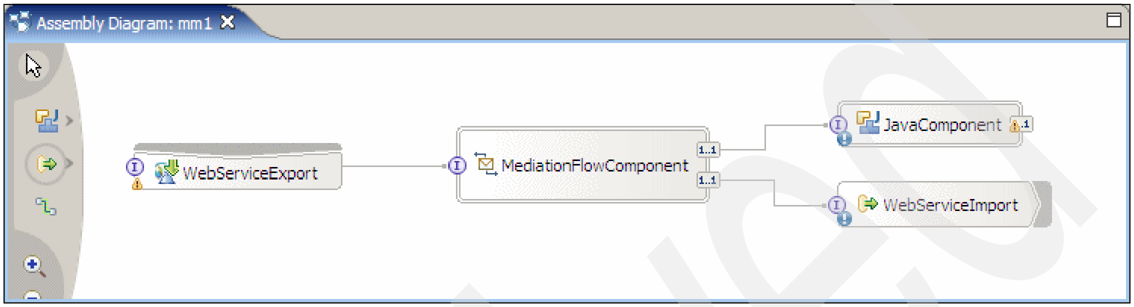


Figure 3-2 Assembly diagram showing a mediation module in WebSphere Integration Developer

> **Note**: The underlying implementation of a WebSphere Enterprise Service Bus mediation module is a Java 2 Enterprise Edition (J2EE) application. We discuss this topic further in 3.7, “J2EE and WebSphere Application Server technology underlying WebSphere ESB” on page 99.

As mentioned, WebSphere Enterprise Service Bus supports two component types:

- Mediation flow components
- Java components

These component types are discussed in more detail in 3.4, “WebSphere Enterprise Service Bus SCA Components” on page 69.

However, this book is primarily concerned with connecting enterprise applications to WebSphere Enterprise Service Bus, so what are of more interest to us are the various transport bindings that are supported on SCA imports and exports, which allow us to connect to applications external to WebSphere Enterprise Service Bus:

- Web services
- Messaging
  - JMS
  - WebSphere MQ
  - WebSphere MQ using JMS

- Stateless Session Bean
- Enterprise information systems (EIS)
- SCA default or native

Each of these binding types is discussed in more detail in 3.5, “WebSphere ESB Transport Binding Types” on page 82.

3.2 Business objects

Chapter 1, “Connecting enterprise applications” on page 3, introduced the concept of a business object. In WebSphere Enterprise Service Bus (and WebSphere Process Server) a business object has a specific meaning: it is the definition of a type of data object. A business object can be defined graphically in WebSphere Integration Developer, and its underlying representation is an XML schema type. If you want to define an interface in WebSphere Integration Developer (which is typically represented by a WSDL file) that uses complex (non-atomic) types, you need to define a business object to do this.

Figure 2-9 on page 60 shows two business objects, a Customer object and an Order object, as they would appear in WebSphere Integration Developer. In XML schema form, the Customer business object would appear as in Example 3-1.

Example 3-1 Customer business object in XML Schema form

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://lib_chap2_dataobjects">
  <xsd:complexType name="Customer">
    <xsd:sequence>
      <xsd:element minOccurs="1" name="firstName" type="xsd:string"/>
      <xsd:element minOccurs="1" name="lastName" type="xsd:string"/>
      <xsd:element minOccurs="1" name="customerID" type="xsd:int"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

The order business object would appear as in Example 3-2.

Example 3-2 Order business object in XML Schema form

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:bons0="http://lib_chap2_dataobjects"
targetNamespace="http://lib_chap2_dataobjects">
    <xsd:include schemaLocation="Customer.xsd"/>
```

```
    <xsd:complexType name="Order">
    <xsd:sequence>
      <xsd:element minOccurs="1" name="customer"
           type="bons0:Customer"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

It is particularly important to be aware of the targetNamespace that business objects use, because the namespace is required when creating data objects based on business objects using code (for example, in a Java SCA component, which we discuss in 3.4.1, "Java components" on page 69). A detailed explanation of XML namespaces is beyond the scope of this book (you can find more information here: http://www.w3.org/TR/REC-xml-names/), but they are similar in many respects to Java packages. By default the namespace used for business objects is based on the module or library they are contained within, but it can be changed in the Properties view for a business object in WebSphere Integration Developer.

In these examples, both of the namespaces used are identical, as they are contained within the same library. Nevertheless, the Order business object refers to the Customer business object using a namespace prefix (essentially a shortcut to the namespace). In this case bons0. bons0 is defined in the first few lines to point towards the http://lib_chap2_dataobjects. Both of the business objects use that namespace as their targetNamespace, which is the namespace used for the business object by default.

It is worth being aware that often the term *business object* and the term *data object* are used interchangeably, even though they technically have a slightly different meaning. This does not normally cause any ambiguity.

3.3 Data Bindings

In order to convert data external to WebSphere Enterprise Service Bus to and from data objects, some transport bindings require a data binding to be specified. The data binding translates between the data provided by the transport binding as part of a message and a data object.

Typically, there are some pre-supplied data bindings, but if these do not suit, you can write a data binding, which conforms to the commonj.connector.runtime.DataBinding interface (in fact, some data bindings conform to a more specific interface that extends this one, that provides methods to get and set transport-specific data). The DataBinding interface specifies two methods: getDataObject() and setDataObject(). The first is used by the WebSphere Enterprise Service Bus to get the data object that should be used within the mediation module, and is typically used when the data binding is specified on an export. The second is used to pass the data object into the data binding to be translated into external data, and is typically used on an import. These roles are swapped when a reply message is handled in a two-way operation.

3.4 WebSphere Enterprise Service Bus SCA Components

WebSphere Enterprise Service Bus supports two SCA component types, Java components and mediation flow components. By themselves, they do not normally connect to Enterprise Applications (although we explore a pattern in Chapter 5, “Business scenarios” on page 129, where a Java component is used to do this), but they are important for mediating data received or sent to enterprise applications, so we cover them here.

3.4.1 Java components

WebSphere Enterprise Service Bus supports Java-based SCA components. These provide the ability to implement service components using Java classes. These classes are like normal Java classes, except that they have to implement service operations using methods that take data objects as parameters (and return data objects as appropriate). Java components are most appropriate when other components do not provide the functionality you require.

Java components are best understood by example. For example, imagine that we have defined the interface shown in Figure 3-3 in WebSphere Integration Developer, with a single two-way operation.

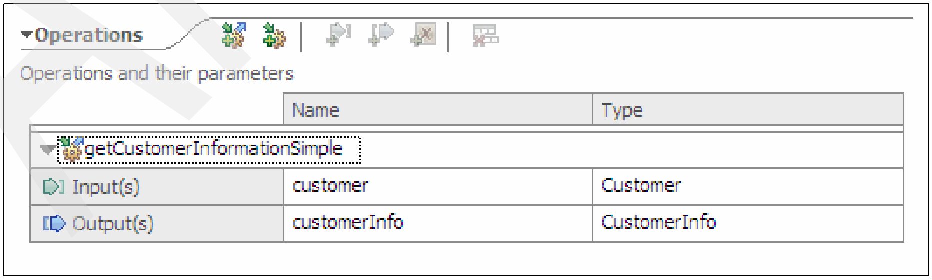


Figure 3-3 FindCustomerInformation interface

The getCustomerInformationSimple operation expects a customer business object, as shown in Figure 3-4, as input.

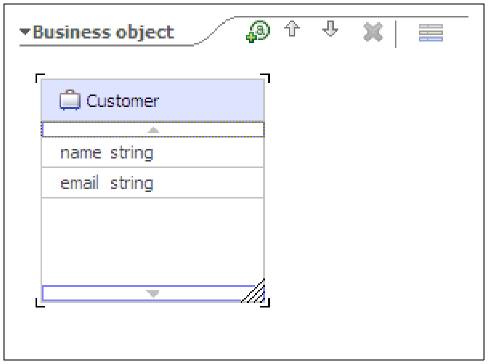


Figure 3-4 Customer business object

It returns a CustomerInfo business object (shown in Figure 3-5).

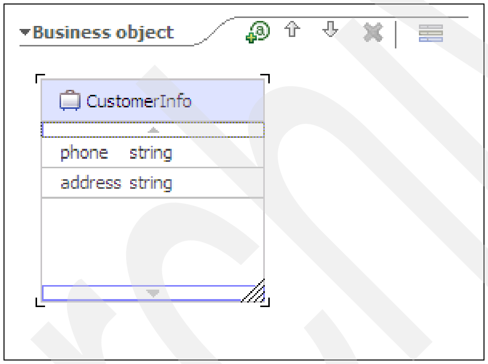


Figure 3-5 CustomerInfo business object

We implement a Java component that is responsible for implementing this operation. It should return the phone number and address based on the customer's name and e-mail. The implementation is shown in Example 3-3. The design is shown in the UML sequence diagram in Figure 3-6.

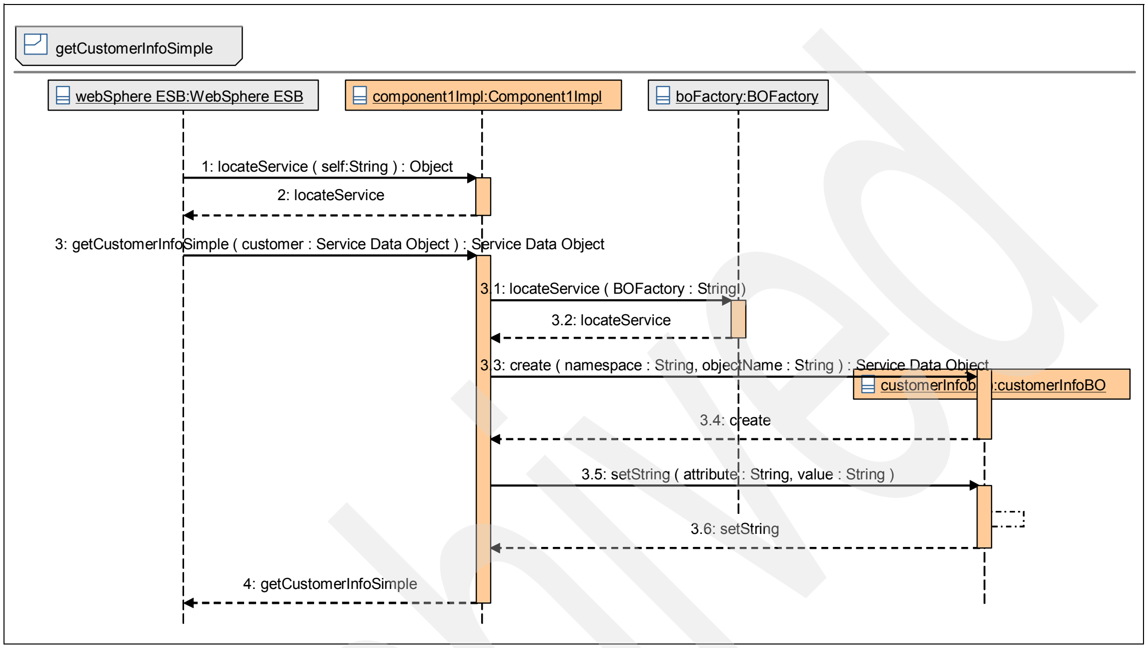


Figure 3-6 Sequence diagram showing operation getCustomerInfoSimple

You can see that the operation name in the interface corresponds to the Java method name (getCustomerInformationSimple). The method deconstructs the incoming Data Object, which contains the data defined by the customer business object. It then creates a CustomerInfo business object and fills in the data. In this case, we merely pass through the values for name and e-mail as the phone number and the address. In reality, this implementation would do something more realistic, such as look up the values in a database.

Example 3-3 Example implementation of SCA Java component

```
package sca.component.java.impl;

import com.ibm.websphere.bo.BOFactory;
import com.ibm.websphere.sca.ServiceManager;
import commonj.sdo.DataObject;

public class Component1Impl {
    public Component1Impl() {
```

```
        super();
    }

    private Object getMyService() {
        return (Object) ServiceManager.INSTANCE
                .locateService("self");
    }

    public DataObject getCustomerInformationSimple(
            DataObject customer) {
        String name = customer.getString("name");
        String email = customer.getString("email");

        BOFactory boFactory = (BOFactory) new ServiceManager()
                .locateService("com/ibm/websphere/bo/BOFactory");
        DataObject customerInfoBO = boFactory.create(
                "http://mm1", "CustomerInfo");

        customerInfoBO.setString("phone", email);
        customerInfoBO.setString("address", name);

        return customerInfoBO;
    }
}
```

Java components can also invoke other SCA components, as long as the composition within a mediation module contains a reference from them to that component. A sample snippet of code showing how this might be done is given in Example 3-4.

Example 3-4 Sample code

```
import com.ibm.websphere.bo.BOFactory;
import com.ibm.websphere.sca.Service;
import com.ibm.websphere.sca.ServiceManager;
import commonj.sdo.DataObject;

...

ServiceManager serviceManager = new
com.ibm.websphere.sca.ServiceManager();
Service service = (Service) serviceManager
        .locateService("OtherJavaComponentToInvoke");
BOFactory boFactory = (BOFactory) serviceManager
        .locateService("com/ibm/websphere/bo/BOFactory");
```

```
DataObject businessObjectToInvokeWith = boFactory
        .create("http://bonamespace", "BOName");
businessObjectToInvokeWith.setString("myfield",
        "myvalue");
DataObject replyBusinessObject = (DataObject) service.invoke(
        "myoperationname", businessObjectToInvokeWith);
```

Asynchronous invocations can also be made using the invokeAsync() method. As the response is asynchronous, it is obtained later using the invokeResponse() method. See the Javadoc™ API documentation for the Service class for more information:

http://publib.boulder.ibm.com/infocenter/dmndhelp/v6rxmx/index.jsp?topic=/com.ibm.websphere.wesb.wps1.doc/doc/com/ibm/websphere/sca/Service.html

Java components are sometimes also used to implement custom mediations. See 3.4.2, “Mediation flow components” on page 73, below for more information.

3.4.2 Mediation flow components

Mediation flow components are the primary facility provided in WebSphere Enterprise Service Bus to perform mediation. By mediation, we mean generalized operations on or to the message flowing through the system. For example, a message can be altered, logged, dynamically routed, and so on.

Restriction: WebSphere Integration Developer supports the creation of two types of SCA modules, called mediation modules and, simply, modules. The first type of module is primarily designed for deployment on WebSphere Enterprise Service Bus and for performing mediation. Therefore, the only types of SCA component it can contain are zero or one mediation flow components, and zero or more Java components. Exports must not be directly connected to Java components (see Figure 3-1 on page 65).

Modules are designed primarily for use with WebSphere Process Server and are more flexible. They support Java components, as well as the other component types described in 3.9, “Other SCA component types in the WebSphere family” on page 105. However, they *cannot* contain mediation flow components.

It is worth noting that mediation flows are also typically the only way to gain access to the headers exposed by various binding types, such as JMS and MQ. Other SCA component types (including Java components and those discussed in 3.9, “Other SCA component types in the WebSphere family” on page 105) do not expose these, and headers will not be preserved after passing through them. This is why a typical implementation pattern when using WebSphere ESB in conjunction with WebSphere Process Server is to make the external interface to the system (for example, an export with a JMS binding) a mediation module containing a mediation flow component, so that any transport-specific header processing or parsing can be carried out and the information preserved.

Mediation flow components contain a single *mediation flow* per operation defined on the interface attached to the component. For one-way operations there is a request flow, and for two-way operations there is a request flow and a response (reply) flow. Mediation flows comprise *mediation primitives* of different types. A single mediation flow is sometimes called a mediation. WebSphere Enterprise Service Bus provides a number of these, and they are designed to be generally useful to do typical mediation tasks. Mediation flows have entry (input) and exit (call out) points. Figure 3-7 is a simple model of these relationships.

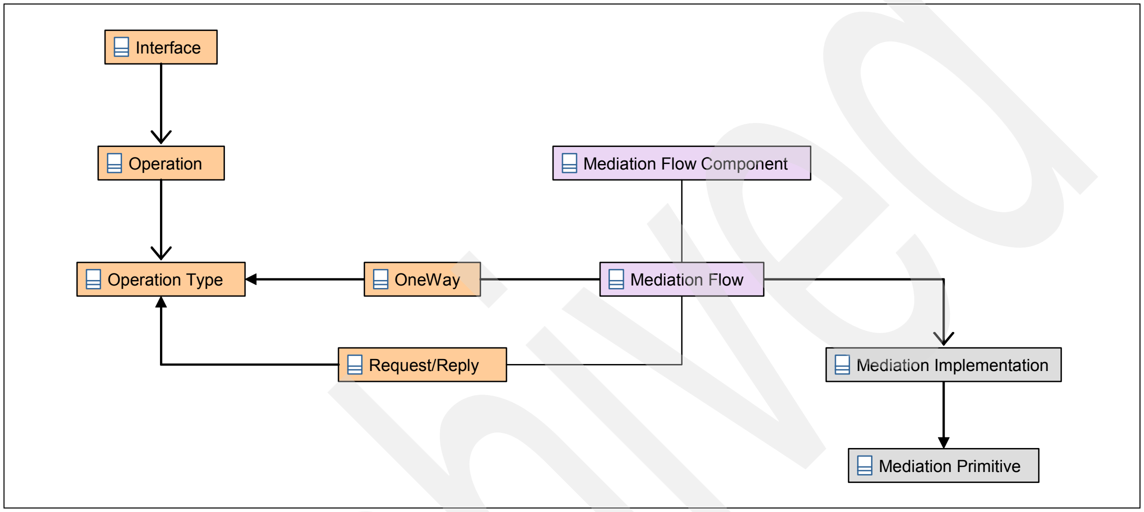


Figure 3-7 Simplified UML model of a mediation flow component

Figure 3-8 illustrates an example mediation flow as it would appear in WebSphere Integration Developer. The diagram illustrates a two-way operation. There are separate request and response flows for the operation. The request flow for operation1 is shown. The message comes in through the input node on the left-hand side of the diagram. The XSL Transformation primitive attempts to transform the message such that it conforms to the interface the callout node expects. If the transformation fails, the message is sent to a failure primitive. These mediation primitives are explained in more detail in “Mediation primitives” on page 77.

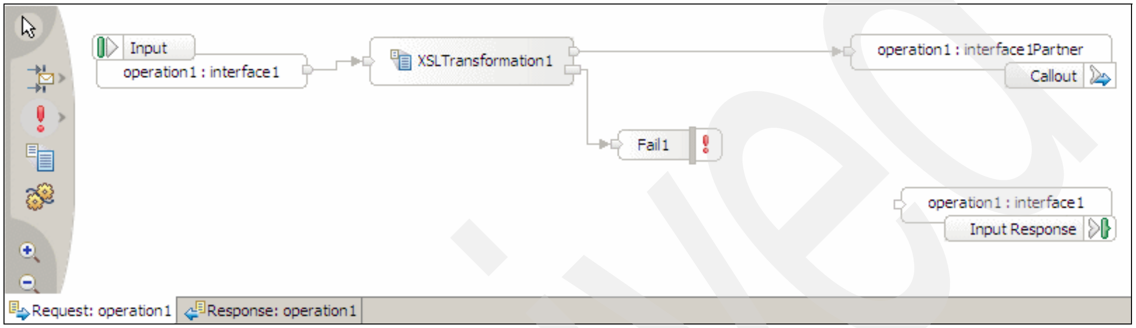


Figure 3-8 Example mediation flow

> **Attention:** Do not confuse the mediation facilities provided by WebSphere Enterprise Service Bus mediation flows with the mediation facilities provided by mediation handlers in WebSphere Platform Messaging. (See 3.7, “J2EE and WebSphere Application Server technology underlying WebSphere ESB” on page 99, for more information about how these relate.) Mediation handlers are sequentially processed via a list, rather than as a flow. However, they also operate at a different conceptual level from WebSphere Enterprise Service Bus mediation flows. They operate on the message directly. (So, for example, JMS message properties like message ID are preserved when using mediation handlers.) For comparison, using WebSphere Enterprise Service Bus mediation flows inside an SCA module involves transforming a message into an SDO at the export, then transforming it back when it leaves the import. This typically destroys properties such as JMS message ID.

Service Message Objects (SMOs) are the units of data transfer within a mediation flow. They flow from one mediation to another. SMOs are discussed in more detail in “Service Message Objects” on page 78.

Mediation primitives

The various mediation primitives that can be used within a mediation flow are:

- EXtensible Stylesheet Language Transformation (XSLT) - This mediation primitive provides a facility to map one message type to another, using a mapping defined using an XSLT file (WebSphere Integration Developer provides a tool to make constructing XSLT files simpler). It is quite powerful, but XSLT is often also a complex approach, in some cases, using the Message Element Setter may be more appropriate (see below).
- Message Element Setter - This primitive allows elements of a message to be set, deleted, or copied from elsewhere in the message. It is a simpler alternative to using the XSLT primitive.
- Message Filter - This primitive allows a message to be filtered based on part of its content, defined via an XPath expression. The message can be sent to different output terminals depending on its content.
- Message Logger - This primitive allows for part of a message to be logged to a database. The schema used for the table that the message is logged to is fixed. Typically, database create or update operations that require more flexibility should use a JDBC adapter. See 4.2.5, “Different types of WebSphere JCA Adapters” on page 115.
- Database Lookup - This primitive allows for information to be looked up in a database table and stored in the message structure, based on a key elsewhere in the message.
- Event Emitter - This primitive allows for events to be emitted via the Common Event Infrastructure (CEI).
- Endpoint Lookup - This primitive enables lookup of endpoints via the WebSphere Services Registry and Repository. It is typically used in conjunction with dynamic endpoint support (see “Dynamic endpoints” on page 81).
- Fail - This primitive forcibly causes a mediation flow to fail and roll back its transaction. The exact behavior is dependent on the module, but if, for example, the mediation flow component was called from a Web services export, the export would return a SOAP fault message.
- Stop - This primitive causes the message to be swallowed. No further processing is performed, but this is not considered a failure either. It is essentially a black hole.
- Custom - This primitive is primarily designed to allow Java code to be placed in a mediation flow, although it can also be used to invoke other SCA components. An API exposes the message structure such that it can be read and manipulated.

Attention: Do not confuse custom mediations with what we call roll-your-own mediations (which are occasionally also called custom mediations, and mediation primitive plug-ins in the Infocenter). Custom mediations can be implemented:

- Visually, using facilities exposed in WebSphere Integration Developer to do simple filtering and logic using a drag-and-drop interface.
- Using a snippet of Java code directly inside the mediation flow.
- By referencing a Java SCA component inside the same SCA module as the mediation flow.
- By referencing another type of SCA component or an import inside the same SCA module. The first possibility is only theoretical, because there are no other types of SCA component (apart from Java) available inside mediation modules. However, it is feasible to connect custom mediations to an import. Be aware that the module you are connecting to via the import will have to be designed to consume the root or the body of a SMO as applicable (see “Service Message Objects” on page 78 for more information).

Typically, custom mediations are implemented in Java, and are not suitable for re-use (although several custom mediations in the same flow can call out to the same Java SCA component).

Roll-your-own mediations (that is, mediation primitive plug-ins) are written entirely from scratch, and appear as a new first-class mediation type in the mediation flow editor in WebSphere Integration Developer. Writing one is beyond the scope of this book, but you can find more information here:

`http://publib.boulder.ibm.com/infocenter/dmndhelp/v6rxmx/index.jsp?topic=/com.ibm.wbit.help.sib.mediation.ui.doc/topics/rcustspi.html`

They are more appropriate when you need to provide a new generic mediation primitive type.

Sometimes it can be hard to see where all the mediation primitives are in WebSphere Integration Developer. Look for small arrows that look like  or  in the mediation primitive palette. They allow you to access more mediation types.

Service Message Objects

Service Message Objects (SMOs) are the unit of data transfer within a mediation flow. They flow from one mediation to another. They are a type of SDO. (SDOs are discussed in 2.2, “Service Data Objects” on page 59.)

Figure 3-9 is a model of an SMO, and an example is shown in Figure 3-10 on page 80.

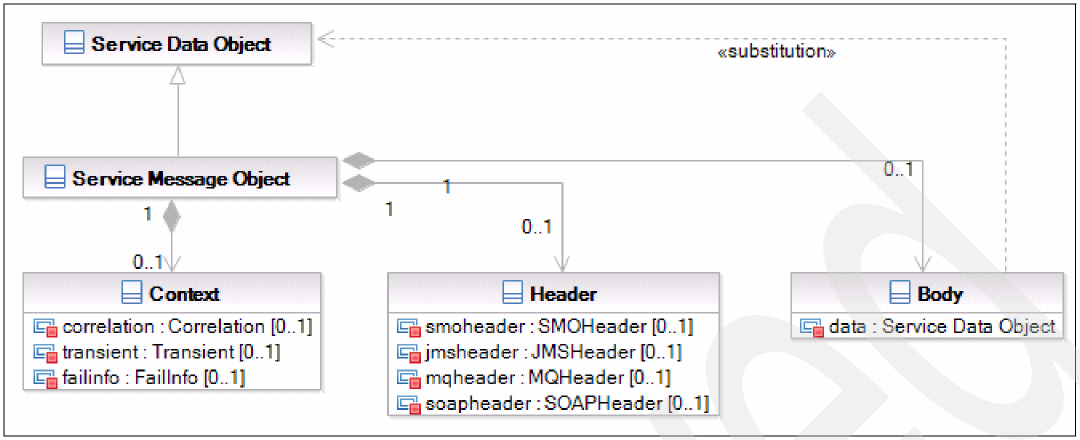


Figure 3-9 SMO structure

The structure is:

- Context. This contains three sub-sections:
  - correlation - This section can be used to store any information that you wish to be stored while the message flows through the mediation flow. It will also be persisted for two-way operations, so when the reply message is processed, the response flow automatically has access to the same correlation context as the request message did.
  - transient - This section is very similar to the correlation, but the contents are not persisted for two-way operations. The performance is slightly improved by keeping data here, so you should use it in preference to the correlation section unless you need the information to be persisted.
  - failInfo - This section is used by ESB to store information about failures, for example, if a message has left a mediation primitive via its fail terminal.
- Header. This can contain:
  - SMOHeader - generic information about this message, such as the operation name, and whether it is request, reply, or one-way
  - JMSHeader - JMS headers, if this message flow is connected to a JMS or MQ/JMS export
  - MQHeader - MQ headers, if this message flow is connected to an MQ export
  - SOAPHeader/SOAPFaultInfo - SOAP header and fault information, if this mediation flow is connected to a Web services export

- Body - This part of the SMO contains the actual message passed into the mediation flow component as an SDO

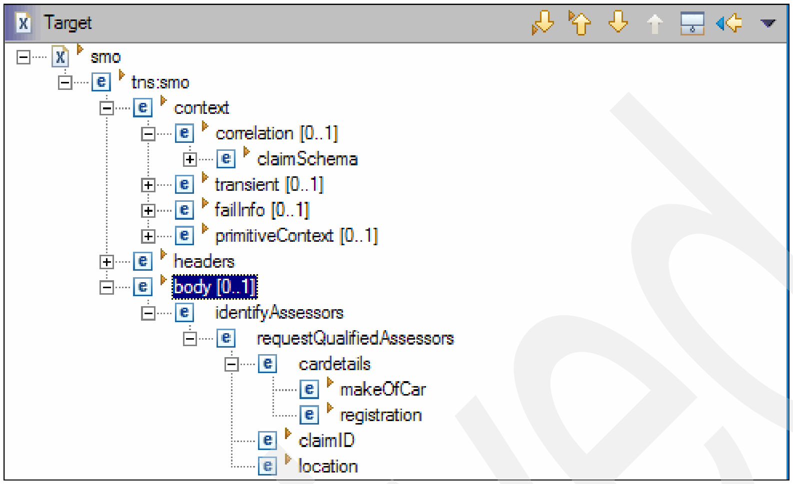


Figure 3-10 Example of SMO viewed in XSLT editor

> **Note:** The full details of the SMO structure are not covered above. Refer to this section of the WebSphere Enterprise Service Bus InfoCenter for more information:
>
> `http://publib.boulder.ibm.com/infocenter/dmndhelp/v6rxmx/index.jsp?topic=/com.ibm.wbit.help.6012.medprim.doc/ref/rwesb_SMOStructure.html`

SMOs are automatically created on entry to a mediation flow, with the incoming SDO being placed in the SMO body section. On call-out from the mediation flow, the body from the SMO is placed in the SDO that is used to invoke the next SCA component (or an import). The headers are also preserved, as long as the mediation flow component is connected directly to an import that understands those headers.

Because SMOs are Data Objects, they are hierarchical. Often an XML-based structure is used to represent them. Indeed, the Extensible Stylesheet Language Transformation (XSLT) primitive (which is discussed in "Mediation primitives" on page 77) operates by *flattening* the SMO into an XML document, applying an XSLT file to it, then reconstructing a SMO from the resultant XML document.

Because this XML representation is often used, XPath is used to describe parts of the tree structure. For example, if you had a mediation flow component that exposed the interface in Figure 3-8 on page 76, and you were defining the request flow, you might access the customer's name by using the following XPath expression:

```
smo/tns:smo/body/getCustomerInformationSimple/customer/name
```

The smo/tns:smo part will always appear, and is preamble. The body section means that we are interested in accessing the body. The getCustomerInformationSimple section refers to the operation on the interface. The customer section refers to the name of the parameter within that operation that we want to access, and the name section is the actual part of the business object we are interested in.

Often you will only want to work with the body of the SMO. The other parts will not interest you. For the mediation primitives where it is relevant (for example, the XSLT primitive), you can specify the root as either / or /body. If you select /body, every XPath is relative to the body. If the primitive in question is one where you have access to the data object (such as the Custom Java primitive), the DataObject you are passed is at the level of the body.

Example 3-10 on page 96 gives an example of how to work with a SMO in a Custom Java mediation.

Dynamic endpoints

As of WebSphere Enterprise Service Bus 6.0.2, dynamic endpoints are supported as part of the mediation flow component. This allows messages to be dynamically routed to different services. There is a target address section in the headers section of the SMO. If this section is filled in with an endpoint, and the Use Dynamic Endpoint property is set on the callout from the mediation flow (one of the right-most nodes shown in the mediation flow editor), then the target address is used for the callout rather than the component the mediation flow is wired to. Dynamic endpoint targets can look like:

- SCA Binding - sca://SCAModulename/exportName
- Web service Binding (SOAP/HTTP) - http://host:port/service
- Web service Binding (SOAP/JMS) - jms:/queue?destination=destinationName&connectionFactory=factory&targetService=service
- Import in this module (any type of import) - moduleName/importName

The dynamic endpoint logic works with the Endpoint Lookup mediation primitive, which fills in the target address section of the SMO. However, you can also do this with a Message Element Setter or another primitive.

3.5 WebSphere ESB Transport Binding Types

In this section, we cover the different transport binding types available. These are the bindings that are used with imports and exports to expose and call services from mediation modules. In other words, they allow interconnection with other services. Some transport bindings require the specification of data bindings. We discuss these where relevant.

3.5.1 SCA default bindings

SCA default bindings can also be thought of as SCA *native* bindings. They are the simplest type of binding to configure. For an SCA export, there is nothing to configure. For an SCA import, the only requirement is to specify the name of the module and SCA export that you wish to connect to at design time in WebSphere Integration Developer. SCA bindings should normally be used when two SCA modules are executing within the same SCA system and need to be connected.

> **Attention:** When we refer to an SCA system, in WebSphere Enterprise Service Bus terminology this is a *cell*. A cell is an administrative concept that groups together *nodes* and *servers*. Servers can also be *clustered*. These concepts are introduced by the WebSphere Application Server underlying WebSphere Enterprise Service Bus. If you are just experimenting with WebSphere Enterprise Service Bus (for example, you are using the Integrated Test Environment), then there is typically a 1:1:1 mapping between the cell, node, and server (you only have one of each), and therefore your cell covers only the server you are using. If you are using a more complex topology, which you typically would when a system is taken into production, you need to understand these concepts more fully. The IBM Redbooks publication *WebSphere Application Server V6 System Management & Configuration Handbook*, SG24-6451, may help:
>
> `http://www.redbooks.ibm.com/abstracts/sg246451.html`

3.5.2 Web services bindings

This type of transport binding allows call-out of SOAP-based Web services (via imports), as well as exposing SOAP-based Web services interfaces (via exports). Both SOAP/HTTP and SOAP/JMS are supported.

Both the HTTP and JMS transport types are presented as one binding type in WebSphere Integration Developer, but when you generate a Web service binding, you are asked which transport type you wish to use. SOAP/HTTP is more common than SOAP/JMSP.

As SOAP is well-defined and maps cleanly into the SCA and SDO model, there is no requirement (or facility) to provide any form of data binding, either for imports or exports[1].

The Web services bindings support the well-known encoding styles: document/literal, doc-lit-wrapped, RPC-literal, and RPC-encoded. For more information about these different styles, this article is excellent:

http://www-128.ibm.com/developerworks/webservices/library/ws-whichwsdl/

The bindings are also mostly compliant with the WS-I Basic Profile 1.1:

http://www.ws-i.org/Profiles/BasicProfile-1.1.html

3.5.3 JMS bindings

The JMS transport bindings allow inbound JMS communication (via exports) and outbound communication (via imports).

The JMS bindings in WebSphere ESB support *plain* JMS messages (that is, not SOAP). They work with the WebSphere Platform Messaging JMS provider (see 3.7, “J2EE and WebSphere Application Server technology underlying WebSphere ESB” on page 99, for more information). They also support third-party JMS providers such as Tibco, Sonic, and Oracle (See 3.7, “J2EE and WebSphere Application Server technology underlying WebSphere ESB” on page 99).

Both point-to-point (via queues) and publish/subscribe (messages sent or received on a topic) communication patterns are supported. Standard JMS terminology collectively refers to queues and topics as destinations.

Two-way point-to-point operations are supported by specifying:

- For JMS bindings on an export (incoming operations) - a receive queue for incoming messages (request) and a send queue for outgoing messages (reply)
- For JMS bindings on an import (outgoing operations) - a send queue for outgoing messages (request) and a receive queue for incoming messages (reply)

[1] SOAP/JMS is undergoing standardization in the manner that the SOAP JMS binding is serialized into the JMS header. It is important to test different implementations of SOAP/JMS interoperate or you may need to tweak the JMS header to some extent.

Tip: Sometimes people get confused about the names of these queues and accidentally switch them around. If you think about them from the perspective of the SCA module, it makes the naming clearer. Send destinations are used to send messages outward, receive destinations to receive messages inward. Figure 3-11 on page 84 may help you visualize the relationship.

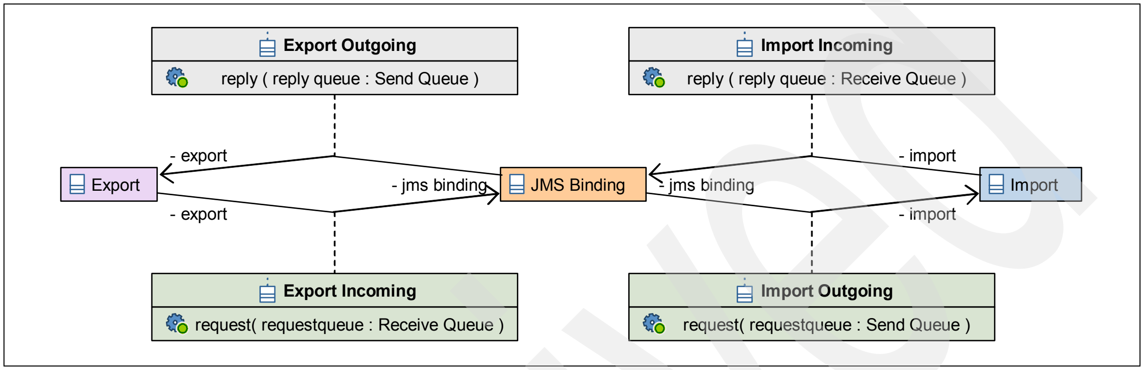


Figure 3-11 JMS binding request and reply queues

One-way operations require only one destination to be specified. Publish/subscribe can only be used with one-way operations.

Because JMS is a fundamentally unstructured protocol (inside the body of the message), data bindings have to be specified to transform JMS data from and to business objects. There are several data binding types supplied with WebSphere ESB, as show in Figure 3-12.

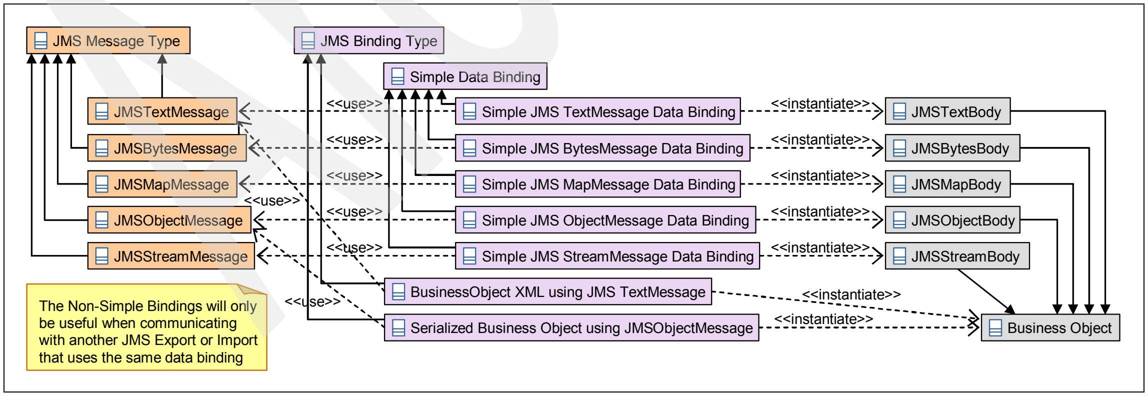


Figure 3-12 Deserializing JMS messages into business objects using predefined bindings

The objects are:

- Serialized business object using JMSObjectMessage - This serializes (or de-serializes) the business object in question using the standard Java serialization mechanism and places it in a JMSObjectMessage. It is typically only useful when communicating with another JMS import or export that uses the same data binding.
- Business object XML using JMSTextMessage - This flattens (or unflattens) the business object in question to XML and places it in a JMSTextMessage. Again, it is typically only useful when communicating with another JMS import/export that uses the same data binding, although it might be useful in limited circumstances where another application supports exactly this format.
- Simple Data Binding (one of TextMessage, BytesMessage, MapMessage, StreamMessage, ObjectMessage) - These data bindings provide the ability to automatically convert data binding from and to predefined business objects (called JMSTextBody, JMSBytesBody, JMSMapBody, JMSStreamBody, and JMSObjectBody, correspondingly). This is most useful when the body content is simple. For example, if a JMS message contains a single string, you can use the TextMessage simple binding, which generates a JMSTextBody business object. You can convert this simply to your real business object using an XSLT or some other mediation in a mediation flow.

Note: You gain access to the predefined business objects described above at design time by creating a library in WebSphere Integration Developer (which you can reference from any module). Then right-click the library and select **Open Dependencies**. Under Predefined Resources, select **Schema for simple JMS Data Bindings**. The business objects will appear in your library.

Frequently, you will want to write your own data binding, as typically you are connecting to a JMS application that does not provide data using one of the supported types. In order to do this, you would write a Java class to implement the interface com.ibm.websphere.sca.jms.data.JMSDataBinding. That Java class would then be specified as the data binding in the properties of the import or export in WebSphere Integration Developer.

Also, by default, if your incoming JMS message contains a TargetFunctionName header property, that is used as the native method name. This is because the JMS export uses the com.ibm.websphere.sca.jms.selector.impl.JMSFunctionSelectorImpl class by default as its function selector. The native method name is mapped to the operation name to invoke by the Method bindings section of the properties of the import or export.

If, for example, your messages do not contain that header, or you wish to select the native method name using a different method, you can write a custom JMS function selector, which implements the commonj.connector.runtime.FunctionSelector class.

It is worth being aware that sometimes this process is shortcut. If the function selector returns a native method name that is not mapped to an operation name, that native method name is used directly as the operation name. You should not rely on this behavior.

Combining some of these concepts into a simple example, imagine an export with a JMS transport binding. Let us reuse the interface shown in Figure 3-3 on page 69 (Figure 3-4 on page 70 and Figure 3-5 on page 70 show what the corresponding business objects look like). Imagine that the export connects to a service (perhaps via a mediation flow component or perhaps the Java component that we implemented in 3.4.1, "Java components" on page 69) that looks up a customer's address and phone number based on the customer's name and e-mail address. The data regarding the customer is sent into the export via a JMS text message, comma separated, and out of the binding (on the reply) with another JMS text message. In other words, the two text messages might look like:

- Request: John Smith,jsmith@somecompany.com
- Reply: 1234,Somewhere Street

The class design model for the CustomerJMSDataBinding will look something like Figure 3-13, with the read method to get the JMSTextMessage on the receive queue and the write method to put the JMSTextMessage onto the reply queue. The get and setDataObject methods make the data object accessible to the rest of the mediation.

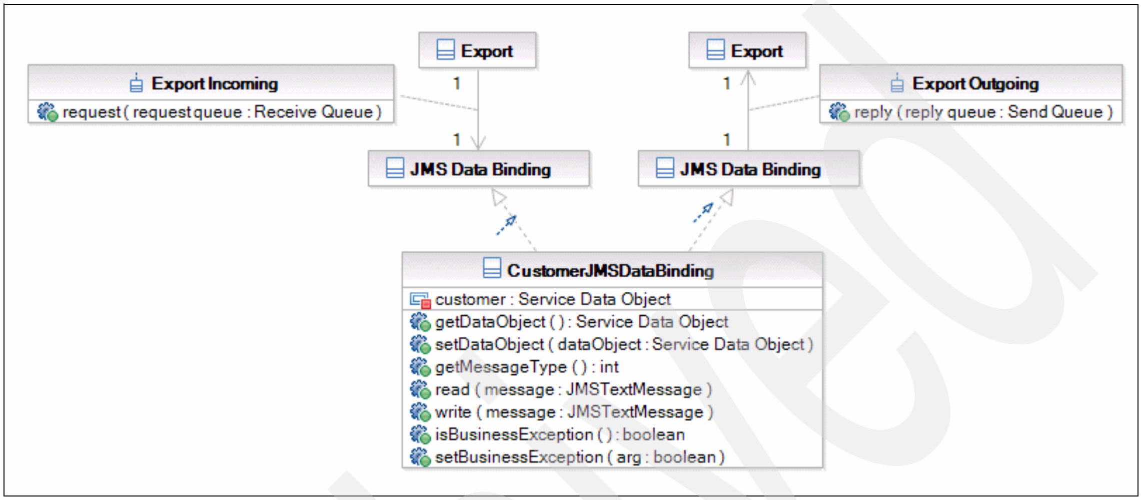


Figure 3-13 CustomerJMSDataBinding class model

An implementation of a data binding for the JMS export might look as shown in Example 3-5. The JMSDataBinding interface extends the DataBinding interface discussed in 3.3, "Data Bindings" on page 68, and provides two additional methods that must be overridden: read() and write(). read() is used to read in a JMS message, and is typically used with an inbound message on an export. write() is used to send out a JMS message, and is typically used with an outbound message on an import. These roles are reversed for a reply message. An instance of this data binding class is created for each message, so the class can use instance data (in this case the member variable dataObject) to share data between the read() and write() methods and the getDataObject() and setDataObject() methods.

Example 3-5 Example data binding to convert between JMS text message and business objects

```
package com.ibm.itso;

import java.util.StringTokenizer;

import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.TextMessage;
```

```
import com.ibm.websphere.bo.BOFactory;
import com.ibm.websphere.sca.ServiceManager;
import com.ibm.websphere.sca.jms.data.JMSDataBinding;
import commonj.connector.runtime.DataBindingException;
import commonj.sdo.DataObject;

public class CustomerJMSDataBinding implements JMSDataBinding {
    private DataObject dataObject;

    // Used with read()
    public DataObject getDataObject()
            throws DataBindingException {
        return dataObject;
    }

    // Used with write()
    public void setDataObject(DataObject dataObjectToSet)
            throws DataBindingException {
        dataObject = dataObjectToSet;
    }

    public int getMessageType() {
        return JMSDataBinding.TEXT_MESSAGE;
    }

    public void read(Message message) throws JMSException {
        String text = ((TextMessage) message).getText();

        StringTokenizer st = new StringTokenizer(text, ",");
        String name = st.nextToken();
        String email = st.nextToken();

        BOFactory boFactory = (BOFactory) new ServiceManager()
                .locateService("com/ibm/websphere/bo/BOFactory");
        DataObject businessObject = boFactory.create(
                "http://customer.bo.namespace", "Customer");

        businessObject.setString("name", name);
        businessObject.setString("email", email);

        this.dataObject = businessObject;
    }

    public void write(Message message) throws JMSException {
```

```
        String phone = dataObject.getString("phone");
        String address = dataObject.getString("address");
        ((TextMessage) message).setText(phone + "," + address);
    }

    public boolean isBusinessException() {
        return false;
    }

    public void setBusinessException(boolean arg0) {
    }
}
```

The implementation of the function selector in this case is hard-coded to just return the operation name we have chosen for the interface. A possible implementation is shown in Example 3-6.

Example 3-6 Example function selector to hard-code operation name

```
package com.ibm.itso;

import commonj.connector.runtime.FunctionSelector;
import commonj.connector.runtime.SelectorException;

public class HardcodedFunctionSelector implements FunctionSelector {
   public String generateEISFunctionName(Object[] arg0)
         throws SelectorException {
      return "getCustomerInformationSimple";
   }
}
```

3.5.4 MQ native and MQ/JMS bindings

The MQ and MQ/JMS Bindings provide a way to communicate directly with WebSphere MQ, via both exports (incoming) and imports (outgoing). The MQ bindings communicate with WebSphere MQ via native MQ messages. The MQ/JMS bindings do the same, but with JMS messages. These bindings are new for WebSphere Enterprise Service Bus 6.0.2, and are designed to provide a direct replacement for the WebSphere MQ Link functionality that is still available as part of the WebSphere Application Server platform. They are more straightforward to configure and are generally recommended over MQ Link (which we do not cover in this book).

These bindings also provide a convenient way to communicate with WebSphere Message Broker because Message Broker also provides the ability to exchange messages with WebSphere MQ (and is tightly bound with it).

Both of these bindings are similar to the JMS binding described above in 3.5.3, “JMS bindings” on page 83. The MQ binding only supports point-to-point messaging (even though WebSphere MQ itself supports publish/subscribe messaging), but the MQ/JMS binding also supports publish/subscribe messaging. (As with the JMS binding, this is for one-way operations only.)

In a similar manner to the JMS bindings, receive and send destination names are specified, but these correspond to destinations on a WebSphere MQ queue manager. Details such as the queue manager name and the host name for the queue manager must also be specified.

The MQ/JMS bindings share most aspects in common with the JMS bindings. They have the concepts of function selectors and data bindings, and these are handled in exactly the same way as we discussed previously. The pre-supplied data bindings are also exactly the same — the same types as discussed in 3.5.3, “JMS bindings” on page 83.

The MQ native transport bindings use a different type of data binding from the JMS bindings. The data binding must implement the interface com.ibm.websphere.sca.mq.data.MQBodyDataBinding. As with the JMS data binding, this extends the interface commonj.connector.runtime.DataBinding. The function selector must implement the class com.ibm.websphere.sca.mq.selector.MQFunctionSelector.

There are four MQ native data bindings supplied with WebSphere ESB:

- Unstructured Text Message - This operates in the same way as the Simple Text Data Binding does for JMS messages. It expects and returns a JMSTextBody business object. See 3.5.3, “JMS bindings” on page 83, for more information.
- Unstructured Binary Message - This operates in the same way as the Simple Bytes Data Binding does for JMS messages. It expects and returns a JMSBytesBody business object. See 3.5.3, “JMS bindings” on page 83, for more information.
- Serialized as XML - This data binding serializes (or deserializes) the business object from and to XML, and stores that in the body of the MQ message.
- Serialized Java Object - This data binding serializes (or deserializes) the business object from and to a Java ObjectMessage, which is stored in the body of the MQ message.

Attention: Do not be confused by the use of the JMSTextBody and JMSBytesBody business objects with the Unstructured Text Message and Unstructured Binary Message bindings. The MQ native bindings have no relationship with the MQ/JMS or JMS bindings. These business objects are simply re-used for this purpose because they have the same structure — a single string in the case of JMSTextBody, or a single hexBinary in the case of JMSBytesBody.

We illustrate a custom MQ data and body binding using the same example used in 3.5.3, “JMS bindings” on page 83, except that this time the data is contained within an MQ message. The data binding looks as in Example 3-7.

Example 3-7 MQ data binding for example scenario

```
package com.ibm.itso;

import java.io.IOException;
import java.util.List;

import com.ibm.mq.data.MQDataInputStream;
import com.ibm.mq.data.MQDataOutputStream;
import com.ibm.websphere.bo.BOFactory;
import com.ibm.websphere.sca.ServiceManager;
import com.ibm.websphere.sca.mq.data.MQBodyDataBinding;
import com.ibm.websphere.sca.mq.structures.MQMD;
import commonj.connector.runtime.DataBindingException;
import commonj.sdo.DataObject;

public class CustomerMQBodyDataBinding implements
        MQBodyDataBinding {
    public void read(MQMD arg0, List arg1, MQDataInputStream arg2)
            throws IOException {
        String name = arg2.readUTF();
        String email = arg2.readUTF();

        BOFactory boFactory = (BOFactory) new ServiceManager()
                .locateService("com/ibm/websphere/bo/BOFactory");
        DataObject businessObject = boFactory.create(
                "http://customer.bo.namespace", "Customer");

        businessObject.setString("name", name);
        businessObject.setString("email", email);

        this.dataObject = businessObject;
```

```
    }

    public void write(MQMD arg0, List arg1,
            MQDataOutputStream arg2) throws IOException {
        String phone = dataObject.getString("phone");
        String address = dataObject.getString("address");
        arg2.writeUTF(phone);
        arg2.writeUTF(address);
    }

    public void setBusinessException(boolean arg0) {
    }

    public boolean isBusinessException() {
        return false;
    }

    public void setFormat(String arg0) {
    }

    public String getFormat() {
        return null;
    }

    // Used with write()
    public DataObject getDataObject()
            throws DataBindingException {
        return dataObject;
    }

    // Used with read()
    public void setDataObject(DataObject dataObjectToSet)
            throws DataBindingException {
        this.dataObject = dataObjectToSet;
    }

    private DataObject dataObject;
}
```

The function selector looks as in Example 3-8. This example is slightly more interesting than the one we illustrated for JMS in Example 3-6 on page 89. It shows how the function selector could vary the operation name dependent on the data in the MQ message received. If the name in the incoming data is Fred, it invokes 'getCustomerInformationSimpleFred'. Otherwise, it invokes 'getCustomerInformationSimple'. (The interface shown in Figure 3-3 on page 69 would have to be modified to add the 'getCustomerInformationSimpleFred' operation.)

Example 3-8 Example function selector for MQ transport binding

```
package com.ibm.itso;

import java.io.IOException;
import java.util.List;

import com.ibm.mq.data.MQDataInputStream;
import com.ibm.websphere.sca.mq.selector.MQFunctionSelector;
import com.ibm.websphere.sca.mq.structures.MQMD;
import commonj.connector.runtime.SelectorException;

public class SemiHardcodedMQFunctionSelector extends
        MQFunctionSelector {
    public String generateEISFunctionName(MQMD md,
            String bodyFormat, List headers, MQDataInputStream input)
            throws IOException, SelectorException {
        String name = input.readUTF();
        String email = input.readUTF();

        if(name.equals("Fred"))
            return "getCustomerInformationSimpleFred";
        else
            return "getCustomerInformationSimple";
    }
}
```

The MQ bindings also provide the ability to specify one or more header bindings. These are classes that implement the com.ibm.websphere.sca.mq.data.MQHeaderDataBinding class, and provide the ability to parse and create MQ header formats other than MQRFH or MQRFH2.

These headers are manifested in the SMO at headers/MQHeader/header[n]. (See 3.4.2, “Mediation flow components” on page 73, for more information about SMOs.) A header binding is responsible for translating between the data found in this section of the SMO and the external MQ representation of a header. This is done with the setDataObject(), getDataObject(), read(), and write() methods in a similar way to the JMS and MQ body data bindings. Example 3-9 shows an example of how a header binding might be implemented. In this example we do not actually read from or write to the data object. We simply use constant values. But in many cases, you would work with the data object in a similar way to previous examples.

Example 3-9 Example MQ header binding

```
package com.ibm.itso;

import java.io.IOException;

import com.ibm.mq.data.MQDataInputStream;
import com.ibm.mq.data.MQDataOutputStream;
import com.ibm.websphere.sca.mq.data.MQHeaderDataBinding;
import commonj.connector.runtime.DataBindingException;
import commonj.sdo.DataObject;

public class ExampleMQHeaderBinding implements
        MQHeaderDataBinding {
    private DataObject dataObject;

    public boolean isSupportedFormat(String format) {
        return format.trim().equals("MQMYFORMAT");
    }

    public void read(String format, MQDataInputStream input)
            throws IOException {
        String headerType = input.readMQCHAR4();
        long constantValue = input.readMQLONG();

        // At this point, you should set dataObject to be the data you
wish to
        // be exposed in
        // headers/MQHeader/header[n]
    }

    public void write(String format, MQDataOutputStream output)
            throws IOException {
        // At this point, dataObject contains the data exposed in
```

```
        // headers/MQHeader/header[n]

        output.writeMQCHAR4("MyHeaderType");
        output.writeMQLONG(5);
    }

    public void setNextFormat(String arg0) {
    }

    public String getNextFormat() {
        return null;
    }

    public void setNextCCSID(int arg0) {
    }

    public int getNextCCSID() {
        return 0;
    }

    public void setNextEncoding(int arg0) {
    }

    public int getNextEncoding() {
        return 0;
    }

    public DataObject getDataObject()
            throws DataBindingException {
        return dataObject;
    }

    public void setDataObject(DataObject arg0)
            throws DataBindingException {
        this.dataObject = arg0;
    }
}
```

It is important to be aware of the way that the isSupportedFormat() method is used in the example above. For each header found in the headers section of the SMO, each header binding specified on the MQ transport binding will have that method invoked in turn. If the method returns `true`, that header binding will be used for the header binding process. It is therefore important to implement this method correctly.

A collorary of this is that if you are trying to create an MQ header from scratch in the data binding (such as we do in Example 3-9 on page 94 in the write() method), you must have defined at least the skeleton of an MQ header in your SMO in order for the header binding to be invoked. You can create this skeleton with a custom Java mediation. A snippet of code is shown in Example 3-10 that shows how to do this.

Example 3-10 Creating MQHeader skeleton for custom header

```
DataObject doHeader = input1.getDataObject("headers");
DataObject mqHeader = doHeader.createDataObject("MQHeader");
DataObject myHeader = mqHeader.createDataObject("header");
myHeader.set("Format", "MQMYFORMAT");
myHeader.setLong("CodedCharSetId", 500L);
myHeader.setLong("Encoding", 785L);
BOFactory boFactory = (BOFactory) ServiceManager.INSTANCE
   .locateService("com/ibm/websphere/bo/BOFactory");
DataObject dO = boFactory.create(
   "http://my.name.space/", "MyBO");
myHeader.set("value", dO);
return input1;
```

Notice that the MQMYFORMAT format type in Example 3-10 corresponds to the format that the isSupportedFormat() method in Example 3-9 on page 94 checks for.

> **Attention:** `‘value’` in the SMO, as set in Example 3-10 above, must be a DataObject. It cannot be a simple type such as a string.

Chapter 8, “Custom CICS integration using WebSphere MQ” on page 215, explains in detail how to construct an MQ body and header data binding for integration with CICS.

3.5.5 Stateless session bean bindings

A stateless session bean binding allows invocation of an EJB™ stateless session bean from an SCA module. It can only be used as an import binding. Typically, you would only use this type of transport binding if a service you wanted to invoke was already written and available as an EJB stateless session bean. If writing the service from scratch, the preferred option would be to write it as a Web service and use the Web service binding. Existing stateless session beans could also be wrapped with a Web service interface and invoked with a Web service binding.

Stateless session bean bindings are typically not created directly, but indirectly by dragging a stateless session bean onto the assembly diagram in WebSphere Integration Developer.

The pros and cons of these approaches, together with more details, are discussed in the WebSphere Integration Developer InfoCenter:

http://publib.boulder.ibm.com/infocenter/dmndhelp/v6rxmx/index.jsp?topic=/com.ibm.wbit.help.migration.ui.doc/topics/toutejb.html

3.5.6 EIS bindings

EIS bindings are used in conjunction with adapters. They are described in more detail in "EIS Data binding" on page 115. Like stateless session bean bindings, they are typically not defined directly, but indirectly via the Enterprise Service Discovery process.

3.5.7 Stand-alone references

Stand-alone references are not strictly a binding type, but they perform a similar role, so we discuss them here.

Stand-alone references can be wired in an SCA module to SCA components to allow them to be invoked by non-SCA components. In that respect, they often take the place of exports with one of the binding types discussed above. A good example of a non-SCA component that might use a stand-alone reference would be a Java Servlet. We discuss this pattern further in 6.6.2, "Solution options" on page 165, and develop the pattern fully in 12.3.1, "Architecture pattern" on page 419. The non-SCA component needs to be packaged with the SCA module.

> **Tip:** In practice, in WebSphere Integration Developer, this would mean that a Dynamic Web Project that contained a servlet that you wanted to use in this way would have to be added as a dependency of the mediation module containing the SCA component you wanted to invoke. It should appear in the list of J2EE dependencies in the Dependency Editor for the mediation module.

3.6 Reference table of possible connection types

Table 3-1 Possible connection types

I want to connect to a...	I should use...
SOAP-Compliant Web service	Web service Binding
Non-SOAP-Compliant or unusual Web service available over HTTP	Proxy Pattern such as that discussed in 6.6, “Connection pattern 6: Web integration” on page 164
JMS Destination (Queue or Topic)	JMS binding
WebSphere MQ using Native Messages	MQ binding
WebSphere MQ using JMS Messaging	MQ/JMS binding
WebSphere Message Broker	MQ binding via WebSphere MQ
SCA Service	SCA binding
EJB	Stateless session bean binding
EIS such as CICS, IMS, JDEdwards EnterpriseOne, Siebel Business Applications, SAP Exchange	The appropriate WebSphere Adapter for your EIS: http://www-306.ibm.com/software/integration/wbiadapters/ CICS integration is discussed in greater detail in 6.1, “Connection pattern 1: integration” on page 144. Other methods such as CICS over MQ, SAP, and Siebel integration are discussed in greater detail in 6.2, “Connection pattern 2: packaged application integration” on page 149.
Another EIS	Built-in binding type if appropriate, or a custom adapter. For more information about custom adapters, the following book is useful: http://www.redbooks.ibm.com/redpieces/abstracts/sg246387.html A WebSphere Business Integration Adapter might also be appropriate.
CORBA Component	Use WebSphere Business Integration Adapter for CORBA

I want to connect to a...	I should use...
Data in a JDBC-compliant database	WebSphere Adapter for JDBC
Data in flat files	WebSphere Adapter for Flat Files
Data in flat files accessible via FTP	WebSphere Adapter for FTP
.Net Application	Use SOAP/Http or the WebSphere Message Service Client for .Net (See 3.8, "Using clients with WebSphere ESB" on page 103, for more information.)

3.7 J2EE and WebSphere Application Server technology underlying WebSphere ESB

Reader: It is not strictly necessary to understand the material in this section to use WebSphere Enterprise Service Bus. However, you may find that it improves your understanding.

The underlying implementation of a WebSphere Enterprise Service Bus mediation module (and indeed a Module in WebSphere Process Server) is a Java 2 Enterprise Edition (J2EE) application. In fact, WebSphere Enterprise Service Bus is based on WebSphere Application Server Network Deployment, a J2EE application server.

Typically, in WebSphere Integration Developer, you will work in the Business Integration perspective. This is the default. In this perspective, you will see that your mediation module projects appear listed in the Business Integration view, as shown in Figure 3-14.

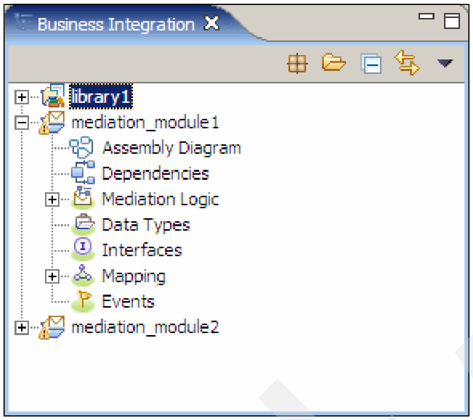


Figure 3-14 Example of Business Integration view from WebSphere Integration Developer

However, if you are familiar with WebSphere Application Server, you may be used to working in the J2EE perspective. If you switch to the J2EE perspective, you will see your mediation module projects listed in the Project Explorer, as shown in Figure 3-15.

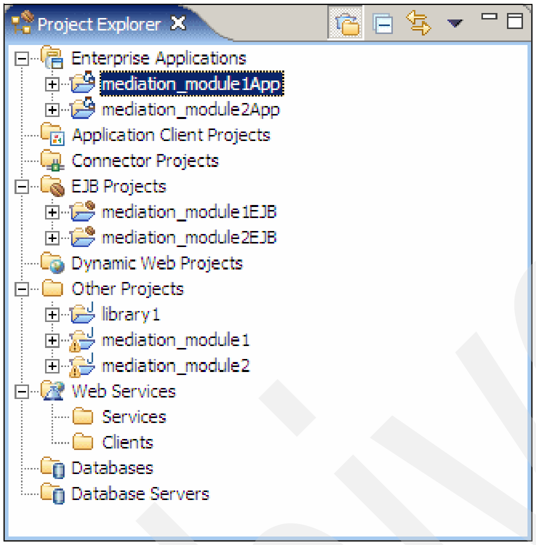


Figure 3-15 Example of the Project Explorer view from the J2EE perspective in WebSphere Integration Developer

The projects that have names ending in App and EJB (you may see other suffixes also) are automatically generated from your mediation module project whenever it is changed. Because they are automatically generated, we do not normally recommend modifying them directly, as your changes will be overwritten as soon as they are rebuilt. However, it is sometimes useful to be aware that they underlie your mediation module. If you are exporting mediation modules from WebSphere Integration Developer for deployment, you should always use the **File** → **Export** → **Integration Module** → **EAR Files for Server Deployment** option. This ensures that the application is exported correctly and should ensure that it can be installed correctly. Never attempt to export projects directly from the J2EE perspective using another export type.

Tip: As an aside, you should also use **File → Export → Integration Module → Project interchange for sharing modules between workspaces** if you wish to create a Project Interchange file (PIF). A PIF should be used if you wish to exchange projects between workspaces or send them to someone else. Using this option in preference to the more apparent **File → Export → Project Interchange** makes sure that only appropriate integration projects are included, as well as giving you the option to include appropriate dependent projects (*which you must normally select*).

Also, because WebSphere Enterprise Service Bus is based on WebSphere Application Server, it contains the WebSphere Platform Messaging functionality (also called SIBus), which means that it contains an embedded JMS provider, referred to as *default messaging*. Typically, this is what the JMS bindings described in 3.5.3, "JMS bindings" on page 83, would communicate with.

There is also a WebSphere MQ messaging provider. This provides some functionality to communicate with WebSphere MQ. You will need to work with this administratively if you use the MQ Bindings (described in 3.5.4, "MQ native and MQ/JMS bindings" on page 89), particularly if you specify to use JNDI names.

Finally, since v6.0.2.1 of WebSphere Enterprise Service Bus it is possible to use third-party JMS message providers. The article by Kevin Barker, et al, at `http://www-128.ibm.com/developerworks/websphere/library/techarticles/0702_barker/0702_barker.html` describes how to do this for the following providers,

- Tibco EMS
- Sonic MQ
- Oracle AQ

You will also find a V5 default messaging JMS provider listed in some places. This is historic and should not be used.

You can obtain more information about J2EE and WebSphere Application Server here:

`http://www-128.ibm.com/developerworks/websphere/library/techarticles/0305_issw/recommendedreading.html`

Attention: WebSphere Enterprise Service Bus 6.0.2 is based off WebSphere Application Server 6.0.2, and WebSphere Enterprise Service Bus 6.0.1 is based off WebSphere Application Server 6.0.1. WebSphere Application Server 6.1 is the most recent version of WebSphere Application Server.

3.8 Using clients with WebSphere ESB

Figure 3-16 shows the various client types available with WebSphere ESB and how they interact with it. (Note that the diagram only shows one-way operations for clarity, but request/reply operations are supported.)

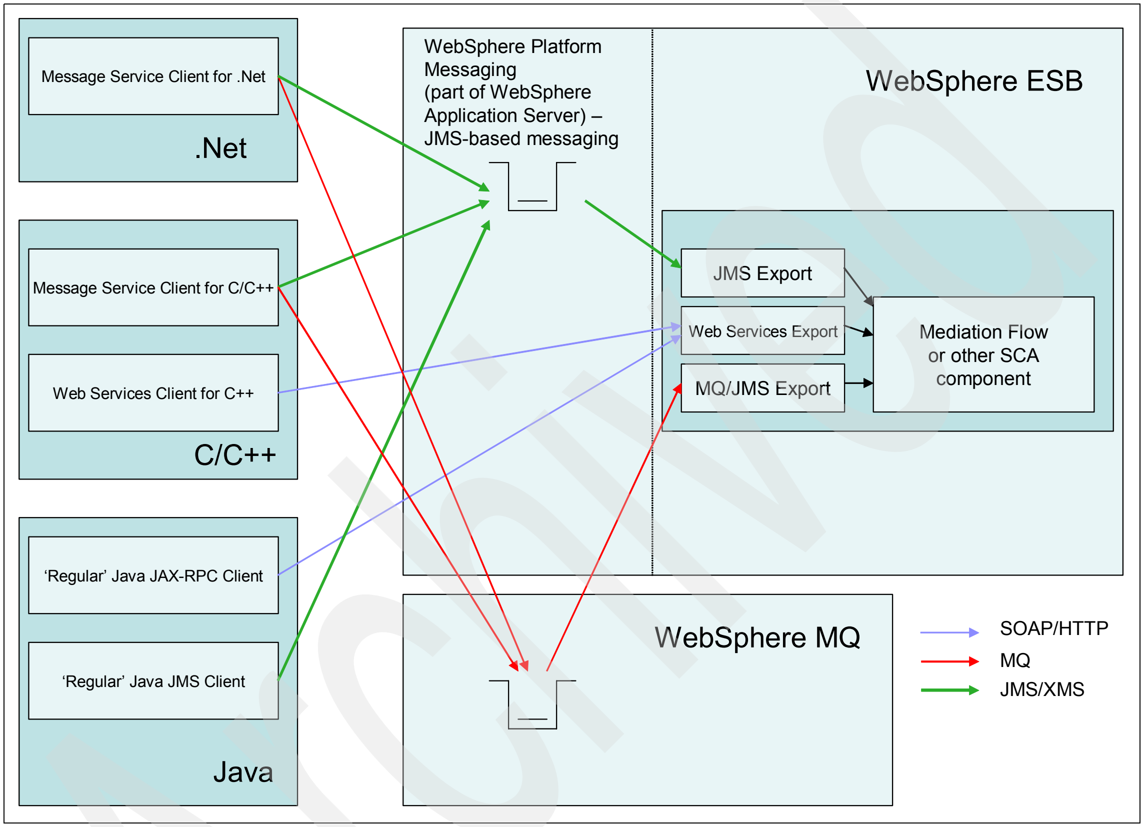


Figure 3-16 Client software supplied with WebSphere ESB and how they interact with it

As with WebSphere Application Server, you can write regular Java Web service clients using the JAX-RPC API, and you can write Java messaging clients using the JMS API. The former can call into Web services exposed by an ESB Web service export, and the latter can interact with destinations defined on ESB JMS exports.

However, WebSphere Enterprise Service Bus is also supplied with three additional pieces of client software:

- IBM Message Service Client for C/C++
- IBM Message Service Client for .Net
- IBM Web services Client for C++

Each of these provides tools and development libraries to enable writing certain types of client in programming languages other than Java.

> **Attention:** If you plan to use these additional clients, bear in mind that they are installed separately from CD 2 in your installation package. They are not part of the standard server install.

3.8.1 XMS clients

There are two XMS clients:

- IBM Message Service Client for C/C++ (also called the XMS C/C++ Client)
- IBM Message Service Client for .Net (also called the XMS .Net Client)

The XMS clients provide a mechanism to communicate with WebSphere Enterprise Service Bus via JMS. Although JMS is a Java-only API, XMS is an API that provides a JMS-like API in other languages (in this case, in C, C++, and .Net). It can communicate with WebSphere MQ and, of most interest to us, WebSphere Platform Messaging, the embedded JMS provider for WebSphere Application Server (and, hence, WebSphere Enterprise Service Bus). The XMS API will be familiar to any developer who is familiar with the JMS API.

Typically, you would use these Message Service Clients with the JMS transport binding on imports or exports in WebSphere ESB.

For more information about the Message Service Clients, visit this URL:

http://www-128.ibm.com/developerworks/websphere/library/techarticles/0509_phillips/0509_phillips.html

3.8.2 Web services client

The IBM Web services client for C++ is a JAX-RPC-like environment for communicating with Web services, but designed for use with C++ rather than Java. It is specifically designed for working with the Web services functionality provided by WebSphere Application Server and WebSphere Enterprise Service Bus.

Typically, you would use the Web services client with the Web services transport binding on imports or exports in WebSphere ESB.

For more information about the Web services Client, visit this URL:

http://publib.boulder.ibm.com/infocenter/iseries/v5r3/index.jsp?topic=/rzamj/rzamjwebservicesintro.htm

3.9 Other SCA component types in the WebSphere family

WebSphere Process Server, which is a functionality super-set of WebSphere Enterprise Service Bus, provides several additional SCA component types. We briefly describe them here, but details on their use is beyond the scope of this book.

- Interface map - This allows mapping from one SCA interface to another. This includes the business object mapping facility.
- Selector - This allows you to select between multiple SCA components, primarily for versioning purposes.
- BPEL - This provides the ability to execute Business Process Execution Language processes.
- State Machine - This allows modelling of event-driven business scenarios.
- Rule Group - This provides the ability to control the timing and implementation of business rules.
- Human Task - Human tasks provide the ability to expose part of a service flow as a task to be done by a human.

For more information, visit the WebSphere Integration Developer InfoCenter:

http://publib.boulder.ibm.com/infocenter/dmndhelp/v6rxmx/index.jsp?topic=/com.ibm.wbit.help.nav.doc/topics/welcome.html

4

Adapters

Connectivity is an important part of integrating existing IT assets into an SOA. Application servers and buses that make up the SOA include ready-built connectivity functionality. Using the functionality described in Chapter 3, “Connecting to the WebSphere Enterprise Service Bus” on page 63, sometimes IT assets can be connected directly to the enterprise service bus and the connections customized to make IT assets available as business objects. But sometimes there is no obvious way to connect an IT asset directly to the service bus, and an adapter might be used. Sometimes, for popular application packages, both direct connection or an adapter are possible, and you need to decide which approach makes more sense.

Adapters extend and compliment the capabilities of our servers and buses. Adapters are part of the SOA connectivity layer and are often seen as a legacy connectivity capability.

In this chapter we define what an adapter is and look at the different types of adapters and different technologies that an adapter can use. Then we discuss how an adapter can be leveraged to be used in a service-oriented architecture to provide the connectivity to and from the ESB.

This chapter includes the following topics:

- Why use an adapter
- WebSphere JCA adapters
- WebSphere Business Integration adapters

4.1 Why use an adapter

Adapters are one of the many ways to link an enterprise information system (EIS) into a service-oriented architecture. In Figure 4-1 we show different solutions to connect an SOA to an EIS.

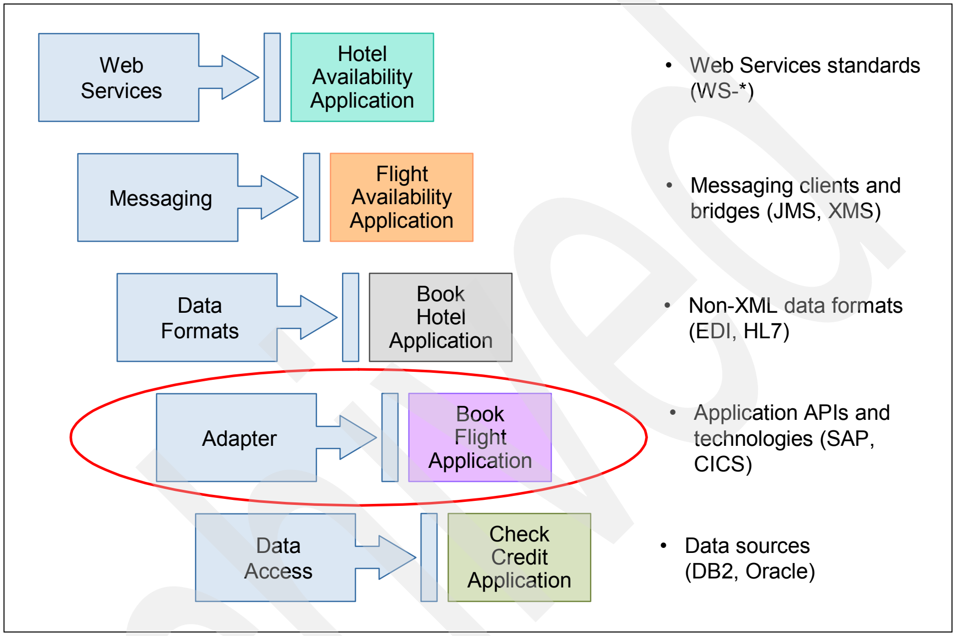


Figure 4-1 Adapters are one of many ways to link into your SOA

The different solutions are:

- Web services, when available, are usually easy to use. However, they are not always available, as a lot of existing older applications are not fully Web services enabled, and may not be enabled soon. Also, the quality of service may not be as desired. There may be performance, security, or transactional requirements that are not met by a Web service.
- Messaging provides an architecturally sound and consistent approach. However, in some cases it involves a lot of configuration in the application and in others, some of the depth of functionality/bus-logic is inaccessible via messaging (or limited).
- Data formats are targeted transformations, and are sometimes available as a package specifically tailored to an integration bus or server.
- Direct Data Access is often used as part of an EAI solution or as part of an data focused solution (see 1.1.2, “Data-focused integration” on page 4).

Whether it is an appropriate choice in an EAI solution depends a lot upon architectural questions to do with data-coupling and quality of service.

- Adapters abstract or expose the low-level EIS functions or events in the form of a service. This enables service components to interact with EIS passing business objects, invoking service operations. It saves the service component having to deal with the low-level APIs and events of the EIS, and it provides a decoupling layer between the EIS and the integration solution.

Adapters simplify the work of the application developer by making EIS functions available in a consistent way as services.

There are two categories of adapter that we discuss in this chapter:

- WebSphere JCA adapters
- WebSphere Business Integration adapters

Built with very different styles and architectures, they use a different programing interface, and they do not provide the same quality of service, at the end, to the application using them. We discuss the differences in the following sections.

4.2 WebSphere JCA adapters

There are three types of WebSphere JCA adapters that we discuss in the following sections:

- Base JCA resource adapters
- WebSphere JCA adapters
- WebSphere custom JCA adapters

The base JCA resource adapters are shipped with Rational Application Developer and supported by WebSphere Application Server, to be used in pure J2EE applications using Enterprise Java Beans or Web artifacts like JSPs or servlets and deployed onto the base application server. Historically, IBM used to ship base JCA 1.0 adapters with WebSphere Studio Application Developer Integration Edition, configured to connect to specific technologies such as CICS, IMS, SAP, and other packaged applications. With Version 6.0, base JCA 1.0 and 1.5 adapters are shipped with Rational Application Developer (see 4.2.3, “Base JCA resource adapters” on page 113).

The second type, called WebSphere adapters, are JCA 1.5 adapters with IBM SCA extensions that allow them to be used in a service-oriented architecture. These adapters are pre-built to connect to specific technologies (such as JDBC or e-mail) or specific applications (such as SAP). These are shipped with WebSphere Integration Developer v6.0.2 (see 4.2.4, “WebSphere JCA Adapter” on page 114).

Finally, the third type, WebSphere custom JCA adapters, are custom built to connect to a non supported end system in a service-oriented application.

4.2.1 JCA background

What we call a JCA Adapter is an adapter based on an open standard: the Java 2 Platform Enterprise Edition (J2EE) Connector Architecture standard (JCA) Version 1.5.

http://java.sun.com/j2ee/connector/

JCA enables *managed bidirectional connectivity* and data exchange between EIS resources and J2EE components:

- Managed, because the JCA adapter leverages the capabilities of the J2EE container to provide quality of service to the J2EE application. (The J2EE server being in our case WebSphere Application Server.)
- Bidirectional, because since JCA 1.5, JCA resource adapter can send events to EIS but also EIS can emit events to the J2EE application through the JCA resource adapter.

The JCA specification defines a set of contracts to define the behavior of an adapter that is discussed in the following section.

The JCA specification also defines a non-managed mode, where the adapter does not run in the J2EE container (a mode that we do not discuss here).

4.2.2 JCA technical overview

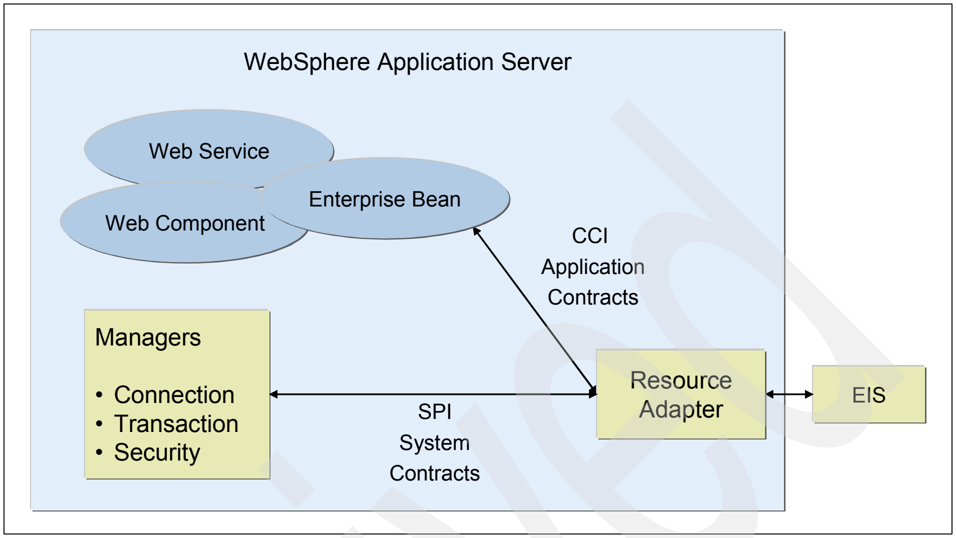


Figure 4-2 Adapter contracts

The JCA specifications define two main contracts that a resource adapter must implement:

- The application contract or *Common Client Interface* (CCI) defines the API through which a J2EE component such as an enterprise bean or a Web application accesses the EIS. This API is the only view that the component has of the EIS. The resource adapter itself and its system contracts are transparent to the J2EE component.
- The system contracts or *Service Provider Interface* (SPI) link the resource adapter to important run-time services like connection life cycle management, transaction, and security management that the J2EE server implements.

The connection life-cycle management contract provide *connection pooling* to the application for best performance and scalability when accessing the EIS. It also provides connection management (establishment, drop, recycle). These capabilities are transparent to the application that relies on the adapter to provide the necessary quality of service. The application simply gets a connection from the pool, uses it, and releases it back to the pool when done.

In JCA, the *transaction* management contract provides distributed transaction (XA). If the EIS and the resource adapter support XA transaction, then a global transaction enclosing other resource managers like databases or messaging transactions can also enclose the EIS resource adapter (like any other XA resource).

For *security*, authentication, authorization, and encryption of the communication are part of the system contract that provide security management to the application.

Managed connection factory

The resource adapter implements the managed connection factory, which in turn delegates connection management and life cycle to the J2EE application server.

The application uses the managed connection factory to get access to a managed connection to the EIS by doing a lookup using the Java Naming and Directory Interface™ (JNDI).

To be in a managed environment the connection must be accessed via the JNDI lookup, otherwise it is not a managed connection.

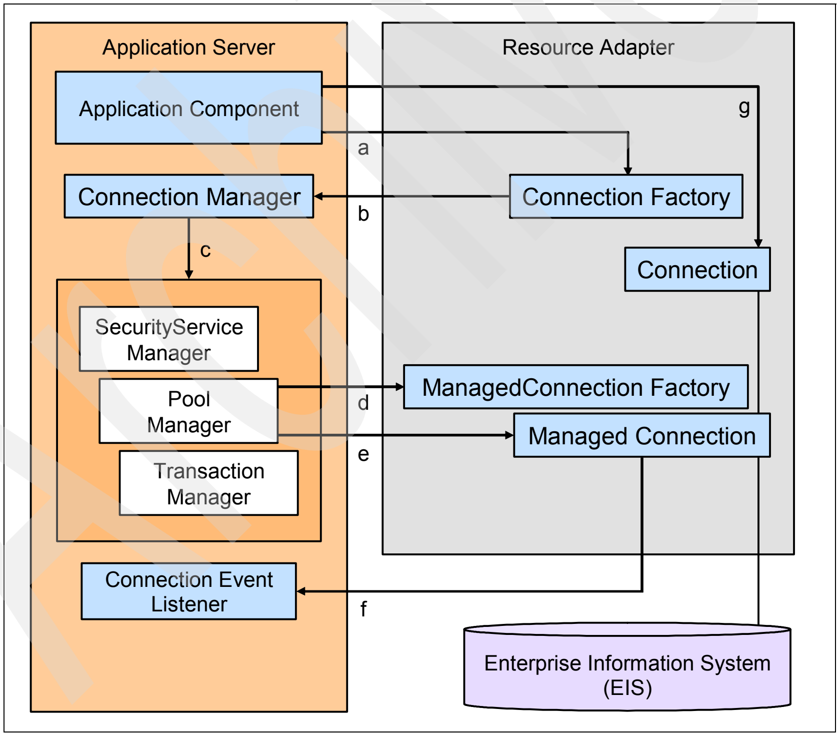


Figure 4-3 Adapter connection management architecture

Interaction specification

The interaction specification holds properties for conducting the interaction with an EIS instance. Every EIS has its own interaction model and the interaction specification is the place where this is stored. The interaction specification is used to control the execution of the specified function on the EIS. It has properties such as:

- Program to call
- Operation verb
- Transaction name
- SQL statement
- Time-out
- Synchronous or asynchronous control

The interaction specification is usually stored in a class provided by the resource adapter and generated by the tooling.

4.2.3 Base JCA resource adapters

Today, available in Rational Application Developer Version 6 as an optional J2C feature, Rational Application Developer Version 6 ships four base JCA resource adapters that can be used with WebSphere Application Server:

- CICS ECI Adapter 5.1 is a JCA Version 1.0 adapter and is only supported by WebSphere Application Server Version 5.0.2 and later.
- CICS ECI Adapter 6.0.1 is a JCA Version 1.5 adapter and is only supported by WebSphere Application Server Version 6.0 and later.
- IMS resource adapter 9.1.0.1.3 is a JCA Version 1.0 adapter and is only supported by WebSphere Application Server Version 5.0.2 and later.
- IMS resource adapter 9.1.0.2.2 is a JCA Version 1.5 adapter and is only supported by WebSphere Application Server Version 6.0 and later.

For more information about the CICS resource adapter, see:

http://publib.boulder.ibm.com/infocenter/radhelp/v6r0m1/topic/com.ibm.etools.j2c.cicseci.doc/topics/cresadap.html#ccicseci

For more information about the IMS resource adapter, see:

http://publib.boulder.ibm.com/infocenter/radhelp/v6r0m1/topic/com.ibm.etools.j2c.ims.doc/concepts/cimswhat.html#cimswhat

4.2.4 WebSphere JCA Adapter

WebSphere JCA Adapters are fully compliant with JCA 1.5 specification, but they differ from the base resource adapters because they are SCA enabled. They are built on top of the Service Component Architecture (SCA) framework to provide powerful and easy-to-use capabilities for the integration developer.

Enterprise Service Discovery

All WebSphere JCA adapters provide a very important Enterprise Service Discovery feature. Completely integrated into the WebSphere Integration Developer platform (WID), this feature is used from the development environment to inspect the remote enterprise information system (EIS) and generate:

- Business objects (XML schemas definitions: XSD files) representing the remote data in XML
- Service definitions (WSDL) representing the remote operations or remote transactions to be executed

Both are directly usable by the integration developer to build the integration solution and exchange data with the EIS by calling the operations defined in the WSDL files.

For the integration developer, everything appears as though it was any other business object or service definition. The integration developer does not need to know the details of the target system to be able to integrate with it.

EIS Data binding

WebSphere Adapters also provide enterprise information system (EIS) protocol binding support. The adapter can directly be used on the assembly diagram and assembled as any other import or export node, as shown in Figure 4-4.

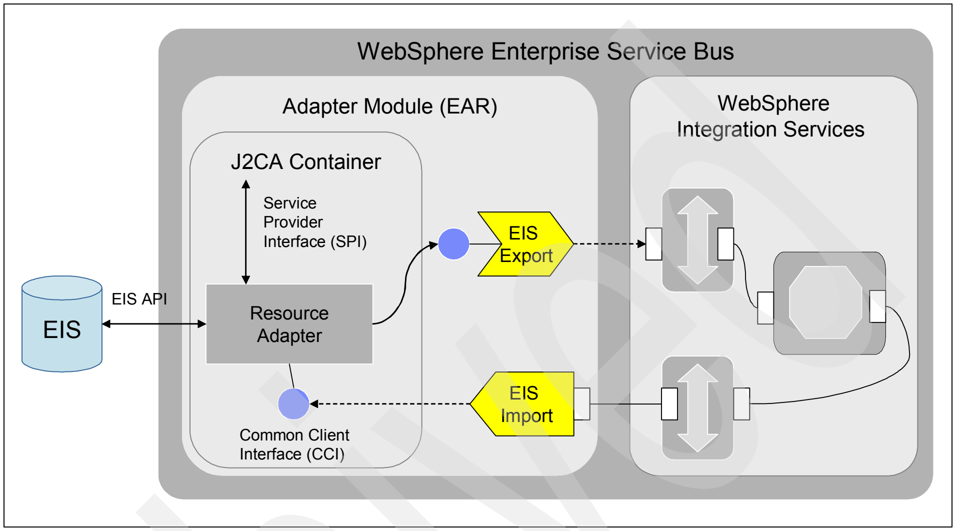


Figure 4-4 WebSphere JCA Adapter and the Service Component Architecture

The EIS binding exposes JCA interaction as a protocol on the Assembly diagram, at the Service Component Architecture level. Wiring enterprise information system destinations directly to the WebSphere Enterprise Service Bus is a valuable simplification of an EAI architecture.

4.2.5 Different types of WebSphere JCA Adapters

There is a distinction between two categories of WebSphere JCA adapter:

- Technology adapters
- Application adapters

In the following sections, we provide a list of all WebSphere JCA adapters and a technical overview for each. But to get more complete information about any WebSphere Adapters, visit the adapter infocenter at:

`http://publib.boulder.ibm.com/infocenter/dmndhelp/v6rxmx/topic/com.ibm.wsadapters602.doc/welcome_wsa602.html`

Technology adapter

Whereas an application adapter connects the enterprise service bus to a specific enterprise information system, a technology adapter is useful with many applications, and in many solutions, where the underlying EIS also has an interface to the same technology. Where there is no pre-packaged adapter, using a technology adapter is always worth considering. It may be more suitable for some simple integrations, and the technology adapters are available at no additional charge for the runtime with WebSphere Integration Developer. A word of caution though: Using technology adapters to connect to *unsupported* interfaces in EIS systems may leave you with an ongoing bill to maintain the interface you have chosen to connect to. Spending time to architect the technology interfaces may prove a good investment in the future.

A technology adapter wraps a technology into a JCA style of invocation, providing the JCA capability and quality of service when accessing resources belonging to the wrapped technology — to the extent that the wrapped technology also supports the qualify of service desired.

Technology adapters, like application adapters, expose the underlying EIS function as service using an EIS import into the ESB.

The JCA Technology adapters are part of WebSphere Adapters Version 6.0.2 and are packaged with WebSphere Integration Developer 6.0.2.

The following technology adapters can be deployed on a WebSphere Enterprise Service Bus runtime environment with no additional licence. The development version of the adapter can be deployed on the server without any additional cost:

- FTP

  The adapter for FTP connects WebSphere Enterprise Service Bus with remote file systems through an FTP server. The adapter connects to the FTP server to retrieve and write to the files. The FTP adapter supports the exchange of business data between remote file systems and the enterprise applications by connecting to the FTP server for retrieving and writing to the files.

- JDBC

  The adapter for JDBC communicates with an EIS that is a database provider. The resource adapter can be used with any database having a JDBC driver that supports the JDBC 2.0 or later specification. Examples of such databases include DB2, Oracle, Microsoft SQLServer, Sybase, and Informix®. The adapter for JDBC supports inbound and outbound processing.

  An example of how the JDBC adapter can be used to provide connectivity into the WebSphere Enterprise Service Bus is described in “Solution 3 - using the WebSphere JDBC Adapter” on page 159.

A practical example of how to use the JDBC adapter is shown in Chapter 11, “Event-driven integration using a JDBC adapter” on page 379.

- EMAIL

  WebSphere Adapter for Email enables enterprise information system connectivity across e-mailing systems. Because the adapter is equipped to handle both inbound and outbound communication with an application server, it can retrieve an e-mail from a mail server based on selective search criteria and then use the information in e-mails to trigger an application. Similarly, it can send a response to the e-mail addresses that initiate the operation.

- Flat File

  The Adapter for Flat Files connects the enterprise service bus with file systems running on an enterprise information system. For example, the J2EE component, when configured to work with the adapter, can create a file with specified contents in the EIS file system. The IBM WebSphere Adapter for Flat Files facilitates the exchange of business data in the form of delimited records in the event file between the file systems and the application. The adapter supports inbound and outbound operations.

Application adapter

Five application adapters are shipped with WebSphere Integration Developer 6.0.2 to connect WebSphere Enterprise Service Bus to five packaged applications. The runtime licence for deployment has to be purchased separately.

Services and data specific to the packaged application are made available to the J2EE application server through the bus and adapter and provide a means for a J2EE component and the remote application to interact.

The five adapters are:

- JD Edwards® EnterpriseOne

  WebSphere Adapter for JD Edwards EnterpriseOne enables bidirectional, real-time integration with JD Edwards EnterpriseOne applications.

- Oracle E-Business Suite

  WebSphere Adapter for Oracle E-Business Suite enables connectivity between the WebSphere system and the Oracle E-Business Suite for both outbound request processing and inbound events. It uses WebSphere Adapter for JDBC and Oracle E-Business Suite database mechanisms together for processing data.

- SAP Software

WebSphere Adapter for SAP Software connects to SAP systems running on SAP Web application servers. The adapter supports the Business Application Programming Interface (BAPI®) for outbound processing, Application Link Enabling (ALE) for both outbound and inbound processing, and the SAP Query Interface (SQI) for outbound processing.

- Siebel Business Applications

  WebSphere Adapter for Siebel Business Applications provides support for Siebel business objects, Siebel business components, and Siebel business services. The adapter supports both inbound and outbound operations.

- PeopleSoft Enterprise

  The WebSphere Adapter for PeopleSoft Enterprise provides support for all business objects in the PeopleSoft Component Interface Java API, PeopleCode methods associated with the underlying components and records, except searches and menu-specific processing options.

4.2.6 Custom WebSphere JCA Adapters

A custom JCA adapter can be developed as a WebSphere JCA compliant adapter using the unique services provided by the IBM WebSphere Adapter Toolkit.

It is possible to extend the generically implemented system contract classes to fit the needs of any custom adapter. The custom adapter can also make use of the built-in utility APIs to handle common adapter tasks, and significantly reduces the development time and effort to create an adapter.

IBM WebSphere Adapter Toolkit enables customers and business partners to develop brand new J2EE Connector Architecture (JCA) adapters to connect to information systems that are not supported by any adapter. The toolkit helps to create either a basic JCA 1.5 adapter or an adapter that leverages the additional capabilities of the Adapter Foundation Classes (SCA) and is utilized by WebSphere Adapters.

The Eclipse-based toolkit includes the following:

- A wizard to create a connector project including the Java code stubs for the appropriate adapter classes.
- The Adapter Foundation Classes that provide a consistent implementation for WebSphere Adapters based on JCA 1.5.
- A graphical Resource Adapter Deployment Descriptor Editor to ease creation and modification of the resource adapter deployment descriptor file.

- Two sample adapters and associated enterprise applications are provided with the source.
  - The Twineball sample adapter, which leverages the Adapter Foundation Classes, implements the Enterprise Metadata Discovery (EMD) specification for wizard-driven configuration and the Service Data Objects (SDO) specification for exchanging data.
  - The KiteString sample adapter, based directly on the JCA 1.5 and EMD interface specifications, implements the Enterprise Metadata Discovery (EMD) specification for wizard-driven configuration and the javax.resource.cci.Record interface for exchanging data.

IBM WebSphere Adapter Toolkit is provided as a no-fee download from IBM developerWorks® to customers and business partners who secure licenses to WebSphere Integration Developer and Rational Application Developer:

http://www-128.ibm.com/developerworks/websphere/downloads/wat/wat602.htm

For additional information about WebSphere Adapter development, including implementation through deployment, see *WebSphere Adapter Development*, SG24-6387-00:

http://www.redbooks.ibm.com/redpieces/abstracts/sg246387.html

4.3 WebSphere Business Integration adapter

WebSphere Business Integration adapters are not J2EE compliant. They are based on a messaging-oriented type of framework. These adapters have been the main-stay of IBM adapter strategy for the past few years. The adapters in this class are very mature and, other than not being J2EE compliant, they provide a very good integration solution, enabling the service view of an existing IT asset for which they are built.

In Figure 4-5 we highlight the main characteristics of WebSphere Business Integration adapter technology:

- WebSphere Business Integration Adapters are distributed. They reside outside of the application server, in a separate process called IBM WebSphere Business Integration Adapter runtime.
- The server runtime, or integration broker, communicates with the adapter through a Java Message Service (JMS) transport layer.

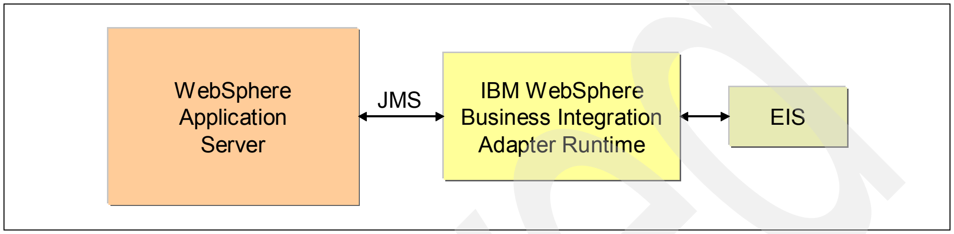


Figure 4-5 IBM WebSphere Business Integration Adapter

4.3.1 WBI Adapter

A WBI Adapter comes with three components:

- The IBM WebSphere Business Integration Framework, common to all WBI adapters, installed once for all adapters, used on the development platform as well as on the production server.
- The adapter-specific runtime that is different for each adapter. It needs to be installed on the development platform only if some tests must be performed.
- The System Manager: an Eclipse-based graphical user interface containing a set of tools used to manage, edit, display, build, test, and monitor a WBI adapter solution.

The System Manager is stand-alone and not connected to any WebSphere development platform. It contains a set of useful components like:

- ODA

  Equivalent to the Enterprise Service Discovery functionality that we discussed in “Enterprise Service Discovery” on page 114, the Object Discover Agent (ODA) is used to introspect the target system and build a repository of business objects (so the user does not have to build them manually).
- Connector Configuration

  This is used to build configuration files specific to each adapter instance; to drive the requests to and from the target system into the adapter; and to specify the user identification, names of queues, and name of business objects supported by the adapter.

- Business Object Designer

  An easy-to-use business object editor to build or edit and display business objects captured or not by the ODA.

- Adapter Monitor tool

  A monitoring tool useful to display messages that are stored on the queues between the adapter runtime and the enterprise service bus runtime.

4.3.2 How to use a WBI Adapter with the enterprise service bus

Once installed, the WBI Adapter needs to be configured using the system manager. Then the business objects can be created either automatically with the ODA or manually with the Business Object Designer.

Using the Enterprise Service Discovery tool of the WebSphere Integration Developer platform (WID), the business objects are loaded into the workspace together with the configuration file. Service definitions are then generated by the Enterprise Service Discovery based on the type of interaction selected (one way, two ways, inbound, or outbound).

At this point the service (WSDL) and the schemas definitions (XSD) are loaded in the integration platform, ready to be assembled in a mediation module, as shown in Figure 4-6.

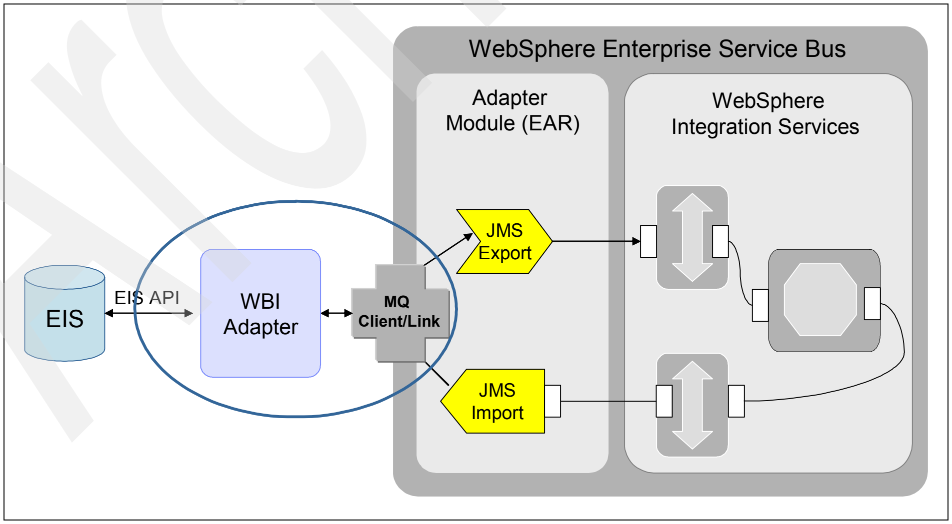


Figure 4-6 WebSphere Enterprise Service Bus and WBI Adapter Architecture

4.3.3 Quality of service of WBI Adapter

Like JCA 1.5 adapters, the WebSphere BI adapters are bi-directional, meaning that WBI adapters can initiate interactions with the enterprise service bus, send an event into the bus, as well as be called by the enterprise service bus.

Unlike JCA adapters, as they are based on an asynchronous transport (JMS), WBI Adapters do not provide the built-in security and transaction support the JCA adapters do.

4.4 Conclusion

In this adapter chapter, we looked at IBM WebSphere Adapters that provides two main classes of adapters:

- WebSphere Adapters

  These are designed around the Java Connector Architecture (JCA). They are standard. They integrate closely with WebSphere Enterprise Service Bus runtime. This is the IBM strategic direction for adapters.

- WebSphere Business Integration Adapters

  These adapters were acquired by IBM as part of a strategic acquisition some years ago. These have been the main-stay of IBM adapter strategy for the past five years. The adapters in this class are the most mature and most prevalent.

Other than being two logical classes of adapters, it is important to notice that IBM supports both classes of adapter. They both enable the Service view of an existing asset, and both extend the connectivity of the enterprise service bus.

IBM WebSphere Adapters are one of the best solutions to extend the connectivity of the enterprise service bus to the enterprise information system providing industry-ready first class adapter support. If a specific WebSphere Adapter is available to connect to your EIS you should look first to see whether it satisfies your needs. Only if it does not would you then go on and consider other options to connect to your EIS.

Building a custom WebSphere Adapter is certainly one option, but there are others that may be much quicker and cheaper to develop, especially if there is no requirement to package the solution as a reusable standardized adapter. In the next section of this book we take six typical business scenarios and develop alternative connectivity solutions for them using WebSphere ESB. We then take a small number of these solutions and show you how to implement the architectures using WebSphere Integration Developer 6.0.2.

Part 2

Scenarios and patterns

The next two chapters describe six WebSphere ESB connectivity scenarios from business and technical angles. In Chapter 5, “Business scenarios” on page 129, we describe six different business scenarios and in Chapter 6, “Connection patterns” on page 143, take each of those business scenarios and match them to six different connection patterns, and then discuss alternative ways of designing the solution.

As a team reasonably versed in EAI, we categorized different EAI connectivity patterns encountered when working with clients in a rough-and-ready empirical exercise. We intend this to help organize the discussion of different ways of using the WebSphere ESB to connect Enterprise Applications, and that by using the classification you can easily find a connection pattern that you can then apply to your current business integration engagement.

To help you further in trying to match the these connection patterns to business problems we have *reverse engineered* the connection patterns to six typical business problems that call for each of the solutions. Table 4-1, which is extracted from the next chapter for easy reference, links the business scenarios and connection patterns together.

If you are reading this online you will find that each cell in the table is hyperlinked to the sections in the text describing each scenario.

Table 4-1 Business scenario reference table

Industries	Business scenarios	Connection patterns
5.1, “Case 1: insurance claims and call center” on page 131	5.1.1, “Scenario 1: claimants’ access to claims information” on page 131	6.1, “Connection pattern 1: integration” on page 144
	5.1.2, “Scenario 2: insurance brokers’ access to customer policies” on page 133	6.2, “Connection pattern 2: packaged application integration” on page 149
5.2, “Case 2: banking and fraud management” on page 134	5.2.1, “Scenario 3: interaction with external fraud agency” on page 135	6.3, “Connection pattern 3: batch integration” on page 151
	5.2.2, “Scenario 4: interaction with packaged fraud application” on page 137	6.4, “Connection pattern 4: event-driven integration” on page 157
5.3, “Case 3: automotive spare parts” on page 138	5.3.2, “Scenario 6: integration of a third-party supplier” on page 141	6.5, “Connection pattern 5: custom application integration” on page 160
	5.3.1, “Scenario 5: integration of the dealer network” on page 140	6.6, “Connection pattern 6: Web integration” on page 164

As we state in the preface of this book, we have further described the business scenarios in terms of whether the solution is to be built by an Independent Software Vendor, a Systems Integrator on behalf of a client, or the client them self. Who is building the solution affects the choice of the technical solution, because, faced with a range of alternative solutions, the choice of the one to build is influenced by the future costs of both reusing the assets developed in creating the solution and of bearing the ongoing maintenance costs of the solution.

Patterns for e-Business

The approach we followed is modelled on that of the Patterns for e-Business (found at `http://www.ibm.com/developerworks/patterns/`) and expanded upon in a large number of IBM Redbooks in the *Patterns* series.

The Patterns for e-Business methodology is a means to select and justify the choice of a type of middleware architecture to meet a business need. The approach is to select the business pattern that most closely matches the business problem to be solved, such as a Self-Service Web application if one is designing a B2C Web site, then identifying the most suitable application pattern to meet the business requirements of the business problem. The next one identifies how to configure the application into a runtime pattern, and then how to select the appropriate middleware components to implement the runtime pattern.

Using the Patterns for e-Business approach, the patterns developed in this book are a variation on the following e-Business patterns:

- Business Pattern: Application Integration

  The business scenarios we identified can be thought of as variations of the general Application Integration business pattern:

  `http://www.ibm.com/developerworks/patterns/application/index.html`

- Application Pattern: Router: Broker Variation

  The application pattern we most closely match within Application Integration is Router: Broker variation:

  `http://www.ibm.com/developerworks/patterns/application/select-application-topology.html`

- Runtime pattern: [SOA]Router runtime pattern (that is, ESB runtime pattern)

  The runtime pattern is clearly closest to the Router runtime pattern:

  `http://www.ibm.com/developerworks/patterns/application/at4-runtime-router.html#soa`

This part of the book can be looked upon as an effort to extend these e-Business patterns into a more detailed analysis of business adapter patterns.

5

Business scenarios

This section looks at the integration challenges faced by an organization needing to connect different business applications. To illustrate these challenges, we introduce three fictitious companies, from three industry sectors:

- Insurance (ITSO-INS)
- Banking (ITSO-BANK)
- Wholesaling (ITSO-AUTO)

We describe their application integration requirements from a business perspective.

We also highlight constraints and requirements that may influence the solution in practice. We look at how the choice of solution may be influenced by the role of the integrator responsible for connecting the applications. We consider three roles:

- Independent Software Vendor (ISV)
- Systems Integrator (SI)
- Client building their own solutions

The ISV has developed its own packaged application that it resells either shrink wrapped or with a bundle of services to integrate it with customers' other applications.

The SI has developed a number of software assets that they r-use to build solutions in their service engagements.

The client runs an IT system with a mixture of package and custom applications.

The subsequent chapters offer practical solutions for each of the business scenarios. The solutions outlined vary in their depth and detail.

Table 5-1 1 links the sections describing business scenarios in this chapter with the technical scenarios in Chapter 6, “Connection patterns” on page 143.

Table 5-1 Business scenario reference table

Industries	Business scenarios	Technical scenarios
5.1, “Case 1: insurance claims and call center” on page 131	5.1.1, “Scenario 1: claimants’ access to claims information” on page 131	6.1, “Connection pattern 1: integration” on page 144
	5.1.2, “Scenario 2: insurance brokers’ access to customer policies” on page 133	6.2, “Connection pattern 2: packaged application integration” on page 149
5.2, “Case 2: banking and fraud management” on page 134	5.2.1, “Scenario 3: interaction with external fraud agency” on page 135	6.3, “Connection pattern 3: batch integration” on page 151
	5.2.2, “Scenario 4: interaction with packaged fraud application” on page 137	6.4, “Connection pattern 4: event-driven integration” on page 157
5.3, “Case 3: automotive spare parts” on page 138	5.3.1, “Scenario 5: integration of the dealer network” on page 140	6.5, “Connection pattern 5: custom application integration” on page 160 6.6, “Connection pattern 6: Web integration” on page 164
	5.3.2, “Scenario 6: integration of a third-party supplier” on page 141	

5.1 Case 1: insurance claims and call center

The ITSO-INS insurance company is a home, car, and small business insurance organisation. ITSO-INS has grown rapidly in size over recent years as a result of the acquisition of several niche organisations within its industry.

The ITSO-INS business model is to engage minimal direct sales employees, while relying heavily on external insurance brokers to sell its service. ITSO-INS also engages the external assessor network to appraise accident repair claims.

As part of this model, ITSO-INS has recently implemented a Web-based interface giving claimants the capacity to interact with ITSO-INS. Each claimant is assigned a unique claims number, user name, and password allowing claimants to view the status of their claims.

Although the company comprises many divisions, for the purpose of our discussion, we focus on the accident claims division and the customer service division.

5.1.1 Scenario 1: claimants' access to claims information

Through the process of acquisition, ITSO-INS has acquired separate client databases and the associated claimant adjudication systems. These systems are comparable in their function in that each system calculates the sum payable to the claimant. The systems have been developed to accept input from accident assessors based on the outcome of the accident assessment process. The final payment sum is calculated from this input.

ITSO-INS is moving towards the deployment of a new single adjudication system, but at present runs both new and old systems in parallel. This has resulted in a requirement for the development of a single user mechanism directing claimants' input to the appropriate adjudication system. (Figure 5-1 illustrates the multiple claims systems.)

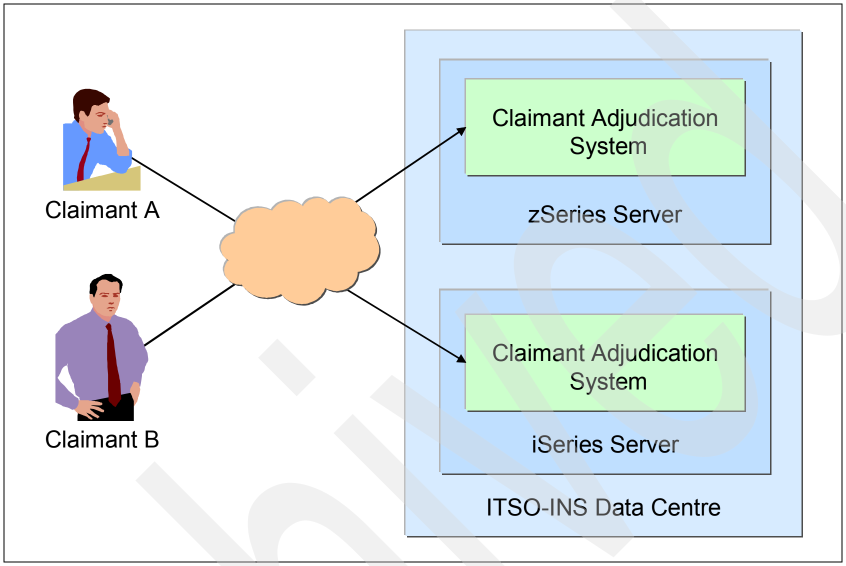


Figure 5-1 Multiple claimant adjudication systems

Business requirements

The identified business requirements include the following:

- To develop a mechanism to direct input from accident assessors to the appropriate adjudication system.
- Claimant's must have the ability to view their claims through this mechanism irrespective of which adjudication system their data is held.

Environment - ITSO-INS claimant adjudication systems

For the environment:

- Each adjudication system engine is operating in a separate environment. System 1 is operating a CICS system - IBM mainframes zSeries®. System 2 is an RPG-based system - IBM AS400 (iSeries®) system.
- For external access to the systems ITSO-INS has stipulated that this must be done via the Web-based transport protocol HTTP.

5.1.2 Scenario 2: insurance brokers' access to customer policies

ITSO-INS has implemented an SAP CRM customer system. This system processes customer policy information, while calculating commissions due to insurance brokers. The application is housed in the ITSO-INS data center.

In addition to previous acquisitions, ITSO-INS has recently purchased a call center facility. This facility uses a call center application deployed to IBM pSeries® server. The application is run at a single remote call center location.

ITSO-INS offers their external insurance brokers access to the customer and policy information database. There are two routes that an external insurance broker may use to access this database:

- Via a Web-based interface.
- Remote users, who do not have access to a Web browser or have specific policy issues, may contact the call center via a dedicated broker hotline.

It has been identified that while creating a call record within the call center application, customer details must be accessed on the SAP CRM system via a second user session. This has resulted in the re-entry of customer data into the call center application, increasing the time required to complete a call. Customer satisfaction has declined as a consequence.

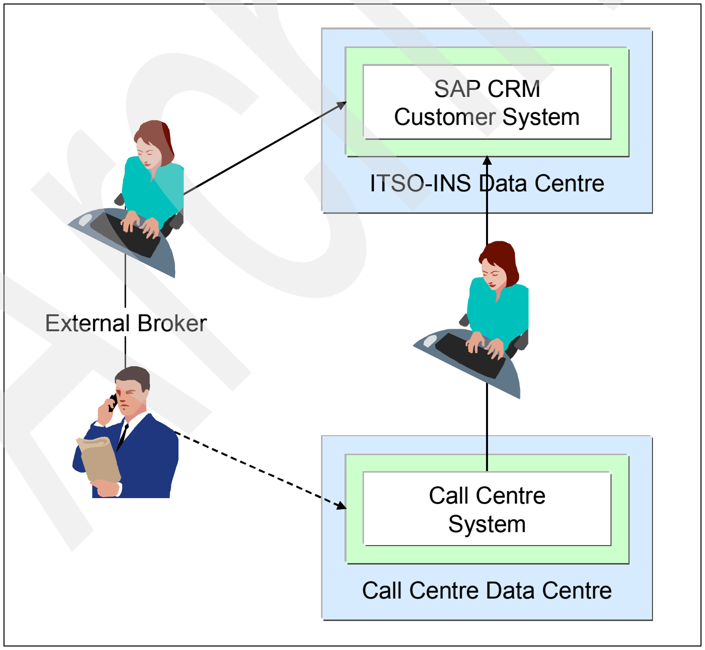


Figure 5-2 Broker accessing SAP CRM customer and call center systems

Business requirements

The identified business requirements include the following:

- To develop a seamless interface between the call center application and the SAP CRM.
- This interface should enable the call center customer services representative to retrieve the details of a customer's policy from the SAP CRM by entering the policy number, while creating a new record in the call center system.

Environment

In the environment:

- The SAP CRM system is running on a separate LPAR on the zSeries server.
- The call center application is deployed on a server within the call center. The call center is geographically isolated from the ITSO-INS data center.
- There is no wide area connection between the two data centers.
- External access to the system is via the Web-based transport protocol, HTTP.

5.2 Case 2: banking and fraud management

The ITSO-BANK has recently implemented an automated transaction tracking system (ATT). This system has been implemented to monitor transaction activities initiated via a branch, Internet banking, electronic transfer, or other external sources.

The ITSO-BANK system is a workflow-based application. On the initiation of a transaction, a new record is created and a unique transaction number assigned. Dependant on the category of the transaction initiated, a predefined transaction classification type is allocated. The routing of the transaction is determined by the classification type allocated to the new record.

As a transaction is routed through the system, a series of business rules is applied. The first and most crucial business rule is one that determines whether the transaction is suspicious in nature. If a transaction is identified as suspicious, further scrutiny to determine the appropriate course of action is required. This process includes the writing of a record to the fraud table within the ATT database, along with the insertion of a record into a flat file.

ITSO-BANK acknowledges that there is a need for a uniform strategy by which external partner systems interact with the ATT system for the purpose of obtaining data. At present external partners may access the data held within the ATT database or the flat file.

5.2.1 Scenario 3: interaction with external fraud agency

At present, the ITSO-BANK business transaction division outsources all suspicious transactions to an external fraud agency. The agency therefore requires access to the ITSO-BANK transactions, and external partners are required to access a flat file when retrieving data.

Flat file creation

At the conclusion of each day, an internal application reads the fraud table and creates a CSV file containing the day's transactions. This file has a predefined name with the current date appended to it, and is placed in a user-configured directory on a shared server.

The external fraud agency has installed a software application held on the ITSO-BANK server. This application polls the directory at regular intervals to establish whether a file exists. Once located, the file is copied to a predefined directory on the external fraud agency's server.

Current external fraud agency process

At present, an external fraud agency administration employee prints the imported file and enters each transaction into the agency's internal fraud system. The fraud system processes each transaction in turn by applying a set of predefined business rules, and an outcome is determined.

Once the day's transactions have been processed and outcomes determined, a flat file is created and copied from the external fraud agency server to a directory in the ITSO-BANK server.

The ITSO-BANK business transaction division acknowledges that the current process is inefficient and time consuming. At present, the turn-around time in determining the status of a transaction (that is, whether it is suspicious) is lengthy and has proven to be ineffective.

The issues identified are:

- The flat file is transferred at the conclusion of each business day. As a result the response time from the fraud agency is a minimum of 24 hours.
- The flat file created is often large, resulting in lengthy transfer times between the two systems. Should the transfer fail, the transfer process must be initiated from the beginning.
- There is an incompatibility between the data formats within the ITSO-Bank system and the fraud agency systems.
- The fraud agency is required to re-enter the transaction from the ITSO-Bank system into the fraud agency system.

- The transactions within the flat file do not contain details indicating whether the transaction is new or whether it is an update to an existing transaction. At present, this necessitates the fraud agency to reference each transaction against its database to determine the status of the transaction.

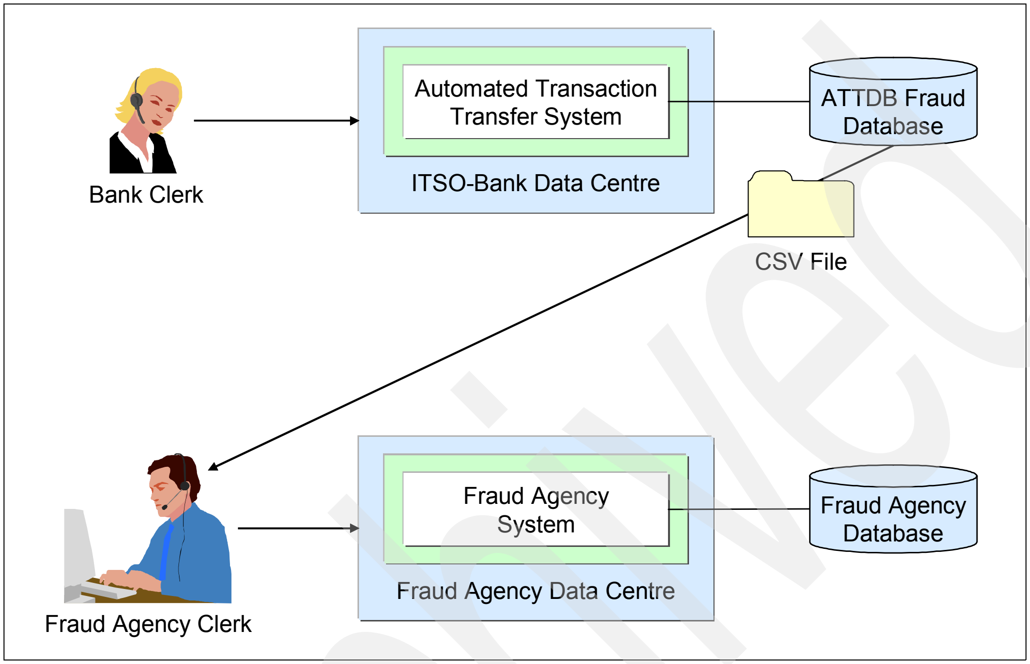


Figure 5-3 Batch transfer

Business requirements

The business division has identified the following requirements:

- After each record is written to the flat file in the ITSO-BANK system, the transaction must be forwarded to the fraud agency system within the hour.
- Fully automatic input of data into the fraud agency system.
- Conversion of incompatible data formats between the two systems.
- The ITSO-BANK system must append a specifier to each record indicating whether the record is new or is an update to an existing record.

Environment

ITSO-BANK has stipulated that all external electronic communications be carried out via the Web-based transport protocol HTTP.

5.2.2 Scenario 4: interaction with packaged fraud application

In conjunction with the existing ITSO-BANK system, the home transaction division of the ITSO-Bank has implemented an internal, automated fraud management system (FMS) to manage suspicious transactions. This standalone system has been developed and implemented by an independent software vendor.

Adhering to pre-determined business rules, this application is designed to determine whether a transaction is fraudulent.

The FMS has been developed as a J2EE-compliant integrated solution that can be deployed into an existing environment or as a standalone application. It is ITSO-BANK's preference to house the FMS on its own server environment.

At present, the home transaction division of ITSO-BANK executes a query at the conclusion of each business day against all new records added to the fraud table. Once the query is complete, details of the records are printed and manually entered into the FMS.

Previous attempts have been made to develop a process in which the ITSO-BANK system records are copied from the existing fraud table to the new FMS. However, the attempts at conversion have been unsuccessful due to incompatible data format issues.

It has been identified that the current process is ineffective in identifying the validity of a transaction due to the lengthy process time.

The issues identified are:

- Transactions are not processed until the following business day, thus limiting the ability to negate invalid transactions.
- Duplication of data held within the existing ITSO-BANK system and the FMS databases.
- Incompatibility between the file formats created in the ITSO-BANK system and new FMS. This requires maintenance of conversion applications.

Business requirements

The home insurance division has identified the following requirements:

- As a transaction is written to the fraud table it must be available to the FMS within the hour, rather than at the conclusion of each business day.
- Fully automatic input of data into the fraud management system.
- Automatic conversion of the ITSO-BANK transaction data into the FMS data format.

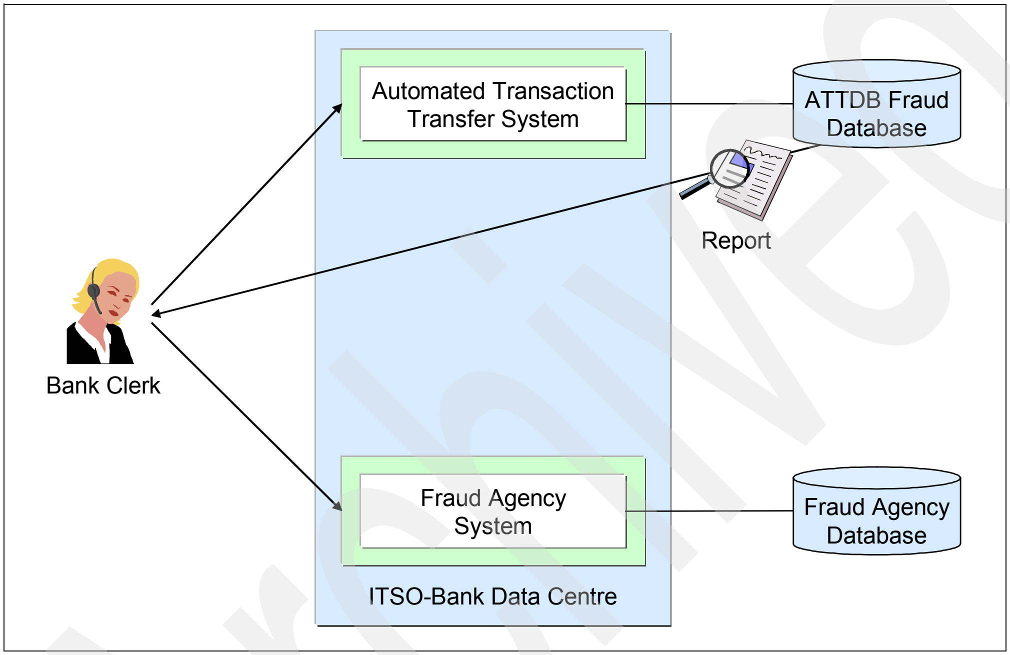


Figure 5-4 Event-driven service requests

Environment

The FMS is housed on a standalone server within the ITSO-BANK data center.

5.3 Case 3: automotive spare parts

The ITSO-AUTO automotive company specializes in the supply of new and pre-owned European vehicles, parts, and maintenance.

The company has multiple divisions:

- New vehicle sales
- Used vehicle sales
- Company leasing
- Spare parts
- Vehicle servicing

ITSO-AUTO has an integrated suite of applications that manages the sales and distribution of new and used cars. This suite has been extended to incorporate the vehicle leasing divisions requirements.

In the past ITSO-AUTO has partnered with external organisations to manage the supply and distribution of spare parts. In partnering with external organisations to manage spare parts, it has become clear to ITSO-AUTO that the maximum potential revenue is not achieved. A decision has been made to bring the management and distribution of spare parts in-house.

The ITSO-AUTO strategic plan does not involve in-house development of software applications. They prefer to procure suitable applications from external vendors and, where possible, to use external system integration companies to manage the integration of new software systems into their existing environment.

5.3.1 Scenario 5: integration of the dealer network

ITSO-AUTO supplies spare parts to its dealer network. Dealers order a part from ITSO-AUTO, and ITSO-AUTO arranges for the part to be supplied, either from its own stocks or from an external suppler. Currently, the order process involves an ITSO-AUTO employee receiving an order and phoning suppliers to achieve the most competitive price. This process has proven to be inefficient and time consuming (Figure 5-5).

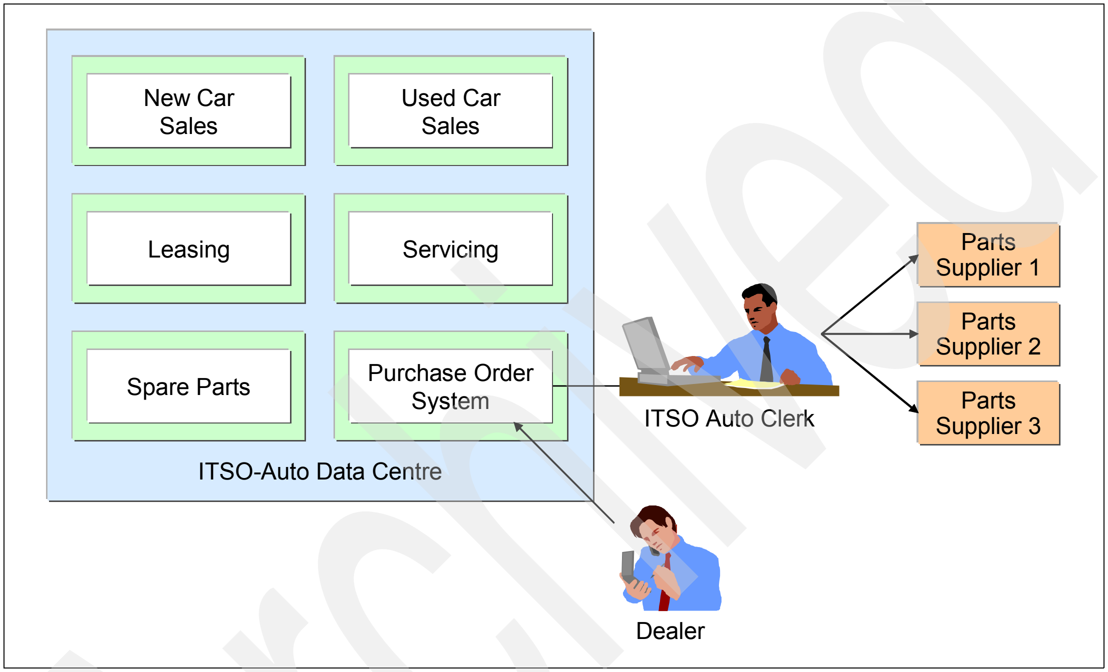


Figure 5-5 Custom application integration - ASIS

ITSO-AUTO has procured a new purchase order system from an Independent Software Vendor. It has engaged the services of an external systems integration company to integrate the system with its dealer network and parts suppliers.

The purchase order system uses a Web interface for registered external automotive dealers to order spare parts, and invokes services offered by external suppliers to select and place a parts order on behalf of the dealer.

Dealers can place orders online by specifying the part number and quantity. Once an order is placed, the purchase order system generates a response to the dealer confirming that the order has been accepted.

Business requirements

The identified business requirements include:

- To offer dealers the most competitive price in the marketplace
- To provide an automated process whereby the purchase order system is able to link with suppliers and obtain the best price
- Guaranteed response time and traceability of purchase orders

5.3.2 Scenario 6: integration of a third-party supplier

ITSO_AUTO uses a number of different parts suppliers. It wants to add a competitive new parts supplier that is entirely Web-based and that has only dealt with manual Web browser based ordering in the past. Attaching an automated ordering system as a client is a new development. The parts supplier can offer a set of lightweight services to ITSO-AUTO as callable http interfaces. These interfaces have previously been used by Web-based *Web 2.0* style clients.

The company has exposed its services over both JSON RPC and XML-RPC. ITSO_AUTO must ensure that its spare parts ordering system is able to integrate with the new supplier so as to offer dealers the most competitive price for parts.

Business requirements

Automatically invoke the Web 2.0 style interface from the automated parts ordering system.

All requests are logged to a database.

Business requirements

The business requirements for connecting the dealers include the following:

- The dealers need a simple and dynamic browser-based interface to allow them to browse parts catalogues quickly and interactively.
- The middleware solution must connect dealers with the ITSO-AUTOs existing SOAP-based dealer registration and authentication services.
- The application must log the details of each request.

Environment

Dealers are connected to ITSO-AUTO over the Internet.

5.4 Summary

This chapter introduced you to six scenarios that are used in the rest of the book. These scenarios are representative of many kinds of integration scenarios involving connecting Enterprise applications to a Web application or to a business process. The software application market is very fragmented, with no vendor holding even a double-figure share. Consequently, most engagements involve connecting a different set of applications and building new connectivity solutions, typically without the luxury of customizing a packaged adapter pre-built for a well-known application. In summary, the scenarios are:

- Reusing an older application as a service by connecting it to a service bus (“Scenario 1: claimants’ access to claims information” on page 131)
- Synchronization of data with a packaged application for use by a new Web-based application (“Scenario 2: insurance brokers’ access to customer policies” on page 133)
- Scheduled batch transfer of data from one application to another (“Scenario 3: interaction with external fraud agency” on page 135)
- Near real-time response to an event in a packaged application (“Scenario 4: interaction with packaged fraud application” on page 137)
- Provision of services to Web 2.0 clients (“Scenario 5: integration of the dealer network” on page 140)
- Integration of external Web-based applications with packaged applications (“Scenario 6: integration of a third-party supplier” on page 141)

6

Connection patterns

In this chapter, we explore different ways of connecting WebSphere ESB to Enterprise applications. These connection methods are based on the various business scenarios explored in Chapter 5, “Business scenarios” on page 129. Some of the scenarios explored here are developed further in Part 3, “Working examples” on page 171.

6.1 Connection pattern 1: integration

In 5.1.1, "Scenario 1: claimants' access to claims information" on page 131, we discussed a business scenario where the ITSO-INS insurance company wishes to connect its CICS system running on zSeries and its iSeries system to a Web-based interface. This business problem is illustrated in Figure 6-1.

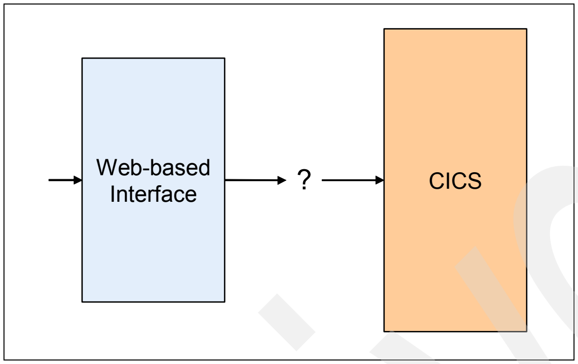


Figure 6-1 Business problem to solve: connecting CICS to a Web-based interface

The CICS scenario is the primary focus of this section. However, we also discuss a scenario for connecting a Web-based interface to a older application that communicates via a comma-delimited MQ message. This is covered in 6.1.3, "Simple WebSphere MQ integration" on page 148.

6.1.1 Challenges and assumptions

In this section we discuss the feasibility of various options for doing this. We assume that the Web-based interface can communicate via SOAP/HTTP-based Web services.

6.1.2 Solution options

There are various options for implementing this scenario using WebSphere ESB:

- Using a CICS JCA adapter
- Using the CICS/MQ bridge
- Using a HATS/3270 bridge
- Using CICS Web services

Analysis of solutions

In this sub-section, we analyze each of the four solutions outlined above, discussing their relative merits.

Solution 1 - using a CICS JCA adapter

There is a WebSphere CICS JCA-based resource adapter that is shipped with WebSphere Integration Developer and the CICS transaction gateway. This can be used for communication with CICS via the CICS transaction gateway in the same way as any other WebSphere JCA adapter. In other words, it appears as an EIS import/export in a mediation module. This solution is illustrated in Figure 6-2.

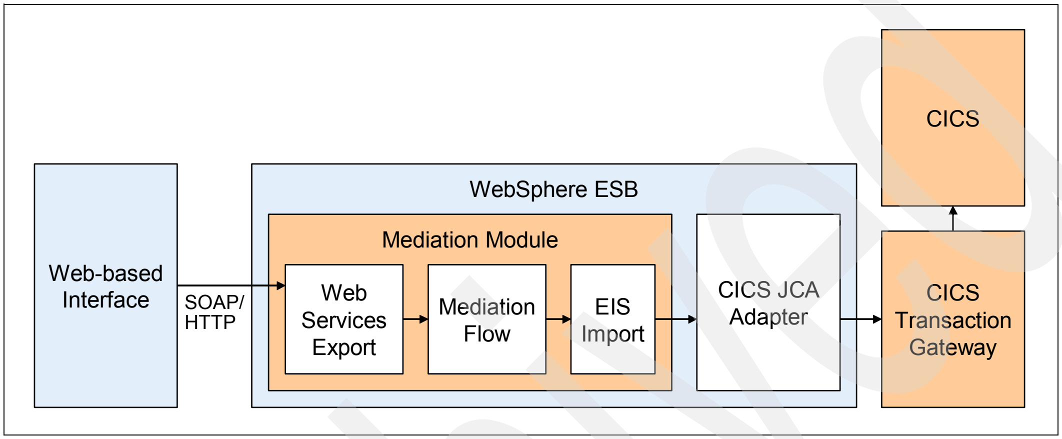


Figure 6-2 Communicating with CICS using CICS JCA-based adapter

Using the WebSphere CICS JCA-based adapter is the generally recommended approach when connecting to CICS from WebSphere ESB or some other J2EE-based environment. It inherits many quality-of-service attributes from the environment, including security and connection pooling.

However, this solution is only suitable when CICS applications are exposed via a CICS transaction gateway, because the CICS adapter requires the transaction gateway to communicate with CICS.

For more information about the CICS resource adapter, see the WebSphere Integration Developer InfoCenter:

http://publib.boulder.ibm.com/infocenter/dmndhelp/v6rxmx/index.jsp?topic=/com.ibm.websphere.wesb.doc/info/welcome.html

Solution 2 - using a HATS/3270 bridge

The Host Access Transformation Services (HATS) product provides the ability to make a 3270-based terminal application available via a Web-based application that runs on a WebSphere-based application server (such as that underlying WebSphere ESB). For a CICS application accessible via a 3270 terminal-based application, WebSphere ESB can interact with it using an HTTP-based pattern and custom code that emulates a user navigating through the Web pages.

There are many more levels of indirection than are desirable here, so this solution is definitely not recommended for connection to CICS. However, we do discuss some HTTP-based connection patterns from WebSphere ESB in 6.6, “Connection pattern 6: Web integration” on page 164, which might be appropriate if you do decide to attempt this communication pattern.

For more information about HATS, see the HATS InfoCenter:

http://publib.boulder.ibm.com/infocenter/hatshelp/v60/index.jsp?topic=/com.ibm.hats.doc/doc/service.html

Solution 3 - using CICS Web services

CICS provides a Web service interface that allows for integration of CICS applications using Web services. CICS applications can appear as service providers or service requestors. Because WebSphere ESB provides native support for Web services, this interface can be easily integrated with it.

The business scenario might be solved with CICS Web services, as illustrated in Figure 6-3.

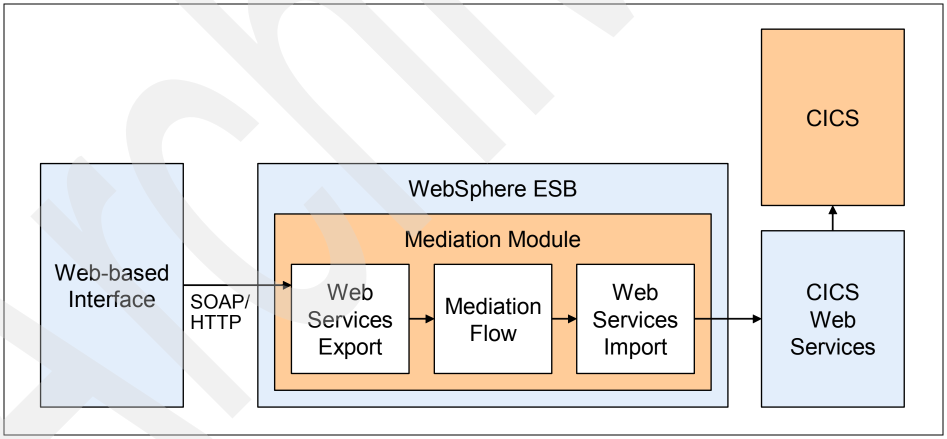


Figure 6-3 Communicating with CICS using CICS Web services

This solution has the advantage that the majority of the work involves working with the CICS Web service support. WebSphere ESB is only required to mediate between the Web service interface provided by CICS and the Web service interface expected by the front-end Web-based interface. However, as a result, it requires skills in using the CICS Web services support. This is beyond the scope of this book, but you can find more information in the excellent IBM Redbooks publication *Implementing CICS Web Services,* SG24-7206, found at:

http://www.redbooks.ibm.com/abstracts/SG247206.html?Open

This solution is relatively future-proof: support for Web services is improving in integration products in general.

Solution 4 - using the MQ-CICS bridge

The MQ-CICS bridge provides an alternative means of communicating with CICS, via WebSphere MQ. Because WebSphere Enterprise Service Bus provides a native means of communicating with CICS via MQ bindings on imports and exports, this is another potential approach for connecting WebSphere Enterprise Service Bus to CICS.

The business scenario might be solved using this approach, as illustrated in Figure 6-4.

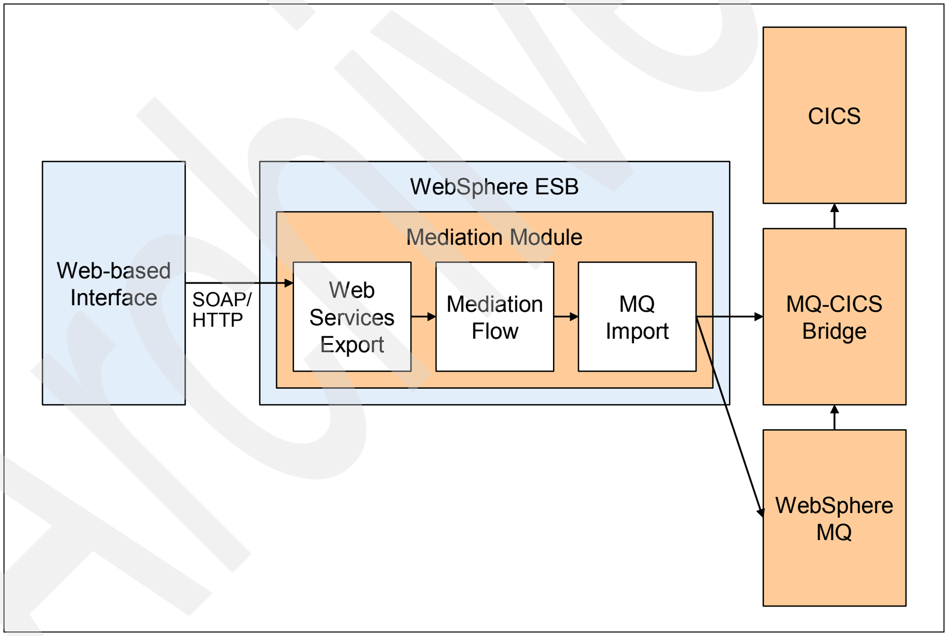


Figure 6-4 Communicating with CICS using the MQ-CICS bridge

The advantage of this approach is that it is robust and is suitable when a CICS transaction gateway is not present, and you do not wish to use CICS Web services. The disadvantage is that it is necessary to develop a MQ body and header data bindings in order to construct the MQ message in the correct format for the bridge. We develop this solution further in Chapter 8, “Custom CICS integration using WebSphere MQ" on page 215, and show how it could be implemented.

6.1.3 Simple WebSphere MQ integration

Solution 4 above (using the MQ-CICS bridge and an MQ import to connect WebSphere ESB) can be generalized to any older application that communicates via WebSphere MQ. This is illustrated in Figure 6-5. In Chapter 7, "Integration using WebSphere MQ" on page 173, we develop this scenario fully, illustrating how to integrate with a typical WebSphere MQ application that communicates using comma-delimited WebSphere MQ messages. In some situations (such as the one that we develop), creating a custom header binding is not necessary because custom headers are not necessary. On rarer occasions, developing a custom body binding is not necessary either, as one of the provided data bindings is suitable (3.5.4, "MQ native and MQ/JMS bindings" on page 89).

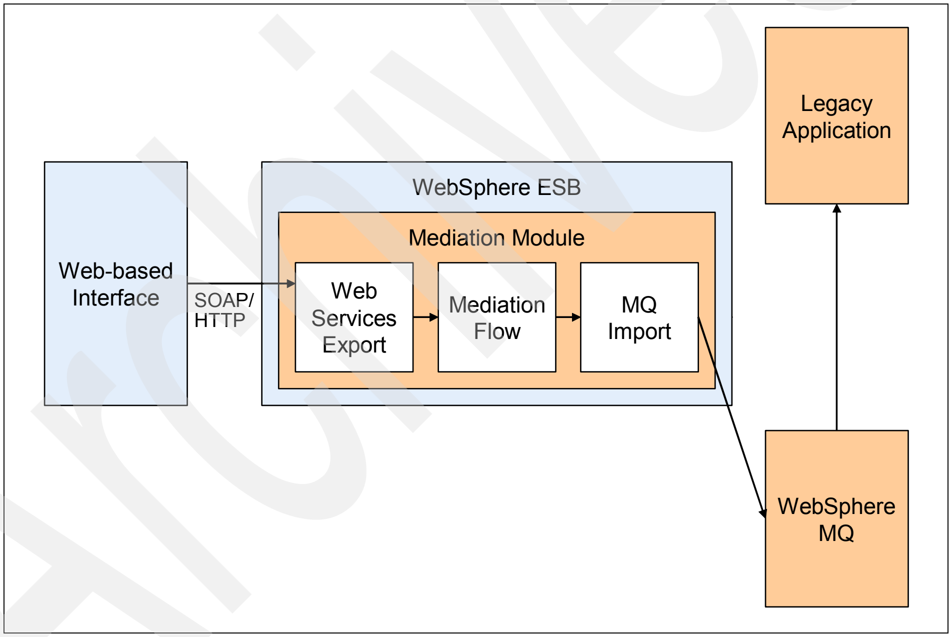


Figure 6-5 Connecting older application to WebSphere ESB via WebSphere MQ

6.2 Connection pattern 2: packaged application integration

In 5.1.2, “Scenario 2: insurance brokers’ access to customer policies” on page 133, we discussed a business scenario where the ITSO-INS insurance company wishes to connect its SAP CRM system to a packaged application. In this section we discuss the feasibility of various options for doing this.

6.2.1 Challenges and assumptions

The challenges and assumptions are:

- The ITSO-INS insurance company has a packaged application that needs to integrate with multiple business EIS applications.
- The packaged application can communicate using SOAP/HTTP Web services.
- The customer has an EIS that is SAP XI.

We assume that the insurance company decided that they wish to use WebSphere ESB as an intermediary, in order to be able to integrate with many customers’ systems. For the sake of this section, we discuss only the customer with SAP.

6.2.2 Solution options

There are four options for implementing this scenario:

- Connecting to SAP using SAP-provided Web services
- Connecting to SAP using SAP-provided JMS
- Connecting to SAP using the SAP JCA-based WebSphere Adapter
- Connecting to SAP using the SAP WebSphere Business Integration Adapter

Analysis of solutions

In this sub-section, we analyze each of the four solutions outlined above, discussing their relative merits.

Scenario 1 - connecting to SAP using Web services

In this scenario, an SAP SOAP adapter is used to expose a business function as a Web service. WebSphere Enterprise Service Bus connects to this using a Web service transport binding on an import. This scenario is illustrated in Figure 6-6.

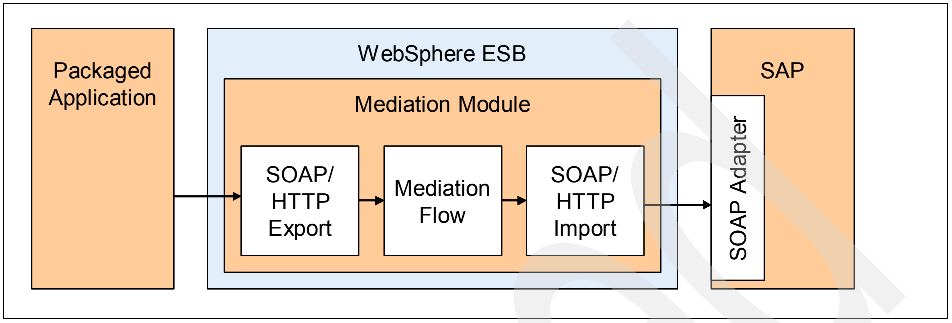


Figure 6-6 Connecting to SAP using SAP SOAP adapter

Scenario 2 - connecting to SAP using JMS

In this scenario, an SAP JMS adapter is used to expose a business function via JMS. The JMS provider used must be supported by both WebSphere ESB and the SAP JMS adapter. WebSphere MQ fulfills this criterion. WebSphere Enterprise Service Bus connects to the JMS provider using an appropriate binding on an import (in the case of WebSphere MQ, an MQ/JMS binding). This scenario is illustrated in Figure 6-7.

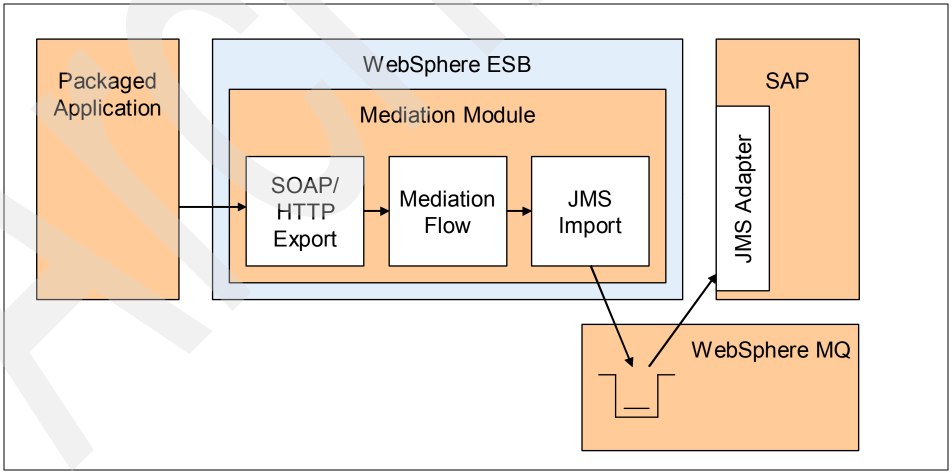


Figure 6-7 Connecting to SAP using SAP JMS Adapter

Scenario 3 - connecting to SAP using JCA-based Adapter

In this scenario, the WebSphere JCA-based Adapter for SCA is used to connect WebSphere ESB to SAP. This scenario is illustrated in Figure 6-8.

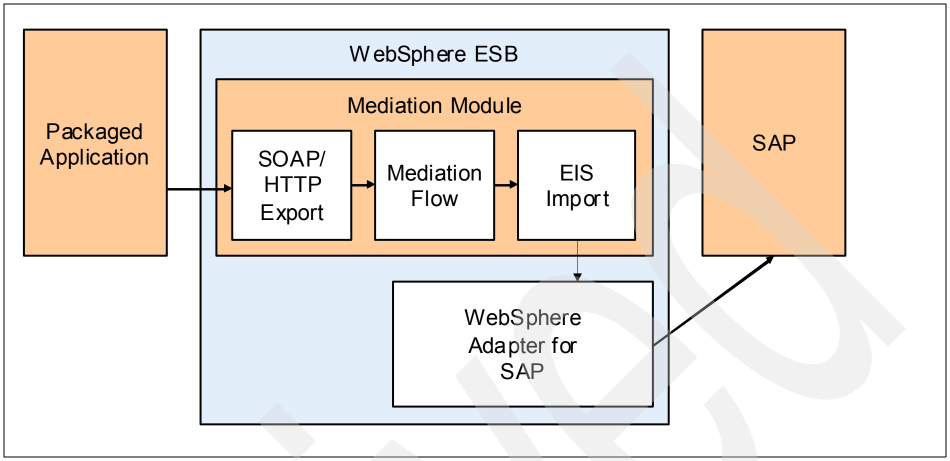


Figure 6-8 Connecting to SAP using WebSphere Adapter for SAP

Solution 4 - connecting to SAP using WebSphere Business Integration Adapter for SAP

This solution involves using the WBI Adapter for SAP. We do not recommend this, as there is a JCA-based adapter that is a more modern approach that integrates well with WebSphere Enterprise Service Bus.

6.2.3 Further information

In this section we discussed solely SAP XI. For much more information about connecting to this and another SAP products, see *Connect WebSphere Service Oriented Middleware to SAP*, SG24-7220.

6.3 Connection pattern 3: batch integration

In 5.2.1, “Scenario 3: interaction with external fraud agency” on page 135, we discussed a business scenario that requires service-oriented requests to be generated from financial transactions stored into CSV files. These requests are to be forwarded to the fraud management system (FMS). The scenario as outlined does not strictly involve batch transfer of data, as requests have to be generated on a timely manner as data is transferred into the file. However, the same solutions could be applicable to a batch transfer scenario (for example, data transferred every day at 6 a.m.), with little or no modification. This business problem is illustrated in Figure 6-9.

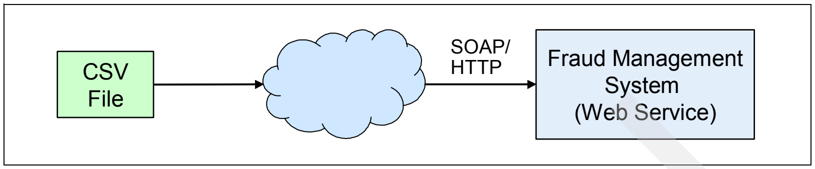


Figure 6-9 Connection required for scenario

In this section we first discuss the assumptions and challenges of this business problem, and then develop it technically, exploring possible solutions.

6.3.1 Challenges and assumptions

The primary defining technical characteristic of this scenario is that a single service-oriented transaction should be initiated for each record in the original file-oriented data.

In Chapter 5, “Business scenarios” on page 129, we introduced the assumption that the records were stored in flat files. We assume the following additional parameters for this scenario:

- Data is stored in a directory. Each record is stored in a separate line in a file. Each field is comma-separated.
- Each record should be used to initiate one request (transaction).
- The file may potentially be quite large (of the order of millions of lines).
- The file might be located remotely or locally to the WebSphere ESB that needs to read it.
- For the purposes of this example, each request should invoke the fraud agency system, which exposes a Web service interface. We do not discuss the details of this in this chapter, but we assume that there is some mediation required between the data stored in the CSV file and the interface exposed by the Web service (for example, two fields may need to be combined into one).

6.3.2 Solution options

There are various ways to implement this scenario:

- Using a standalone application that communicates with the FMS directly
- Using a standalone application that communicates with the FMS via WebSphere ESB

- Reading from the file using a custom JCA adapter that connects into WebSphere ESB
- Using WebSphere ESB with the Flat File Adapter

Analysis of solutions

In this sub-section we analyze each of the four solutions outlined above, discussing their relative merits.

Solution 1 - standalone application with direct FMS communication

This solution involves writing a standalone application, perhaps in Java, that consumes data from the comma-separated files, performs the necessary mediation between data formats and interfaces, then acts as a Web service client and communicates with the back-end Web service. This solution is illustrated in Figure 6-10.

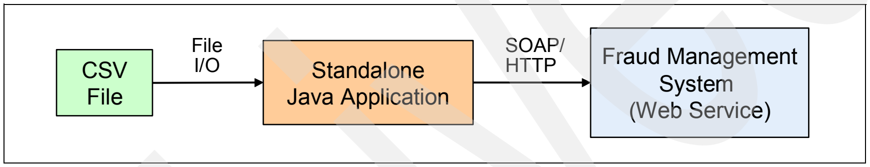


Figure 6-10 Standalone Java application solution

The CSV file could be local to the same host as the Java application, or remotely mounted. If the file were only remotely accessible using a protocol such as FTP, the standalone Java application would also have to act as a client for this.

The primary disadvantage of this approach is its inflexibility: if the structure or the locale of the files change, or the back-end service changes, the solution has to be re-worked. It is also harder to add additional data sources. In addition, it is a non-trivial undertaking to write a Web service client, particularly with high performance and scalability requirements.

Solution 2 - standalone with communication via WebSphere ESB

This solution involves writing a standalone J2EE application that consumes transactions from the CSV file in a similar way to solution 1. However, instead of performing any mediation necessary and directly connecting to the FMS, it connects into WebSphere ESB via a stand-alone reference, which allows the application to communicate with an SCA-based mediation module. The mediation module performs the mediation and connects to the fraud management system using an import with a Web services binding. This solution is illustrated in Figure 6-11.

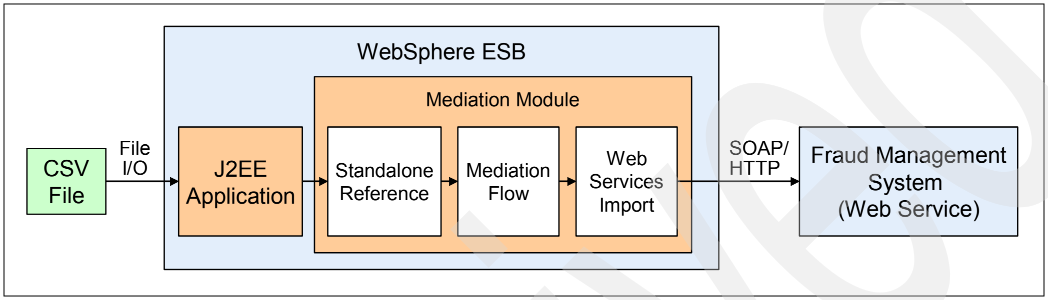


Figure 6-11 Consuming batch data from CSV file via J2EE application and standalone reference

This option suffers from the same problems as solution 1: it is inflexible and cumbersome. If the scenario changes in any way, the code that implements the J2EE application would need to be changed. Also, the J2EE application would still need to act as, for example, an FTP client if the file were only remotely accessible.

Solution 3 - writing a custom JCA adapter

This solution involves writing a custom JCA adapter that provides the ability to read in from the file. Requests are then passed from this into a mediation module, which performs the mediation and communicates with the Web service via an import with a Web services binding. This solution is illustrated in Figure 6-12.

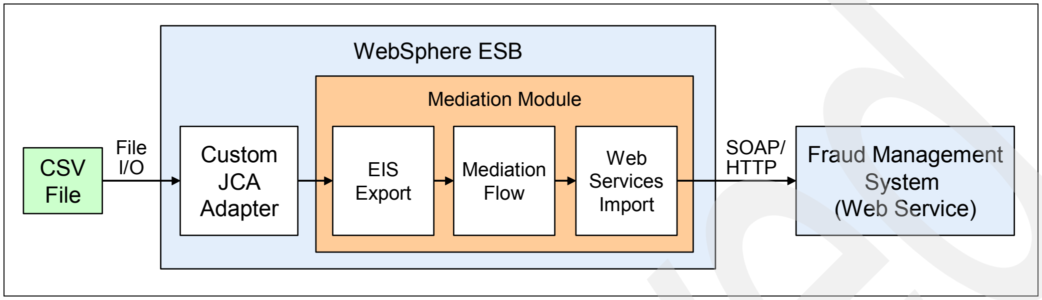


Figure 6-12 Solution

This method provides a lot of control over the entire process, and integrates well with WebSphere ESB. It would also mean that writing a data binding would not be necessary, as the adapter could handle this internally. However, writing a custom adapter is not a non-trivial task. There is probably little benefit to this approach in this scenario.

Solution 4 - use the Flat File Adapter

This solution involves using WebSphere ESB in combination with the WebSphere Flat File Adapter. This solution is illustrated in Figure 6-13.

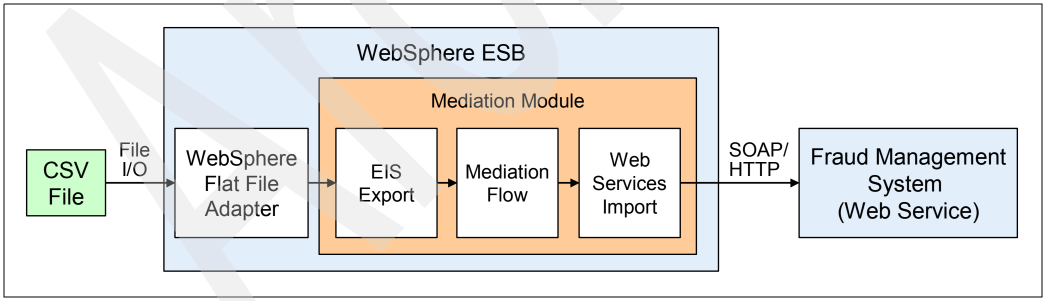


Figure 6-13 Reading from a CSV file using the Flat File Adapter

We recommend this solution. It provides the maximum flexibility.

A nominated directory is set up as the event directory in the flat-file adapter, and files created there are used as input for the scenario. The adapter automatically polls these flat files for new records. For each record to result in a single service invocation, use delimiter-based file splitting. This is a standard feature in the Flat File Adapter. Set the file splitting delimiter to \n, which indicates a new line.

The file the adapter is reading from is typically very large. To reduce the impact of this on overall performance, the inflow of requests into the mediation module can be throttled back. To do this, change the poll period and ‘poll quantity parameters for the adapter (these are typically set during the Enterprise Service Discovery process when the adapter is first set up). The first parameter alters the frequency with which polls are done for new messages, and the second how many messages are input by the Flat File Adapter during each poll. Set the second parameter to 1, and the first to a value that causes an even message rate (this could best be determined by experimentation, as it depends on the performance of the hardware in question, as well as many other factors).

The WebSphere Flat File Adapter only supplies a data binding for XML-based data. Because our data is not XML-based, you would need to write a custom data binding to convert the data into data objects for the incoming requests.

The custom data binding overrides the WBIDataBindingImpl class and the getDataObject() method. The getDataObject() method first calls the superclass implementation of the same method. This returns a FlatFileBG data object. The string corresponding to the first line of the file can be extracted from this, and parsed into fields. The data binding then constructs a business object appropriate for processing in a mediation module.

This solution also provides a convenient way of reading from the file if it is accessible only via FTP, because the WebSphere FTP Adapter can be used in place of the Flat File Adapter. Its mode of operation is very similar. Conceptually, the main difference is that the directory that contains the input files is remote, accessed via FTP or SFTP, rather than local. Otherwise, the setup of the adapter is much the same.

You can find more information about the FTP adapter in the WebSphere Adapter for FTP User Guide:

http://publib.boulder.ibm.com/infocenter/dmndhelp/v6rxmx/index.jsp?topic=/com.ibm.wsadapters602.jca_ftp.doc/doc/stftp_welcome.html

This solution provides simple scalability and is the recommended approach.

6.4 Connection pattern 4: event-driven integration

In 5.2.2, “Scenario 4: interaction with packaged fraud application” on page 137, we discussed a banking scenario that requires events to be triggered from a database table such that the Fraud Management System (FMS) initiates a transaction and sends a response to the banking system. This business problem is illustrated in Figure 6-14.

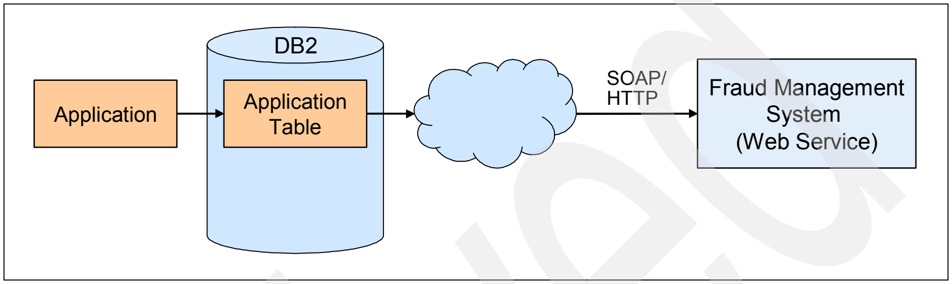


Figure 6-14 Connection required for scenario

In this section we discuss the assumptions and challenges of this business problem and then describe various solution options — ways of triggering service requests from database events. The scenario is fully explored in Chapter 11, “Event-driven integration using a JDBC adapter” on page 379, which describes how to implement it.

6.4.1 Challenges and assumptions

The technical challenge in this section is to initiate a transaction when an event occurs in the banking system.

We assume the following parameters for the scenario:

- Each transaction has a unique transaction number.
- The customer’s database (where the fraud table is hosted) is running on a database that can be accessed via JDBC.
- As per scenario 2, the FMS is implemented as a Web service application.
- The bank’s application exposes a Web service interface to return information about whether a transaction is fraudulent.
- The banking application already writes information to the Fraud table.

6.4.2 Solution options

There are various ways to implement this scenario:

- Using a DB2 database trigger and a Java user-defined function (UDF)
- Using a DB2 database trigger that generates MQ messages using an MQ UDF
- Using the WebSphere JDBC adapter

Analysis of solutions

In this sub-section, we analyze each of the three solutions outlined above, discussing their relative merits.

Solution 1 - using a DB2 database trigger and a Java UDF

This solution involves creating a DB2 database trigger on the fraud table that is triggered when a change happens in the application's tables. This executes a Java-based user-defined function (UDF). In turns, this UDF retrieves any additional data from the database, and acts as a Web service client to the Fraud Management System. This solution is illustrated in Figure 6-15.

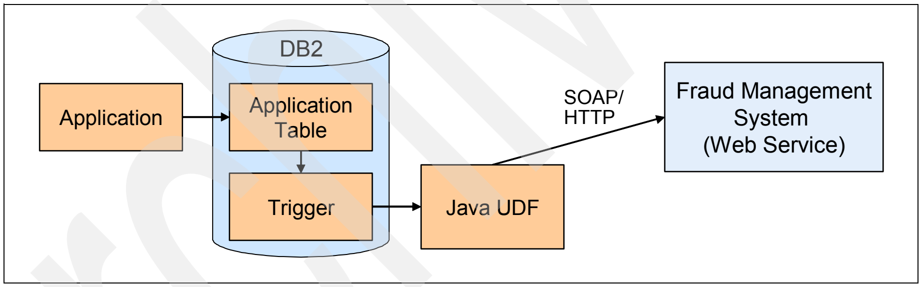


Figure 6-15 Connecting to FMS using triggers and a Java UDF

This solution is cumbersome and awkward. The Java UDF has to make synchronous calls to the Web service, and it is unclear that this will scale well if there are a large number of changes being made to the database. Also, if the interface on the FMS changes, the Java UDF will have to change. This makes the system quite fragile.

You can learn more about Java UDFs in DB2 here:

`http://www.db2mag.com/story/showArticle.jhtml?articleID=15300024`

Solution 2 - using a DB2 database trigger and an MQ UDF

This solution involves creating a DB2 database trigger that outputs a message to a WebSphere MQ queue using an MQ user-defined function (UDF) when an event happens in the application's tables. WebSphere ESB can then initiate a service request by using an export with an MQ binding.

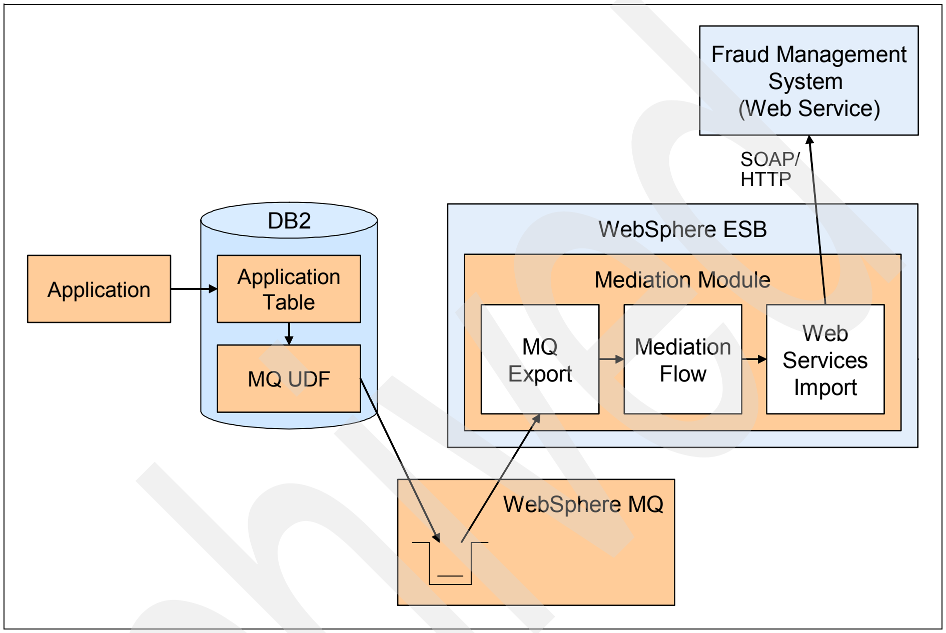


Figure 6-16 Connecting to FMS using triggers, an MQ UDF, and WebSphere ESB

The advantage of this approach is that very little code will have to be written (possibly none, if the MQ message is simple enough and provided data bindings can be used). The disadvantage is that it requires installation and maintenance of an additional MQ queue manager, if one is not already present.

For more information about using MQ UDFs with DB2 in this way, see the DB2 InfoCenter:

http://publib.boulder.ibm.com/infocenter/db2luw/v8/index.jsp?topic=/com.ibm.db2.ii.doc/ad/ciiwsdb2.htm

Solution 3 - using the WebSphere JDBC Adapter

The most straightforward solution, and the one we go on to develop further in Chapter 11, "Event-driven integration using a JDBC adapter" on page 379, is to use WebSphere ESB and the WebSphere JDBC Adapter. WebSphere ESB will be used to convert data between the business objects that the JDBC adapter outputs and the interface that the FMS expects. This solution is illustrated in Figure 6-17.

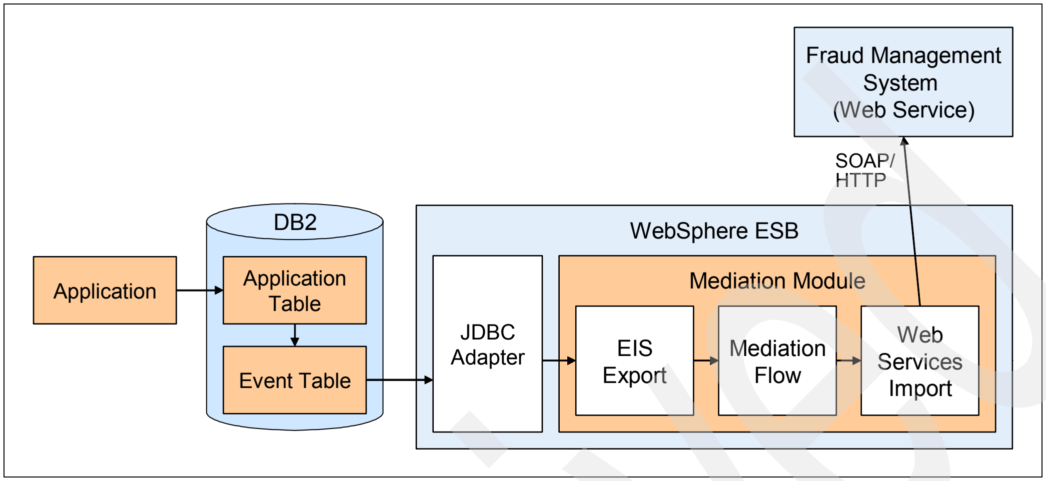


Figure 6-17 Triggering service requests using the JDBC Adapter

This solution has the advantage of maximum flexibility and scalability. The JDBC adapter can easily be replaced with a different type of adapter or binding if the solution needs to be changed. The callout to the FMS can also easily be changed.

6.5 Connection pattern 5: custom application integration

In 5.3, "Case 3: automotive spare parts" on page 138, we discussed a business scenario where ITSO-AUTO wishes to integrate a purchase order system that it has acquired with its dealerships. In this section we discuss the feasibility of various options for doing this.

6.5.1 Challenges and assumptions

For this scenario we do not assume any technical restrictions, as we assume that the purchase order system and the dealership systems can be altered as necessary. In other words, we assume that this is a *green-field* scenario as far as WebSphere ESB is concerned.

In other words, the business scenario can be expressed as shown in Figure 6-18.

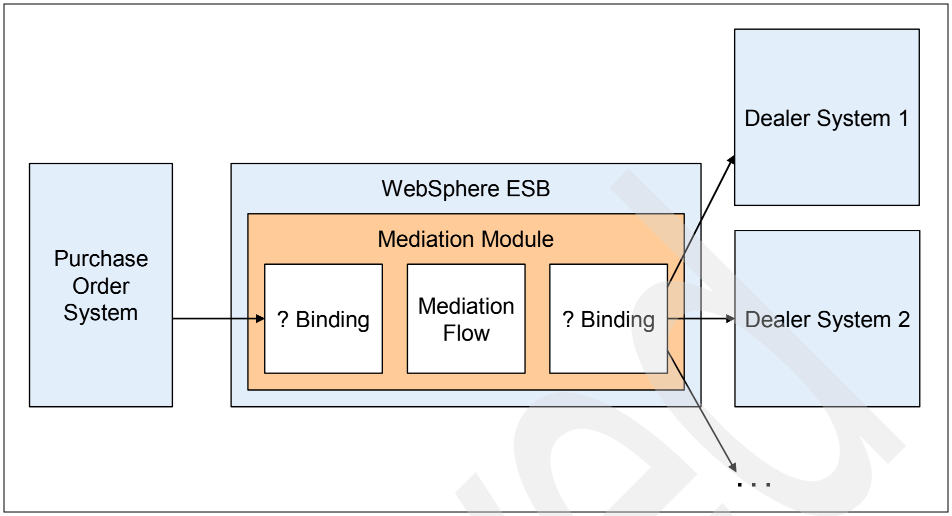


Figure 6-18 Integrating the purchase order system and dealership systems

6.5.2 Solution options

In this section we discuss the relative merits of the different ways of integrating the purchase order system and the dealership system using WebSphere ESB. As there are no technical restrictions, we are free to discuss the relative merits of different connection types.

WebSphere ESB supports a variety of connection types natively by virtue of its provision of out-of-the-box transport bindings. We discussed these various transport bindings in 3.5, "WebSphere ESB Transport Binding Types" on page 82, but to briefly recap, they include:

- Messaging bindings, such as JMS, MQ Native, and MQ/JMS
- Web services bindings
- EIS bindings (used with WebSphere Adapters)

The SCA native bindings are not relevant here, as they are only used for integration within an SCA-based system, such as the WebSphere Process Server runtime.

Of course, it would not be necessary to use the same binding type to connect the purchase order system to WebSphere Enterprise Service Bus and to connect WebSphere Enterprise Service Bus to the dealer systems. However, we ignore that complication here and assume that we are using the same binding type throughout.

Messaging bindings

There are three messaging bindings that provide a way to integrate with asynchronous messaging systems:

- JMS connects WebSphere Enterprise Service Bus to WebSphere Platform Messaging.
- MQ native connects WebSphere Enterprise Service Bus to WebSphere MQ using native MQ messages
- MQ/JMS connects WebSphere Enterprise Service Bus to WebSphere MQ using JMS messages.

The JMS API is a more modern and standards-driven approach to constructing messaging clients that work with JMS-style messages, so generally you would choose that in preference to MQ Native messaging unless there were a compelling reason otherwise — primarily, that you wish to integrate with existing applications that already communicate using non-JMS-style messages. Since this is a green-field scenario, we discard the MQ Native option.

Essentially, therefore, there are two messaging options open to us: the JMS bindings, which are used with WebSphere Platform Messaging, the embedded JMS provider in WebSphere Enterprise Service Bus; and the MQ/JMS bindings, which are used with WebSphere MQ.

Again, the decision as to which to use is driven primarily by existing technology in use. If the enterprise in question already had a WebSphere MQ-based messaging backbone, then using the MQ/JMS bindings would make sense. WebSphere MQ is comparatively more mature than WebSphere Platform Messaging. However, because the latter is more tightly integrated with the WebSphere Application Server platform, it would generally be the recommended approach in a green-field scenario such as this. The JMS bindings also have the advantage that they require less configuration than the alternatives.

In many circumstances, using messaging bindings requires extra work because data bindings need to be written (header or body bindings, or both). However, in this case, that may not be a relevant problem, because this is a green-field scenario. In this case, the provided data bindings could be used, such as the bindings that serialize a business object as XML. This would be acceptable as long as the systems that were integrated (in this case, the purchase order system and the dealer system) had the ability to construct and parse XML-based business objects.

Web services bindings

The Web services bindings provide a way to integrate applications directly using synchronous SOAP-based Web services invocations. They support SOAP/JMS as well as SOAP/HTTP. However, SOAP/JMS is generally not as common as SOAP/HTTP, so in general we do not recommend it when the overriding goal is interoperability. But SOAP/http interoperability comes at a price of poorer robustness and performance than SOAP/JMS.

SOAP/HTTP Web services do not rely on external infrastructure as the messaging bindings do, because there is no messaging backbone. So in the case we discuss here the purchase order system could integrate directly with WebSphere ESB, which in turn could integrate directly with the dealer systems. This is a slight advantage: there is less infrastructure to maintain and set up, assuming that the TCP/IP network is taken as a given. However, HTTP is also a transport mechanism that does not provide guaranteed delivery. This may make the system as a whole less reliable.

The Web services bindings do not require writing any form of data binding, and integrate neatly with the SCA programming model used in WebSphere ESB. However, they do have the disadvantage of being synchronous. This makes them most suitable for short transactions. If the interaction between the purchase order system and the dealer is likely to run on for a long time, then an asynchronous method of communication such as the messaging bindings may be more appropriate, as otherwise system resources may be tied up unnecessarily.

EIS bindings

The EIS bindings are used with the WebSphere Adapters, generally to integrate with older or nonstandard systems. They are generally robust and have many Quality of Service attributes, but if you were using them with a new system designed from scratch, as we are discussing here, you would not generally use them, as they are more complex to configure than the options discussed above and require more levels of indirection.

Summary

Unfortunately, there is no simple answer for which transport binding method to use to connect these applications. Web services are often a good choice because they integrate neatly with the SCA model used in WebSphere Enterprise Service Bus, but they have the disadvantage of being synchronous and relying on a non-guaranteed transport mechanism. However, it is fair to say that the situation here — a green-field scenario where there are no particular technical requirements — is atypical. Normally, performance requirements, reliability requirements, or existing infrastructure dictate one connection type or another. The only general piece of advice is to pick the simplest communication mechanism possible that fulfills the requirements of a scenario.

6.6 Connection pattern 6: Web integration

The provision of Web services on the Internet, and on company intranets, is increasing. Not all of these services are standards-compliant or integrate with WebSphere ESB out-of-the-box.

In 5.3.1, "Scenario 5: integration of the dealer network" on page 140, we discussed a situation where an automotive company needed to expose a packaged application to its dealerships in the form of services that could be used by Web
2.0 based browser user interfaces. We then discussed another situation in which the auto company needs to invoke external services originally provided by a service provider to be consumed by Web 2.0 based browser interfaces.

In this section we first discuss the assumptions and challenges of this business problem. We then describe possible patterns for connecting to a service on the Internet using WebSphere ESB (as well as expanding on a high-level design for our preferred method) when a Web service import binding cannot be used because the service is not SOAP/HTTP-compliant. These are *outbound* patterns. We also look at corresponding patterns for *inbound* services — where services on the Internet need to connect to WebSphere ESB — and a Web service export binding cannot be used for the same reason.

This business problem is illustrated in Figure 6-19.

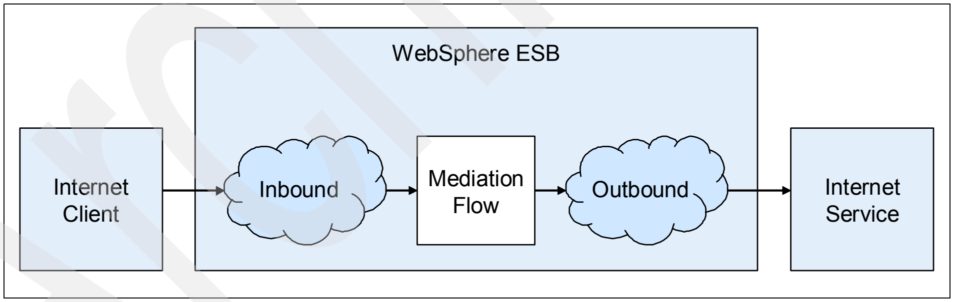


Figure 6-19 Connection required for scenario

6.6.1 Challenges and assumptions

The technical challenge involved in this scenario is to integrate WebSphere ESB with service requestors or providers of non-SOAP/HTTP-based Web services.

For the purposes of discussion we assume that the service we want to invoke from ESB, or expose via ESB, uses HTTP as its underlying transport (although some of the techniques we describe could be extended to other protocols). Three examples of services that fit this model are:

- Web services with a plain XML/HTTP interface (see `http://www.xmlrpc.com/` for more information). These Web services are not SOAP-compliant.
- Services with a Representational State Transfer (REST) interface. We do not discuss REST in detail here, but more background information can be found here:

  `http://en.wikipedia.org/wiki/Representational_State_Transfer`
- Mostly SOAP-compliant Web services that require extra, unusual SOAP headers, or are not WS-I Basic profile 1.1 compliant in some exotic way (we call this *dirty* SOAP). For the purposes of this scenario, we assume a service that requires a particular extra custom SOAP header.

We collectively call these the *source* or *target protocol* (depending on direction).

6.6.2 Solution options

Assuming that we have a WebSphere ESB mediation module that wants to call out to the target service (outbound), there are three possible solutions:

- Writing a custom JCA adapter to handle the target protocol. This adapter could be specific to the scenario or more generic, and would be used as an EIS import in the mediation module.
- Invocation of the target service via a Web service proxy. In other words, ESB calls out to a stand-alone Web service via an import with an Web services binding. This Web service proxy in turn calls out to the target service.
- Writing a Java SCA component to act as a client for the target protocol.

These solutions fit the three examples discussed in 6.6.1, “Challenges and assumptions” on page 164, equally. However, for the dirty SOAP example, there is an additional possibility: use the regular Web services binding on an import, and add the additional SOAP header in a mediation flow.

There are three corresponding patterns for when one of these HTTP-based protocols wants to invoke ESB (inbound):

- Writing a custom JCA adapter to handle the source protocol. In fact, if this adapter were sufficiently generic, it could be shared with solution 1 above.
- Invocation of ESB from the source service via a servlet. In other words, the source service invokes the servlet using the source protocol. This is

translated by the servlet into a Web services invocation onto a Web service export on ESB. This is the corresponding solution to solution 2 above.

- Exposing an ESB mediation flow via a servlet and a stand-alone reference. This corresponds to solution 3 above, although it is not quite the opposite because the servlet is not an SCA component. However, solution 3 is different from solution 2, because it is more tightly bound.

These solutions fit the three examples discussed in 6.6.1, “Challenges and assumptions” on page 164, equally. Unlike the outbound situation, there is no extra possibility for the dirty SOAP case.

Analysis of solutions

In this sub-section we analyze each of the three solutions outlined above, discussing their relative merits.

Solution 1 - custom JCA HTTP Adapter

This outbound version of this solution is illustrated in Figure 6-20.

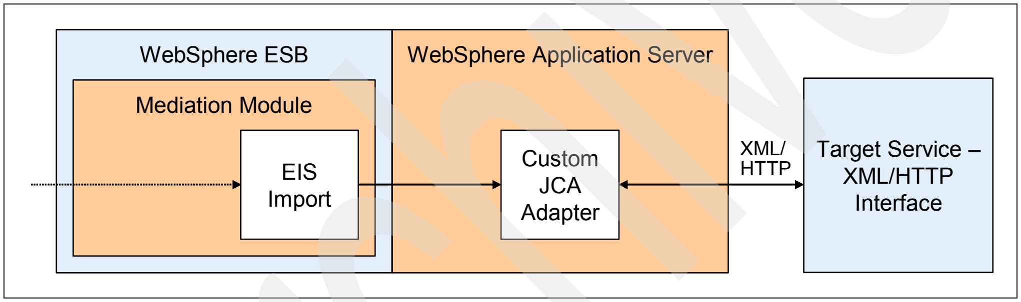


Figure 6-20 Attaching to the target service using a Custom JCA Adapter

This is the most *architecturally pure* approach to connecting to HTTP. This adapter, if well constructed, could allow any application (including J2EE applications that are not SCA-based) to connect via HTTP.

The inbound version of this solution is shown in Figure 6-21. This solution is the opposite of the outbound version.

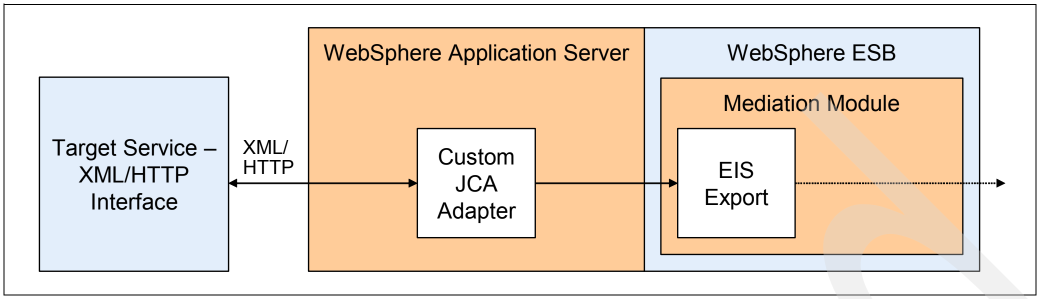


Figure 6-21 Attaching from a source service using a Custom JCA Adapter

It is not a trivial undertaking to write a custom adapter, and this solution is therefore probably the most involved of the four solutions. It is most suitable when a high Quality of Service is required, such that the adapter can integrate with the transactionality and security features provided by the J2EE layer. This adapter would be most appropriate when it is expected that connection via HTTP is something very common that is going to be needed many times in the future.

Solution 2 - invocation of the target service via a Web service proxy

The outbound version of this solution is illustrated in Figure 6-22.

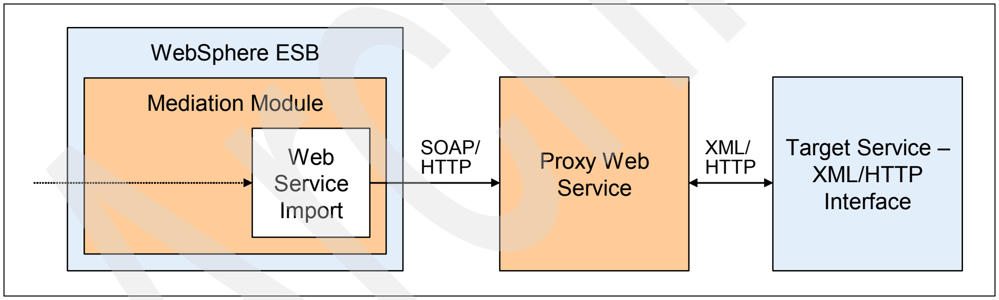


Figure 6-22 Attaching to the target service using a Web service proxy

In this solution, the mediation module would simply call out to an external Web service using a regular SOAP/HTTP-based Web service import binding, and the Web service would do the heavy lifting of converting the request to the target protocol and converting any reply back into SOAP/HTTP.

The inbound version of this solution is illustrated in Figure 6-23. This solution is almost the opposite of the outbound version, except that we use a Servlet rather than a Web service because a generic HTTP interface has to be exposed, rather than a SOAP/HTTP interface.

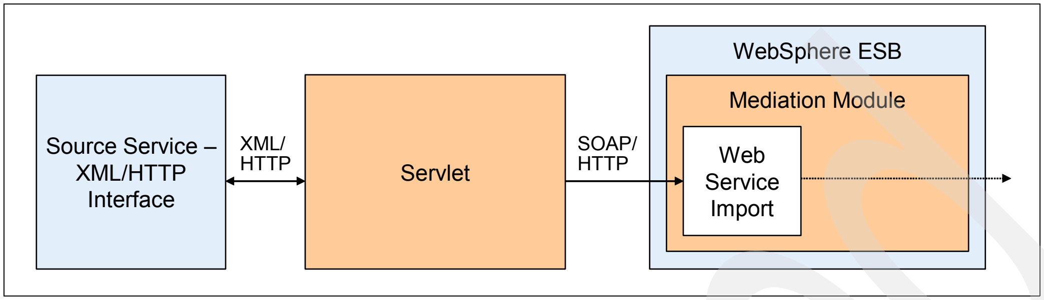


Figure 6-23 Attaching from the source service using a servlet

Because this solution requires a large part of the implementation to be outside ESB, it is suitable when the ESB mediation module cannot be modified beyond including a Web service import. Essentially, this solution uses the proxy pattern (`http://en.wikipedia.org/wiki/Proxy_pattern`). However, we decided not to investigate this solution further, as it requires an additional invocation outside the scope of SCA that has a performance and maintainability impact.

In fact, this solution does not have to use Web service bindings. It could use any of the supported binding types provided by WESB (MQ, MQ/JMS, and so on). The main reason that we have chosen Web services here is simply because it integrates neatly with the SCA model. For example, Web service transport bindings do not require you to write a data binding.

Solution 3 - Java SCA component acting as a target protocol client

The outbound version of this solution is illustrated in Figure 6-24.

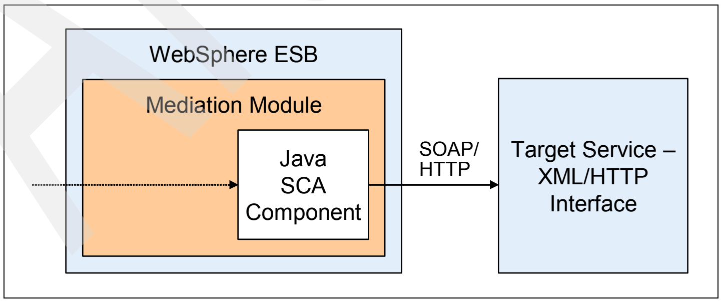


Figure 6-24 Attaching to the target service using a Java SCA component

In this solution, a Java SCA component acts directly as an HTTP client to perform the communication necessary for the target protocol.

The inbound version of this solution is illustrated in Figure 6-25. This is slightly different from the outbound version, because the servlet sits outside the mediation module. However, the stand-alone reference makes it able to invoke an SCA component (in this case the mediation flow).

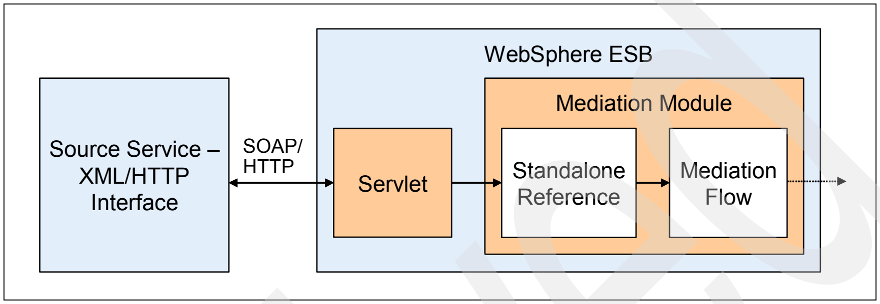


Figure 6-25 Attaching from source service using a Servlet and a standalone reference

This solution is the simplest of the four we outlined, and it also the easiest to implement. The best way to implement this, and the way we discuss further in 12.3, “Implementing inbound communication” on page 418, is to package the Java component in a separate module and to expose the module via an export. If the module were to be used only within the scope of WESB, an SCA export should be used to do this, as it is efficient, but a Web service export could also be used to expose the module outside of WESB. In the latter case, this solution is essentially equivalent to solution 2, except that the construction is simpler.

Part 3

Working examples

In this part of the book we build seven examples of connecting Enterprise Applications to WebSphere ESB. The first four example use messaging connectors -— three examples use WebSphere MQ, and one uses JMS. The fifth example uses the JDBC WebSphere Adapter to demonstrate how to use a WebSphere Adapter, and to customize the JDBC adapter to respond to events triggered by database updates. The final two examples use http connections — one inbound and one outbound.

All the examples are available as a WebSphere Integration Developer v6.0.2 Project Interchange file, the details of obtaining which you will find in Appendix A, "Additional material" on page 499.

7

Integration using WebSphere MQ

In Chapter 5, “Business scenarios” on page 129, we described an insurance company, ITSO-INS, that needs to connect a new self-service Web application to integrate with its earlier applications.

The connection pattern is described in 6.1, “Connection pattern 1: integration” on page 144, and is shown again in Figure 7-1 for convenience. As shown in the pattern, WebSphere MQ is commonly used to connect to earlier applications. See Figure 7-1.

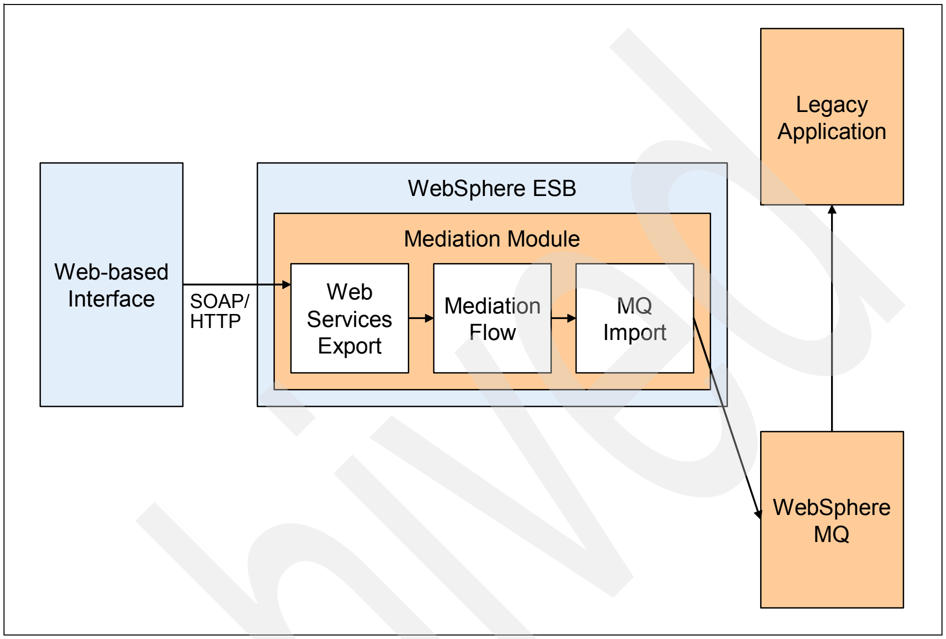


Figure 7-1 Connecting older application to WebSphere ESB via WebSphere MQ

In this chapter we show how to develop a WebSphere MQ data binding for an earlier application and use this to connect WebSphere ESB to a simple WebSphere MQ application using a regular expression parser (Figure 7-2).

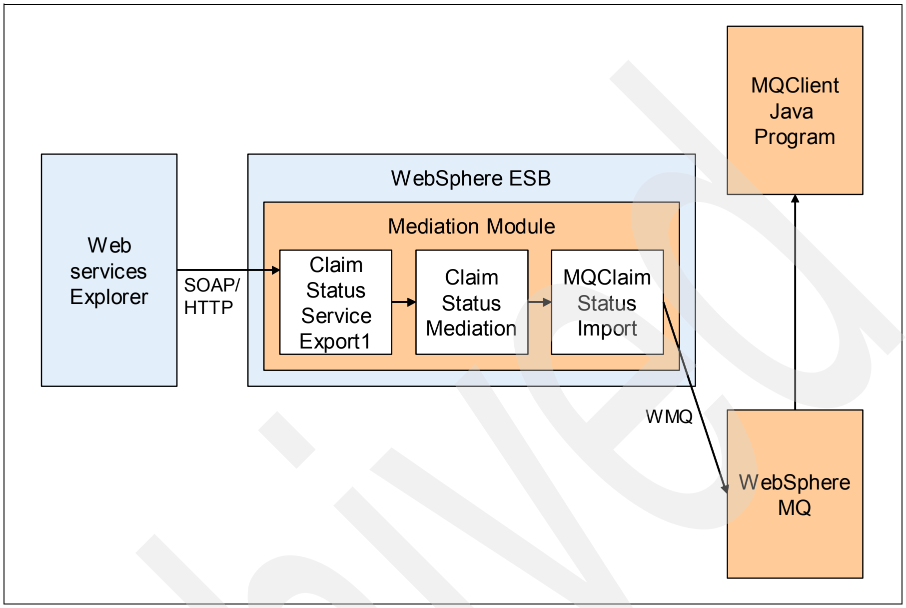


Figure 7-2 ClaimStatus realization

7.1 Custom data bindings

The business objects are part of the Service Component Architecture in SOA. Business objects define the data flowing in the Service Component Architecture. If a business object is serialized as XML then moving it in and out of WebSphere Enterprise Service Bus is easy. But in many cases that is not the case. Applications written before the advent of XML technologies represent data in various formats, for example, as tagged data, delimited data, name value pairs, or in fixed width records, and so on.

How do we connect applications that exchange data in these formats with the WebSphere Enterprise Service Bus in a simple way? The pattern we demonstrate uses WebSphere MQ to connect to the application and a custom data binding to convert the data exchange format used by the application to and from the business objects that are used in the Service Component Architecture. WebSphere Enterprise Service Bus mediates between the exchange data in the MQ message and the business objects in the service components. We need to create a custom data binding to define how WebSphere Enterprise Service bus parses the data in the WebSphere MQ message and converts it to a business object, and conversely takes a business object and serializes it into the chosen exchange format in the WebSphere MQ message.

The principles involved are the same, whatever the format of the data in the WebSphere MQ message, but the complexity of the code you need to write in the custom data binding will vary depending on the complexity of the data exchange format. One of the differentiators between the WebSphere Enterprise Service Bus as an implementation of an ESB for the Service Component Architecture and WebSphere Message Broker performing the same task is that many non-XML *data bindings* (or wire formats in WebSphere MQ Broker parlance) are either built into the WebSphere Message Broker product or can be bought as add-ons to the product from IBM or other systems integrators, whereas in the WebSphere Enterprise Service Bus, the custom data mediations are coded in Java.

In our example we use a WebSphere MQ application with delimited data as its exchange format. We show how to create a custom MQ Data Binding to transform delimited data into a business object and vice versa.

Here is an sample delimited MQ message that the Custom Data Binding builds from a business object:

```
claimID ~ 110011 ~ SA2001 ~ VinodMachineni
```

7.2 Create mediation module project

In this section we create a mediation module project and create the necessary settings to develop our application. This is the project in WebSphere Integration Developer where all of the artifacts for this chapter are saved.

7.2.1 Enable the J2EE capabilities of your workbench

To do this:

1. Open **WebSphere Integration Developer** and switch to the **Business Integration Perspective**.
2. Click **Window** → **Preferences** → **WorkBench** → **Capabilities**. Check **Advanced J2EE** and select **Apply** → **OK** (Figure 7-3).

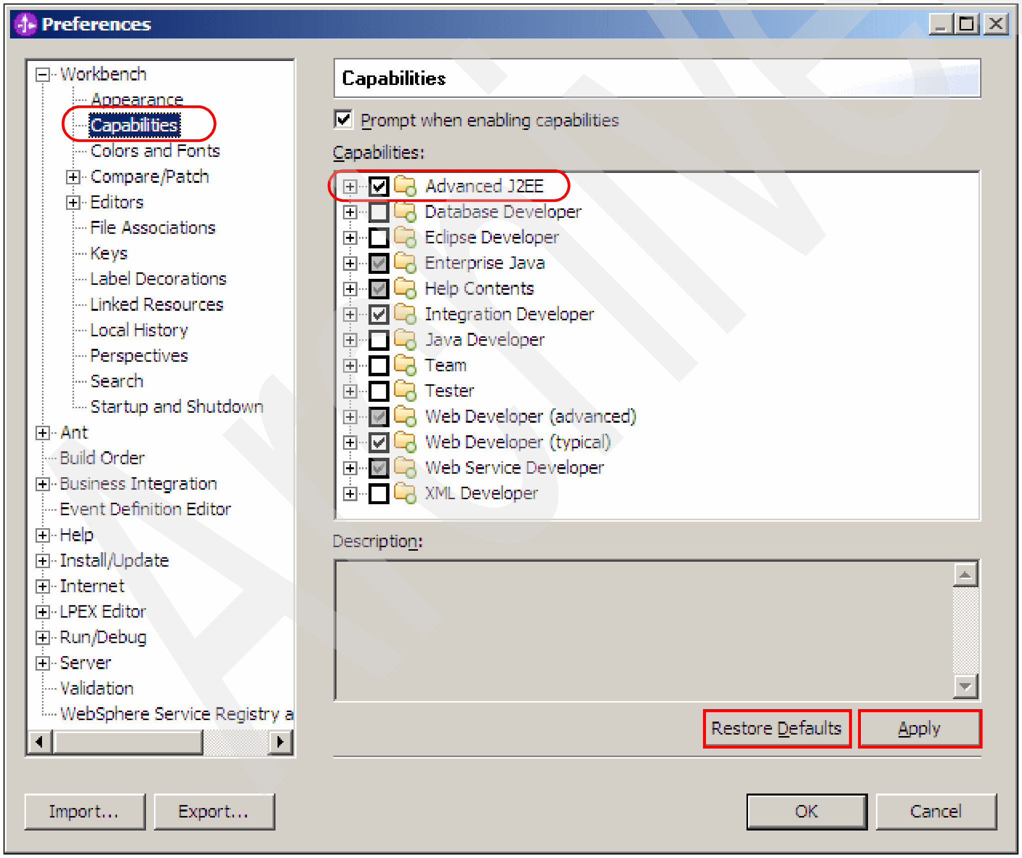


Figure 7-3 Enable workbench capabilities

7.2.2 Create the mediation module project

To do this select **File** → **New** → **Other** → **Business Integration** → **Mediation Module** → **Next** (Figure 7-4).

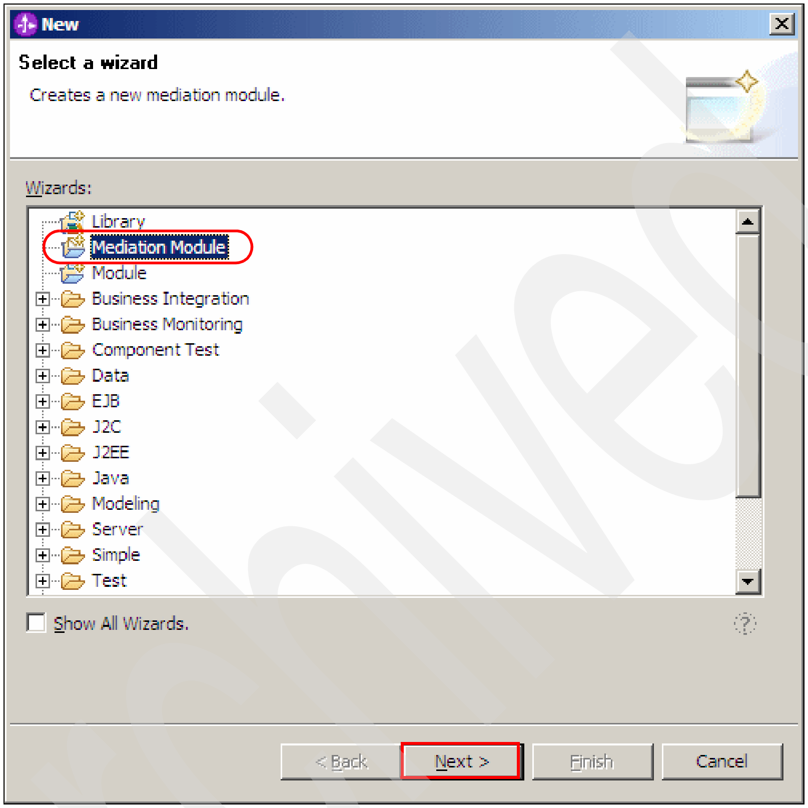


Figure 7-4 Mediation Module Creation wizard

In the New Mediation Module Panel type `ClaimStatus` in the Module Name field and uncheck Create Mediation flow component. Ensure that the target runtime environment is WebSphere ESB server v6.0 and click **Finish** (Figure 7-5).

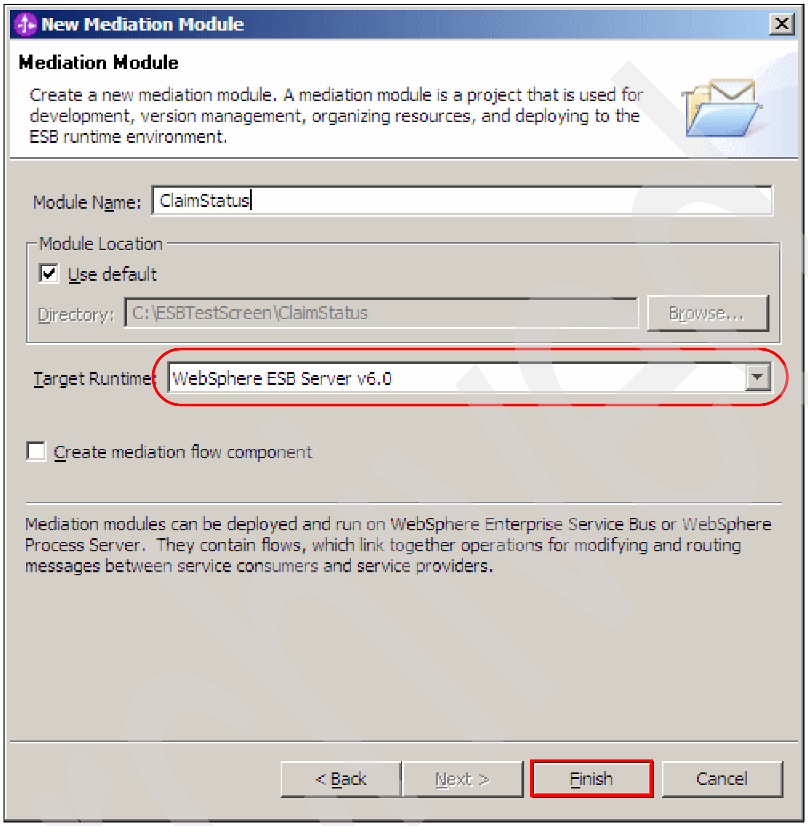


Figure 7-5 New Mediation Module panel

7.3 Create business objects

In this section we create two business objects, claimID and ClaimResult, that we use in the custom MQ Data Binding.

7.3.1 Create claimID business object

We create a new business object called claimID, with three child attributes of type String:

- policyNumber
- claimNumber
- lastName

First, create the business object:

1. Right-click **Data Types** and select **New** → **Business Object**.
2. Fill in the fields in the new Business Object Editor panel as in Figure 7-6.

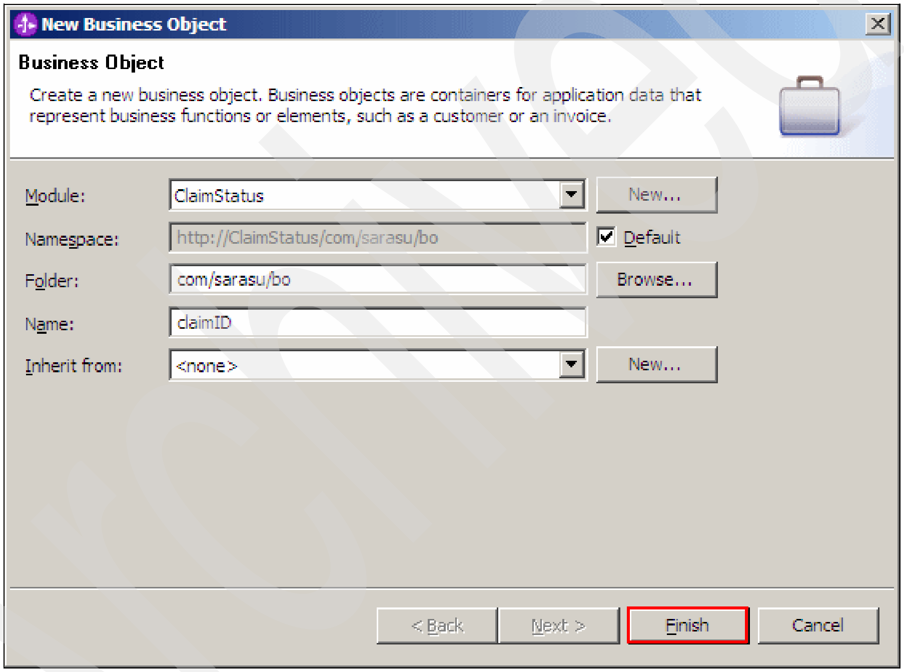


Figure 7-6 New Business Object - claimID

3. Add the attributes to the object. In the claimID BO editor either right-click the business object **claimID** and select **Add attribute** (Figure 7-7), or use the Add attribute icon in the toolbar.

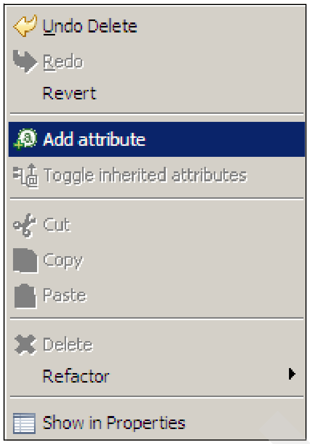


Figure 7-7 Add attribute

4. Type `policyNumber` as the name of the attribute. Check the **Required** check box (Figure 7-8).

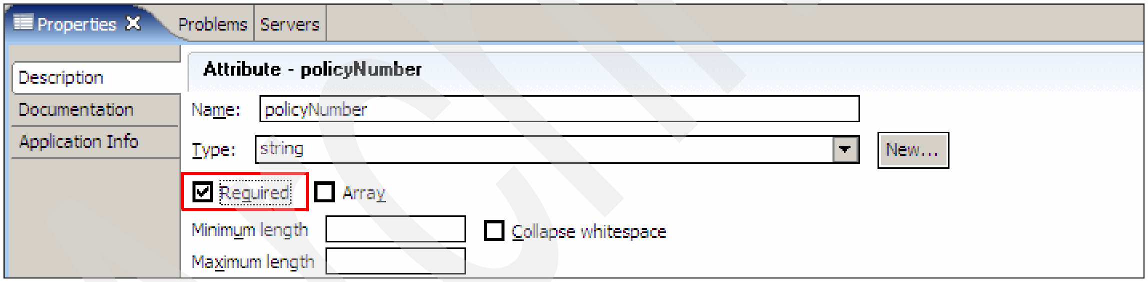


Figure 7-8 Add attribute values

5. Repeat the steps to add the attributes claimNumber and lastName to the claimID object (Figure 7-9) then select **Save** → **Close**.

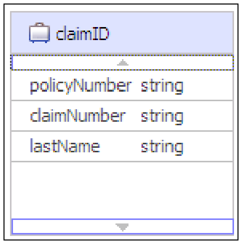


Figure 7-9 tclaimID business object

7.3.2 Create claimResult business object

Create business object claimResult with attributes claimNumber and claimStatus of type String.

1. Select **Data Types** → **New** → **Business Object**.
2. Fill in the fields in the New Business Object panel as shown in Figure 7-10.

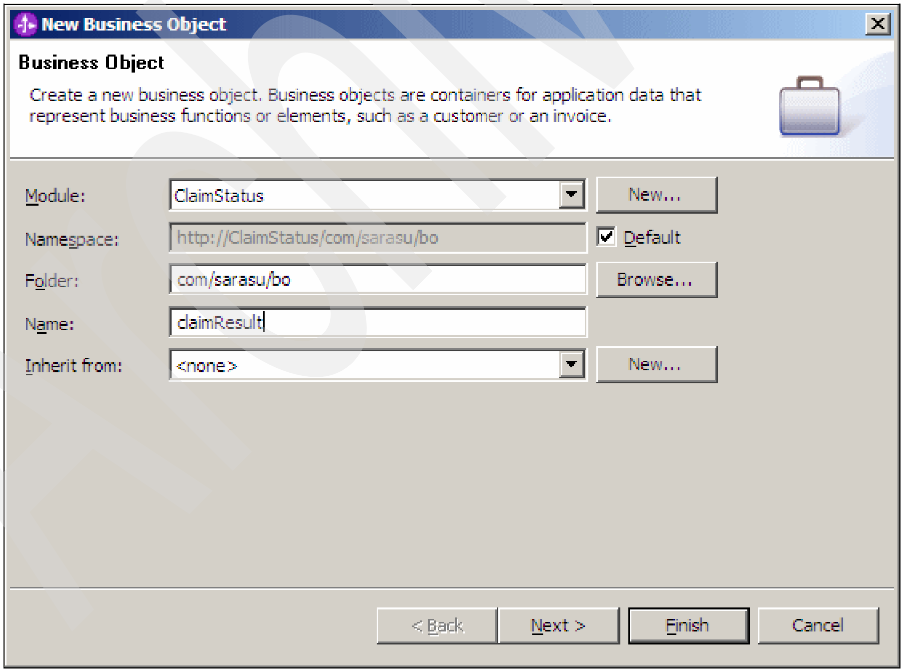


Figure 7-10 New Business Object Editor - claimStatus

3. As before, add the two required child attributes claimNumber and claimStatus and click **Save** → **Close** (Figure 7-11).

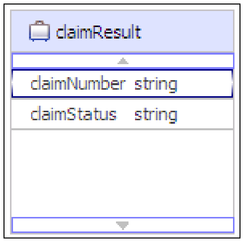


Figure 7-11 claimResult business object

7.4 Create interface

We now create an interface to the mediation component and create its imports and exports. This interface has one operation, getClaimStatus, and the input for this operation is claimID and the output is claimResult.

1. Select **Interfaces** → **New** → **Interface**.
2. Complete the New Interface Wizard panel as shown in Figure 7-12 and click **Finish**.

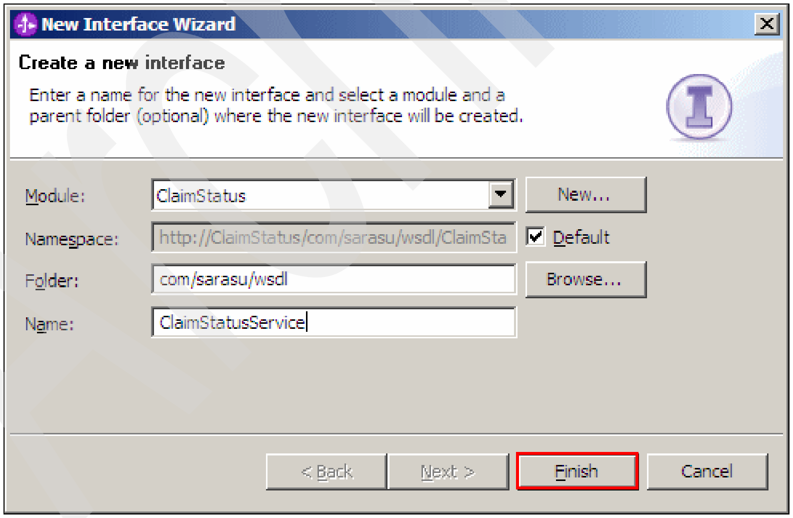


Figure 7-12 New interface wizard

3. In the Interface editor, right-click the white space and select **Add Request Response Operation** or click the Add Request Response Operation icon in the toolbar.

   The operation name is getClaimStatus.

4. For input, accept the default name (iutput1). Set the type to **claimID** using the **Browse** button.

5. For output, accept the default name (output1). Set the type to **claimResult** using the **Browse** button (Figure 7-13). Click **Save** → **Close**.

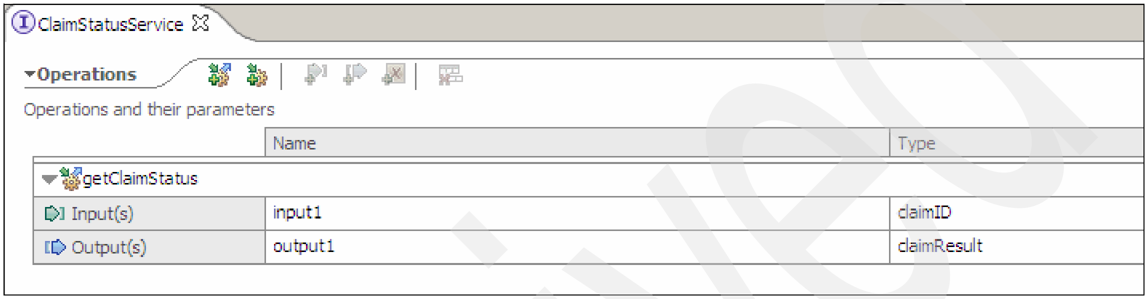


Figure 7-13 Completed interface

7.5 Implement WebSphere MQ Data Binding

We now need a custom data binding to transform the claimID business object into the following delimited text:

```
claimID ~ 110011 ~ SA2001 ~ BhagathNamburu
```

We also need a Custom Data Binding to transform the delimited text below to a claimResult business object.

```
claimResult ~ SA2001 ~ Approved
```

1. Select **ClaimStatus project** → **New** → **Other** → **Java** → **Class** → **Next** (Figure 7-14).

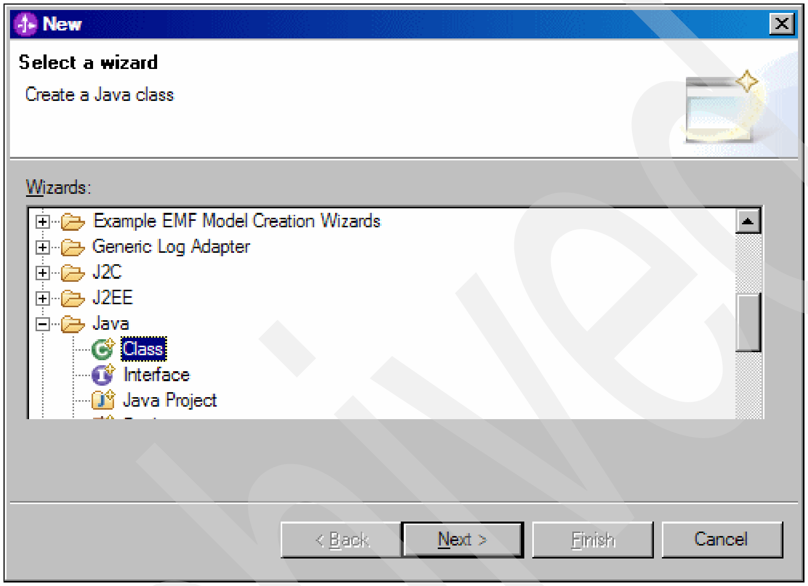


Figure 7-14 Select a Java class wizard

2. In the New Java Class form select **Source Folder** → **Browse** → **ClaimStatus** → **OK** (Figure 7-15).

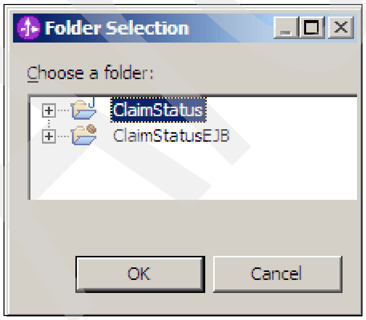


Figure 7-15 Folder Selection for binding

3. Select **Package** → **Browse** → **com.sarasu** → **OK**.

4. Type `MQBodyDataBindingDelimited` as the name.

5. Select **Interfaces** → **Add**. Select the **MQBodyDataBinding** interface, then **OK** (Figure 7-16).

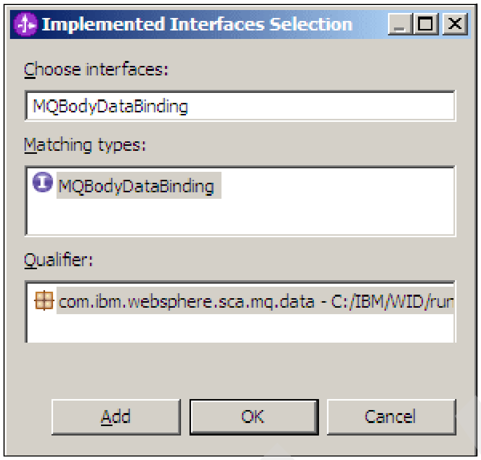


Figure 7-16 Implemented Interfaces Selection

6. Deselect Constructors from superclass and click **Finish** (Figure 7-17).

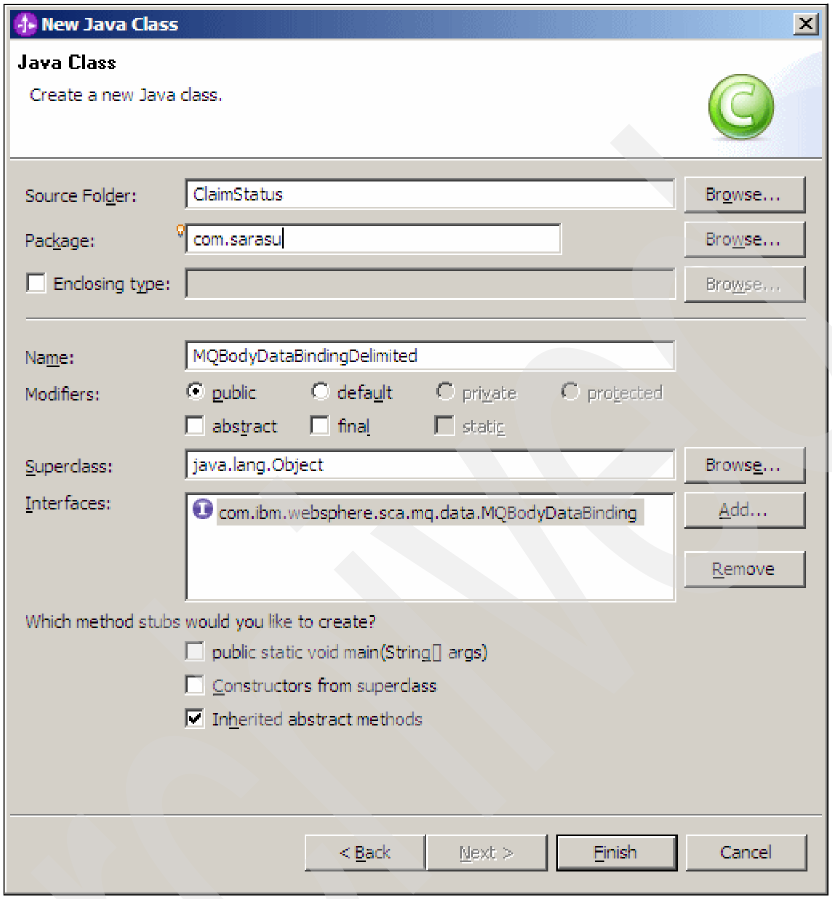


Figure 7-17 New Java Class - MQBodyDataBindingDelimited

Add the following static constants and a dataObject instance of the business object to the MQBodyDataBindingDelimited class, which has been opened in the editor panel (Figure 7-18).

```
public class MQBodyDataBindingDelimited implements MQBodyDataBinding {
   private static final String DELIMITER = "~";
   static final private String SPACEDDELIMITER = " " + DELIMITER + " ";
   static final private String REGEXDELIMITER = "\\s" + DELIMITER + "\\s";
   static final private int CLAIMFIELD = 1;
   static final private int CLAIMSTATUS = 2;
   static final private String NAMESPACE = "http://ClaimStatus/com/sarasu/bo";
   static final private String OBJECTNAME = "claimResult";
   // DataObject that is built from the request message
   // It is also used to build the response messages
   public DataObject dataObject;
```

Figure 7-18 Class and instance variables in MQBodyDataBindingDelimited

It is important that the format matches the format string set up in the SMO as described in Example 3-9 on page 94 and used by the isSupportedFormat() method on the MQBodyDataBinding class.

Next you need to implement the code for read() and write(). Remember that this is understood from the ESB point of view, so read() reads from the WebSphere MQ queue and write() writes to the WebSphere MQ queue. The read code is given in Figure 7-19 and the write code in Figure 7-20 on page 189. You need to organize imports to pick up all the referenced types.

```
   public void read(MQMD md, List headers, MQDataInputStream input)
         throws IOException {
      dataObject = DataFactory.INSTANCE.create(NAMESPACE, OBJECTNAME);
      System.out.println("Building a Business Object from MQ Text");
      String myInputStringMessage = new String(input.readBytes(input.available()));
      // Incoming message like: claimResult ~ SA2001 ~ Approved
      String[] inputTokens = myInputStringMessage.split(REGEXDELIMITER);
      dataObject.setString(CLAIMNUMBER, inputTokens[CLAIMFIELDn]);
      dataObject.setString(CLAIMSTATUS, inputTokens[CLAIMSTATUSn]);
      System.out.println(CLAIMNUMBER + " is: "
            + dataObject.getString(CLAIMNUMBER) + " "
            + CLAIMSTATUS + " is: " + dataObject.getString(CLAIMSTATUS));
   }
```

Figure 7-19 read() method

The read() method creates a new business object if one is needed, and then parses the delimited WebSphere MQ message data body, which is passed as a ByteArray in arg2.

The other methods you need to alter are the set and getDataObject to create a DataObject instance in the DataBinding object for a mediation that will be writing the DataObject onto a WebSphere MQ queue and to return the DataObject instance from the DataBinding object to the ESB once it has been created by reading the data from a WebSphere MQ queue. These methods are shown in Figure 7-21.

The complete class is given in Example 7-1.

```
public void write(MQMD arg0, List arg1, MQDataOutputStream arg2)
      throws IOException {
      System.out.println("Building MQ Text from a business object");
      final String S1 = "claimID" + SPACEDDELIMITER
            + dataObject.getString("policyNumber") + SPACEDDELIMITER
            + dataObject.getString("claimNumber") + SPACEDDELIMITER
            + dataObject.getString("lastName");
      System.out.println(S1);
      arg2.writeMQCHAR(S1);
}
```

Figure 7-20 write() method

```
public DataObject getDataObject() throws DataBindingException {
   return dataObject;
}
public void setDataObject(DataObject arg0) throws DataBindingException {
   this.dataObject = arg0;
}
```

Figure 7-21 Getting and setting the dataObject instance

Example 7-1 MQBodyDataBindingDelimited.java

```
package com.sarasu;
import java.io.IOException;
import java.util.List;
import com.ibm.mq.data.MQDataInputStream;
import com.ibm.mq.data.MQDataOutputStream;
import com.ibm.websphere.bo.BOFactory;
import com.ibm.websphere.sca.ServiceManager;
```

```
import com.ibm.websphere.sca.mq.data.MQBodyDataBinding;
import com.ibm.websphere.sca.mq.structures.MQMD;
import commonj.connector.runtime.DataBindingException;
import commonj.sdo.DataObject;

public class MQBodyDataBindingDelimited implements MQBodyDataBinding {
   private static final String DELIMITER = "~";
   static final private String SPACEDDELIMITER = " " + DELIMITER + " ";
   static final private String REGEXDELIMITER = "\\s" + DELIMITER + "\\s";
   static final private int CLAIMFIELD = 1;
   static final private int CLAIMSTATUS = 2;
   static final private String NAMESPACE = "http://ClaimStatus/com/sarasu/bo";
   static final private String OBJECTNAME = "claimResult";
   // DataObject that is built from the request message - It is also used to
   // build the response messages
   public DataObject dataObject;

   public void read(MQMD md, List headers, MQDataInputStream input)
         throws IOException {
      dataObject = DataFactory.INSTANCE.create(NAMESPACE, OBJECTNAME);
      System.out.println("Building a Business Object from MQ Text");
      String myInputStringMessage = new String(input.readBytes(input.available()));
      // Incoming message like: claimResult ~ SA2001 ~ Approved
      String[] inputTokens = myInputStringMessage.split(REGEXDELIMITER);
      dataObject.setString(CLAIMNUMBER, inputTokens[CLAIMFIELDn]);
      dataObject.setString(CLAIMSTATUS, inputTokens[CLAIMSTATUSn]);
      System.out.println(CLAIMNUMBER + " is: "
            + dataObject.getString(CLAIMNUMBER) + " "
            + CLAIMSTATUS + " is: " + dataObject.getString(CLAIMSTATUS));
   }
   public void write(MQMD arg0, List arg1, MQDataOutputStream arg2)
         throws IOException {
         System.out.println("Building MQ Text from a business object");
         final String S1 = "claimID" + SPACEDDELIMITER
               + dataObject.getString("policyNumber") + SPACEDDELIMITER
               + dataObject.getString("claimNumber") + SPACEDDELIMITER
               + dataObject.getString("lastName");
         System.out.println(S1);
         arg2.writeMQCHAR(S1);
   }
   public void setBusinessException(boolean arg0) {
      // TODO Auto-generated method stub
   }
   public boolean isBusinessException() {
      // TODO Auto-generated method stub
```

```
      return false;
   }
   public void setFormat(String arg0) {
      // TODO Auto-generated method stub
   }
   public String getFormat() {
      // TODO Auto-generated method stub
      return null;
   }
   public DataObject getDataObject() throws DataBindingException {
      return dataObject;
   }
   public void setDataObject(DataObject arg0) throws DataBindingException {
      this.dataObject = arg0;
   }
}
```

7.6 Assemble the mediation

We have created the business objects, interface, and custom MQ DataBinding we need to create the mediation flow for the scenario.

The mediation flow has a Web services export and an MQ Messaging export with our Custom data binding.

7.6.1 Add mediation flow component

In the ClaimStatus project double-click **Assembly diagram** to open the Assembly diagram editor.

1. From the left-hand palette click the Mediation flow icon  and drop it on the canvas. Name the mediation `ClaimStatusMediation` (Figure 7-22).

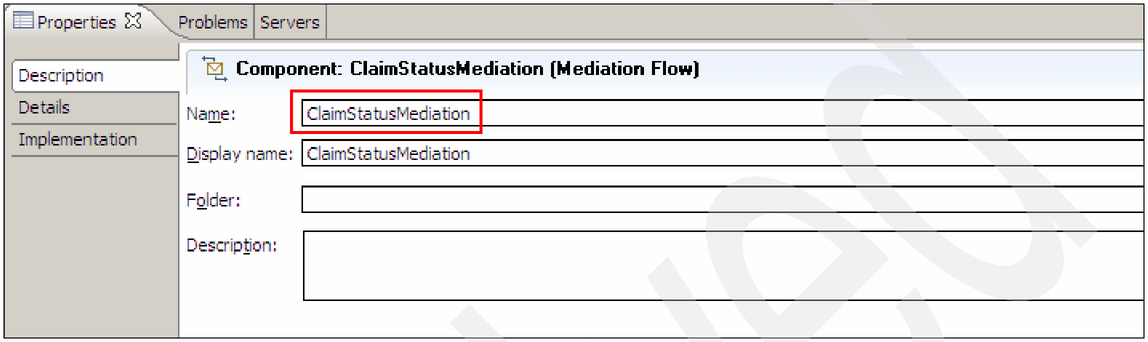


Figure 7-22 Claim Status Mediation name and description

2. Right-click **ClaimStatusMediation** and select **Add** → **Interface** (Figure 7-23). Select the **ClaimStatusService** interface you created earlier (Figure 7-24 on page 193).

   Alternatively, you could drag the interface from Explorer and drop it on the mediation flow.

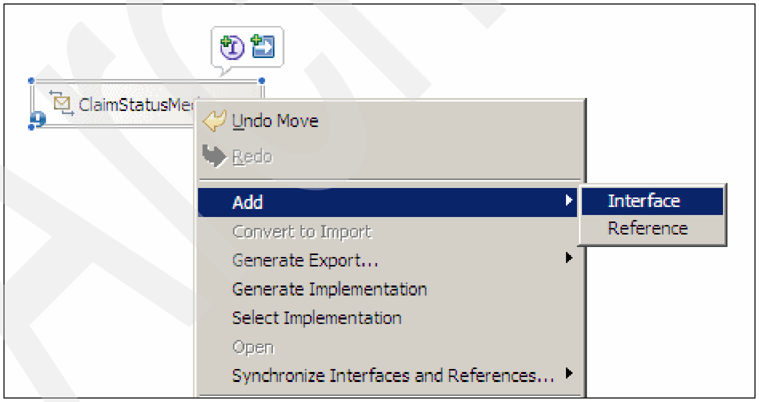


Figure 7-23 Add interface to ClaimStatusMediation

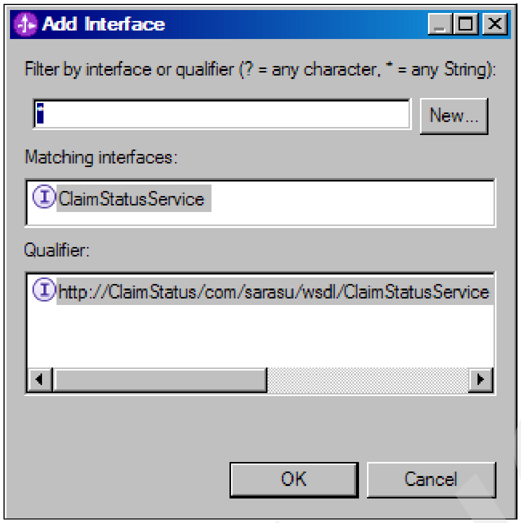


Figure 7-24   Add interface to mediation

7.6.2 Generate a SOAP/http Web service export

To do this right-click **ClaimStatusMediation** and select **Generate Export** → **Web service binding** (Figure 7-25). In the Transport selection pane select **soap/http** (Figure 7-26 on page 195).

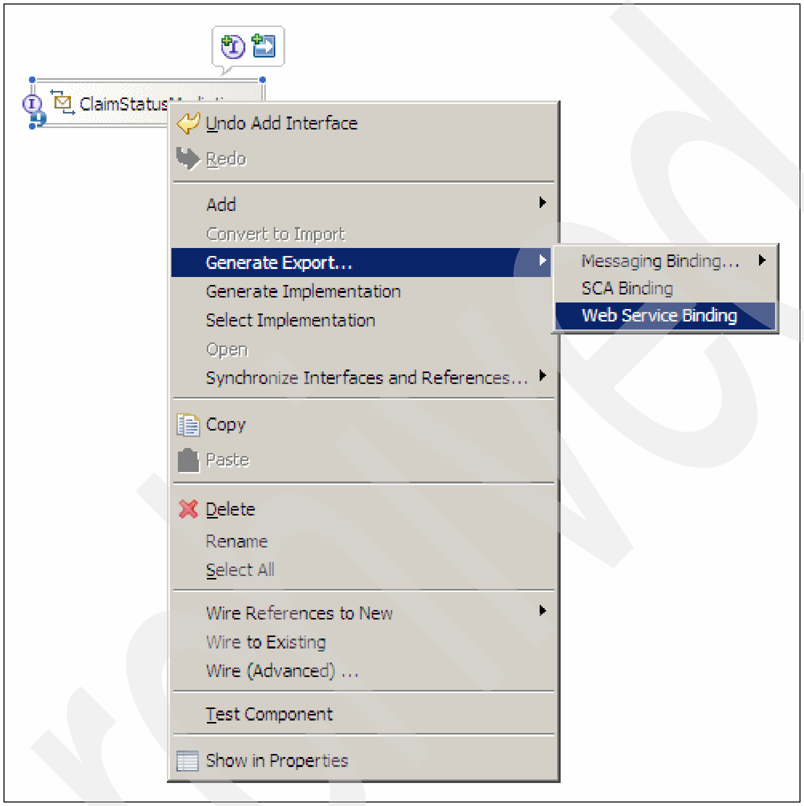


Figure 7-25 Generate Web service Export

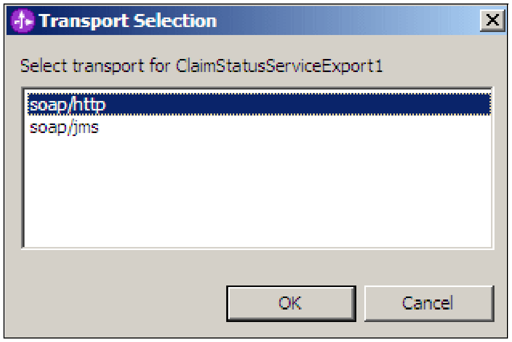


Figure 7-26 SOAP/HTTP Transport Selection

7.6.3 Create an import

To do this:

1. From the left-hand palette click the Import flow icon and drop it on the canvas. Name the import `ClaimStatusServiceMQImport`.
2. Add a wire from the mediation ClaimStatusService to the import ClaimStatusServiceMQImport. You are prompted to add a matching reference (Figure 7-27). Click **OK**.

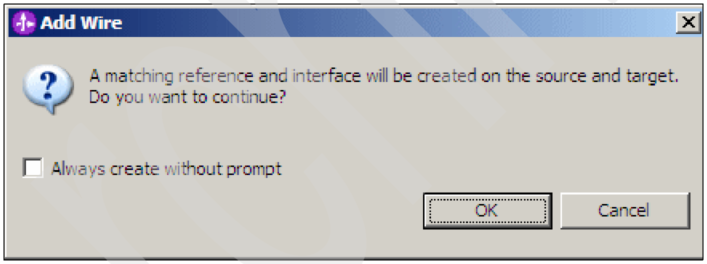


Figure 7-27 Add Wire to import prompt

3. In the Add Reference pane select **ClaimStatusService** from the Matching interfaces and click **OK** (Figure 7-28).

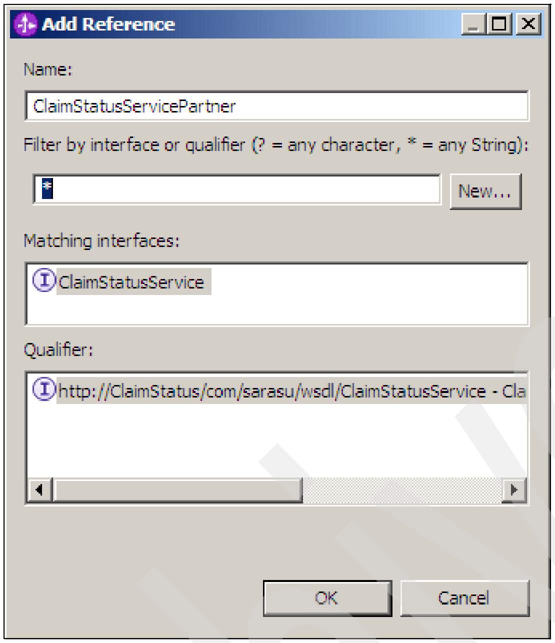


Figure 7-28 ClaimStatusService Reference

7.6.4 Bind the import to WebSphere MQ

To do this:

1. Right-click **ClaimStatusServiceMQImport** and select **Generate Bindings** → **Messaging Binding** → **MQ Binding** (Figure 7-29).

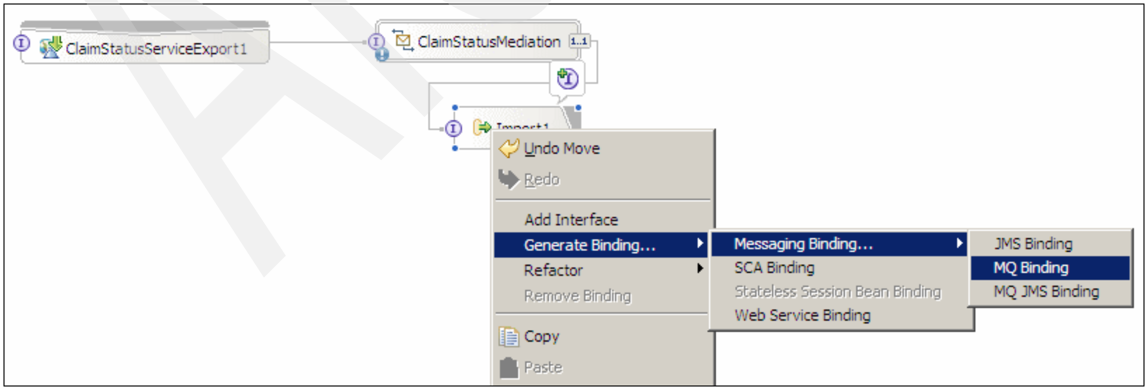


Figure 7-29 Generate MQ binding

2. In the MQ Import Bindings settings pane leave the radio button next to “Specify properties for configuring WebSphere MQ Resources” selected.
3. Fill in the form as shown in Figure 7-31 on page 198. When you select **User Supplied** as the serialization type use the spin boxes to select the **MQBodyDataBindingDelimited** class you have just written as the serialization routine (Figure 7-30).

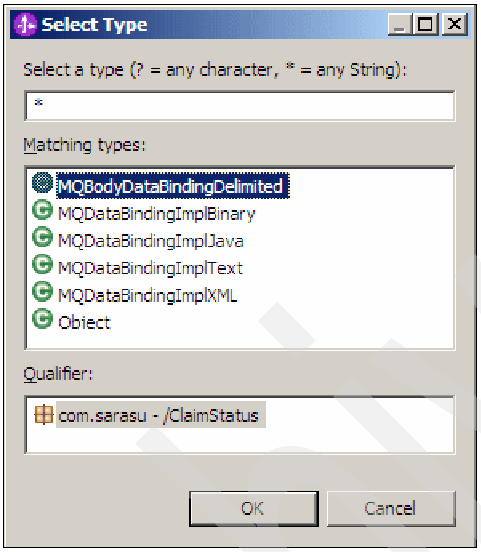


Figure 7-30 Select MQBodyDataBindingDelimited as the user-defined serializer class

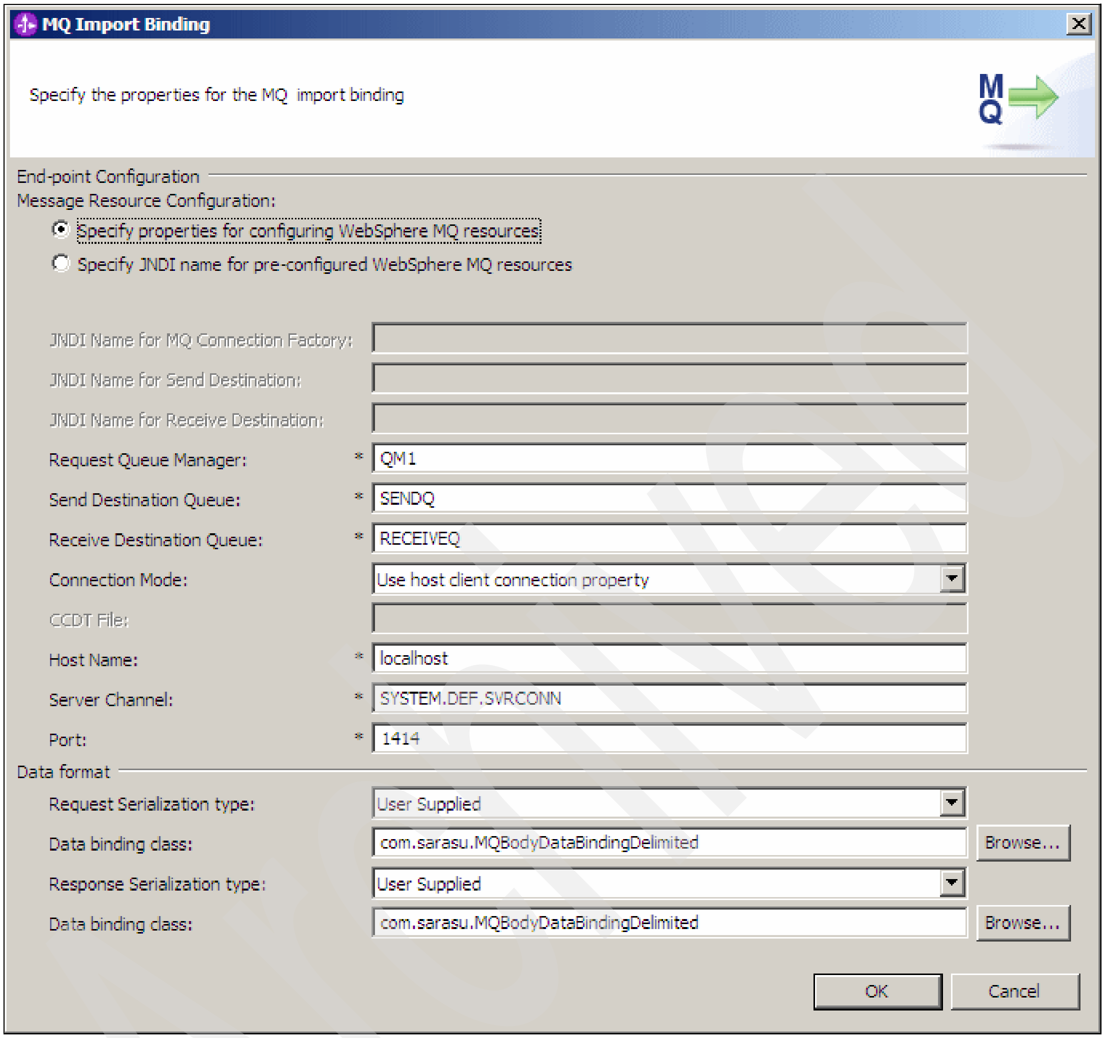


Figure 7-31 MQ Data binding properties

7.7 Implement the mediation flow

The next step is to implement the mediation flow component.

1. Double-click the **ClaimStatusMediation** component in the assembly diagram. Click **Yes** when you are prompted to do the implementation (Figure 7-32).

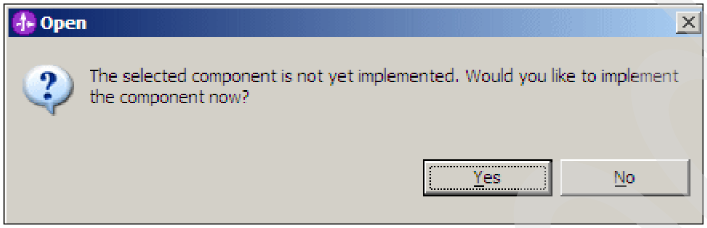


Figure 7-32 Prompt for the implementation of mediation

2. In the Generate Implementation panel (Figure 7-34 on page 200) click **New Folder**. Type `com/sarasu/mediation` and then click **OK** (Figure 7-33). The mediation flow editor opens.

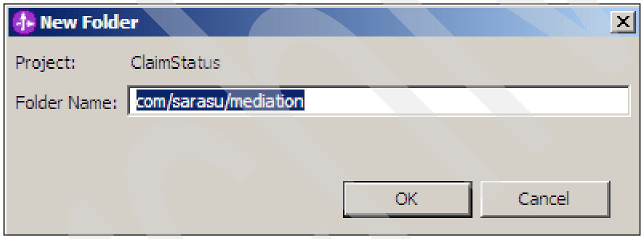


Figure 7-33 Mediation folder creation

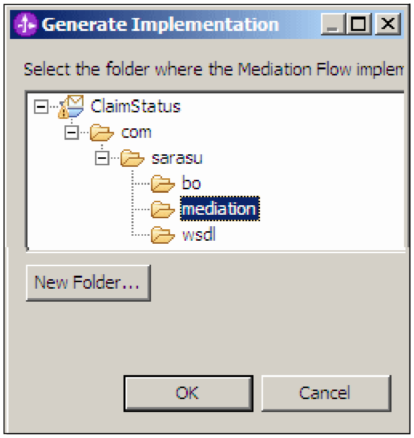


Figure 7-34 Select mediation folder

This mediation is to be *no-op*. The input data is passed through to the output.

1. Drag a wire from the operation getClaimStatus in ClaimStatusService to operation getClaimStatus in ClaimStatusServicePartner (Figure 7-35).

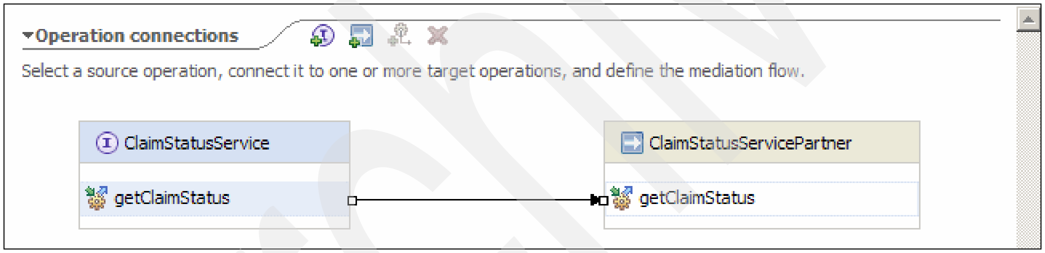


Figure 7-35 Operation connections

2. In the lower section of the editor select the **Request** tab, select the **out** terminal of the Input, and wire to the **in** terminal of the Callout (Figure 7-36).

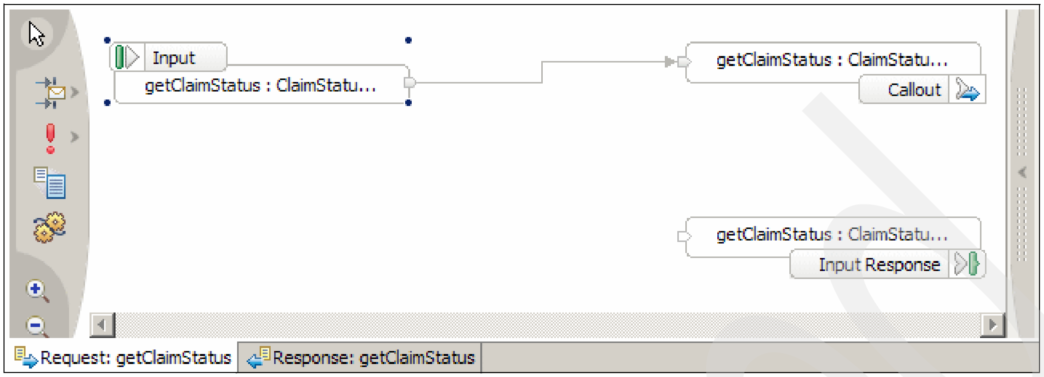


Figure 7-36 Wire request input to callout

3. Repeat this for the Response tab to wire the out terminal of the CalloutReponse to the in terminal of the InputResponse (Figure 7-37). Then click **Save** → **Close**.

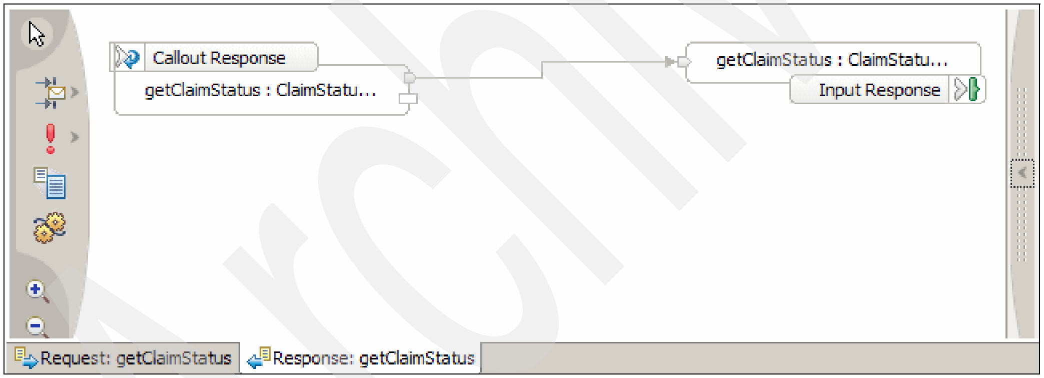


Figure 7-37 Wire callout response to InputResponse

7.8 MQClient implementation

Before testing the mediation we need to set up an WebSphere MQ application to receive the outbound WebSphere MQ message on SENDQ from the mediation, and another application to send the results of the application to RECEIVEQ. We have already set up an SOAP/http export to the mediation that we can test with a Web service explorer.

The message in the SENDQ is in this format:

```
claimID ~ 110011 ~ SA2001 ~ RameshTangellamudi
```

The message in the RECEIVEQ is in this format:

```
claimResult ~ SA2001 ~ Approved
```

7.8.1 Create MQClient Java program

To do this:

1. Right-click the **ClaimStatus** project and select **New** → **Other**. In the Select a Wizard panel, expand **Java** → **Class** → **Next** (Figure 7-38).

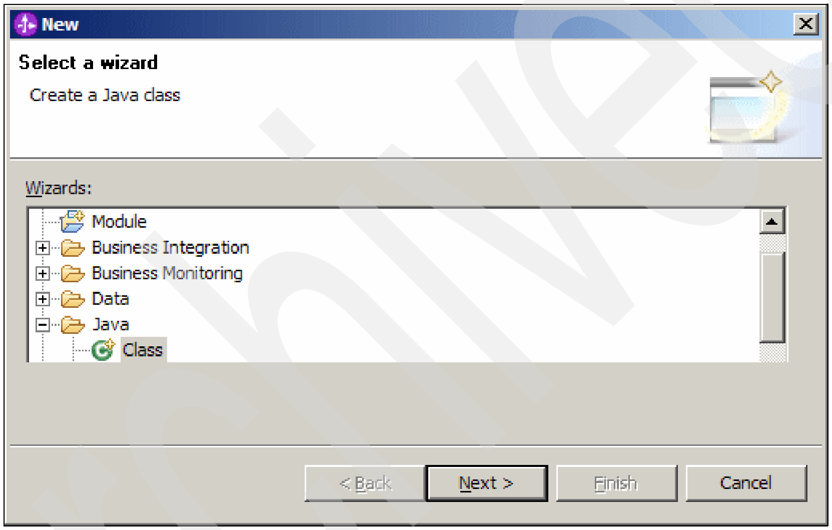


Figure 7-38 Select wizard

2. Click **Browse** → **Source Folder** → **ClaimStatus** → **OK** (Figure 7-39).

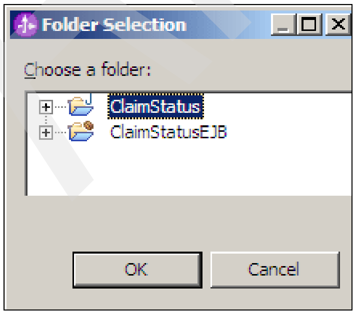


Figure 7-39 Folder selection for binding class

3. Browse to com.sarasu for the package type MQClient (Figure 7-40) and click **Finish**.

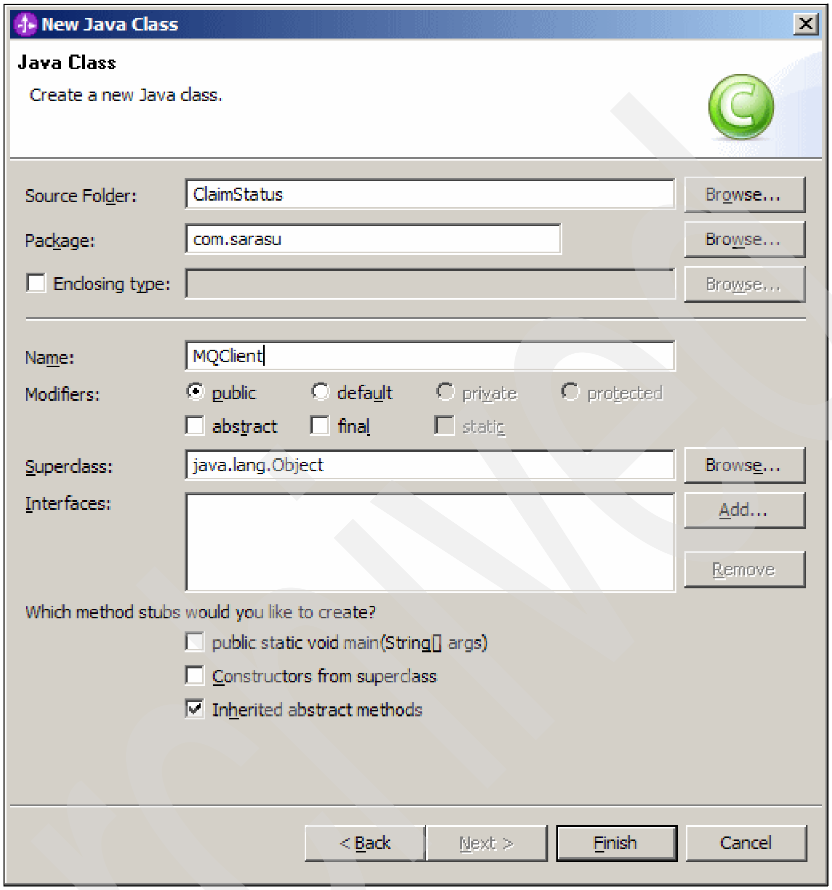


Figure 7-40 New Java class details for MQClient

Add the WebSphere MQ jars we need to the build path.

1. Right-click the **ClaimStatus** and select **Properties** → **Java Build Path**. Select the libraries tab **Add External JARs** (Figure 7-41).

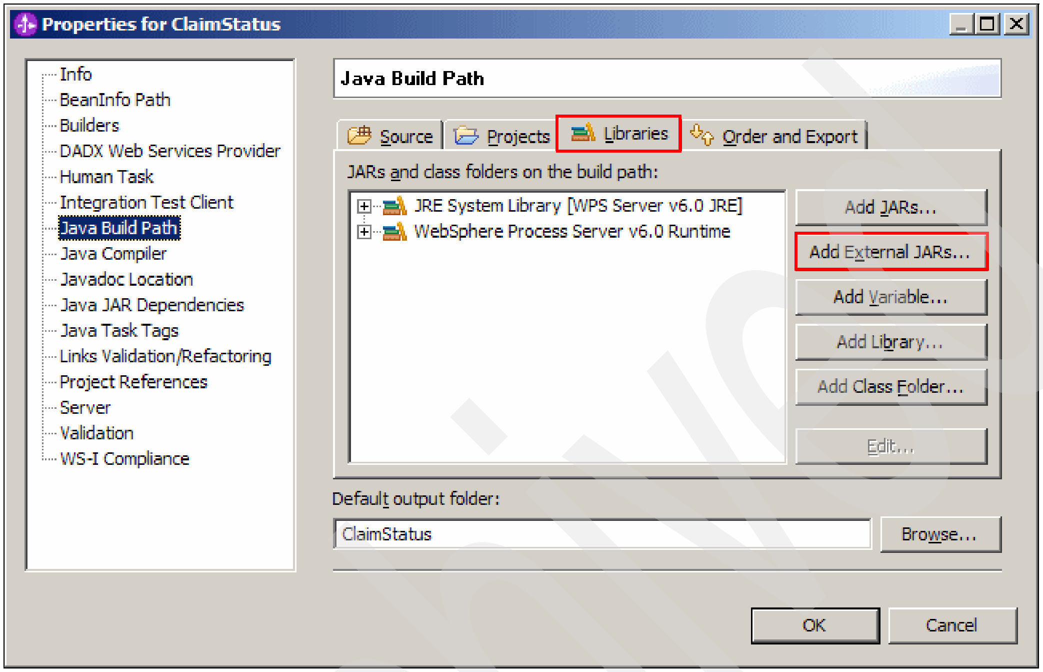


Figure 7-41 Java Build Path

2. Browse to <WebSphere MQ Install Root>\Java\lib. Select the .jar files shown in Figure 7-42 and click **Open** → **OK**. The resulting build path is shown in Figure 7-43 on page 206.

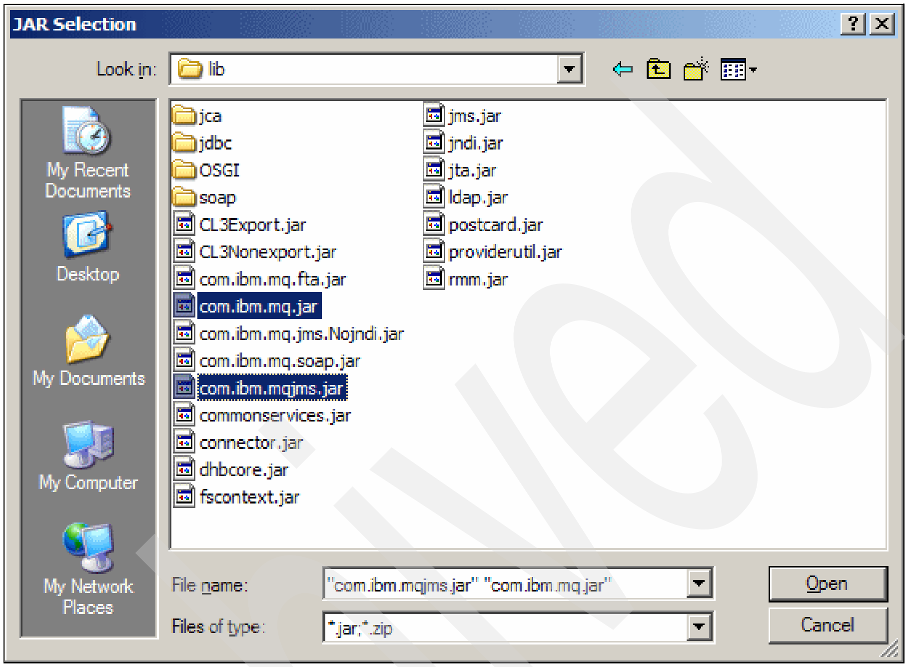


Figure 7-42 Jar file selection

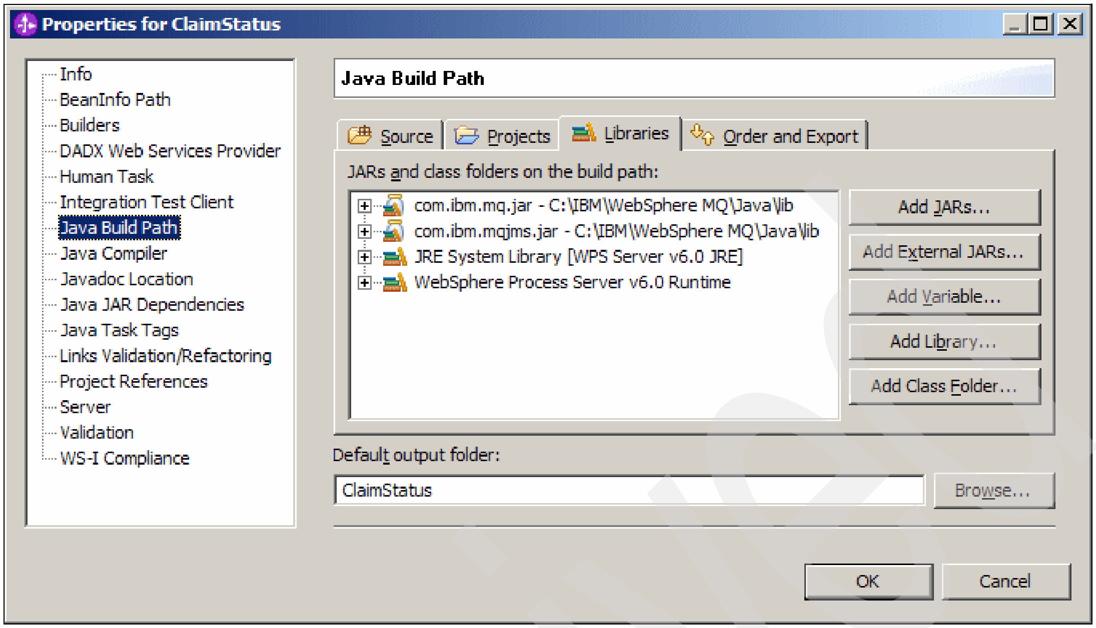


Figure 7-43 Properties for ClaimStatus project

Replace the Java code in the MQClient.java program with the code in Example 7-2. The program reads in the delimited text from a WebSphere MQ message on SENDQ, tokenizes it using the regular expression parser provided by the *split* method, and writes the result onto the RECEIVEQ.

Example 7-2 MQClient java program

```
package com.sarasu;
import com.ibm.mq.MQC;
import com.ibm.mq.MQEnvironment;
import com.ibm.mq.MQGetMessageOptions;
import com.ibm.mq.MQMessage;
import com.ibm.mq.MQQueue;
import com.ibm.mq.MQQueueManager;
public class MQClient {
   static final private String QMGR = "QM1";
   static final private String HOSTNAME = "localhost";
   static final private String CHANNEL = "SYSTEM.DEF.SVRCONN";
   static final private String TRANSPORT = MQC.TRANSPORT_MQSERIES_CLIENT;
   static final private String SENDQ = "SENDQ";
   static final private String RECEIVEQ = "RECEIVEQ";
   static final private String DELIMITER = "~";
   static final private String SPACEDDELIMITER = " " + DELIMITER + " ";
```

```
static final private int CLAIMFIELD = 2;
static final private int CLAIMSTATUS = 2;
private MQQueueManager qMgr;

public static void main(String args[]) {
   System.out.println("Starting MQClient");
   new MQClient();
   System.out.println("Stopping MQClient");
}

public MQClient() {
   try {
      // Session Constants
      MQEnvironment.properties.put(MQC.TRANSPORT_PROPERTY, TRANSPORT);
      MQEnvironment.properties.put(MQC.CHANNEL_PROPERTY, CHANNEL);
      MQEnvironment.hostname = HOSTNAME;
      qMgr = new MQQueueManager(QMGR);
      // Get Bytes message from SENDQ, convert into String
      MQMessage retrievedMessage = new MQMessage();
      MQQueue sendQ = qMgr.accessQueue(SENDQ, MQC.MQOO_INPUT_AS_Q_DEF);
      MQGetMessageOptions gmo = new MQGetMessageOptions();
      gmo.options = MQC.MQGMO_WAIT + MQC.MQGMO_FAIL_IF_QUIESCING;
      gmo.waitInterval = MQC.MQWI_UNLIMITED;
      sendQ.get(retrievedMessage, gmo);
      byte[] myByte = new byte[retrievedMessage.getTotalMessageLength()];
      for (int i = 0; i < myByte.length; i++) {
         myByte[i] = retrievedMessage.readByte();
      }
      String myInputStringMessage = new String(myByte);
      System.out.println("The input message is: " + myInputStringMessage);
      // claimID ~ 110011 ~ SA2001 ~ RameshTangellamudi
      // Parse using regular expression
      String [] inputTokens =  myInputStringMessage.split("\\s"+DELIMITER+"\\s");
      String myOutputStringMessage = "The message is claimResult" +
      SPACEDDELIMITER + inputTokens[CLAIMFIELD] + SPACEDDELIMITER + "Approved";
      // claimResult ~ SA2001 ~ Approved
      System.out.println("The output message is: " + myOutputStringMessage);
      // Now output the result message onto RECEIVEQ
      MQQueue recvQ = qMgr.accessQueue(RECEIVEQ, MQC.MQOO_OUTPUT);
      MQMessage sendMessage = new MQMessage();
      sendMessage.correlationId = retrievedMessage.messageId;
      sendMessage.writeBytes(myOutputStringMessage);
      recvQ.put(sendMessage);
      // Close down gracefully
      sendQ.close();
```

```
            recvQ.close();
            qMgr.disconnect();
         } catch (Exception e) {
            e.printStackTrace();
         }
      }
   }
```

7.9 Deploy and test the application

We are now ready to deploy the mediation on to the server, run it, and test it. The first step is to set up the servers.

7.9.1 Create servers

To do this:

1. Create a new WebSphere ESB server and start it in WebSphere Integration developer, if necessary.
2. Launch WebSphere MQ Explorer and create Queue Manager: QM1 and local queues SENDQ and RECEIVEQ (Figure 7-44).

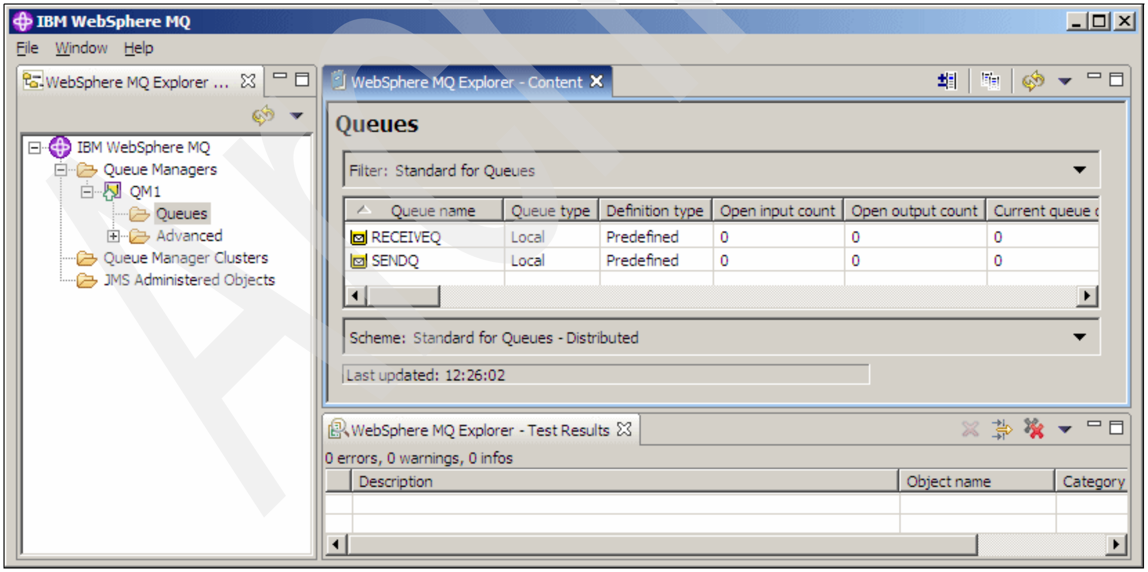


Figure 7-44 QM1, SENDQ, and RECEIVEQ

7.9.2 Deploy applications

First deploy the mediation and then create a test configuration to run the MQClient Java program.

1. In the Servers view select **WebSphere ESB server** → **Add and Remove projects**. In the Add and Remove Projects panel choose to add **ClaimStatusApp** and then click **Finish** (Figure 7-45).

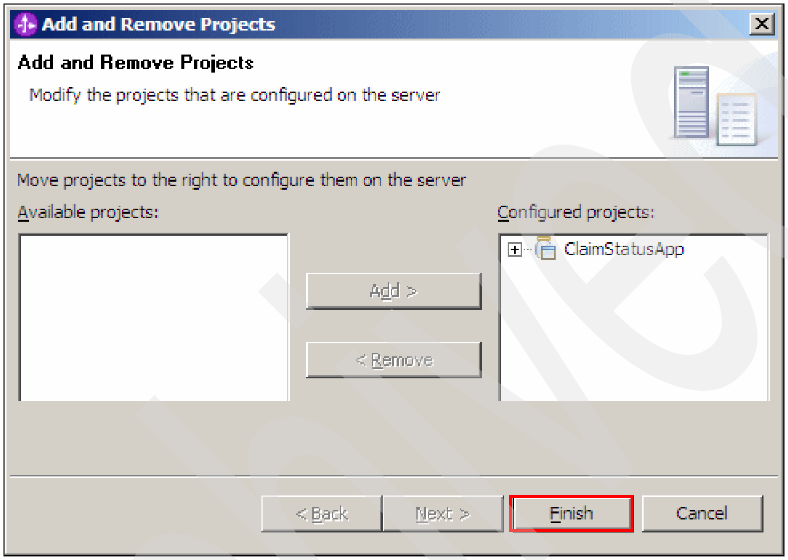


Figure 7-45 Add ClaimStatusApp project

2. Click the **Publish** button on the server view. Wait for the application to start cleanly. This may take several minutes.

3. Switch to the Java perspective. Right-click **MQClient.Java** and select **Run** → **Java Application** (Figure 7-47 on page 211). On the Create, manage, and run configurations panel select **Run** (Figure 7-46).

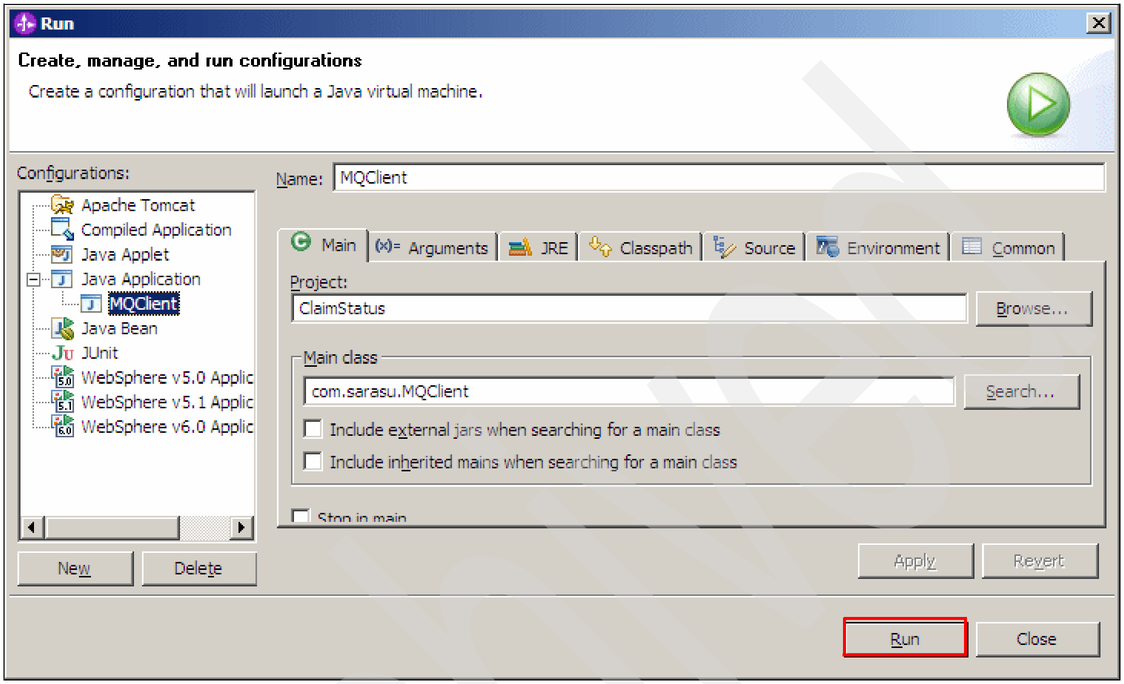


Figure 7-46 Run the MQClient Java program

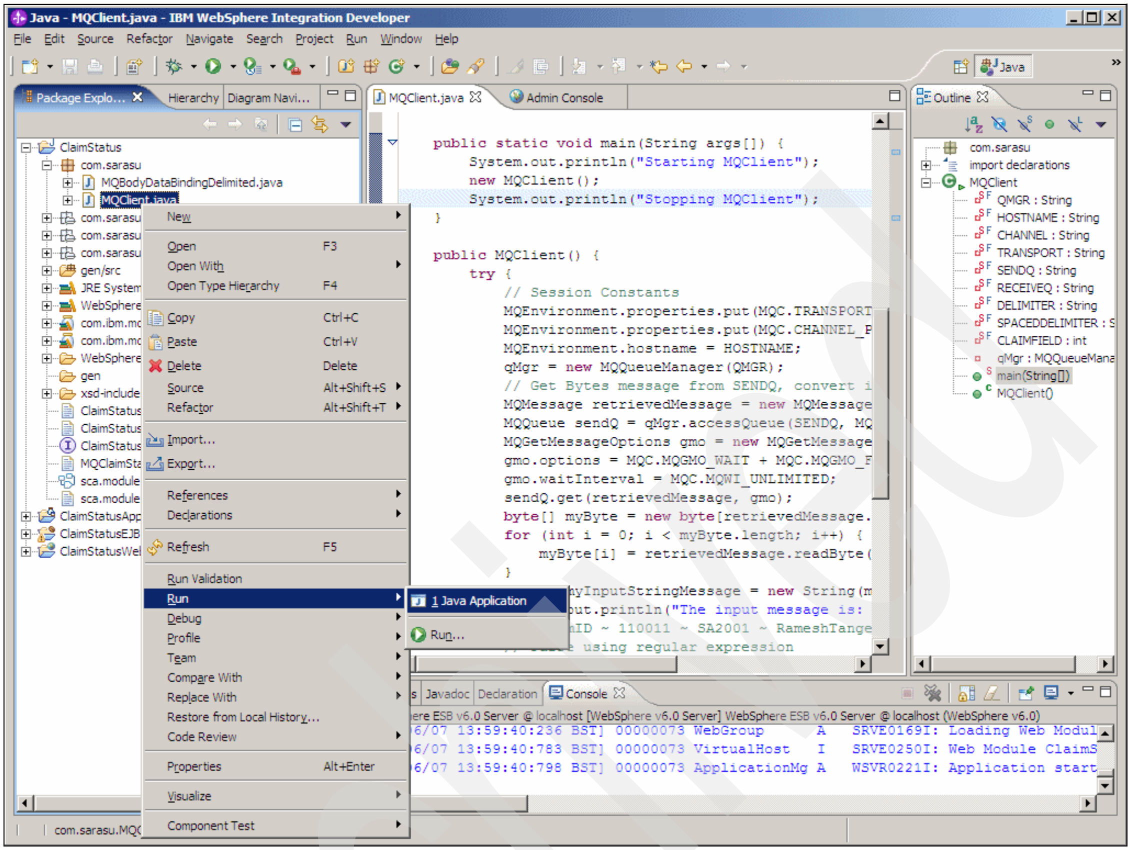


Figure 7-47 Run MQClient

7.9.3 Testing the application

We use the Web services Explorer to test the solution. First you need to enable the Web Service Developer capabilities in the workbench. Then we create a SOAP/http request and run it through the mediation, which drives the MQClient program that is waiting on a message to arrive in SENDQ. The MQClient program returns the response to the RECEIVEQ, which is processed through the reply mediation. Then the result is returned to the reply window in the Web services explorer.

1. Select **Window** → **Preferences** → **Capabilities** and place a check mark in Web Service Developer, then click **OK** (Figure 7-48).

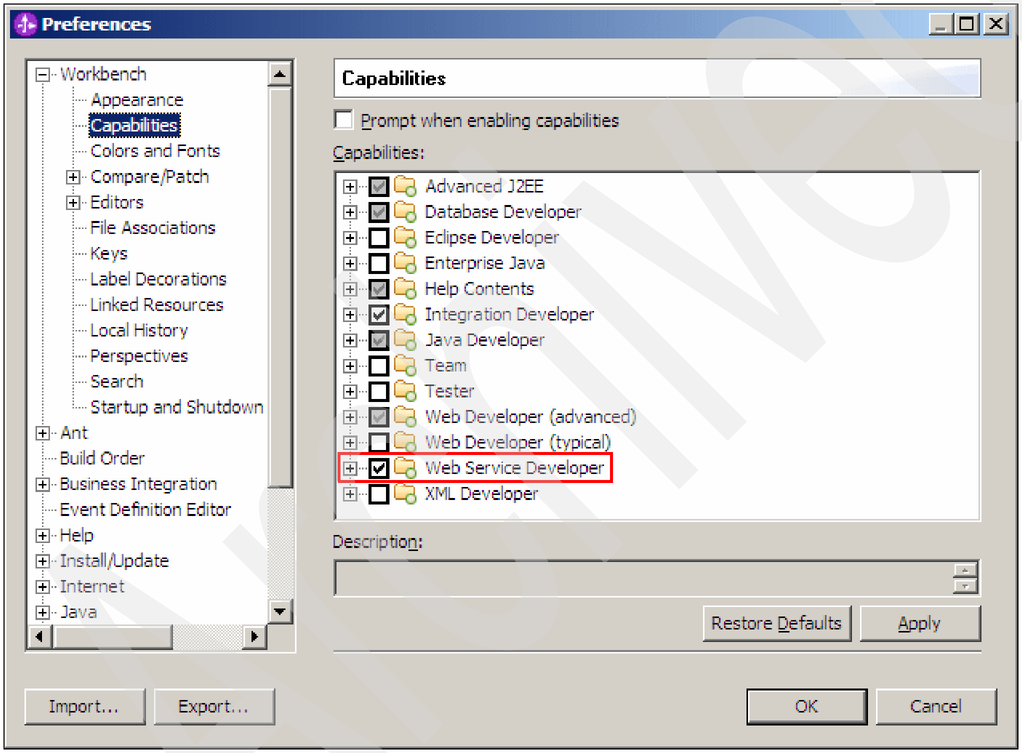


Figure 7-48 Web Services Developer capabilities

2. Launch the Web services Explorer using either the icon in the title bar , or in (for example) the Java Perspective open the **ClaimStatusWeb** project. Right-click

ClaimStatusServiceExport1_ClaimStatusServiceHttp_Service.wsdl and select **Test with Web services explorer**. Navigate to the getClaimStatus method and insert input values in the fields, then select **Go** (Figure 7-49).

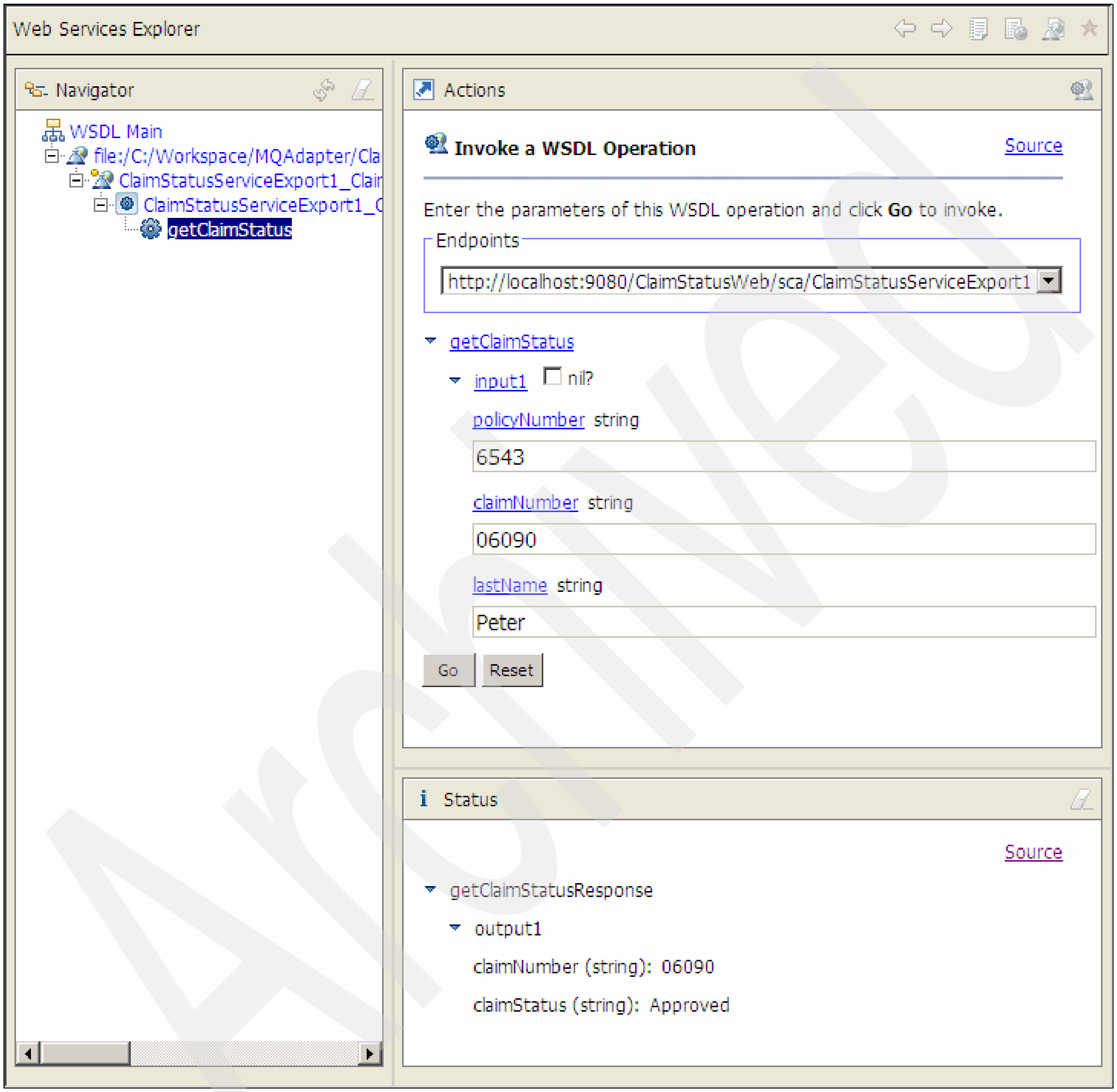


Figure 7-49 Test the claimStatus mediation

The console output from the MQClient application is shown in Figure 7-50.

```
Starting MQClient
The input message is: claimID ~ 6543 ~ 06090 ~ Peter
The output message is: The message is claimResult ~ 06090 ~ Approved
Stopping MQClient
```

Figure 7-50 Console output from MQClient

The output from the mediations is shown in Figure 7-51.

```
Building MQ Text from a business object
claimID ~ 6543 ~ 06090 ~ Peter
Building a Business Object from MQ Text
claimNumber is : 06090 claimStatus is : Approved
```

Figure 7-51 Console output from MQBodyDataBindingDelimited

7.9.4 Conclusion

In this chapter we implemented an example of our first integration pattern — connecting to an earlier application using WebSphere MQ. We used the Java regular expression parser to convert the WebSphere MQ message to and from a Service Data Object that is passed through an ESB mediation. There were no mediation primitives in the mediation flow. We just focus on how to get WebSphere MQ messages in and out of WebSphere Enterprise Service Bus. Once that is accomplished, then the legacy application is available as a service to any other programs connected to the ESB.

There are three main directions in which you may want to elaborate on this pattern:

- Alternative parsing schemes. One could use an XSLT mediation or a custom mediation to leverage existing parsing tools. To use an XSLT mediation you could pass the WebSphere MQ message in as essentially a single field, and then use an XSLT or custom mediation as a second parsing pass.
- Using data in the WebSphere MQ headers that are passed to and from the custom data binding. The headers are held in arg1 as a list.
- Integrating with more sophisticated legacy applications for which you may be able to mechanically generate the parser from some kind of Enterprise Metadata Discovery tool.

  The next two chapters follow on the second and third points to develop two scalable solutions to integrate with CICS using the CICS/MQ bridge.

8

Custom CICS integration using WebSphere MQ

In this chapter we explain how to implement protocol switching and data transformation between a modern Web services application and a legacy application using the WebSphere Enterprise Service Bus.

As described in "Environment - ITSO-INS claimant adjudication systems" on page 132, our legacy application is a CICS system running on a mainframe z/OS platform where COBOL transactions are deployed. Those transactions implement the precious business logic written in COBOL that we want to re-use without changing anything.

The constraints that we face throughout this chapter are:

- The only way to access the CICS system is WebSphere MQ.

  The CICS mainframe is accessed via a MQ/CICS bridge that acts as an MQ destination on the MQ network. On receipt of a well-formatted message, the bridge is able to invoke a CICS transaction passing in the user data contained in the message payload. The transaction results follow the reverse path and a message containing the result is built by the MQ Bridge and sent back to the reply queue, specified on the incoming message.
- There is not a WebSphere Adapter that supports COBOL CICS transactions over the WebSphere MQ protocol.

However, the WebSphere CICS JCA adapter can be used to generate the business object EIS data binding classes that are needed to transform the CICS business objects in a format that the CICS transaction will understand. We plan to marry this tooling with a solution built to use WebSphere ESB and WebSphere MQ.

In this chapter, we learn:

- How to set up and use the Enterprise Metadata Discovery to generate EIS data binding helper classes and use them in a custom MQ data binding class
- How to configure the WebSphere Enterprise Service Bus WebSphere MQ transport binding to build and send well-formed MQ-CICS bridge messages through the bridge to a CICS system and start a CICS transaction

Assumption

We assume that:

- IBM WebSphere Integration Developer Version 6.0.2 is installed, with an integrated test environment: WebSphere ESB server Version 6.0.2.1 runtime.
- We have access to a WebSphere MQ Queue Manager where the queues to send and receive messages from and to CICS are well defined.

In this example, we use WebSphere MQ Version 6.0.2.1 as our WebSphere MQ transport provider.

8.1 WebSphere J2EE Connector Tools

To obtain the CICS transaction metadata, we plan to use the WebSphere CICS JCA adapter.

This adapter is packaged with the J2EE Connector Tools as an optional feature of IBM WebSphere Integration Developer and is installed using the IBM Rational Product Updater.

For complete information about installing the J2EE Connector Tools feature, refer to:

http://www3.software.ibm.com/ibmdl/pub/software/rationalsdp/rad/60/install_instruction/J2C6013/install.html

Figure 8-1 shows the Rational Product Updater with the CICS JCA Connector tools installed.

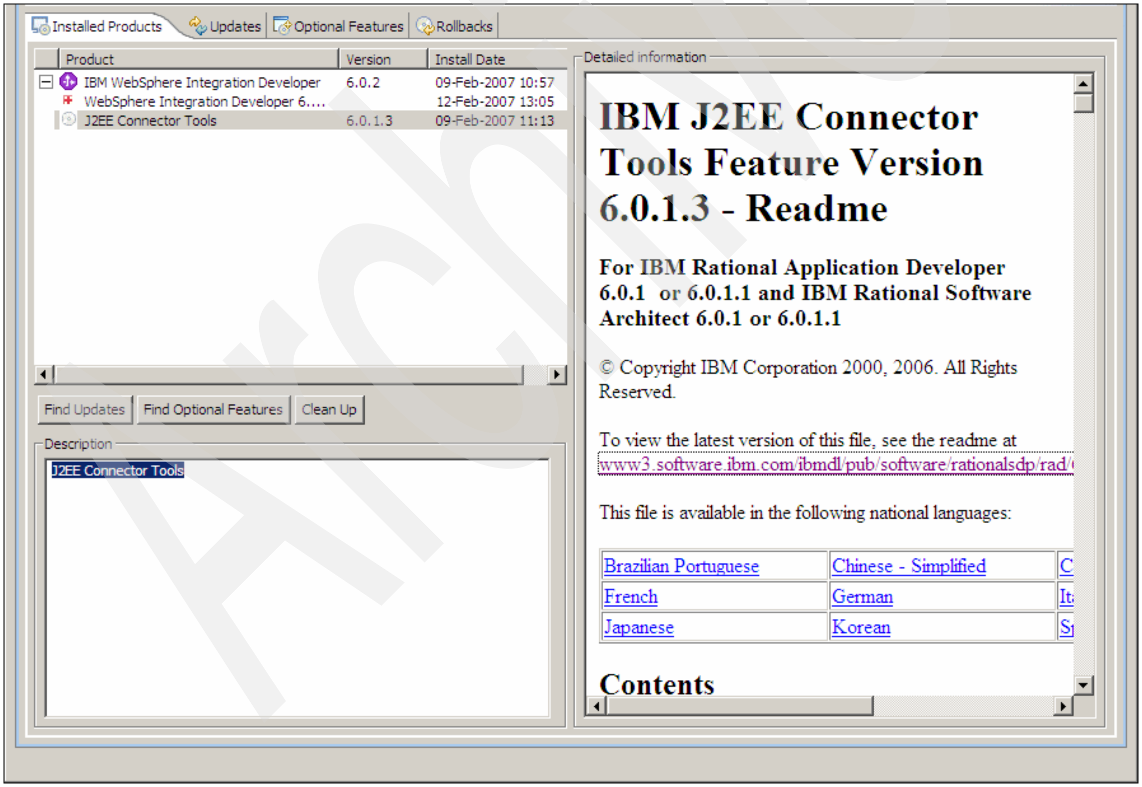


Figure 8-1 Rational Software Development Platform Product Updater with JCA Connector tools

8.2 Solution design

In Figure 8-46 on page 261 we show a detailed view of the interaction between the WebSphere Enterprise Service Bus runtime, the MQ-CICS bridge, the CICS system, and the CICS program that we want to access.

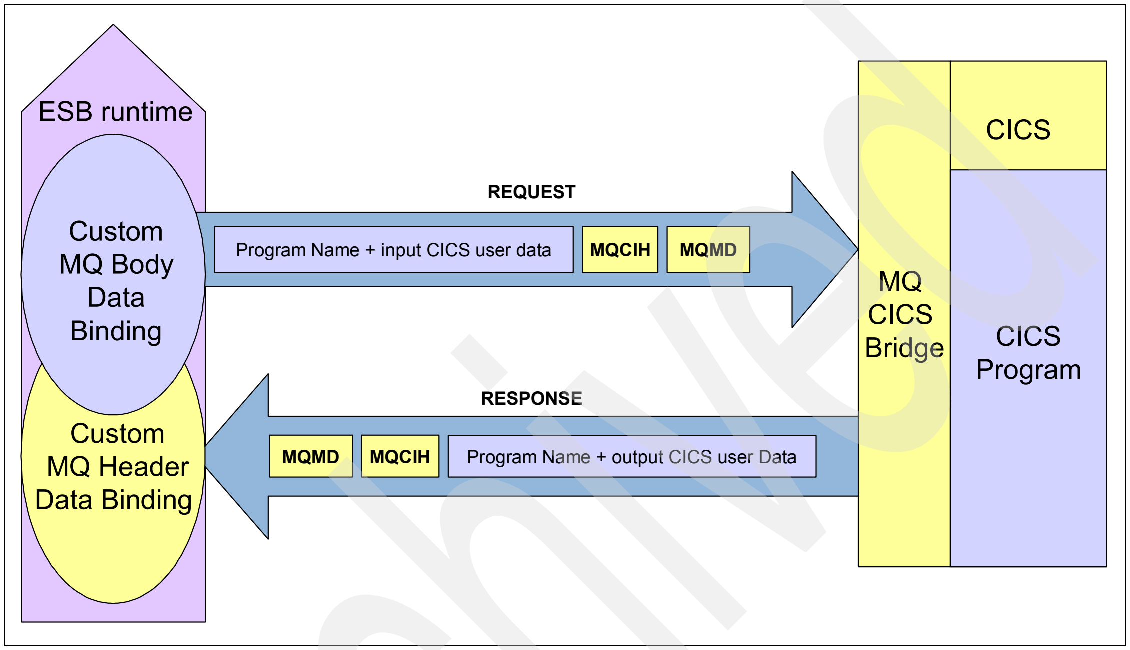


Figure 8-2 Detail of the interaction with CICS and the structure of the MQ message

The message that is being sent to CICS over WebSphere MQ has to have a specific format stored in the MQMD structure and a dedicated header, called the MQ CICS header or MQCIH structure. Finally, the body of the message contains the CICS COBOL user data. WebSphere Enterprise Service Bus provides support for WebSphere MQ headers so that the header data is managed separately from the request and reply bodies[1]. We need to create our own implementation to support the MQCIH header.

In our solution we:

- Implement the mediation module that does the protocol transformation to go from a Web service request/reply type of access to an MQ transport protocol.

[1] WebSphere Enterprise Service Bus Version 6.0.2 has limited support of WebSphere MQ headers, and MQCIH headers are not supported as of the writing of this book. The WebSphere MQ binding of WebSphere Enterprise Service Bus Version 6.0.2 currently provides support only for the MQRFH and MQRFH2 headers.

- Implement the custom MQ body data binding class needed to transform the business objects into a stream of bytes that the CICS application can understand.
- Build the custom MQ header data binding class that adds or removes the CICS header to or from the MQ message, in order to go across the MQ/CICS bridge.

8.3 Mediation module design

Before we start to build the mediation, we need to create the business objects and decide on the interfaces that we are going to use. Then we can wire them together and build the mediation module, as shown in Figure 8-3.

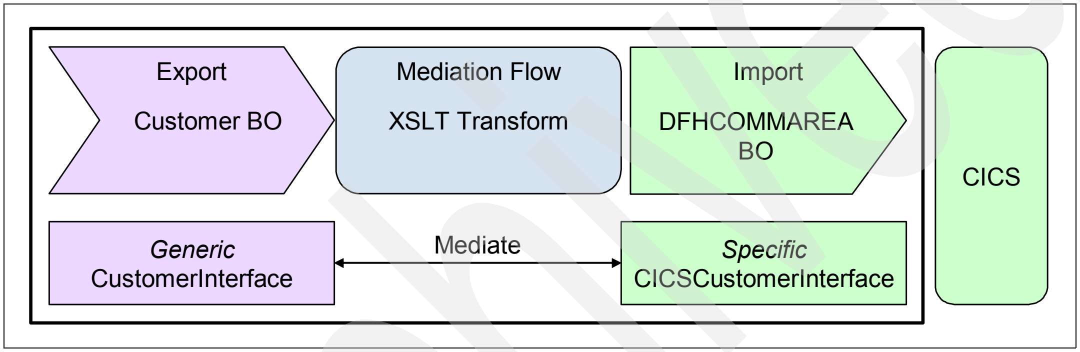


Figure 8-3 Business objects and interfaces of the mediation module

We start by looking at the CICS transaction to build the CICS specific business object and interface that we need, and then we define the more generic business object and interfaces that the mediation will expose.

The CICS transaction we use accesses a customer database. Given a request identification and a customer number, it returns a customer record.

In the Example 8-1 we show the COBOL definition of the user data of our CICS transaction. It is called the DFHCOMMAREA.

Example 8-1 Source file of the CICS transaction

```
*****************************************************************
       01 DFHCOMMAREA.
           03 CA-REQUEST-ID            PIC X(6).
           03 CA-RETURN-CODE           PIC 9(2).
           03 CA-CUSTOMER-NUM          PIC 9(10).
           03 CA-FIRST-NAME            PIC X(10).
           03 CA-LAST-NAME             PIC X(20).
           03 CA-DOB                   PIC X(10).
           03 CA-HOUSE-NAME            PIC X(20).
           03 CA-HOUSE-NUM             PIC X(4).
           03 CA-POSTCODE              PIC X(8).
```

> **Attention:** The COBOL definition shown in Example 8-1 is only an example of a transaction that we use to illustrate our scenario. It has to be replaced by the actual transactions that you run in your environment.
>
> But if you want to implement our solution, you can cut this COBOL structure and paste it into a file with the extension *cpy* (like LGEICUS01.cpy) and use it in the next section to generate the business object.
>
> *Make sure* that you include the first eight blank characters before the level 01 block in order for the COBOL importer to work properly.

On receipt of the SOAP request, the mediation module builds a DFHCOMMAREA data structure and sends it to the CICS program using MQ.

We call that first business object DFHCOMMAREA. As this business object is very specific to the CICS interaction, we do not want to expose it as-is on the mediation module interface, so we define a generic customer business object.

The customer business object is derived from the DFHCOMMAREA business object, but it does not expose some of the very specific CICS fields, like the REQUEST-ID and the CA-RETURN-CODE that we do not need in our business logic. We can also use the customer business object to modify the way other fields are presented to the mediation module interface. As we showed in Figure 8-3 on page 219, there are two business objects and two interfaces for the mediation module.

On the left-hand side, we see the generic interface and generic business object of the solution, and on the other side the CICS-specific interaction and data types. The mediation flow is in the middle.

8.3.1 Application-specific DFHCOMMAREA business object

We start with the CICS-specific side of the mediation module, as shown in Figure 8-4.

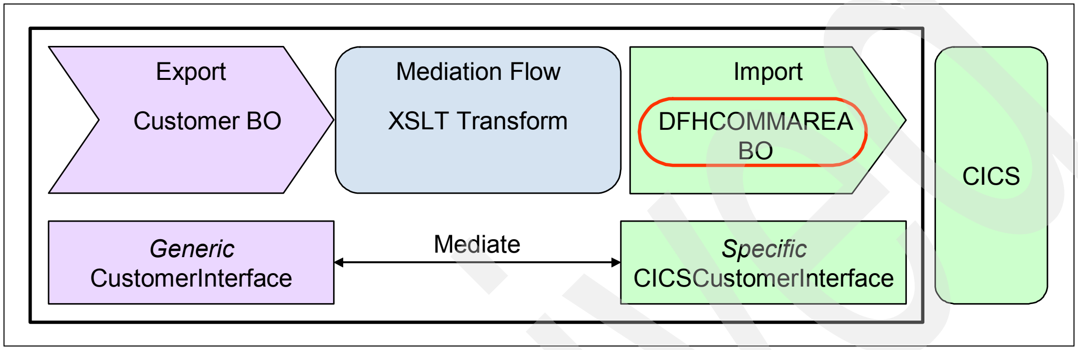


Figure 8-4 Business objects and interfaces of the mediation module

Start WebSphere Integration Developer in a new workspace. Switch to the Business Integration Perspective, where the mediation implementation is performed.

The first thing we do is create a Library Module where we store the business objects and interfaces of our mediation module.

1. Create a Library Module named CICSLibrary.

> **Note:** For convenience and re-usability, it is good to put all business objects and interfaces in a Library module so that they can be re-used by several mediation modules later on.

We can now create the CICS business object that is going to hold the user data information of the CICS transaction.

2. Open the CICSLibrary module. Right-click **Data Types** and select **New** → **Enterprise Data Discovery,** as shown in Figure 8-5.

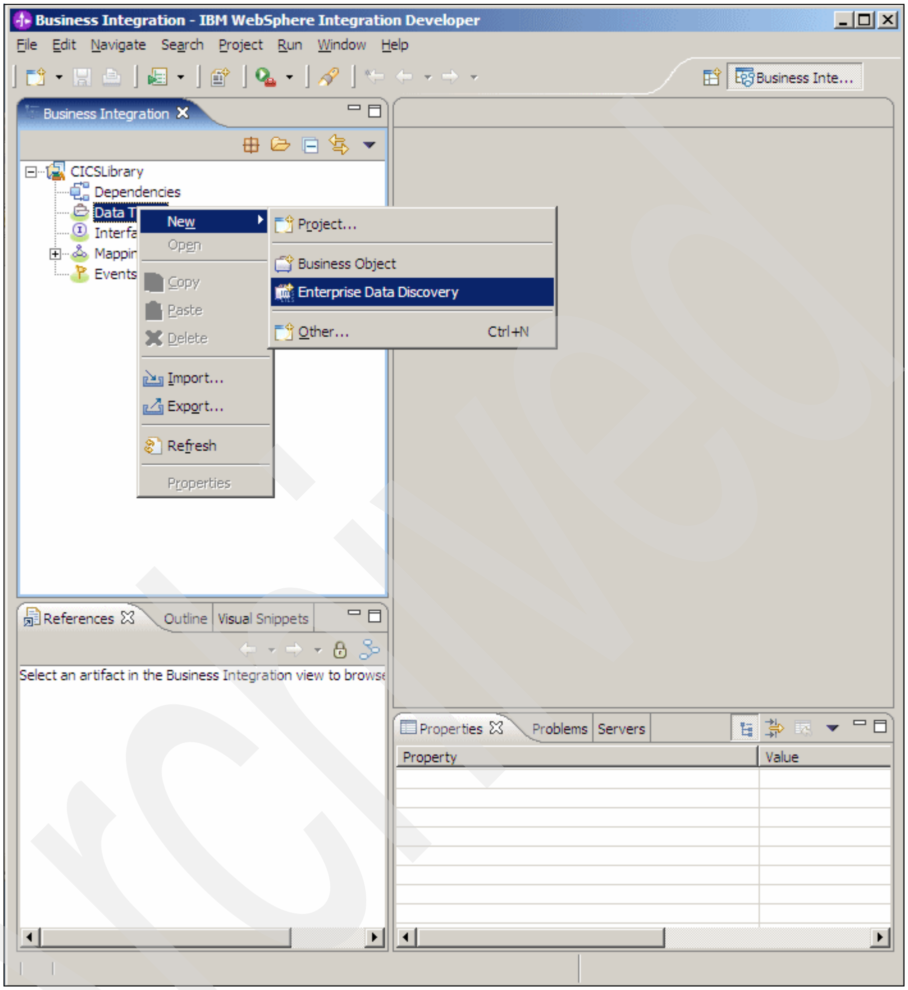


Figure 8-5 Enterprise Data Discovery launcher

3. Click **Browse** to navigate to your CICS COBOL COMMAREA file (Figure 8-6).

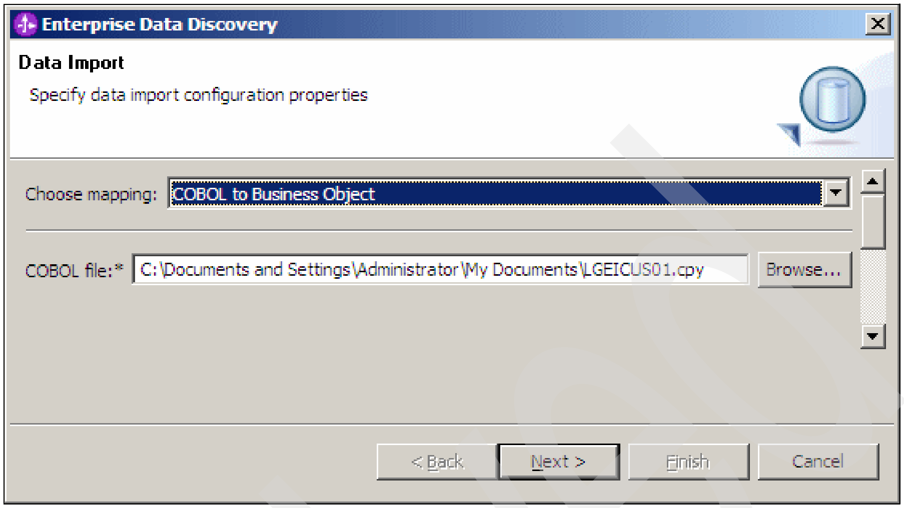


Figure 8-6 Enterprise Data Discovery file selection panel

> **Note:** In Example 8-3 we provide the COBOL structure that we use in the rest of the chapter. You may want to use your own copy book here.

4. Make sure that **COBOL to Business Object** mapping is selected in the Choose mapping field and click **Next**.
5. On the Importer page, select the platform and code page of your target host. As we use a z/OS system, we change the platform to z/OS. We leave all of the other fields unchanged.

6. Click the **Apply** button to parse the COBOL file and get access to the data structures that are defined in the COBOL file (only one data structure in our case) (Figure 8-7). Select **DFHCOMMAREA** → **Next**.

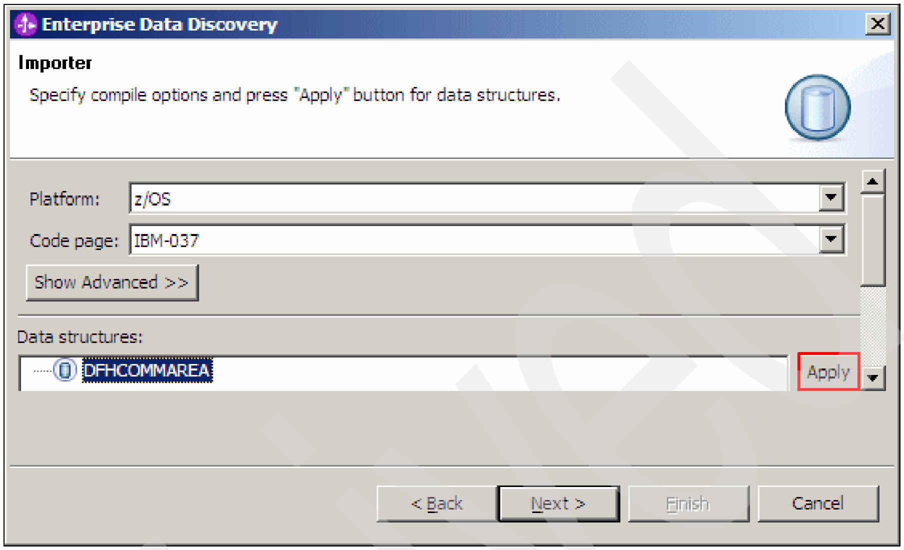


Figure 8-7 Enterprise Data Discovery importer panel

7. In the Saving Properties panel click **Finish**.

> **Note:** If you want to give a name to your business object other than the name proposed in the Name field (which is by default the name of the Cobol COMMAREA structure), you can change it here.

The DFHCOMMAREA business object is now created in the Data Types folder of the library (Figure 8-8).

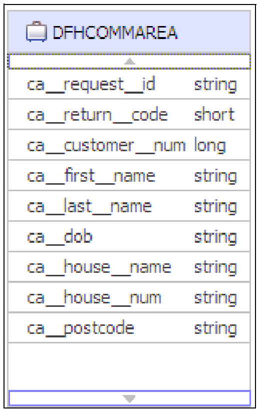


Figure 8-8 DFHCOMMAREA displayed in the business object editor

8.3.2 Generic customer business object

We continue with the generic business object, Customer, as highlighted in Figure 8-4 on page 221.

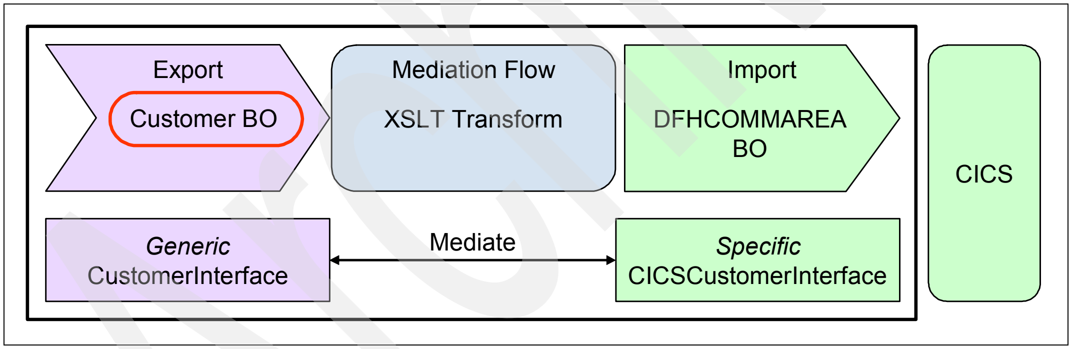


Figure 8-9 Business objects and interfaces of the mediation module

The generic business object is derived from the specific CICS business objects generated in the previous steps. To create it:

1. Open the CICSLibrary folder, right-click **Data types**, and select **New** → **Business Object**. Name it `Customer` and click **Next** (Figure 8-10).

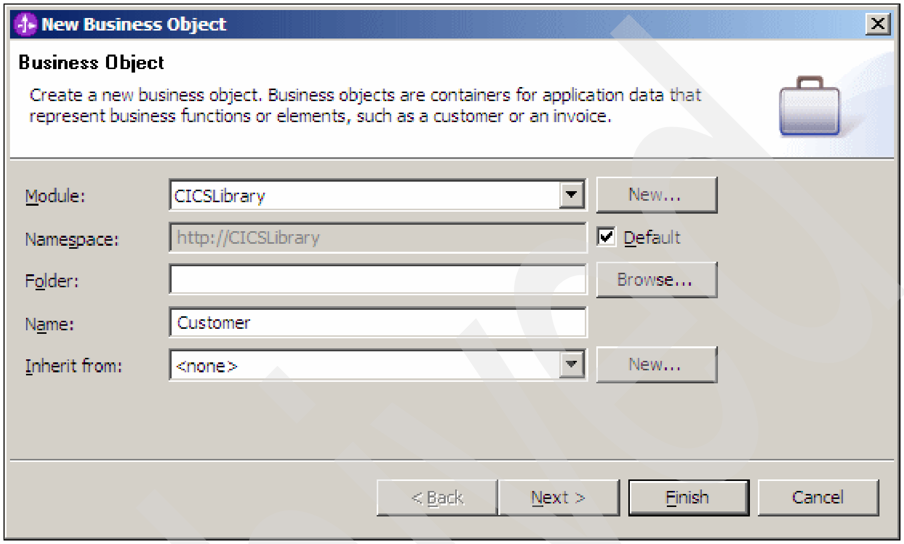


Figure 8-10 Create new customer GBO

2. In the Derived Business Object Panel select the **DFHCOMMAREA** business object, and in the Attributes to include panel, deselect the two first fields, as shown in the Figure 8-11. Click **Finish**.

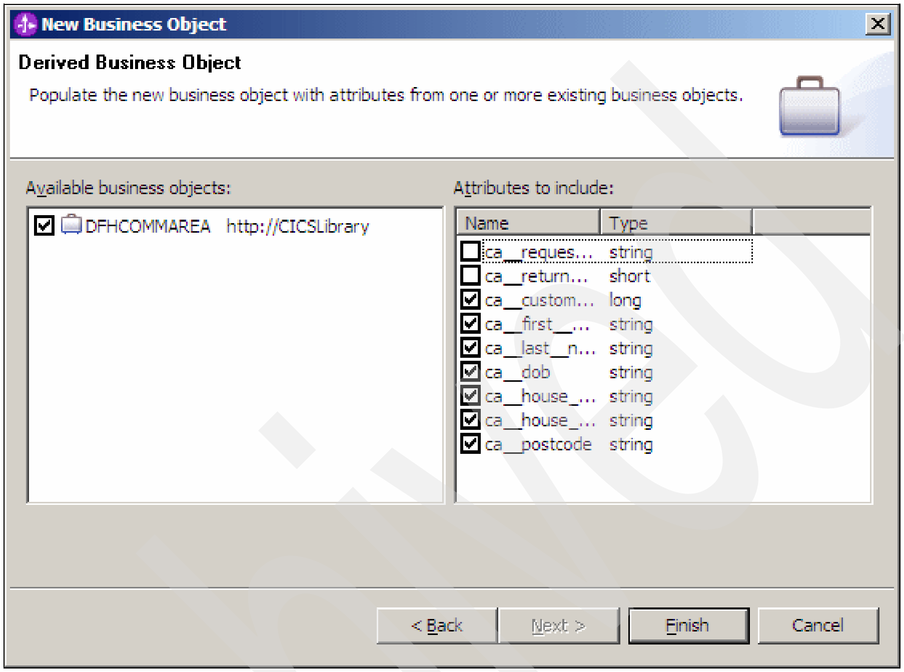


Figure 8-11 Generic business object derived from the specific business object

If we look at the properties of every field of the business object, in the properties panel, we see that there is a Required parameter that defines whether the field should always be filled with values.

In our case, only the ca__customer__num will be always filled as an input parameter. The other fields are output fields and should not be set as Required here.

1. Select every field of the business object in turn but the ca__customer__num and un-check the Required parameter for each.
2. When complete, save the business object and close the business object editor.

The customer business object now appears in the list of data types in the CICSLibrary. We built this generic business object as we designed it, removing the CICS-specific fields that our business application is not willing to see. We use the mediation module capability to map the generic business object into a specific business object.

8.4 CICS interfaces

We are now ready to create the interfaces that we need to build our scenario:

- One specific interface that uses the DFHCOMMAREA business object, used to access the CICS application CICSCustomerInterface.
- The other one, generic, that uses the customer business object, used to expose our module to the outside world: CustomerInterface.

See in Figure 8-12 for a complete view of the solution.

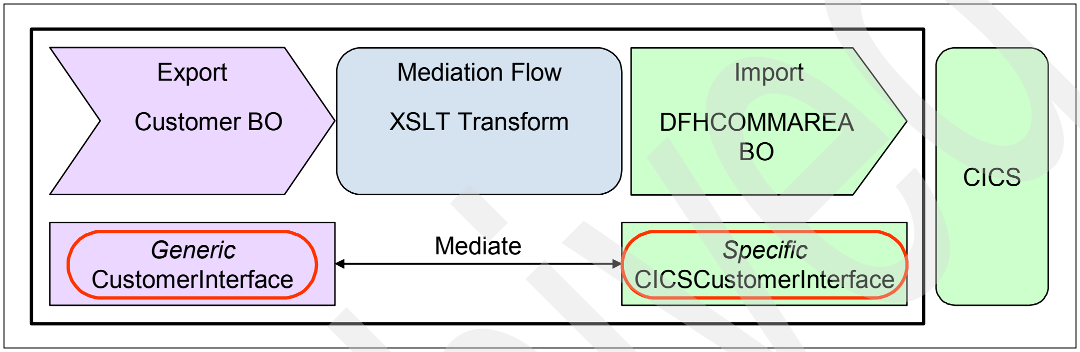


Figure 8-12 Business objects and interfaces of the mediation module

Still in the Business Integration perspective, right-click the **CICSLibrary**, click **New** → **Interface**, and name it `CICSCustomerInterface` (Figure 8-13). Click **Finish**.

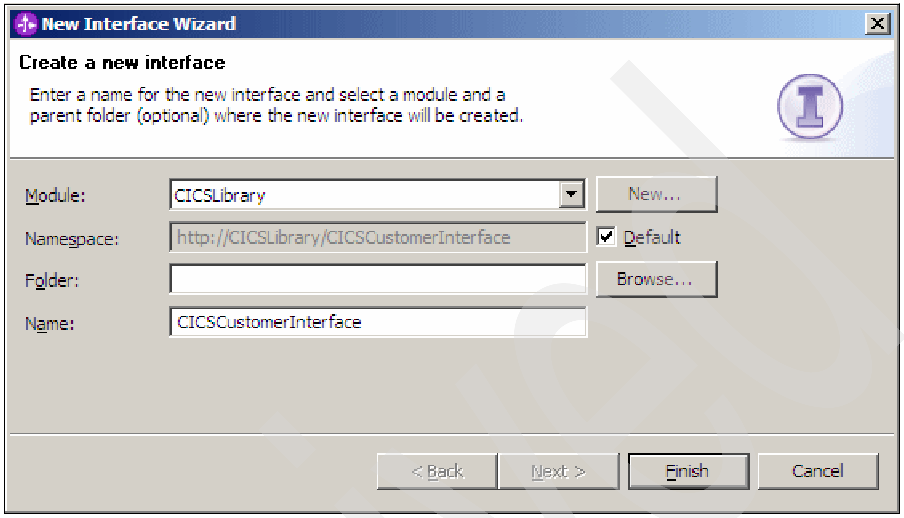


Figure 8-13 New CICSCustomerInterface

The interface editor panel opens to let us create the operation of the interface. Using the information in Table 8-1, create the CICSCustomerInterface accessCICS operation by clicking the Add Request Response operation icon .

Table 8-1 Interface parameters

Interface name	**CICSCustomerInterface**	**CustomerInterface**
Request/response name	accessCICS	access
Input parameter name	cicsInput	customerIn
Input parameter type	`DFHCOMMAREA`	Customer
Output parameter name	cicsOutput	customerOut
Output parameter type	`DFHCOMMAREA`	Customer

Figure 8-14 shows the interface just created, opened in the interface editor.

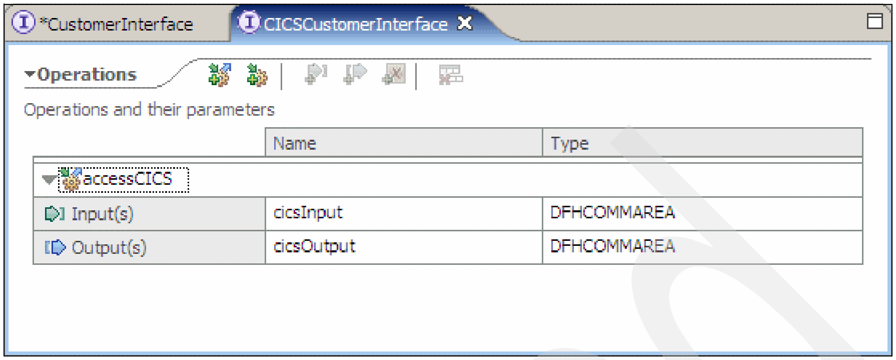


Figure 8-14 CICSCustomerInterace interface definition

Save the interface and close the interface editor.

Following the same steps, we define our generic interface using the information in Table 8-1 on page 229 (Figure 8-15).

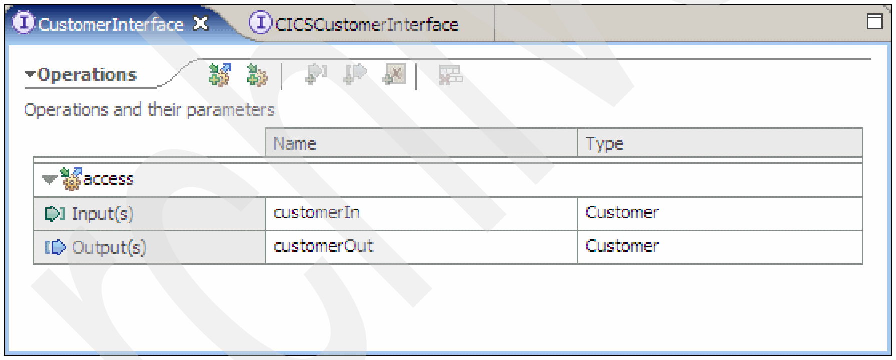


Figure 8-15 CustomerInterface interface definition

Figure 8-16 shows both interfaces and business objects in the CICSLibrary.

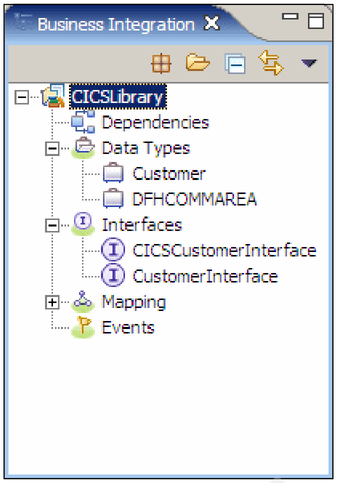


Figure 8-16 CICSLibrary

8.5 Mediation module

As defined earlier in 8.2, “Solution design” on page 218, and shown again here in Figure 8-17, we have to implement a mediation module that transforms a customer into a DFHCOMMAREA, and transforms Web service access into MQ interaction.

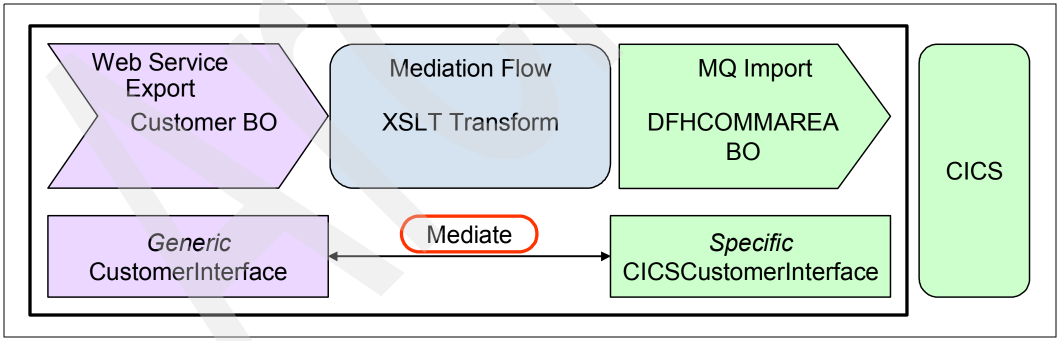


Figure 8-17 Business objects and interfaces of the mediation module

In this section we first implement the mediation module that does the protocol transformation to go from a Web service request/reply type of access to an MQ transport protocol.

We also implement the custom MQ body data binding class needed to transform the business objects into a stream of bytes that the CICS application can understand.

And, finally, we build the CICS MQ header class that adds or removes the CICS header to or from the MQ message, in order to go across the MQ/CICS bridge.

For a more complete explanation of data binding concepts and implementation in the enterprise service bus refer to 3.5, “WebSphere ESB Transport Binding Types” on page 82.

8.5.1 Mediation implementation

In the Business Integration perspective, we use the assembly editor to add components to our module and wire them together to implement the mediation module.

Create the CICSMQAccess mediation module

To do this:

1. In WebSphere Integration Developer, switch to the Business Integration Perspective.
2. Right-click in the **Business Integration explorer** view and select **New** → **Mediation Module**.
3. On the Mediation Module panel enter `CICSMQAccess` as the module name. Leave all the other fields as default and click **Next**.
4. On the Select Required Libraries panel select **CICSLibrary** → **Finish**.

   This creates the mediation module called CICSMQAccess that has a dependency on CICSLibrary.

Create the CICSMQAccess Assembly Diagram

Double-click the Assembly Diagram in the CICSMQAccess project to open its assembly diagram (Figure 8-18).

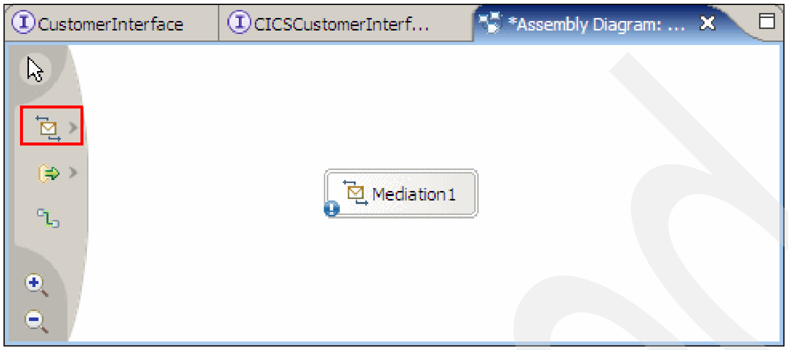


Figure 8-18 Create a mediation flow component in the Assembly diagram

You probably have a mediation flow component called Mediation1 in the diagram already. If not:

1. Add a mediation flow component to the diagram by dragging the highlighted mediation flow icon from the palette onto the canvas.
2. Change the name of the mediation flow component to `CICSMediation`.
3. Right-click the flow component and select **Add** → **Interface**. Select **CustomerInterface** → **OK** (or drop the interface from the Business Integration explorer onto the flow, or select the hover icon above the flow component).
4. Select the Add Reference hover icon. Select **CICSCustomerInterface** → **OK**.
5. Drag and drop an import component from the palette onto the assembly diagram and rename it to `MQ`.

6. Wire the CICSMediation to the MQ import using the yellow (hover) wire tool. If you select the reference to hover the wire tool respond **OK** to the dialog to create a matching interface on the MQ import (Figure 8-19).

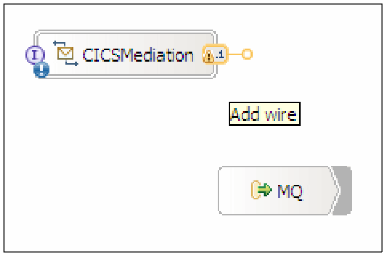


Figure 8-19 Write import to existing reference

7. Drag and drop an **Export** component from the palette onto the assembly diagram and rename it to `WS`.
8. Right-click the **WS** export component, click **Add Interface**, and select **CustomerInterface** → **OK**.
9. Right-click the CICSMediation flow component **Wire to existing**. This connects the mediation flow component to the import. (We have already connected the export to demonstrate an alternative approach to wiring components.) Click **Save** (Figure 8-20).

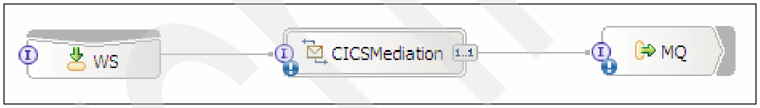


Figure 8-20 Assembly diagram of the mediation module

> **Note:** The exclamation marks on the CICSMediation component and on the MQ Import node remind us that we need to provide an implementation for those two components. This is what we do next. We also need to generate a binding for the export to a transport (SOAP/Http).

Mediation flow implementation

In the implementation of the mediation flow itself, we want to map the generic interface CustomerInterface into the specific CICSCustomerInterface.

Figure 8-21 shows all the business objects and interfaces involved in the solution and where our implementation fits.

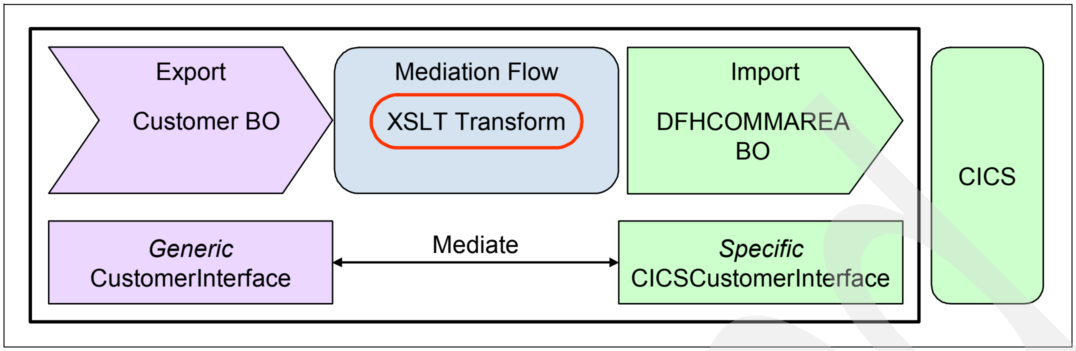


Figure 8-21 Business objects and interfaces of the mediation module

To build the mediation flow implementation, we need to open the assembly diagram of the mediation module in the business integration perspective.

1. If not already opened, double-click the Assembly Diagram in the CICSMQAccess project. The assembly diagram of the mediation module opens as shown in Figure 8-22.

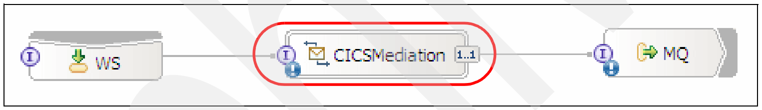


Figure 8-22 Assembly diagram of the mediation module

2. Double-click the **CICSMediation** component. Respond **Yes** to the question `Would you like to implement the component now?` and **OK** (default) to the dialog about where to create the mediation flow implementation.

The CICSMediation flow component is opened in the Mediation Flow Editor (Figure 8-23).

3. On the left-hand side we have the CustomerInterface, showing the access operation we defined in 8.4, “CICS interfaces” on page 228. If you hover the mouse over the right most part of the interface, a yellow circle appears. Drag and drop this circle to connect it to the accessCICS operation in the CICSCustomerIntefacePartner reference on the right-hand side. This creates a connection arrow between the two operations, as shown in Figure 8-23.

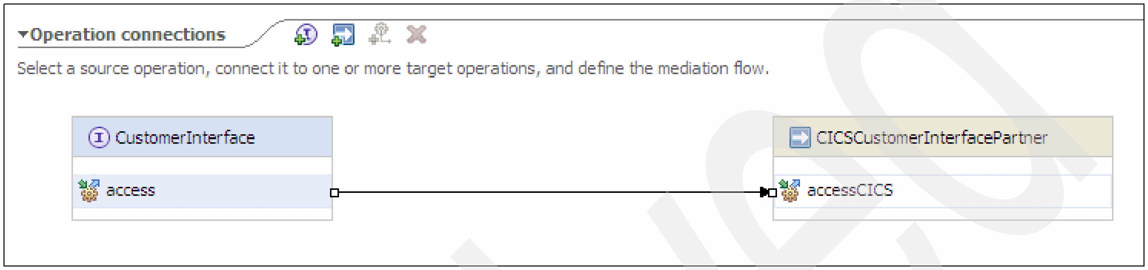


Figure 8-23 Operation connections editor panel

4. Select the connection arrow you just created (it should be black, as shown in Figure 8-23). In the implementation panel, in the palette area (on the left), select the **XSL Transformation mediation primitive**, and move your mouse onto the canvas area to drop it. An XSL Transformation primitive called XSLTransform1 is added to the diagram, as shown in Figure 8-24.

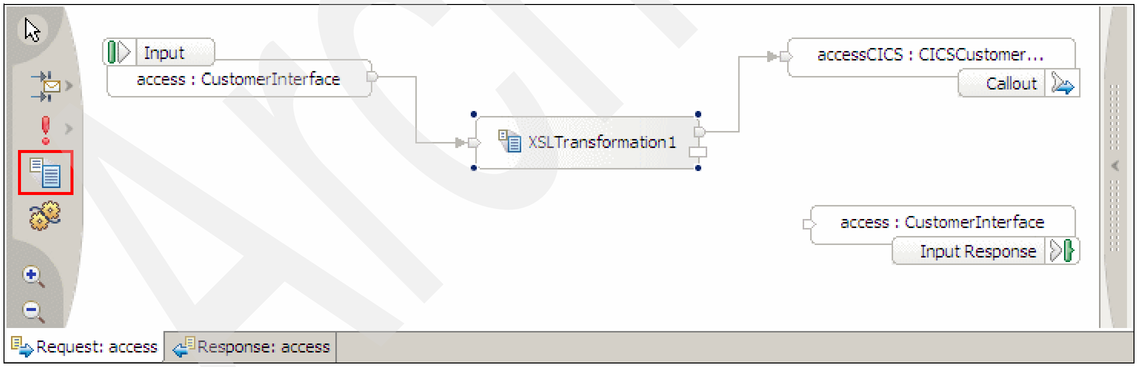


Figure 8-24 Request connection flow details

5. If you hover the mouse over the right most part of the input node, on the CustomerInterface interface, a yellow circle appears. Drag and drop that circle to connect it to the XSL transformation primitive just created.

6. Do the same with the output node. If you hover the mouse over the right most part of the XSLTransformation1 component, a yellow circle appears. Make sure that you select the top out terminal, not the fail terminal. Drag and drop that circle to connect it to the accessCICS:CICScustomerInterfacePartner callout node (Figure 8-24 on page 236).

We now implement the XSLT mapping components using the XSLT map editor.

Request mediation flow

To do this:

1. Click **XSLTransformation1**, select its **Properties** panel, select the **Details** tab, and click **New** to create a new XSLT file. On the New XSLT mapping panel accept all the defaults and click **Finish**.

   The XSLT map editor opens. On the left side, expand the body of the source smo until you see the fields of customerIn. On the right side, expand the body of the target smo until you see the fields of cicsInput.

2. Select and drag the first field of the customerIn structure **ca__customer__num** and drop it onto the fourth field of the cicsInput structure named bo:ca__customer__num.

3. With the same technique, map all of the fields of customerIn to cicsInput fields (Figure 8-25).

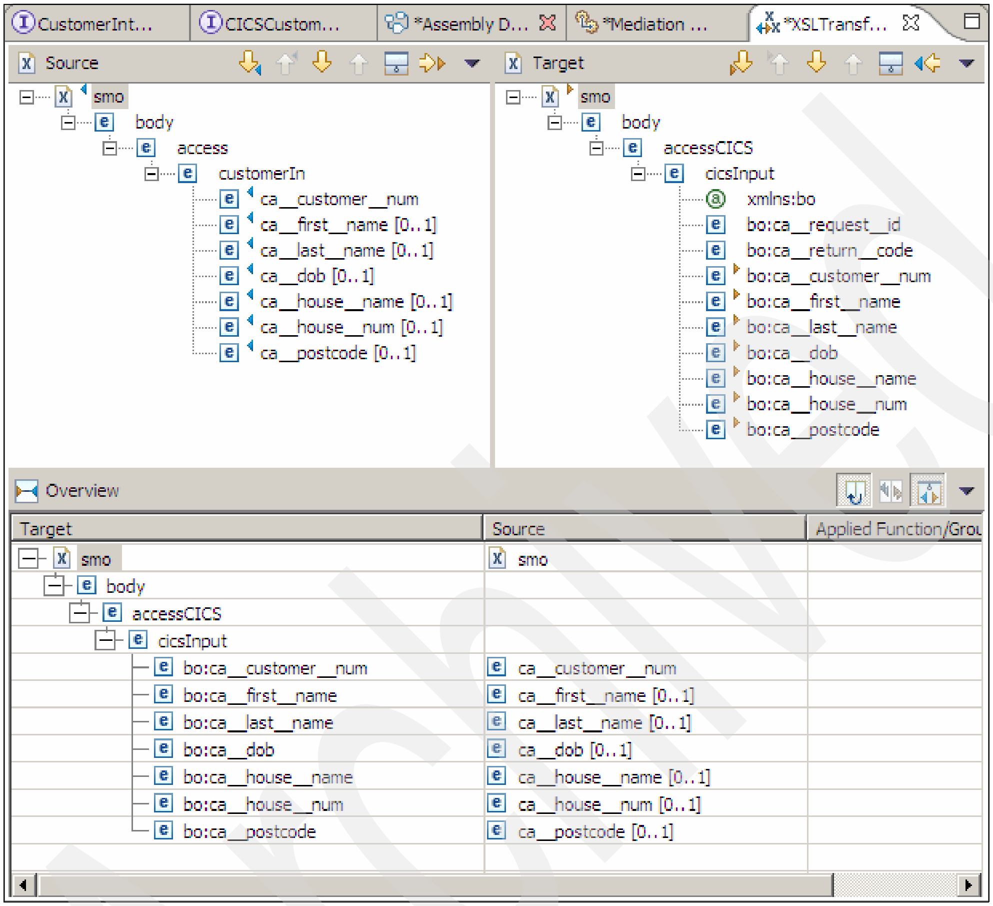


Figure 8-25 XSLT file opened in the map editor

On the target side, the cicsInput bo:ca__request__id and cicsInput bo:ca__return__code fields are not mapped, as they are not part of the interface. They are constant fields.

Referring to Figure 8-5 on page 222, we decided not to expose those two fields, as they are CICS specific, and they should not appear on the generic CusomerInterface, so we define XSLT functions to assign a constant value to these fields.

4. Right-click **bo:ca__request__id** and select **Define XSLT Function**.

5. In the Define a Function for this Mapping panel (Figure 8-26) select **String** → **Next**.

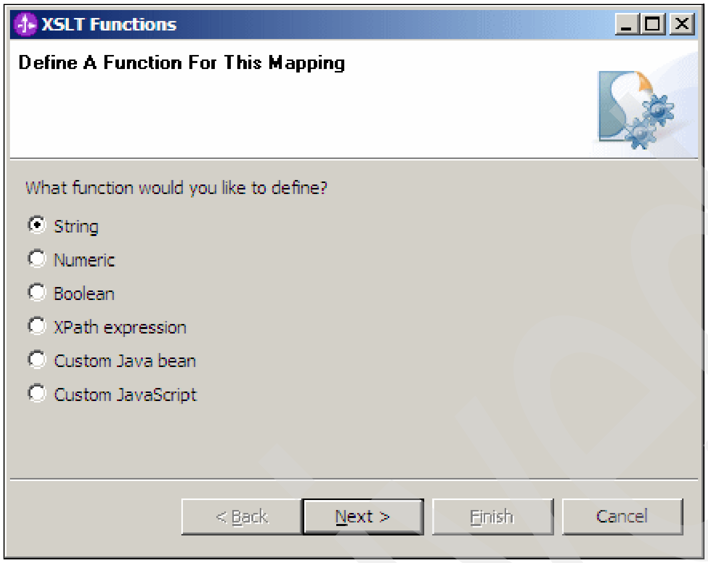


Figure 8-26 Select a String function for the XSLT mapping

6. In the Value Dialog enter the value `'ICUS01'` (make sure to include the single quotation marks; Figure 8-27) and click **OK**.

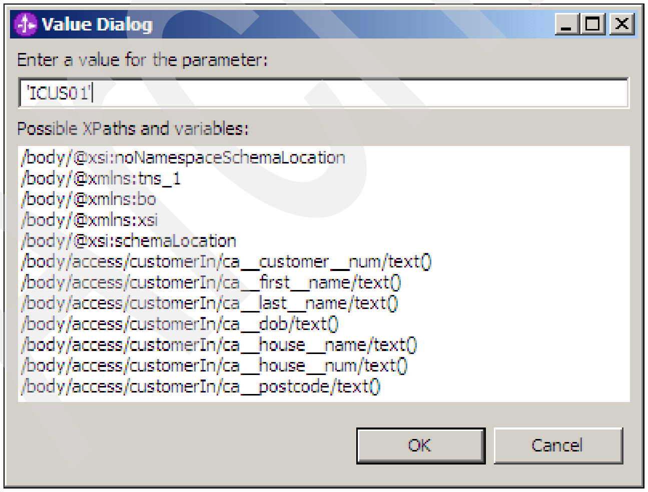


Figure 8-27 Enter constant String value

7. In the XSLT Functions panel select **String** as the function name and click **Add** (Figure 8-28).
8. Click **Finish** (Figure 8-28).

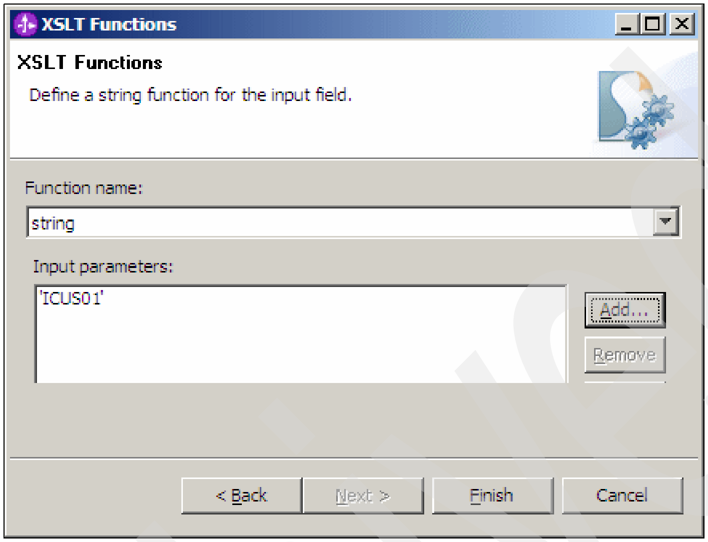


Figure 8-28 Defining XSLT constant function

9. Now define a numeric constant ‘00’ for the bo:ca__return__code using a similar procedure (Figure 8-29). Select a numeric rather than a string function in the Define A Function For This Mapping dialog.

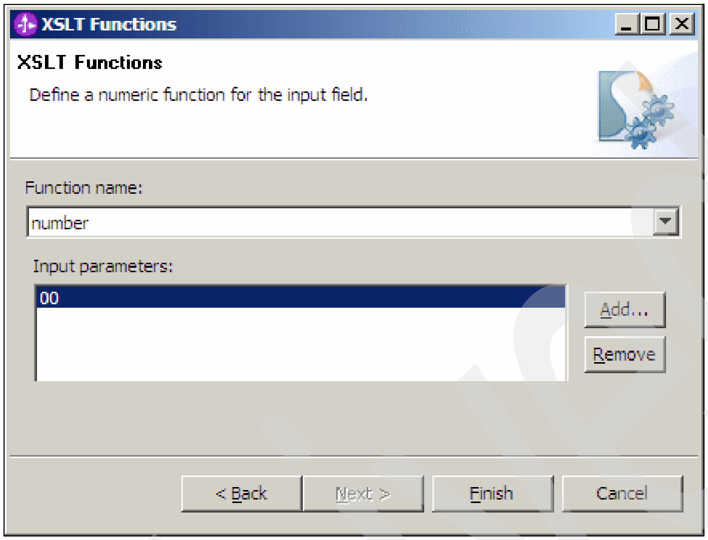


Figure 8-29 Adding numeric constant of 00

The XSLT file should like Figure 8-30. Note the symbols that have been added in front of bo:ca__request__id and bo:ca__return__code to show the presence of an XSLT function. All fields of the target smo are now filled with mapped values or constant values.

10.Save the file and close the XSLT map editor.

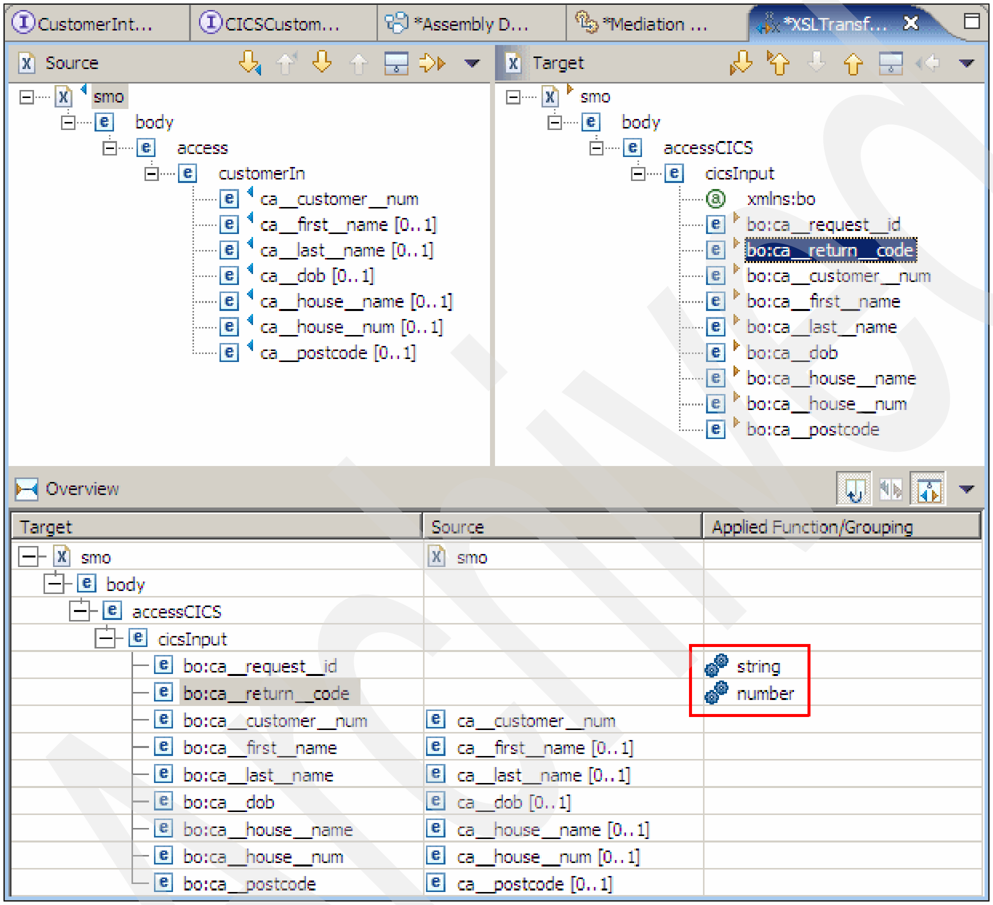


Figure 8-30 XSLT file opened in the map editor

Response mediation flow

We continue the implementation of the flow by providing the same type of implementation for the response flow.

1. As shown in Figure 8-31, click the **Response:access** tab to have access to the response flow diagram.

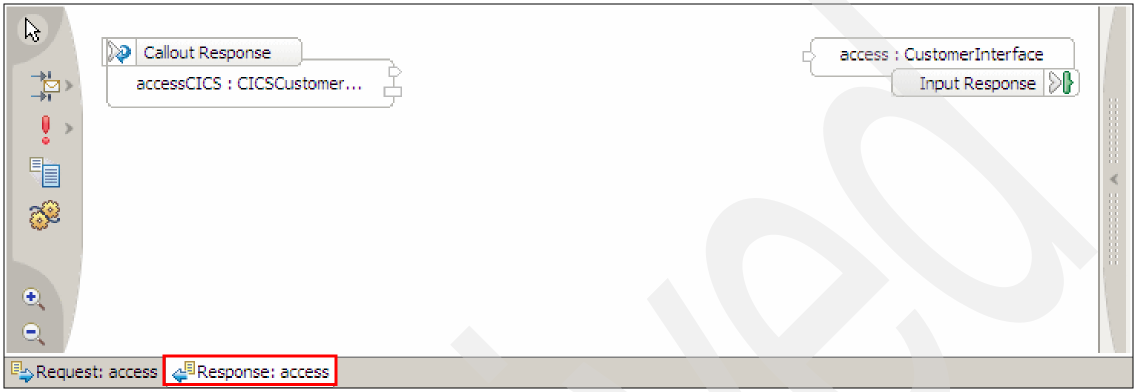


Figure 8-31 Response connection flow diagram

2. As we did for the request flow, add a XSLT mediation primitive to the canvas, connect the Callout response to it, and connect it to the Input Response node (Figure 8-32).

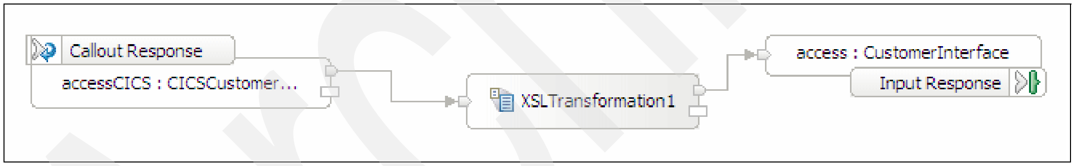


Figure 8-32 Response connection flow details

3. Click **XSLTransformation1** → **Properties** panel → **Details** tab → **New**. This creates a new XSLT file. On the New XSLT mapping panel accept all the defaults and click **Finish**. The XSLT map editor opens.
4. On the left side, expand the body of the source smo until you see the fields of cicsOutput. On the right side, expand the body of the target smo until you see the fields of customerOut.
5. Select and drag the field of the CicsOutput structure **ca__customer__num** and drop it onto the first field of the customerOut structure named ca__customer__num.

6. With the same technique, map all the fields of CicsOutput to customerOut fields (Figure 8-33).
7. Save and close all changes. This rebuilds the workspace.

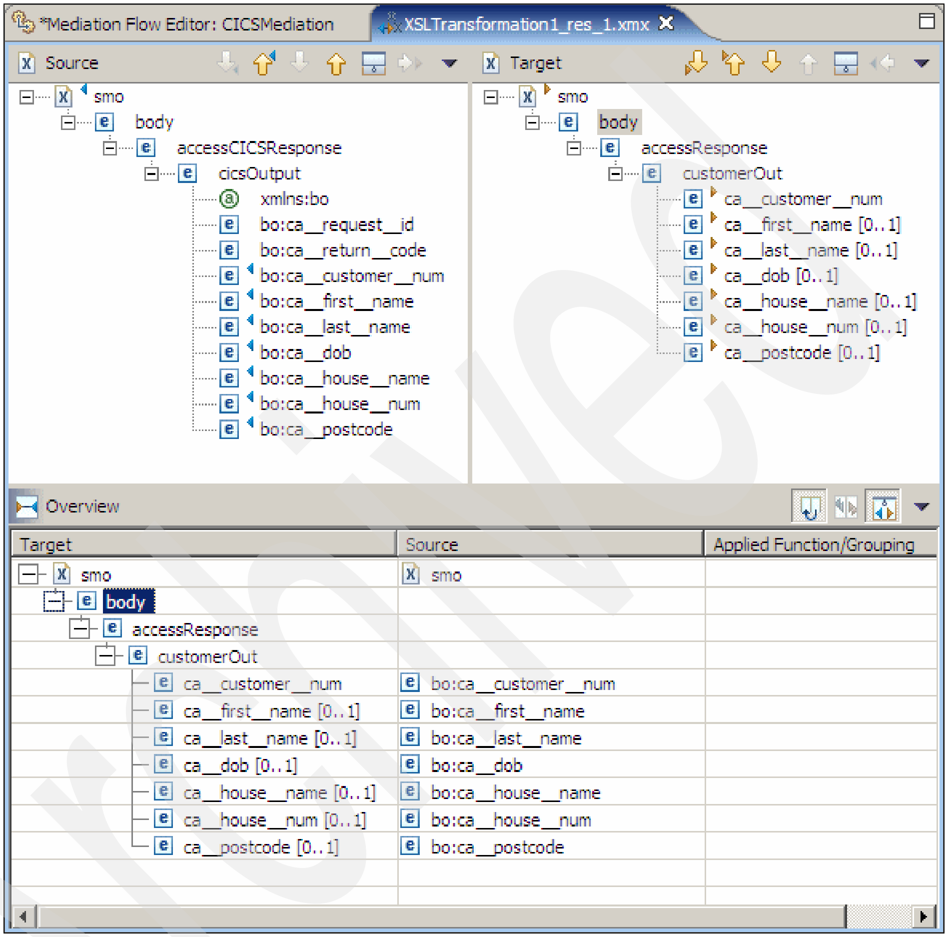


Figure 8-33 XSLT file opened in the map editor

The implementation of the mediation module is now complete. We have built the mediation flow and implemented the request and response flows using XSLT transformations.

We now have to prepare our mediation module before going to the next step and generate the binding implementations for both input and output nodes.

8.5.2 Configure the mediation module project

We need to create a package to store the custom Java code, and update the Java build path to be able to compile this code.

Create com.ibm.itso package

We put all the custom Java classes for this example in one package called com.ibm.itso attached to the CICSMQAccess module. This package is automatically deployed on the server with the application.

> **Note:** In a production situation we would put that code in a separate Java project and define it as a Java dependency on the mediation module, using the dependency editor.

To create the package and add classes in it, we use the Physical Resources view of the Business Integration perspective:

1. In the Business Integration Perspective, click **Window** → **Show View** → **Physical Resources**.
2. In the Physical Resources panel expand **CICSMQAccess**, right-click **CICSMQAccess**, and select **New** → **Other** → **Java** → **Package** → **Next**.
3. Name the package `com.ibm.itso` and click **Finish**.

Update the mediation module Java build path

The custom MQ Body Data Binding implementation class accesses the MQ Java API from the custom Java code, and as a consequence we need to add two MQ.jar files to the mediation module Java build path, as we did in 7.8.1, “Create MQClient Java program” on page 202.

1. Right-click **CICSMQAccess** and select **Properties** → **Java Build Path**. Select the **Libraries** tab → **Add External JARs**.
2. Navigate to the <WebSphere MQ installation>/java/lib directory. Select **com.ibm.mq.jar and com.ibm.mqjms.jar** → **Open** → **OK** (Figure 7-43 on page 206).

   > **Note:** In this example, we use the external WebSphere MQ Version 6 product as our MQ transport provider. We do not use the WebSphere MQ Version 5.3 libraries shipped with WebSphere Enterprise Service Bus. Using Version 6 is the recommended approach, but Version 5.3 libraries can still be used.

The mediation module build path is now updated to include the MQ jars.

8.5.3 Generate Export: Web service binding

At this point we are ready to specify the import and export binding of the mediation (that is to say, specify the protocols that this mediation is going to support). We start with the export as shown in Figure 8-34.

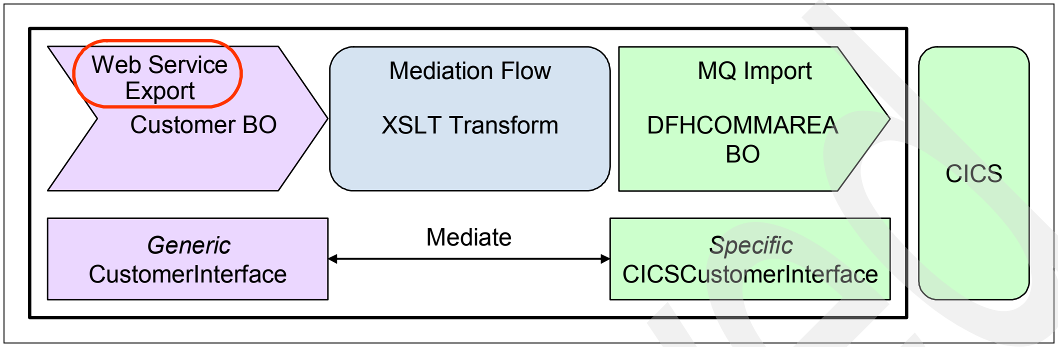


Figure 8-34 Business objects and interfaces of the mediation module

The Web service binding is straightforward to implement as there are no specific options that need to be specified. We just use the default SOAP/http binding here.

In the CICSMQAccess Assembly Diagram right-click the **WS export** component and select **Generate Binding** → **Web Service Binding**. Select **soap/http** on the transport selection panel and click **OK**.

As we see in Figure 8-35, the Web service port is added to the mediation module, together with a wsdl and a soap binding so that the mediation module can now be invoked with SOAP over HTTP.

1. Under the CICSMQAccess mediation module, expand the **Web service Ports** to see the Web Service Port we just generated: WS_CustomerInterfaceHttpPort.
2. Double-click **WS_CustomerInterfaceHttpPort** to open it with the WSDL editor. Look at the different fields, as shown in Figure 8-35. Close the editor when you are done.

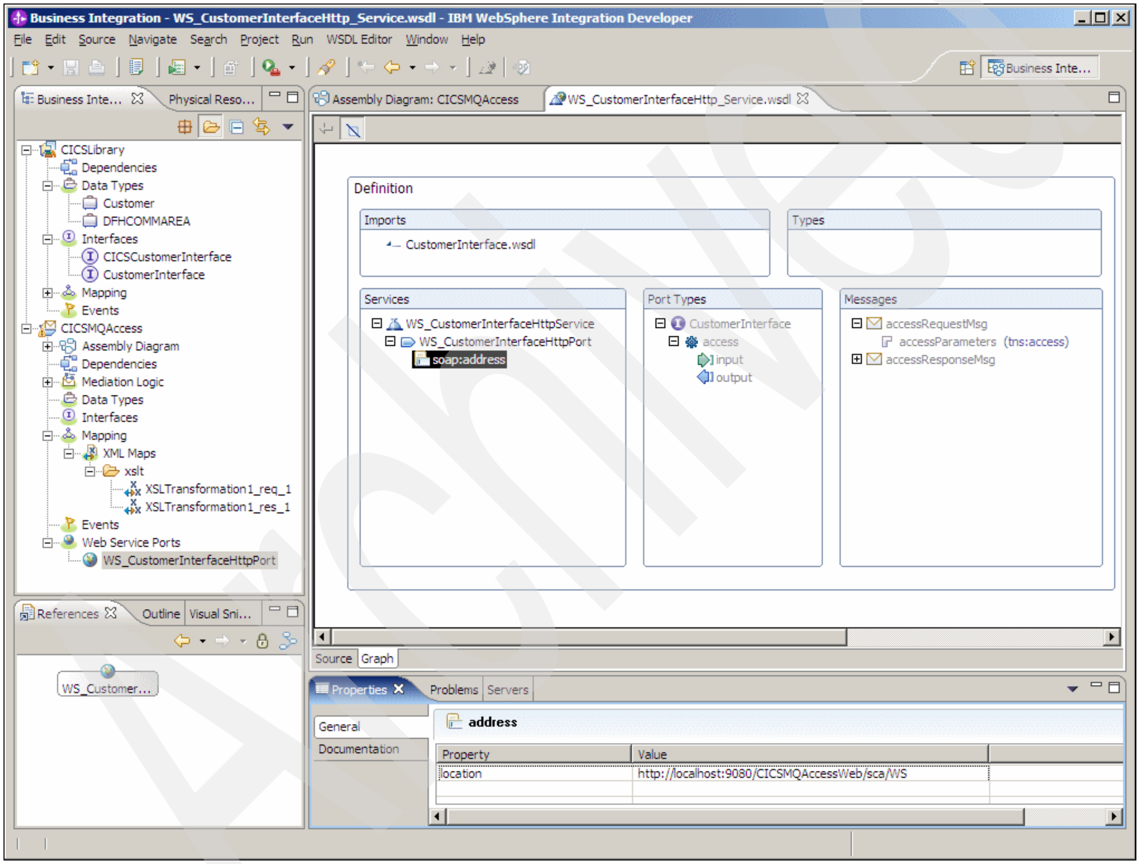


Figure 8-35 Web service export details

8.5.4 Generate import: CICS MQ binding

On the other side of the mediation module we have the MQ import (Figure 8-36).

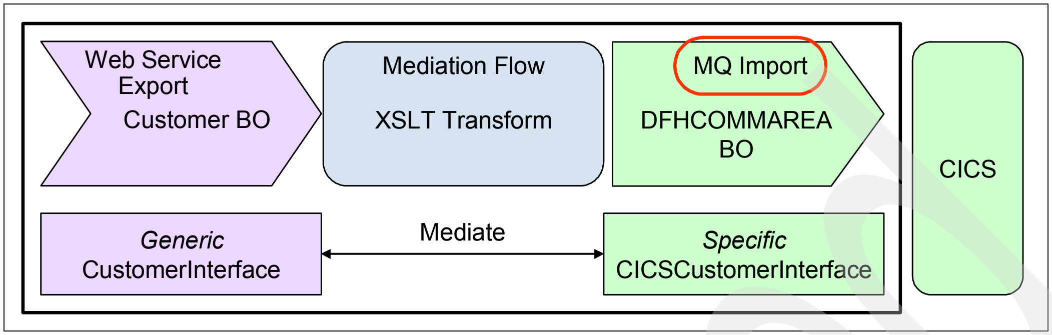


Figure 8-36 Business objects and interfaces of the mediation module

The WebSphere MQ import is more complex to implement, as it needs to provide the capability of an adapter that communicates with CICS over WebSphere MQ that WebSphere Integration Developer tooling does not provide. In Chapter 9, “Code-free CICS integration using WebSphere MQ” on page 299, we explore how to use an Eclipse plug-in that eliminates the need to write a data-binding for communicating with CICS over MQ. Figure 8-37 illustrates the design detail.

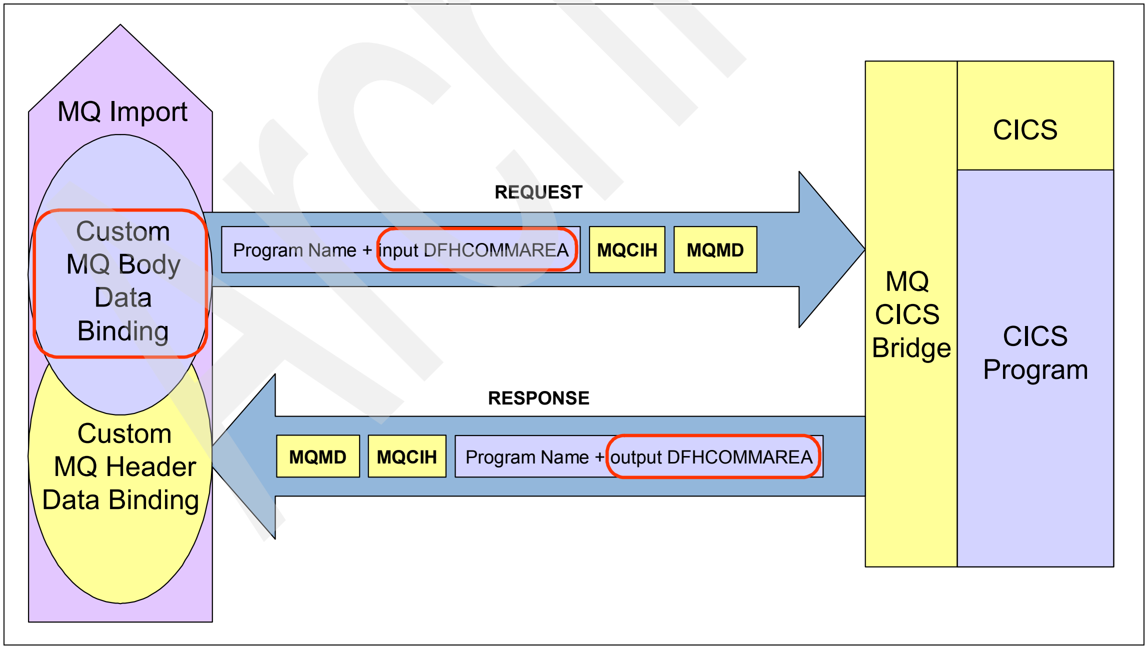


Figure 8-37 Detail of the interaction with CICS and structure of the message

Two Java classes are involved:

- A helper class used to serialize/deserialize the COBOL data types into Java. This class is generated by the Enterprise Data Discovery process as described in "CICS helper class" below. We call that class DFHCOMMAREA.
- An custom WebSphere MQ body data binding implementation class, which we provide. This class uses the CICS helper class to build the body of the message. This class is configured on the binding section of the MQ import node, as described in "Custom MQ body data binding class" on page 251. We call that class CustomerMQBodyDataBinding.java. It is similar to the class we developed in the previous chapter.

In the following three sections we discuss:

- Implementing the helper class
- Implementing the custom MQ body data binding
- Generating the MQ binding

CICS helper class

In this section we generate the helper class that serializes and deserializes COBOL types to Java objects. This class is crucial to handle the CICS interaction: encoding/decoding, ASCII/EBCDIC conversions, and code page support.

This class is generated by the CICS adapter, with the J2EE Connector Tool COBOL parser, part of the J2C Connector Tools plug-in imported in WebSphere Integration Developer as an optional feature. Refer to 8.1, "WebSphere J2EE Connector Tools" on page 217, for information about how to install this feature if you have not already.

We generate the DFHCOMMAREA.java helper class in this section and we use it in the next section, in the custom MQ body data binding implementation.

To generate this class:

1. In the Business Integration perspective, on the Physical Resources panel, expand the **com.ibm.itso** package.
2. Right-click **com.ibm.itso**, select **New** → **Other** → **J2C** → **CICS/IMS Java Data Binding**, and click **Next**.

   > **Tip:** Click **show all wizards** if you do not see the J2C folder.

3. If asked, respond **Yes** to enable the Enterprise Java Capabilities.
4. On the Importer page, make sure that the **COBOL to Java** mapping is selected.

5. Click the **Browse** button to navigate to the CICS Cobol COMMAREA file structure. Use the same file as in 8.3.1, “Application-specific DFHCOMMAREA business object” on page 221.
6. On the Importer page, select the platform and code page of your target host. As we use a z/OS system, we change the platform to z/OS, which might alter the default code page. Do not change the other fields.
7. Click the **Query** button to parse the cobol file and get access to the data structures that are defined in COBOL.

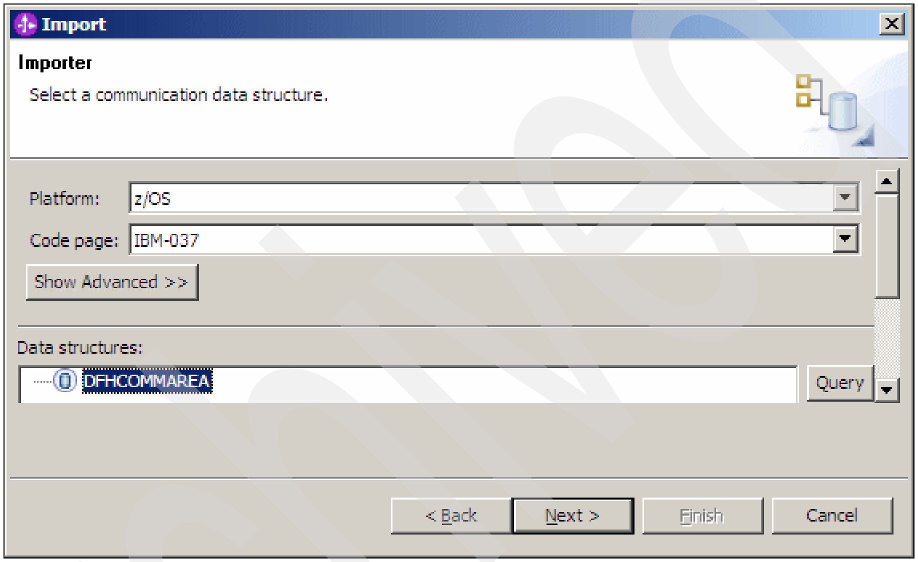


Figure 8-38 Cobol to Java J2C importer panel

8. In the Data structures panel, select **DFHCOMMAREA** and click **Next**.

9. In the Saving Properties panel specify a package named com.ibm.itso, accept all the other defaults (Figure 8-39), and click **Finish**.

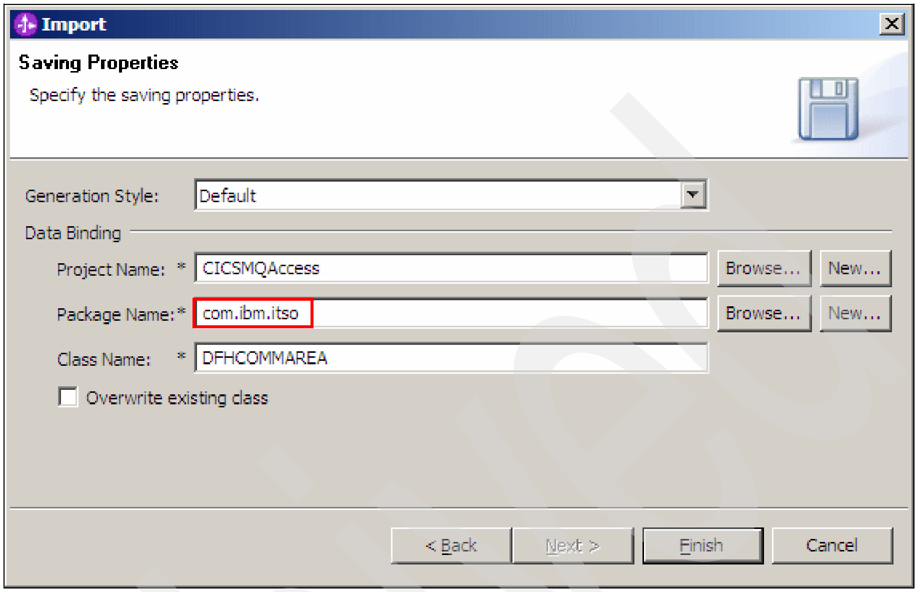


Figure 8-39 Specify the package name for the help class

A class called DFHCOMMAREA is added to the com.ibm.itso package, under the CICSMQAccess mediation module. This class is a Java representation of the DFHCOMMAREA business object we created earlier (see 8.3.1, "Application-specific DFHCOMMAREA business object" on page 221).

We parsed the same COBOL file two times:

- To generate the business object schema definition (XSD) file using the COBOL to Business Object Importer
- To generate the helper class, using the COBOL to Java Importer

The DFHCOMMAREA class is used by the custom MQ body data binding class that we discuss in the following section.

Custom MQ body data binding class

The custom MQ body data binding class is called by the SCA runtime on the MQ import node to build the body part of the MQ message. This class must implement the com.ibm.websphere.sca.mq.data.MQBodyDataBinding interface.

- Every time the WebSphere MQ import sends a WebSphere MQ message, the write() method of this class is invoked. The DFHCOMMAREA business object is dumped to the WebSphere MQ stream. We use the DFHCOMMAREA Helper class to encode the values of the business object.

- Every time the WebSphere MQ export receives a WebSphere MQ message, the read() method of this class is invoked. A DFHCOMMAREA business object is instantiated from the WebSphere MQ stream. The DFHCOMMAREA helper class is used to decode the WebSphere MQ message and build the DFHCOMMAREA business object.

  Example 8-2 on page 255 shows our implementation of this class.

To add the CustomerMQBodyDataBinding class to our project:

1. Right-click the **com.ibm.itso** folder and select **New** → **Other** → **Java** → **Class** → **Next**.

2. Name the class `CustomerMQBodyDataBinding` and click **Add**. Select the interface named **com.ibm.websphere.sca.mq.data.MQBodyDataBinding** and click **OK** (Figure 8-40) → **Finish** → **Save All**.

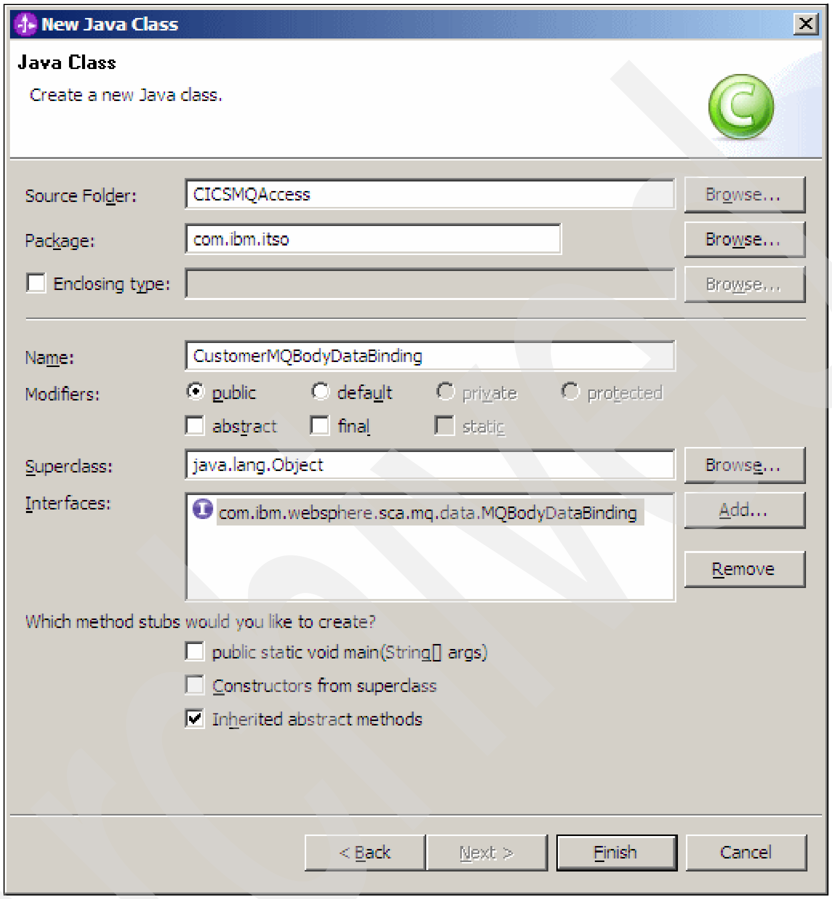


Figure 8-40 Create the CustomerMQBodyDataBinding class

In Example 8-2 on page 255 we provide the complete implementation of this class. But first let us see how the helper class assists us to create the DataObject from the WebSphere MQ message (Figure 8-41).

```
public void read(MQMD md, List headers, MQDataInputStream input)
      throws IOException {
   // Read a MQ Message - Convert message to DataObject
   System.out.println("CustomerMQBodyDataBinding.read()started");
   DFHCOMMAREA dfh = new DFHCOMMAREA();
   dfh.read(input);
   // Create DataObject using fields extracted from Bytearray by the helper class
   BOFactory boFactory = (BOFactory) new ServiceManager()
         .locateService("com/ibm/websphere/bo/BOFactory");
   this.dataObject = boFactory.create("http://CICSLibrary", "DFHCOMMAREA");
   this.dataObject.setString("ca__request__id", dfh.getCa__request__id());
   this.dataObject.setShort("ca__return__code", dfh.getCa__return__code());
   this.dataObject.setLong("ca__customer__num", dfh.getCa__customer__num());
   this.dataObject.setString("ca__first__name", dfh.getCa__first__name());
   this.dataObject.setString("ca__last__name", dfh.getCa__last__name());
   this.dataObject.setString("ca__dob", dfh.getCa__dob());
   this.dataObject.setString("ca__house__name", dfh.getCa__house__name());
   this.dataObject.setString("ca__house__num", dfh.getCa__house__num());
   this.dataObject.setString("ca__postcode", dfh.getCa__postcode());
   System.out.println("Write to dataObject: " + this.dataObject.toString());
}
```

Figure 8-41 CustomerMQBodyDataBinding read() method

The code pattern is very similar to that in Figure 7-19 on page 188. The WebSphere MQ message is read into a Bytearray, but this time the Bytearray is in effect subclassed into a DFHCOMMAREA Java object, which has a set of methods to read and write named fields to the DFHCOMMAREA Java object. The helper class is a parser of the Bytearray, using the information from the COBOL copybook to implement the record-oriented get and set methods.

Using the get and set methods, our read() method is able to build the DFHCOMMAREA business object using the get methods on the DFHCOMMAREA Java object created from the WebSphere MQ message.

The write() method is similar. It uses the set methods on the DFHCOMMAREA Java class to create a Bytearray from the DFHCOMMAREA business object (Figure 8-42).

```
public void write(MQMD md, List headers, MQDataOutputStream output)
      throws IOException {
    // Write a MQ message - convert a DataObject to a message
     System.out.println("CustomerMQBodyDataBinding.write() started");
     // Build DFHCOMMAREA with DataObject
     DFHCOMMAREA dfh = new DFHCOMMAREA();
     dfh.setCa__request__id(this.dataObject.getString("ca__request__id"));
     dfh.setCa__return__code(this.dataObject.getShort("ca__return__code"));
     dfh.setCa__customer__num(this.dataObject.getLong("ca__customer__num"));
     dfh.setCa__first__name(this.dataObject.getString("ca__first__name"));
     dfh.setCa__last__name(this.dataObject.getString("ca__last__name"));
     dfh.setCa__dob(this.dataObject.getString("ca__dob"));
     dfh.setCa__house__name(this.dataObject.getString("ca__house__name"));
     dfh.setCa__house__num(this.dataObject.getString("ca__house__num"));
     dfh.setCa__postcode(this.dataObject.getString("ca__postcode"));
     System.out.println("Write to WMQ message: " + dfh.toString());
     output.write(dfh.getBytes());
 }
```

Figure 8-42 CustomerMQBodyDataBinding write() method

There are also the get() and set() methods to complete (Figure 8-43).

```
public DataObject getDataObject() throws DataBindingException {
   return this.dataObject;
}
public void setDataObject(DataObject dataObject) throws DataBindingException {
   this.dataObject = dataObject;
}
```

Figure 8-43 CustomerMQBodyDataBinding get() and set() methods

Copy the implementation in Example 8-2 and paste it into the Java editor. Click **Save** → **Close**.

Example 8-2 Implementation of CustomerMQBodyDataBinding class

```
package com.ibm.itso;
import java.io.IOException;
import java.util.List;
```

```
import com.ibm.mq.data.MQDataInputStream;
import com.ibm.mq.data.MQDataOutputStream;
import com.ibm.websphere.sca.mq.data.MQBodyDataBinding;
import com.ibm.websphere.sca.mq.structures.MQMD;
import commonj.connector.runtime.DataBindingException;
import commonj.sdo.DataObject;
import com.ibm.websphere.bo.BOFactory;
import com.ibm.websphere.sca.ServiceManager;

public class CustomerMQBodyDataBinding implements MQBodyDataBinding {
   private DataObject dataObject;

   public void read(MQMD md, List headers, MQDataInputStream input)
         throws IOException {
      // Read a MQ Message - Convert message to DataObject
      System.out.println("CustomerMQBodyDataBinding.read()started");
      DFHCOMMAREA dfh = new DFHCOMMAREA();
      dfh.read(input);
      // Create DataObject using the fields extracted from the Bytearray by
      // the helper class
      BOFactory boFactory = (BOFactory) new ServiceManager()
            .locateService("com/ibm/websphere/bo/BOFactory");
      this.dataObject = boFactory.create("http://CICSLibrary", "DFHCOMMAREA");
      this.dataObject.setString("ca__request__id", dfh.getCa__request__id());
      this.dataObject.setShort("ca__return__code", dfh.getCa__return__code());
      this.dataObject
            .setLong("ca__customer__num", dfh.getCa__customer__num());
      this.dataObject.setString("ca__first__name", dfh.getCa__first__name());
      this.dataObject.setString("ca__last__name", dfh.getCa__last__name());
      this.dataObject.setString("ca__dob", dfh.getCa__dob());
      this.dataObject.setString("ca__house__name", dfh.getCa__house__name());
      this.dataObject.setString("ca__house__num", dfh.getCa__house__num());
      this.dataObject.setString("ca__postcode", dfh.getCa__postcode());
      System.out
            .println("Write to dataObject: " + this.dataObject.toString());
   }

   public void write(MQMD md, List headers, MQDataOutputStream output)
         throws IOException {
      // Write a MQ message - convert a DataObject to a message
      System.out.println("CustomerMQBodyDataBinding.write() started");
      // Build DFHCOMMAREA with DataObject
      DFHCOMMAREA dfh = new DFHCOMMAREA();
      dfh.setCa__request__id(this.dataObject.getString("ca__request__id"));
      dfh.setCa__return__code(this.dataObject.getShort("ca__return__code"));
```

```
      dfh.setCa__customer__num(this.dataObject.getLong("ca__customer__num"));
      dfh.setCa__first__name(this.dataObject.getString("ca__first__name"));
      dfh.setCa__last__name(this.dataObject.getString("ca__last__name"));
      dfh.setCa__dob(this.dataObject.getString("ca__dob"));
      dfh.setCa__house__name(this.dataObject.getString("ca__house__name"));
      dfh.setCa__house__num(this.dataObject.getString("ca__house__num"));
      dfh.setCa__postcode(this.dataObject.getString("ca__postcode"));
      System.out.println("Write to WMQ message: " + dfh.toString());
      output.write(dfh.getBytes());
   }

   public void setBusinessException(boolean arg0) {
   }

   public boolean isBusinessException() {
      return false;
   }

   public void setFormat(String arg0) {
   }

   public String getFormat() {
      // TODO Auto-generated method stub
      return null;
   }

   public DataObject getDataObject() throws DataBindingException {
      return this.dataObject;
   }

   public void setDataObject(DataObject dataObject)
         throws DataBindingException {
      this.dataObject = dataObject;
   }
}
```

Generate MQ binding

The next step is to generate the WebSphere MQ binding for the MQClaimStatusImport using the data binding class we have just written.

1. In the Business Integration perspective switch to the Business Integration view. Re-open the Assembly editor for the CICSMQAccess mediation module.

2. Right-click **MQClaimStatusImport** and click **Generate Binding** → **Messaging Binding** → **MQ Binding**, and the MQ Import Binding panel opens.
3. Fill in all of the required values. These depends on your CICS/MQ configuration. Figure 8-44 on page 259 shows the value we used in our tests. Click **OK** → **Save**.

> **Note:** For our tests, we did not use JNDI names for pre-configured WebSphere MQ resources on the server for simplicity, but the recommended approach is to use them.

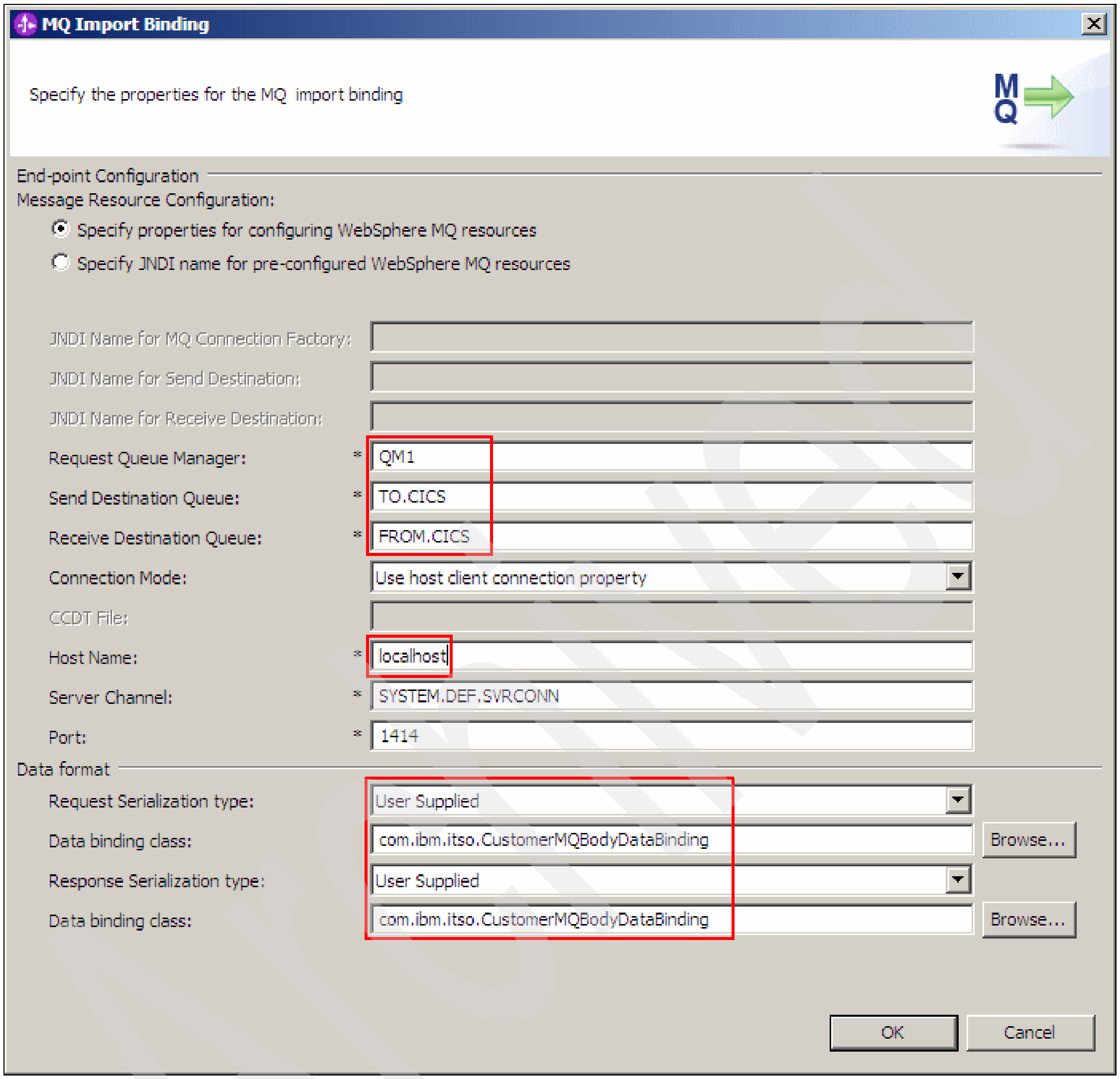


Figure 8-44 MQ Import Binding panel

The assembly diagram of the mediation module should look like Figure 8-45.

Figure 8-45 Assembly diagram of the mediation module

Tip: Note that all exclamation marks have been removed since we have provided an implementation for all components.

8.6 Add a MQCIH header to an MQ message

We now have a mediation module able to build a WebSphere MQ message containing the CICS user data (COMMAREA) in a form that the COBOL transaction can understand.

We now need to add the specific MQCIH structure to the message and fill it with adequate values so that the message can go across the CICS/MQ bridge to the CICS transaction and come back with the result.

Three steps are discussed in this section:

1. To add the MQCIH structure to the message, we add a *dummy* MQCIH to the Service Message Object (SMO) in the mediation flow before it gets delivered to the MQ Import node. To do that we need to add a custom mediation primitive to the mediation flow, that calls a Java class and adds the MQCIH header to he SMO.
2. To fill in the MQCIH structure with adequate values, we implement a custom MQ Header Data Binding Java class.

3. The custom MQ header data binding class is configured on the MQ import to be invoked by the SCA runtime like a handler, as shown in Figure 8-46.

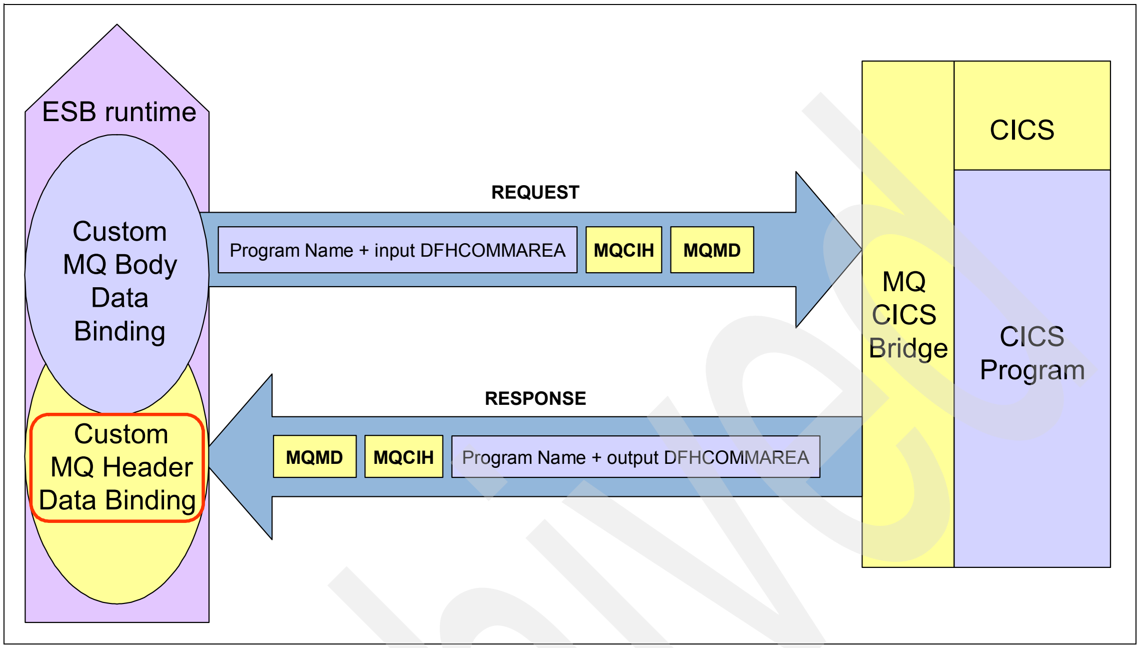


Figure 8-46 Detail of the interaction with CICS and structure of the message

The result is that we build a MQ message containing both the MQCIH structure and the CICS user data generated in our CustomerMQBodyDataBinding class.

8.6.1 Add the MQCIH header to the MQ message

To add a new header structure to the Service Message Object (SMO), we change the mediation flow. We add a custom mediation primitive with an external Java implementation where we can access the SCA API and modify the structure of the SMO.

In the CICSMQAccess Assembly Diagram:

1. Double-click the **CICSMediation** mediation flow component to open it in the mediation flow editor. Click the flow between the CustomerInterface and CICSCustomerInterfacePartner to open the Request:access flow.

2. From the palette, select the custom mediation primitive icon and drop it on the canvas area to create a custom mediation component. Name it `AddMQCIHeader`. Wire it up as shown in Figure 8-47.

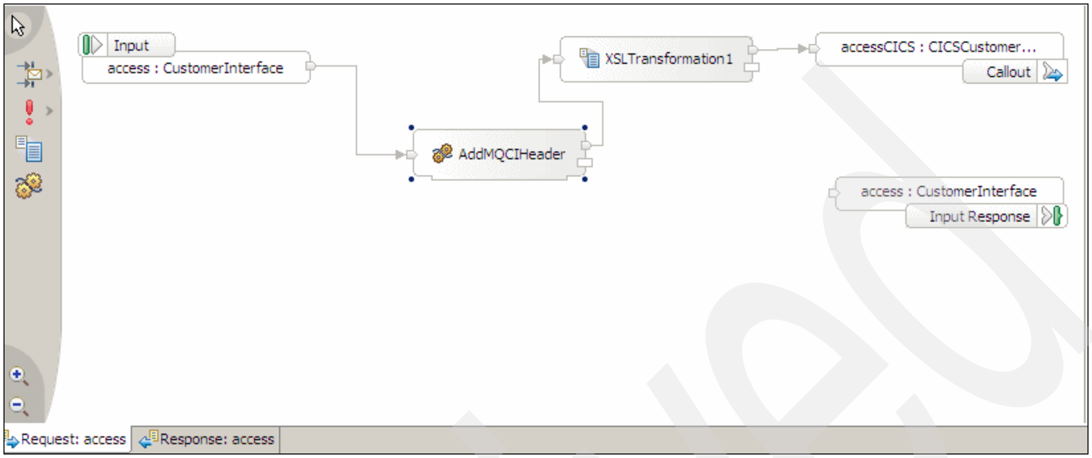


Figure 8-47 Add custom mediation primitive to the flow

3. Select **AddMQCIHHeader** → **Properties** view → **Details** tab. Select the Implementation radio button **Invoke** and answer **YES** to any pop-up.
4. In the modified Details tab click the **Reference** pull-down and select **Add a new reference**.

5. On the Add reference panel, select the second radio button, **Generate a new interface for this reference**. Type `CICSMQAccess` as the Module Name and `AddMQCIHHeaderInterface` as the Interface Name (Figure 8-48). Click **OK** → **Save** → **Close**.

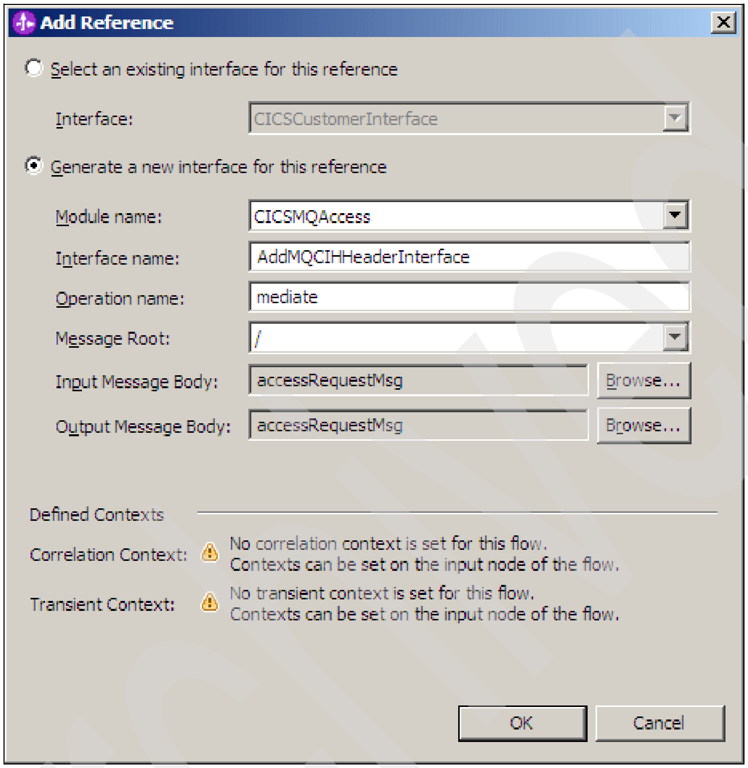


Figure 8-48 Add reference to customer mediation Java program

Next create the Java component that implements the AddMQCIHHeaderInterface interface.

1. Open the assembly diagram. From the palette, select the Java component icon , drop it onto the assembly diagram, and name it `AddMQCIH`.
2. Drag **AddMQCIHHeaderInterface** from the Explorer and drop on the AddMQCIH Java component in the assembly diagram.
3. Right-click **CICSMediation** and select **Add** → **Reference**. Select **AddMQCIHHeaderInterface** → **OK**.

4. Right-click **CICSMediation** and select **Wire to existing** → **Save** (Figure 8-49).

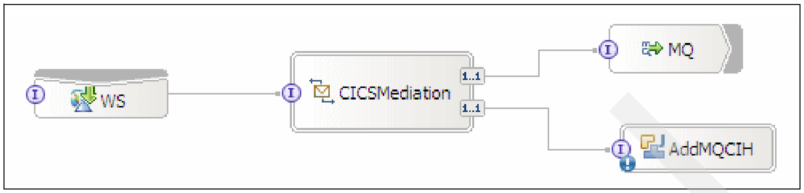


Figure 8-49 Mediation module with the Java component

> **Tip:** If you get a compilation error when saving your mediation module, right-click the **CICSMediation** flow component, select **Synchronize Interfaces and References** → **From implementation**, save the mediation module, and do a clean build of the project.

5. Double-click the Java component to provide its implementation. Answer **Yes** when asked to implement the component now. Select the **com.ibm.itso** package for the folder name.

   A new Java class is generated in the project called AddMQCIHHeaderImpl. This class is the actual implementation of the custom mediation primitive AddMQCIH.

> **Tip:** If you get a core Eclipse exception here, delete back to the customer mediation primitive, removing the generated AddMQCIHHeaderInterface, do a workspace clean (invoke WebSphere Integration Developer with the -clean option), and redo the steps again.

The Java editor should now be opened on the AddMQCIHHeaderImpl class. Keep that window open and proceed to the next step.

Generate the MQCIH helper class

To add a MQCIH header structure to the SMO, we build a MQCIH data object based on an MQCIH XML schema definition file (or xsd file) for that data object. We generate it from the COBOL definition:

- The MQCIH schema is generated using the Enterprise Data Discovery tool that we have already used in this chapter (refer to 8.3.1, “Application-specific DFHCOMMAREA business object” on page 221).
- The COBOL definition of the MQCIH header that we use here is shipped with WebSphere MQ for z/OS in the MQ target library named

SCSQCOBC(CMQCIHL). For more information about this library, refer to the WebSphere MQ for z/OS documentation.

Example 8-3 shows the MQCIH Cobol data structure that we used for our tests and that can be re-used to generate the MQCIH schema.

> **Tip:** You can cut the COBOL data structure given in Example 8-3 on page 265, paste it into a file with a cpy extension (like mqcih.cpy), and use it in the next section to generate the schema definition file.
>
> Make sure that you include the first height white characters before the level 01 block in order for the parser to work okay.

Example 8-3 Cobol source file of the MQCIH structure

```
        01  MQCIH.
           15  MQCIH-STRUCID PIC X(4).
           15  MQCIH-VERSION PIC S9(9) BINARY.
           15  MQCIH-STRUCLENGTH PIC S9(9) BINARY.
           15  MQCIH-ENCODING PIC S9(9) BINARY.
           15  MQCIH-CODEDCHARSETID PIC S9(9) BINARY.
           15  MQCIH-FORMAT PIC X(8).
           15  MQCIH-FLAGS PIC S9(9) BINARY.
           15  MQCIH-RETURNCODE PIC S9(9) BINARY.
           15  MQCIH-COMPCODE PIC S9(9) BINARY.
           15  MQCIH-REASON PIC S9(9) BINARY.
           15  MQCIH-UOWCONTROL PIC S9(9) BINARY.
           15  MQCIH-GETWAITINTERVAL PIC S9(9) BINARY.
           15  MQCIH-LINKTYPE PIC S9(9) BINARY.
           15  MQCIH-OUTPUTDATALENGTH PIC S9(9) BINARY.
           15  MQCIH-FACILITYKEEPTIME PIC S9(9) BINARY.
           15  MQCIH-ADSDESCRIPTOR PIC S9(9) BINARY.
           15  MQCIH-CONVERSATIONALTASK PIC S9(9) BINARY.
           15  MQCIH-TASKENDSTATUS PIC S9(9) BINARY.
           15  MQCIH-FACILITY PIC X(8).
           15  MQCIH-FUNCTION PIC X(4).
           15  MQCIH-ABENDCODE PIC X(4).
           15  MQCIH-AUTHENTICATOR PIC X(8).
           15  MQCIH-RESERVED1 PIC X(8).
           15  MQCIH-REPLYTOFORMAT PIC X(8).
           15  MQCIH-REMOTESYSID PIC X(4).
           15  MQCIH-REMOTETRANSID PIC X(4).
           15  MQCIH-TRANSACTIONID PIC X(4).
           15  MQCIH-FACILITYLIKE PIC X(4).
           15  MQCIH-ATTENTIONID PIC X(4).
```

```
15  MQCIH-STARTCODE PIC X(4).
15  MQCIH-CANCELCODE PIC X(4).
15  MQCIH-NEXTTRANSACTIONID PIC X(4).
15  MQCIH-RESERVED2 PIC X(8).
15  MQCIH-RESERVED3 PIC X(8).
15  MQCIH-CURSORPOSITION PIC S9(9) BINARY.
15  MQCIH-ERROROFFSET PIC S9(9) BINARY.
15  MQCIH-INPUTITEM PIC S9(9) BINARY.
15  MQCIH-RESERVED4 PIC S9(9) BINARY.
```

To generate the MQCIH.xsd file:

1. In the Business Integration perspective, open the CICSMQAccess module, right-click the **Data Types** folder, select **New** → **Enterprise Data Discovery** → **Browse**, navigate to the MQCIH cobol file, and click **Open**.
2. Choose **COBOL to Business Object** → **Next**.
3. On the Importer page, select the platform and code page of your target host. As we use a z/OS system, change the platform to z/OS. Leave all of the other fields unchanged. Click **Apply** to parse the COBOL file.
4. In the Data structures panel select **MQCIH**, then click **Next** → **Finish**.

The MQCIH business object is now created in the Data Types folder of the mediation module. Browse it to check the contents (Figure 8-50).

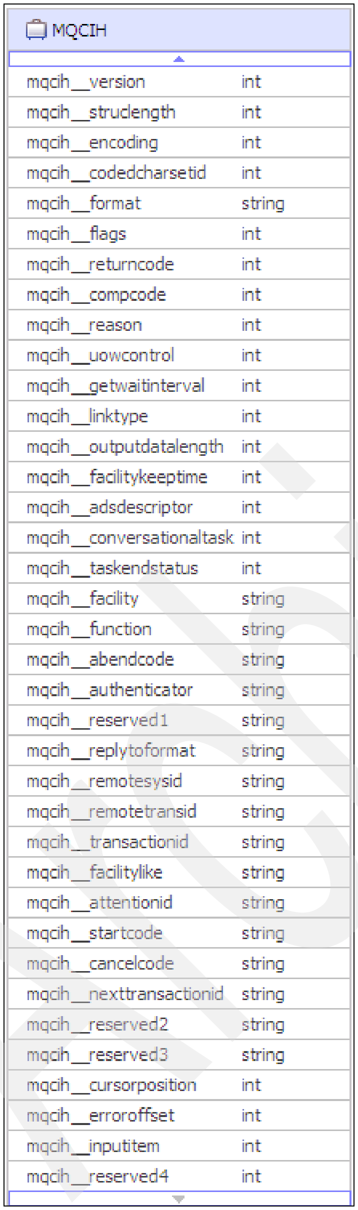


Figure 8-50 MQCIH business object

Custom mediation implementation

Using the XML schema definition file we have generated for the MQCIH structure, we can now provide an implementation for our custom mediation primitive.

In Figure 8-4 on page 221 we give an example of an implementation in which the requirements in Table 8-2 are met.

Table 8-2 Required fields in MQMD and MQCIH example

Field	Structure	Value
CCSID	MQMD	500
encoding		785
correlation		MQCI_NEW_SESSION
CCSID	MQCIH	500
encoding		785
program name		LGICUS01

> **Attention:** Change this code to accommodate any other requirements that you may have to follow for your own case.

In the mediate() method, we create an instance of an MQCIH data object using the SCA API and add it to the Service Mediation Object instance.

1. Find the Editor tab with the Java editor opened on the AddMQCIHHeaderImpl.Java class.

   Otherwise, open the CICSMQAccess assembly diagram. Double-click the custom mediation Java component **AddMQCIH**. The Java editor opens and displays the implementation of the AddMQCIHHeaderImpl.Java class.

2. Copy the mediate method provided in Figure 8-52 on page 270 and paste it in place of the skeleton mediate method in AddMQCIH.java. Click **Save** → **Close**.

The keys to understanding this code are twofold. The interface to this class was defined in the AddMQCIHeader custom mediation we added to the Request:access flow in CICSMediation (Figure 8-51 on page 269). This defined the operation, and the data that is passed in root. A slash (\) is passed to the mediate method. What is a slash?

A slash is the service message object. It is defined in the WebSphere Integration Developer Infocenter and can be found at:

http://publib.boulder.ibm.com/infocenter/dmndhelp/v6rxmx/index.jsp?topic=/com.ibm.wbit.help.medprim.doc/ref/rwesb_SMOStructure.html

The complete AddMQCIH.java class is given in Figure 8-53 on page 271.

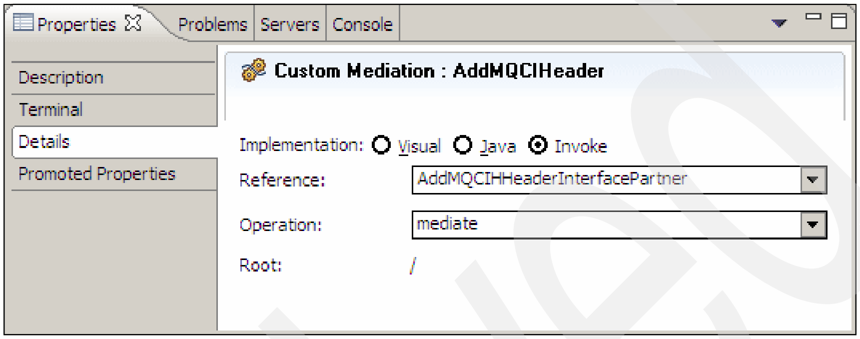


Figure 8-51 Interface to AddMQCIHImpl.java

Referring to the numbers in Figure 8-52 and using the information from the Infocenter in the hyperlink above:

1. *headers* is one of the immediate children of \ on a level with the message context and body, and contains header information associated with the message.
2. This statement creates an MQHeader as a child of headers that models WebSphere MQ message header information.
3. *control* is a child of MQheader and has a field that describes the format and encoding information describing the message body.

```
public DataObject mediate(DataObject input1) {
   DataObject mqHeader = input1.getDataObject("headers"). 1
         createDataObject("MQHeader"); 2
   DataObject control = mqHeader.createDataObject("control"); 3
   control.setLong("CodedCharSetId", 500L); 4
   control.setLong("Encoding", 785L); 4
   DataObject md = mqHeader.createDataObject("md"); 5
   md.setBytes("CorrelId", MQC.MQCI_NEW_SESSION);
   DataObject cihHeader = mqHeader.createDataObject("header"); 6
   cihHeader.set("Format", MQC.MQFMT_CICS.trim()); 4
   cihHeader.setLong("CodedCharSetId", 500L); 4
   cihHeader.setLong("Encoding", 785L); 4
   BOFactory bof = (BOFactory) ServiceManager.INSTANCE.
         locateService("com/ibm/websphere/bo/BOFactory");
   DataObject cih = bof.create("http://CICSMQAccess", "MQCIH"); 7
   cihHeader.set("value", cih); 8
   return input1;
}
```

Figure 8-52 AddMQCIHImpl.java

4. These are leaves of the SMO, and are set with the values from Table 8-2 on page 268.
5. *md* is a child of the MQHeader and contains MQ message descriptor (MQMD), excluding message format information.
6. *header* is another child of MQHeader and contains either predefined headers (*rfh, rfh2*), an *opaque* binary MQHeader, or *value*, which is an arbitrary DataObject representing the message header, usually requiring a user-provided MQ header data binding. We use value.
7. This creates the arbitrary DataObject we supply as *value*.

8. Finally, the cih structure, created as a DataObject, is inserted into the MQHeader structure as a *header value*.

```
package com.ibm.itso;
import commonj.sdo.DataObject;
import com.ibm.mq.MQC;
import com.ibm.websphere.bo.BOFactory;
import com.ibm.websphere.sca.ServiceManager;
public class AddMQCIHImpl {
   public AddMQCIHImpl() {
      super();
   }
   private Object getMyService() {
      return (Object) ServiceManager.INSTANCE.locateService("self");
   }
   public DataObject mediate(DataObject input1) {
      DataObject mqHeader =
            input1.getDataObject("headers").createDataObject("MQHeader");
      DataObject control = mqHeader.createDataObject("control");
      control.setLong("CodedCharSetId", 500L);
      control.setLong("Encoding", 785L);
      DataObject md = mqHeader.createDataObject("md");
      md.setBytes("CorrelId", MQC.MQCI_NEW_SESSION);
      DataObject cihHeader = mqHeader.createDataObject("header");
      cihHeader.set("Format", MQC.MQFMT_CICS.trim());
      cihHeader.setLong("CodedCharSetId", 500L);
      cihHeader.setLong("Encoding", 785L);
      BOFactory bof = (BOFactory) ServiceManager.INSTANCE.
            locateService("com/ibm/websphere/bo/BOFactory");
      DataObject cih = bof.create("http://CICSMQAccess", "MQCIH");
      cihHeader.set("value", cih);
      return input1;
   }
}
```

Figure 8-53 AddMQCIH.java complete class

8.6.2 Custom MQ header data binding implementation

Next, we need to provide an implementation of the custom MQ header data binding class, where we fill all the MQCIH header fields with values. This class implements the com.ibm.websphere.sca.mq.data.MQHeaderDataBinding interface. Refer to Figure 8-46 on page 261 for a complete view of the interaction.

Add the CICSMQHeaderDataBinding class to the project

To do this:

1. In the Business Integration perspective, on the Physical Resources panel right-click **com.ibm.itso**, select **New** → **Other** → **Java** → **Class** → **Next**, and name the class `CICSMQHeaderDataBinding`.
2. In the Interfaces field, click **Add** → com.ibm.websphere.sca.mq.data.MQHeaderDataBinding → **OK** → **Finish** (Figure 8-54).

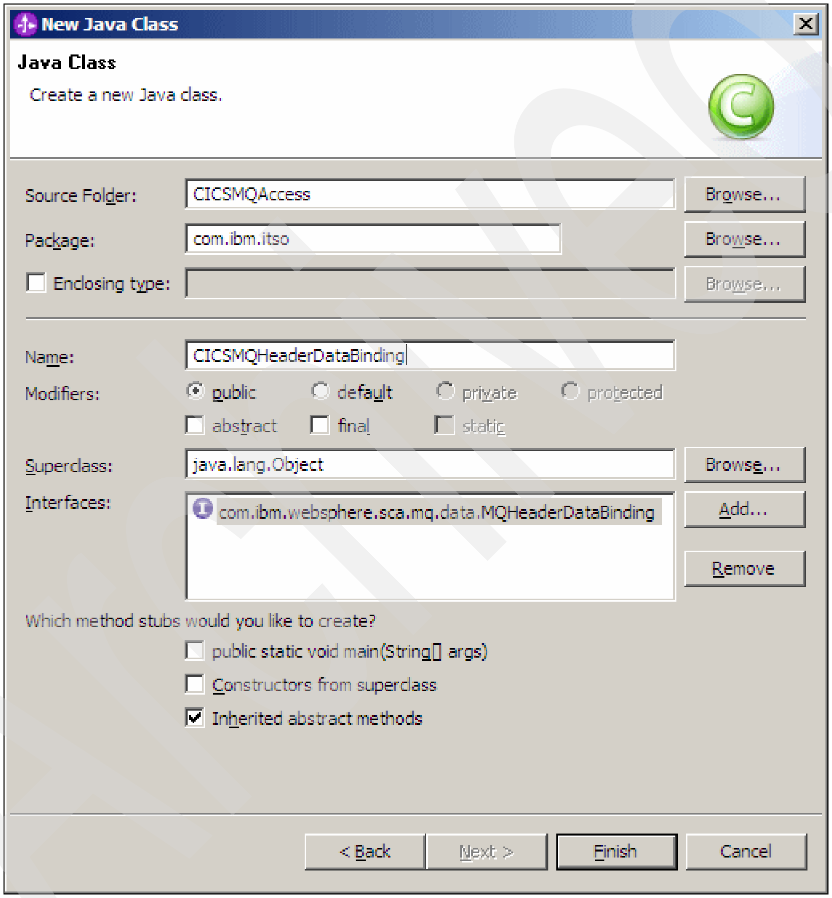


Figure 8-54 Create the CICSMQHeaderDataBinding class

Direct implementation of CICSMQHeaderDataBinding

In Example 8-4 on page 276 we provide the implementation used during our tests. The following methods are invoked in the following order:

1. Request (outbound)

   – The CustomerMQBodyDataBinding::write() method is invoked first. The body of the message is built.

   – Then the header class.isSupportedFormat() is invoked by the runtime.

     If the format string contained in the SMO matches the format implemented by this class, we return true, and the next methods are called:

     - setNextCCSID()
     - setNextEncoding()
     - setNextFormat()
     - setDataObject()
     - write()

     The setNextXXXX() methods tell the CICSMQHeaderDataBinding class what the CCSID, encoding, and format are for the next structure in the MQ header chain after the MQCIH header so that we can write them into fields in our header that describe the next structure (in our case the data body).

     The MQCIH is exceptional in that the header and the body (that is, the COMMAREA) must be in the same encoding and character set, which turns out to be the values set in the MQMD. For this reason the encoding and ccsid fields in the MQCIH are reserved, and any values we set are ignored.

     The MQCIH header structure is written to the MQ output stream. Its values are all constants, so we do not read them from the dataObject instance variable (it has not been initialized anyway, as the read() method has not been called. The MQCIH was not present in the input stream.).

2. Response (inbound)

   – On the response, the header class isSupportedFormat() is invoked first.

     If the format string contained in the SMO matches the format implemented by this class (that is, ‘MQCICS’), we return true, and the next methods are called.

     - read()

       In the read() method, we read the MQCIH header from the input stream and should set the values of the next format, CCSID, and encoding from the inbound MQCIH header into the class attributes for these fields. But since the encoding and CCSID fields are reserved we should check that the values are correctly returned in the MQCIH by tracing the output on the console during development.

We do not need to create a business object for the MQCIH on read(), as it is never used in the mediation. However, the example code shows how you would do this.

The following methods are then called: getDataObject(),getNextFormat(), getNextCCSID(), and getNextEncoding().

– The CustomerMQBodyDataBinding::read()

  The body of the message is read in the custom body data binding class and decoded using the DFHCOMMAREA helper class.

An example trace of the methods is given in Figure 8-55.

```
AddMQCHIImpl started
AddMQCHIImpl finished
CustomerMQBodyDataBinding.setFormat()
CustomerMQBodyDataBinding.setBusinessException() false
CustomerMQBodyDataBinding.setDataObject BO type = DFHCOMMAREA,...
CustomerMQBodyDataBinding.write() started
CustomerMQBodyDataBinding.write() finished
CustomerMQBodyDataBinding.getFormat()
CICSMQHeaderDataBinding.isSupportedFormat: MQCICS   must equal MQCICS
CICSMQHeaderDataBinding.setNextCCSID() 500
CICSMQHeaderDataBinding.setNextEncoding() 785
CICSMQHeaderDataBinding.setNextFormat()
CICSMQHeaderDataBinding.setDataObject() BO type = MQCIH,...
CICSMQHeaderDataBinding.write() started
CICSMQHeaderDataBinding.write() finished
============== CICS/MQ Bridge ===============================================
CICSMQHeaderDataBinding.isSupportedFormat: MQCICS must equal MQCICS
CICSMQHeaderDataBinding.read() started
CICSMQHeaderDataBinding.read() finished
CICSMQHeaderDataBinding.getDataObject() BO type = MQCIH,mqcihStrucid = CIH ,...
CICSMQHeaderDataBinding.getNextFormat() MQSTR
CICSMQHeaderDataBinding.getNextCCSID() 500
CICSMQHeaderDataBinding.getNextEncoding() 546
CICSMQHeaderDataBinding.isSupportedFormat: MQSTR must equal MQCICS
CustomerMQBodyDataBinding.read()started
DFHCOMMAREA: [B@71b9d11d length: 90
CustomerMQBodyDataBinding.read()finished
CustomerMQBodyDataBinding.isBusinessException(): false
CustomerMQBodyDataBinding.isBusinessException(): false
CustomerMQBodyDataBinding.getDataObject BO type = DFHCOMMAREA,...
CustomerMQBodyDataBinding.isBusinessException(): false
```

Figure 8-55 Trace of example code

Example 8-4 shows a straightforward brute force implementation of the heading binding. A briefer and less error-prone implementation can be attempted using the CICS/IMS Java importer we used to generate the DFHCOMMAREA helper class. This is explained in “Using a helper class to implement the MQ Header data binding” on page 279. This section also explains more customization to meet the needs of the CICS/MQBridge interface.

Example 8-4 Implementation of CICSMQHeaderDataBinding class

```
package com.ibm.itso;
import java.io.IOException;
import com.ibm.mq.data.MQDataInputStream;
import com.ibm.mq.data.MQDataOutputStream;
import com.ibm.websphere.sca.mq.data.MQHeaderDataBinding;
import com.ibm.ws.sib.comms.mq.util.MQConstants;
import commonj.connector.runtime.DataBindingException;
import commonj.sdo.DataObject;

public class CICSMQHeaderDataBinding implements MQHeaderDataBinding {
   private DataObject dataObject;
   private int ccsid;
   private int encoding;
   private String format;

   public boolean isSupportedFormat(String format) {
      System.out.println("CICSMQHeaderDataBinding.isSupportedFormat: "
            + format);
      return format.trim().equals("MQCICS");
   }

   public void read(String format, MQDataInputStream input) throws IOException {
      System.out.println("CICSMQHeaderDataBinding.read()");
      readMQCIH(input);
   }

   private void readMQCIH(MQDataInputStream input) throws IOException {
      // We will just consume the input, no need to create the
      // dataObject. It is not referenced
      String strucId = input.readMQCHAR4();
      input.readMQLONG(); // Version
      int strucLength = input.readMQLONG();
      System.out.println(strucId + " length: " + strucLength);
      // Either consume all the fields at once
      // input.readBytes(strucLength-4);
      // Or, one by one, if you need the values
```

```
   encoding = input.readMQLONG();
   ccsid = input.readMQLONG();
   format = input.readMQCHAR8();
   input.readMQLONG();
   input.readMQLONG();
   input.readMQLONG();
   input.readMQLONG();
   input.readMQLONG();
   input.readMQLONG();
   input.readMQLONG();
   input.readMQLONG();
   input.readMQLONG();
   input.readMQLONG();
   input.readMQLONG();
   input.readMQLONG();
   input.readMQBYTE8();
   input.readMQCHAR4();
   input.readMQCHAR4();
   input.readMQCHAR8();
   input.readMQCHAR8();
   input.readMQCHAR8();
   input.readMQCHAR4();
   input.readMQCHAR4();
   input.readMQCHAR4();
   input.readMQCHAR4();
   input.readMQCHAR4();
   input.readMQCHAR4();
   input.readMQCHAR4();
   input.readMQCHAR4();
   input.readMQCHAR8();
   input.readMQCHAR8();
   input.readMQLONG();
   input.readMQLONG();
   input.readMQLONG();
   input.readMQLONG();
   input.readMQCHAR8();
}

public void write(String format, MQDataOutputStream output)
      throws IOException {
   System.out.println("CICSMQHeaderDataBinding.write()");
   buildMQCIH(output);
}

private void buildMQCIH(MQDataOutputStream output) throws IOException {
```

```
      output.writeMQCHAR4(MQConstants.MQCIH_STRUC_ID);
      output.writeMQLONG(MQConstants.MQCIH_VERSION_2);
      output.writeMQLONG(180);
      output.writeMQLONG(546); // encoding, value ignored - used by scaffold
      output.writeMQLONG(500); // ccsid, value ignored - used by scaffold
      output.writeMQCHAR8(MQConstants.MQFMT_STRING);
      output.writeMQLONG(MQConstants.MQCIH_NONE);
      output.writeMQLONG(0);
      output.writeMQLONG(0);
      output.writeMQLONG(0);
      output.writeMQLONG(MQConstants.MQCUOWC_ONLY);
      output.writeMQLONG(0);
      output.writeMQLONG(MQConstants.MQCLT_PROGRAM);
      output.writeMQLONG(MQConstants.MQCODL_AS_INPUT);
      output.writeMQLONG(0);
      output.writeMQLONG(0);
      output.writeMQLONG(MQConstants.MQCCT_NO);
      output.writeMQLONG(MQConstants.MQCTES_NOSYNC);
      output.writeMQBYTE8(new byte[0]);
      output.writeMQCHAR4("");
      output.writeMQCHAR4("");
      output.writeMQCHAR8("");
      output.writeMQCHAR8("");
      output.writeMQCHAR8(MQConstants.MQFMT_STRING);
      output.writeMQCHAR4("");
      output.writeMQCHAR4("");
      output.writeMQCHAR4("");
      output.writeMQCHAR4("");
      output.writeMQCHAR4("");
      output.writeMQCHAR4("");
      output.writeMQCHAR4("");
      output.writeMQCHAR4("");
      output.writeMQCHAR8("");
      output.writeMQCHAR8("");
      output.writeMQLONG(0);
      output.writeMQLONG(0);
      output.writeMQLONG(0);
      output.writeMQLONG(0);
      output.writeMQCHAR8("LGICUS01");
   }

   public void setNextFormat(String format) {
      this.format = format;
   }
```

```
   public String getNextFormat() {
      return this.format;
   }

   public void setNextCCSID(int ccsid) {
      this.ccsid = ccsid;
   }

   public int getNextCCSID() {
      return this.ccsid
   }

   public void setNextEncoding(int encoding) {
      this.encoding = encoding;
   }

   public int getNextEncoding() {
      return this.encoding
   }

   public DataObject getDataObject() throws DataBindingException {
      return this.dataObject;
   }

   public void setDataObject(DataObject dataObject) throws DataBindingException {
      this.dataObject = dataObject;
   }
}
```

Using a helper class to implement the MQ Header data binding

Refer to "CICS helper class" on page 249, where we explained how to generate the helper class from the LGEICUS01.cpy COMMAREA copybook to see in detail the steps we retake now to generate a helper class from the mqcih.cpy copybook. Here we give a brief summary of the steps.

Generate the MQCIH.java helper class

To do this:

1. Switch to the Physical Resources view and right-click the **com\ibm\itso** folder where all the other classes are. Click **New** → **Other** → **J2C** → **CICS/IMS Java Data Binding**.
2. On the Data Import panel browse to the MQCIH.cpy file. You can create it by cutting and pasting from Example 8-3 on page 265. Then click **Next**.

3. Switch the platform to z/OS. Click **Query** → **Next** → **Browse** to the com.ibm.itso Package name and then click **Finish**.

Subclass the generated MQCIH.java class

The methods in the generated class do not include one to read only a portion of the available input stream. We want to extract the MQCIH header from the stream, but leave the COMMAREA for the data binding to process.

1. Right-click the **com\ibm\itso** folder and select **New** → **Other** → **Java** → **Class**. Name the new class `MQCIHExt`. Browse to MQCIH as the Superclass and click **Finish**.
2. Add the new method read(int length), as shown in (Figure 8-56).

```
package com.ibm.itso;

public class MQCIHExt extends MQCIH {
   /** Added to read MQCIH not at the end of the inputStream
    *
    * @param inputStream
    * @throws java.io.IOException
    */
   public void read(java.io.InputStream inputStream, int length)
         throws java.io.IOException {
      byte[] input = new byte[length];
      inputStream.read(input);
      super.setBytes(input);
   }
}
```

Figure 8-56 New MQCIHExt class

Java defaults the constructor, exception, and other methods to the superclass.

Using the helper class

Example 8-5 on page 281 shows the complete CICSMQHeaderDataBinding class[2]. It is much shorter and less prone to error than the brute force code. Also, the handling of the encoding and ccsid fields has been improved.

[2] The additional materials have all of these classes available, and include the tracing statements shown in examples. The trace statements have been omitted in the examples to make the text more compact.

We now examine some of the important, or *tricky*, lines of code.

- return format.trim().equals(MQConstants.MQFMT_CICS.trim()) tests to see that this header binding handles the current header passed in by WebSphere Enterprise Service Bus.
- MQCIHExt mqCih = new MQCIHExt() uses our subclass of the MQCIH helper class.
- mqCih.read(input, mqCih.getSize()) uses the helper class to read just the MQCIH from the input stream. mqCih.getSize() is 180 bytes and does not include the eight bytes of eyecatcher and version fields. These fields are also read out of the input stream.
- After reading the MQCIH, there is a further 8-byte program name that is optionally present when calling the CICS/MQ bridge. These eight bytes are consumed here (see `http://publib.boulder.ibm.com/infocenter/wmqv6/v6r0/index.jsp?topic=/com.ibm.mq.csqzal.doc/csq3dpl.htm`).
- As already explained, nextEncoding = mqCih.getMqcih__encoding() returns the encoding of the following MQBody — the COMMAREA. The same in the following line for the CCSID. These values should be checked in a console trace to see whether they correspond to the values required.
- Also as explained, creating the instance dataObject is optional, and this code only creates the eyecatcher. The MQCIH is not used in the mediation, so it is not worth creating it. Leaving dataObject as a reference to null is fine.
- The write() method is very similar to the dataBody write method. Only the fields that are changed from their zero/blank values need to be set. The outgoing encoding and ccsid fields are reserved by the CICS-MQ bridge. The values we set are ignored, but are there to make the scaffolding work by returning these values as though they were set on return by the bridge.
- The 8-byte DPL program name (see above) is written out after the MQCIH structure.

Example 8-5 CICSMQHeaderDataBinding using helper class

```
package com.ibm.itso;
import java.io.IOException;
import com.ibm.mq.data.MQDataInputStream;
import com.ibm.mq.data.MQDataOutputStream;
import com.ibm.websphere.bo.BOFactory;
import com.ibm.websphere.sca.ServiceManager;
import com.ibm.websphere.sca.mq.data.MQHeaderDataBinding;
import com.ibm.ws.sib.comms.mq.util.MQConstants;
import commonj.connector.runtime.DataBindingException;
import commonj.sdo.DataObject;
```

```
public class CICSMQHeaderDataBinding implements MQHeaderDataBinding {
   private DataObject dataObject;
   private int nextCcsid;
   private int nextEncoding;
   private String nextFormat;

   public boolean isSupportedFormat(String format) {
      return format.trim().equals(MQConstants.MQFMT_CICS.trim());
   };

   public void read(String format, MQDataInputStream input) throws IOException {
   // just consume the CIH - build a dummy DataObject
         MQCIHExt mqCih = new MQCIHExt();
         mqCih.read(input, mqCih.getSize());
         String progName = input.readMQCHAR8(); // progname
         nextEncoding = mqCih.getMqcih__encoding();
         nextCcsid = mqCih.getMqcih__codedcharsetid();
         nextFormat = mqCih.getMqcih__format();
   // The following code starts to build a header dataObject to pass back if desired
         BOFactory boFactory = (BOFactory) new ServiceManager().
               locateService("com/ibm/websphere/bo/BOFactory");
         this.dataObject = boFactory.create("http://CICSMQAccess", "MQCIH");
         this.dataObject.setString("mqcih__strucid", mqCih.getMqcih__strucid());
      }

   public void write(String format, MQDataOutputStream output)
      MQCIH mqCIH = new MQCIH();
      mqCIH.setMqcih__strucid(MQConstants.MQCIH_STRUC_ID);
      mqCIH.setMqcih__version(MQConstants.MQCIH_VERSION_2);
      mqCIH.setMqcih__struclength(MQConstants.MQCIH_LENGTH_2);
      mqCIH.setMqcih__encoding(546); // Ignored : make scaffolding work
      mqCIH.setMqcih__codedcharsetid(500); // Ignored: make scaffolding work
      mqCIH.setMqcih__format(MQConstants.MQFMT_STRING);
      mqCIH.setMqcih__flags(MQConstants.MQCIH_NONE);
      mqCIH.setMqcih__uowcontrol(MQConstants.MQCUOWC_ONLY);
      mqCIH.setMqcih__linktype(MQConstants.MQCLT_PROGRAM);
      mqCIH.setMqcih__outputdatalength(MQConstants.MQCODL_AS_INPUT);
      mqCIH.setMqcih__conversationaltask(MQConstants.MQCCT_NO);
      mqCIH.setMqcih__taskendstatus(MQConstants.MQCTES_NOSYNC);
      mqCIH.setMqcih__facility(MQConstants.MQCFAC_NONE);
      mqCIH.setMqcih__replytoformat(MQConstants.MQFMT_STRING);
      output.write(mqCIH.getBytes());
      output.writeMQCHAR8("LGICUS01"); // progname
   }
```

```
   public void setNextFormat(String format) { nextFormat = format; }
   public String getNextFormat() { return nextFormat; }
   public void setNextCCSID(int ccsid) { nextCcsid = ccsid; }
   public int getNextCCSID() { return nextCcsid; }
   public void setNextEncoding(int encoding) { nextEncoding = encoding; }
   public int getNextEncoding() { return nextEncoding; }

   public DataObject getDataObject() throws DataBindingException {
      return this.dataObject;
   }
   public void setDataObject(DataObject dataObject)
         throws DataBindingException {
      this.dataObject = dataObject;
   }
}
```

8.6.3 Custom MQ Header Data Binding configuration

Next we configure the custom MQ header data binding class on the mediation module MQ binding Import node.

In the assembly editor for the mediation module, we change the MQ binding configuration of the import node to add the CICSMQHeaderDataBinding class. Right-click the **MQ** Import node. Click **Show in Properties**. In the Properties panel click the **Binding** tab. Then click **User Supplied MQ Header Implementation** → **Add** → **CICSMQHeaderDataBinding** → **OK** (Figure 8-57) → **Save** → **Close**.

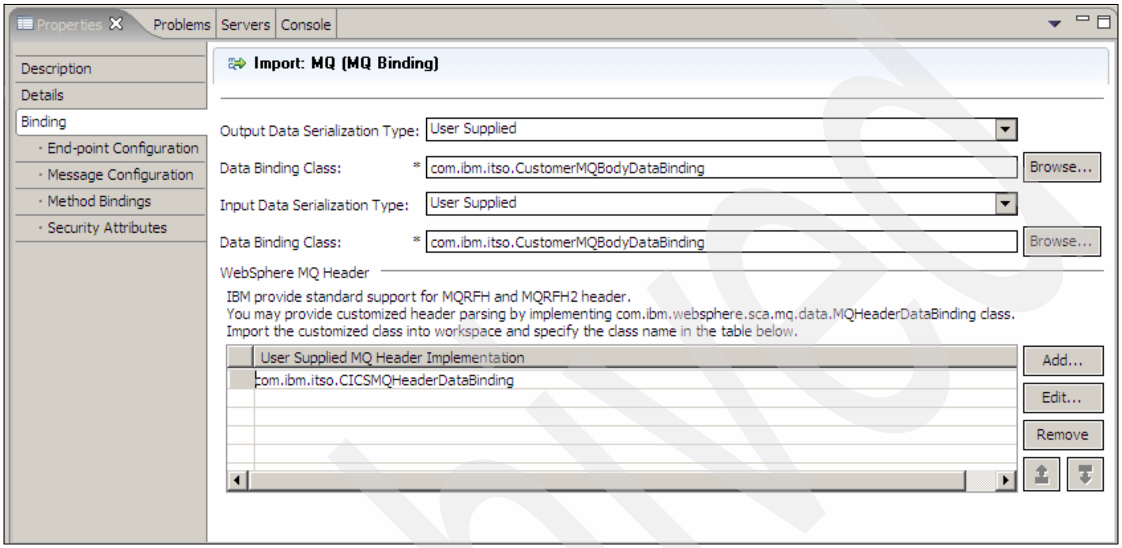


Figure 8-57 Configure the CICSMQHeaderDataBinding class on the MQ import

8.7 Test the mediation

We are ready to test the solution. We can use any Web service client to generate a SOAP request message and invoke the Web services exported port of the mediation module.

Loaded as a SMO into the WebSphere Enterprise Service Bus runtime, our message is then transformed by the mediation flow, a MQCIH header is added, and finally the whole message is written into a WebSphere MQ queue, to be sent to MQ/CICS bridge to the final CICS program (Figure 8-58).

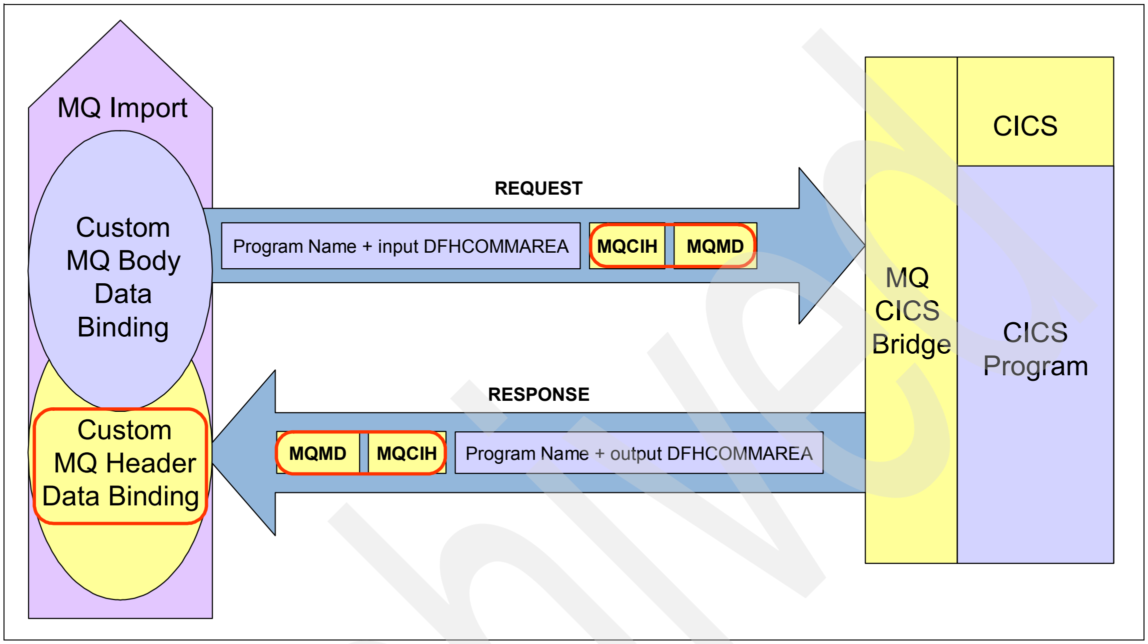


Figure 8-58 Detail of the interaction with CICS and structure of the message

Following the same path on the return, the CICS response message is sent to the WebSphere Enterprise Service Bus runtime, transformed by the mediation, and then forwarded as a SOAP response to the Web services client (see Figure 8-58).

The request/reply interaction of the mediation interface gives us a synchronous invocation flow on the Web services side, while using an asynchronous protocol on the MQ, CICS side. In a real solution one could choose to invoke the mediation using an asynchronous client.

To test the solution, we use the WebSphere ESB server integrated test environment that installed with WebSphere Integration Developer. The code has been tested with a real CICS/MQ bridge connected to a z/OS system. In the additional materials you will find a sample scaffolding program MQCICServer.java, which echoes the request back to WebSphere Enterprise Service Bus. Alternatively, you can use Jim McNair's excellent RFHUTIL utility to browse and scaffold the test (Supportpac IA03, found at
`http://www-1.ibm.com/support/docview.wss?rs=203&uid=swg24000637&loc=en_`

US&cs=utf-8&lang=en). It is possible to introduce data errors using RFHUTIL if you enter the reply data incorrectly, so the best approach is to use a combination of tools, and then deploy to a real CICS/MQ bridge.

Note: The mediation module can also be deployed on a WebSphere Process Server.

We configure the server with the WebSphere MQ libraries that we want to use, and then we deploy the mediation module.

It is important to check what WebSphere MQ com.ibm.mq and com.ibm.mqjms libraries you are using, as typically the libraries shipped with WebSphere Enterprise Service Bus are down level on the latest level of WebSphere MQ. We found that our version of WebSphere Enterprise Service Bus had WebSphere MQ v5 libraries.

1. On the Business Integration Perspective, click the **Servers** tab, right-click **WebSphere ESB Server v6.0**, and click **Start**.
2. Monitor the console until you see that the server starts. Look for the message `Server server1 open for e-business`.
3. Click the **Servers** tab, right-click **WebSphere ESB server v6.0**, and select **Run Administrative console**.
4. On the Welcome panel, enter any user ID (for example, your name), and click **Log in**.

5. Some of the administration you need to do (such as recycle message listeners) requires access to all the administrative functions. On the Welcome page select **All** → **Apply** on the Task filtering selection (Figure 8-59).

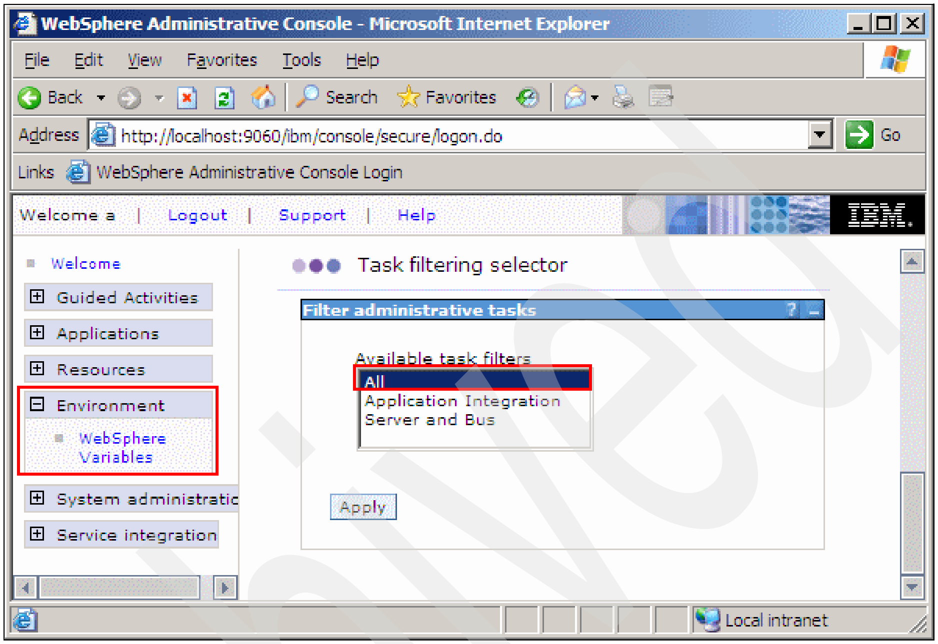


Figure 8-59 Task filtering selection

6. On the Admin Console panel, on the left-hand side expand **Environment** → **WebSphere Variables**, as shown in Figure 8-59 on page 287 and Figure 8-60. You will see in Figure 8-60 that the set of tasks has expanded on the left-hand side.

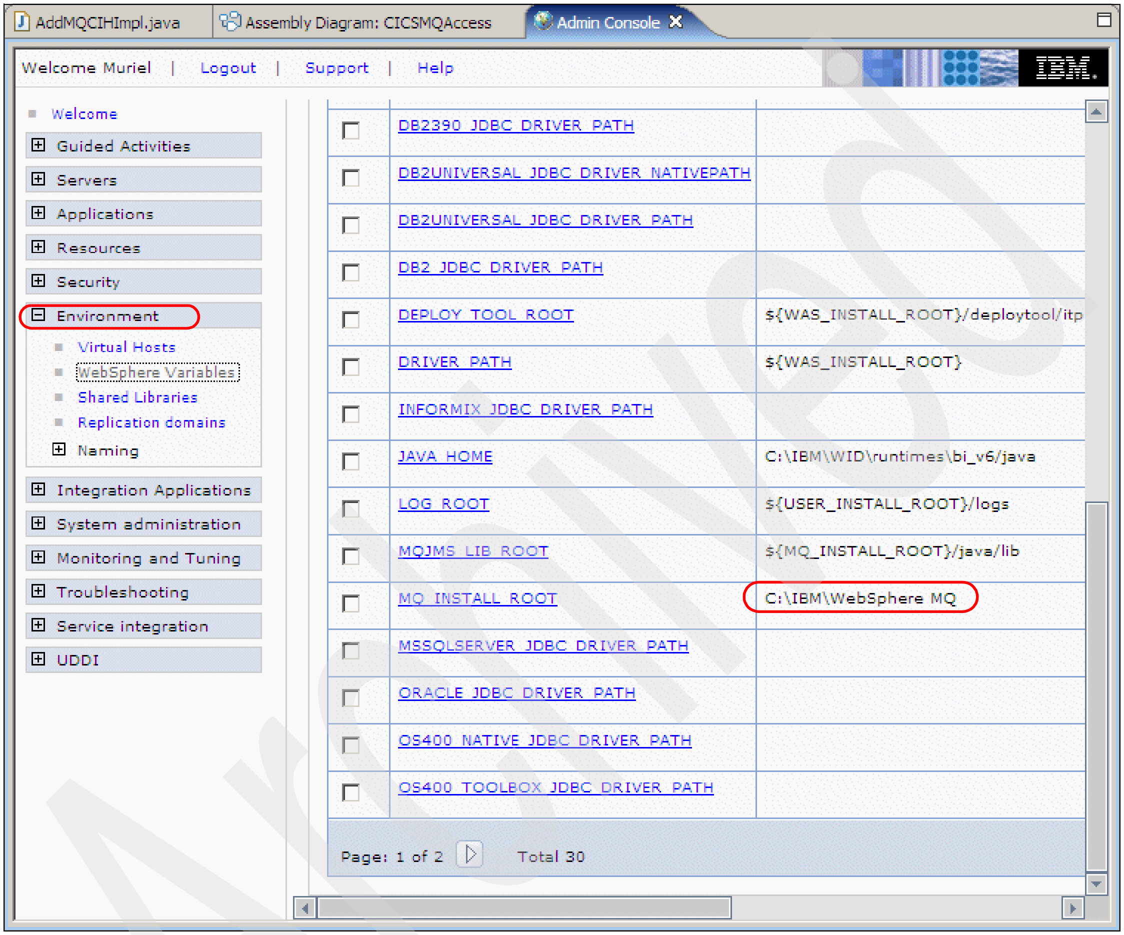


Figure 8-60 Administrative Console showing MQ_INSTALL_ROOT variable

7. Click **MQ_INSTALL_ROOT** and change the Value field to the actual WebSphere MQ installation path for your environment (C:\IBM\WebSphere MQ in our case). Click **Apply** → **Save** → **Save** again.

8. Log out from the administrative console and close the window.

9. Back on the Servers tab, restart the WebSphere ESB Server v6.0.

We did the testing on a CICS z/OS system loaned to us by the Scenario Analysis Lab in IBM Hursley. For this book we describe the testing using the

MQCICSServer scaffolding program included in the additional materials. We used WebSphere MQ Alias queues do the configurations. This way, once the solution has been developed with local scaffolding, it can easily be switched from running on a local queue manager (QM1 in our case) to resolving to a remote Queue Manager where the CICS/MQ bridge resides, without changing the definitions in WebSphere Enterprise Service Bus.

- An alias queue called TO.CICS configured to send message to the MQ/CICS bridge
- An alias queue called FROM.CICS configured to receive message from the MQ/CICS bridge

To deploy the mediation module:

1. Monitor the console until you see that the server started. Look for the message `Server server1 open for e-business`.
2. Click the **Servers** tab, right-click **WebSphere ESB server v6.0**, and select **Add and Remove Projects**.
3. On the left-hand side, under Available projects select **CICSMQAccessApp** → **Add** to move it to the right-hand side under Configured Projects (Figure 8-61). Click **Finish**.

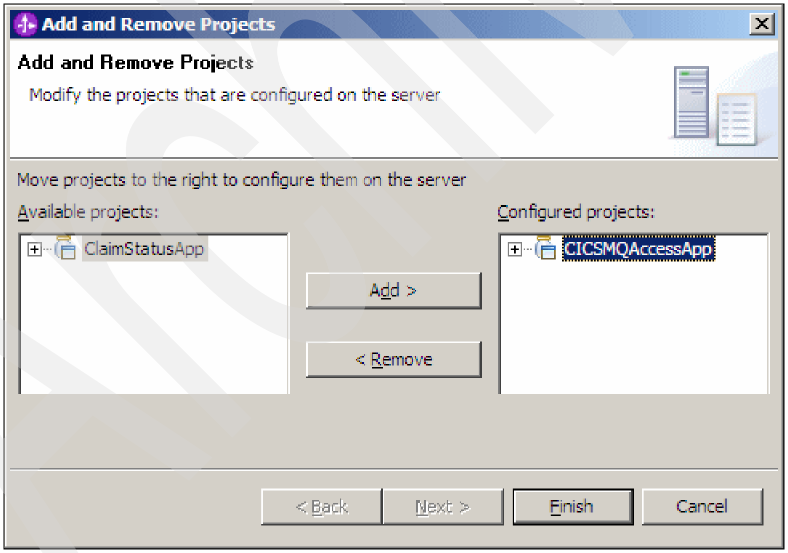


Figure 8-61 Adding MCSMQAccessApp

4. Wait until the publishing of the application is complete. Monitor the console log to verify that there is no error.

Tip: If the queue manager is not running when you start the application, you will see errors in the console. Just start your queue manager and restart the application.

The mediation module is now installed and started on the server. We can use the integrated test client tool of WebSphere Integration Developer to test the mediation module.

5. On the Business Integration Perspective, in the Business Integration view, right-click the **CICSMQAccess** mediation module and select **Test** → **Test Module** (Figure 8-62).

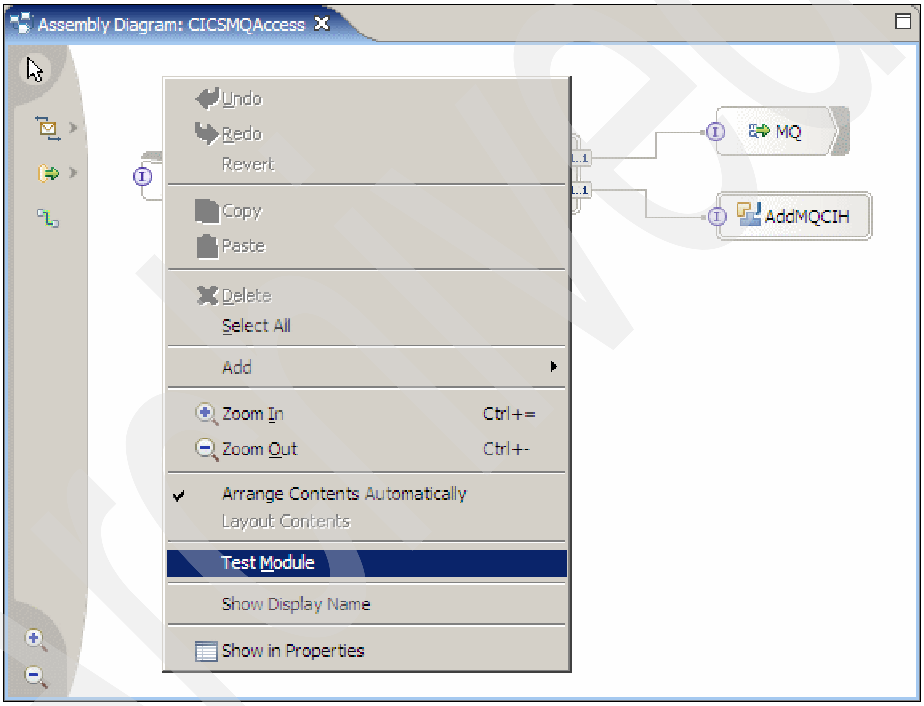


Figure 8-62 Test module

A CICSMQAccess_Test window opens.

We can use the Integrated test client to test all the components of a mediation module. As we have many components in the mediation module, we have to select the component that we want to test. In our case, we want to test the mediation flow, called CICSMediation.

Attention: If you are using a standalone WebSphere Enterprise Service Bus runtime you need to install the correct Integration Test Client update:

`http://www-1.ibm.com/support/docview.wss?rs=2308&context=SSQQFK&dc=D400&uid=swg24010662&loc=en_US&cs=UTF-8&lang=en`

Tip: You can also right-click the WS export and test that using the Test Module tool, or you can open the Web Services explorer and test the SOAP/http service export by locating the WSDL file on the server or in WebSphere Integration Developer.

6. On the CICSMQAccess_Test window make sure to select the **CICSMediation** component using the drop-down list, as shown in Figure 8-63.

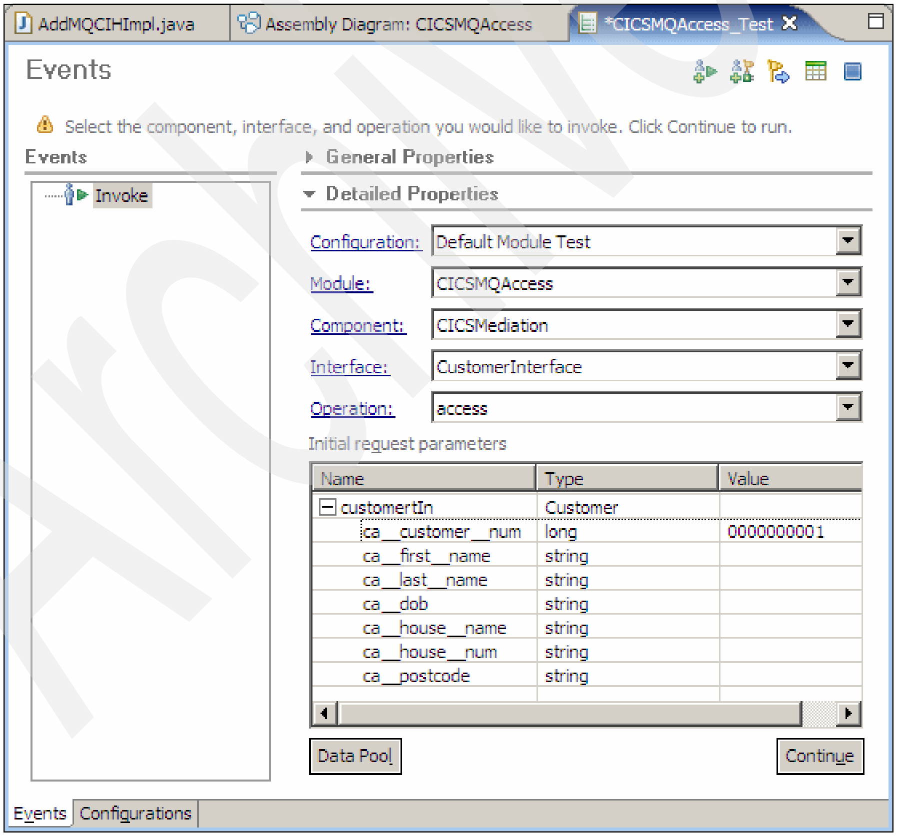


Figure 8-63 Integrated test client request

7. Enter adequate values for your test case. In our case, we provide a 10-digit number in front of the ca__customer__num and click **Continue**.

> **Tip:** If you like to enter realistic values for the CustomerIn structure, right-click **CustomerIn** and select **Add value to pool** → **OK**. The next time that you run the test right-click **CustomerIn**, select **Use Value from Pool**, and select **CustomerIn** → **OK**.

CICS/MQ Bridge test

At this point, the test starts. A WebSphere MQ message is built by the ESB runtime and sent onto the destination queue where the CICS bridge is listening. The answer from the bridge is put on the reply queue specified on the message itself. The name of the reply queue is configured in the MQ Import binding, and overrides the reply queue originally set in the request message. (See the properties of the MQ Import, under the Binding section and Message configuration, as shown in Figure 8-64.)

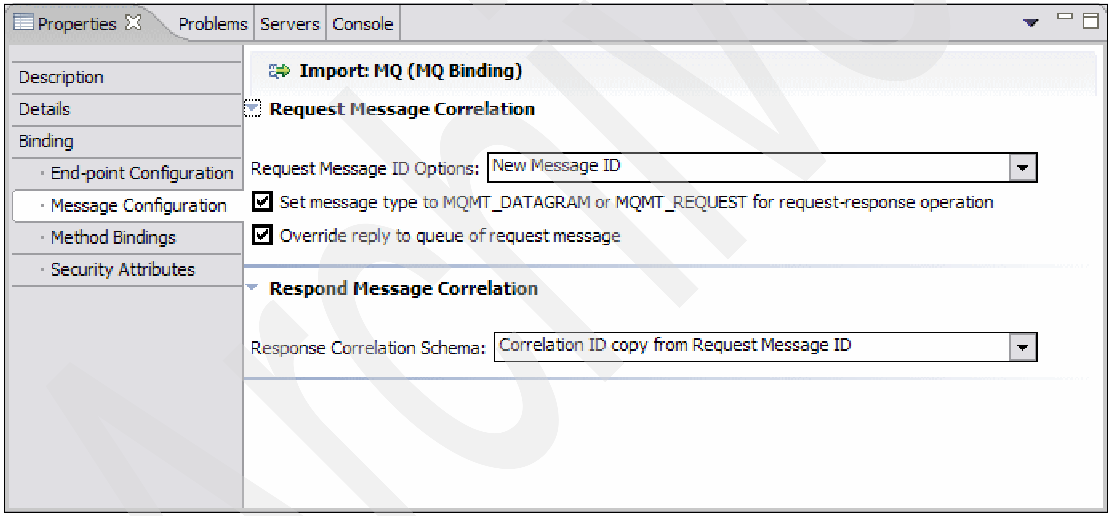


Figure 8-64 MQ binding message configuration

8. The answer from the CICS system is displayed on the Integrated Test client, as shown in Figure 8-65.

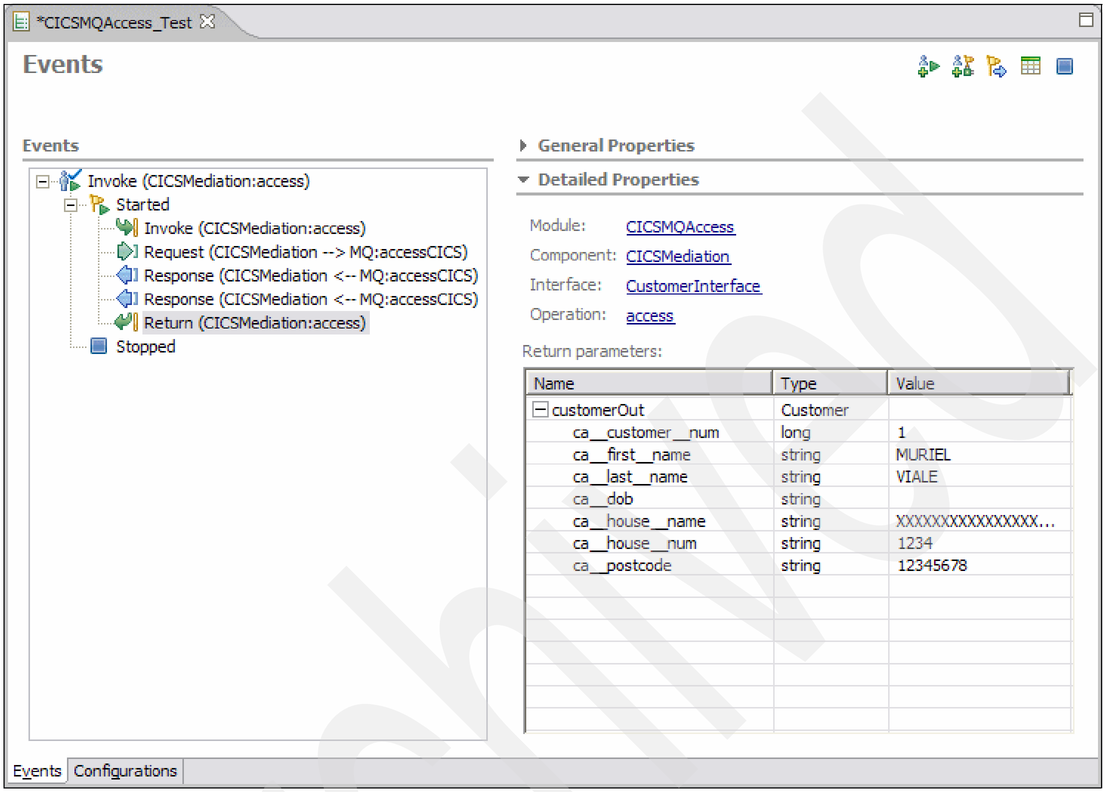


Figure 8-65 Integrated test client response

MQCICSServer scaffolding test

The full listing of the MQCICSServer.java program is in Example 8-6 on page 295. The program replies to just one message at a time.

1. In the Physical Resources view right-click **MQCICSServer.java** and select **Run** → **Java Application** (Figure 8-66 on page 294).

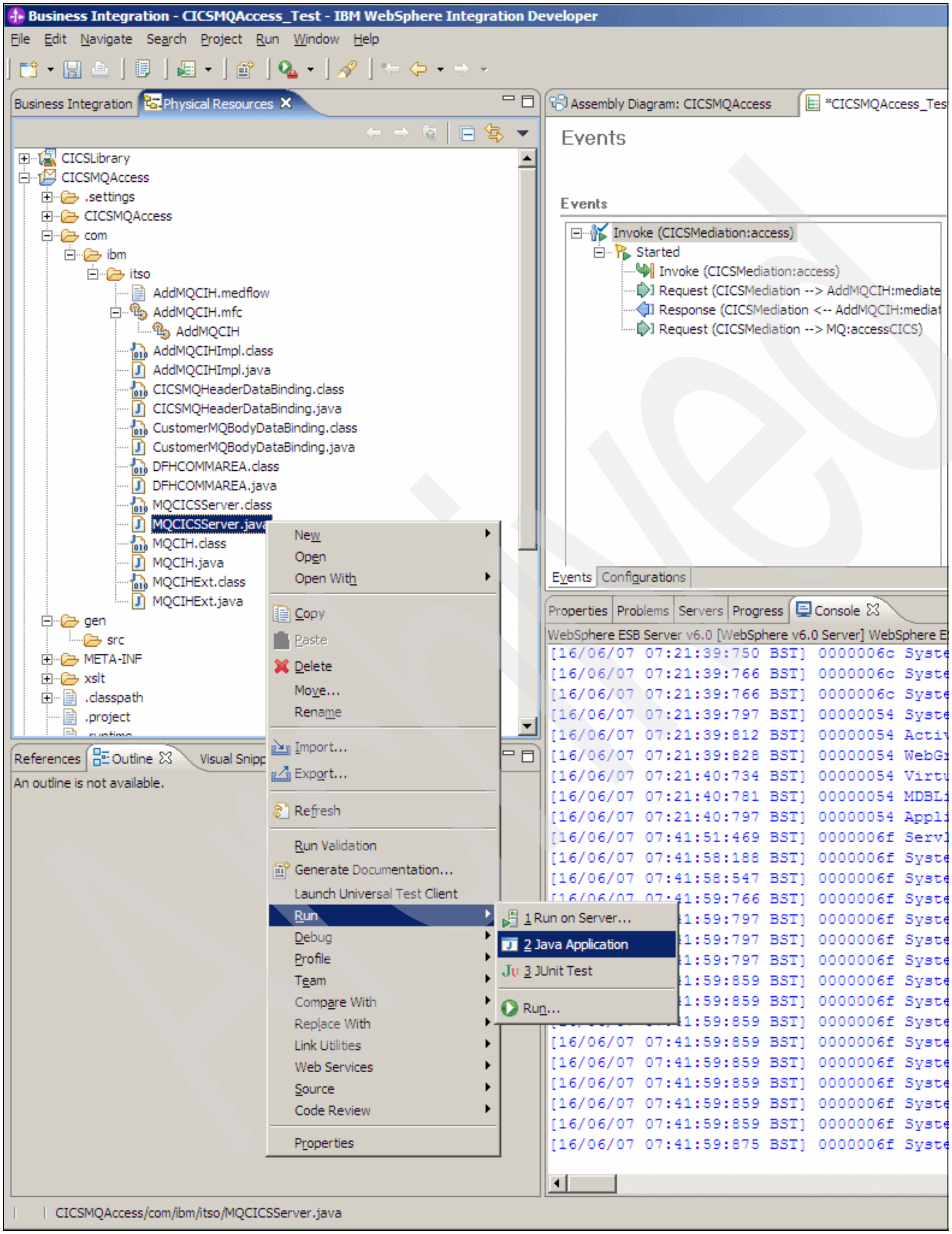


Figure 8-66 Run the MQCICSServer Java application

2. The console trace will then pick up from “CICSMQHeaderDataBinding.write() finished” and complete the response processing. The results are shown in Figure 8-55 on page 275.

> **Tip:** The response fails while you are developing, and the message listener closes down. What do you do then?
>
> If you have bugs during development, and the response message is not consumed, first WebSphere Enterprise Service Bus exhausts the retry count for the queue, and then it shuts down the message listener. Short of recycling the server, how can you get up and running again?
>
> 1. Firstly, using the WebSphere MQ Explorer, or the RFHUTIL program, clear the queue.
> 2. Open the Administrative console and select **Filter Administrative tasks** → **All**.
> 3. Click **Servers** → **Application servers** → **server1** → **Communications** → **Messaging** → **Message Listener Service** → **Listener ports**. Put a check next to CICSMQAccess.MQ_MQIMPORT_LP and click **Start** (Figure 8-67).

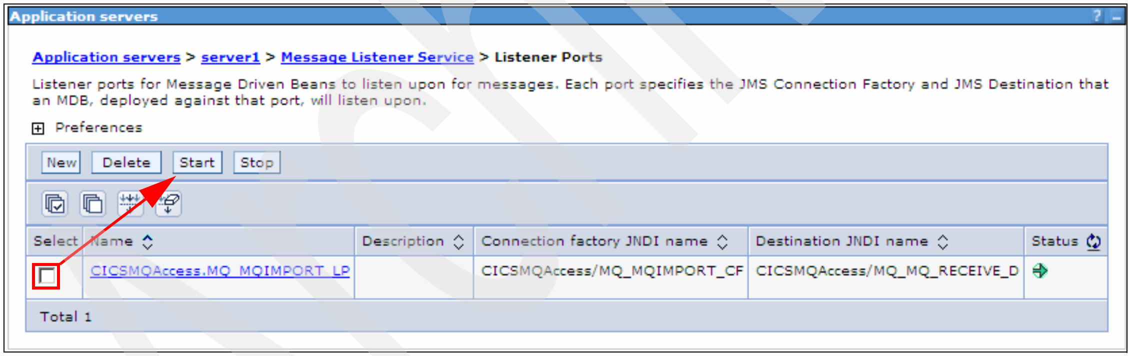


Figure 8-67 Restarting the message listener port

Example 8-6 MQCICSServer.java

```
package com.ibm.itso;

import com.ibm.mq.MQC;
import com.ibm.mq.MQGetMessageOptions;
import com.ibm.mq.MQMessage;
import com.ibm.mq.MQPutMessageOptions;
import com.ibm.mq.MQQueue;
```

```
import com.ibm.mq.MQQueueManager;
import com.ibm.ws.sib.comms.mq.util.MQConstants;

public class MQCICSServer {
   static final private String QMGR = "QM1";
   static final private String RECEIVEQ = "TO.CICS";
   static final private String SENDQ = "FROM.CICS";
   private MQQueueManager qMgr;
   private MQQueue sendQ;
   private MQQueue receiveQ;
   private MQGetMessageOptions gmo;
   private MQPutMessageOptions pmo;
   private MQMessage message;

   public static void main(String[] args) {
      System.out.println("Starting MQCICSServer");
      MQCICSServer mqCICSServer = new MQCICSServer();
      mqCICSServer.process();
      mqCICSServer.terminate();
      System.out.println("Stopping MQCICSServer");
   }
   private void terminate() {
      try {
         sendQ.close();
         receiveQ.close();
         qMgr.disconnect();
      } catch (Exception e) {
         e.printStackTrace();
      }
   }
   private void process() {
      try {
         receiveQ.get(message, gmo);
         System.out.println("Processing message, length is "
               + message.getTotalMessageLength());
         message.correlationId = message.messageId;
         message.encoding = 785;
         message.characterSet = 500;
         message.format = MQConstants.MQFMT_CICS;
         message.messageType = MQC.MQMT_REPLY;
         sendQ.put(message, pmo);
      } catch (Exception e) {
         e.printStackTrace();
         this.terminate();
      }
```

```
    }

    public MQCICSServer() {
       try {
          qMgr = new MQQueueManager(QMGR);
          receiveQ = qMgr.accessQueue(RECEIVEQ, MQC.MQOO_INPUT_AS_Q_DEF);
          sendQ = qMgr.accessQueue(SENDQ, MQC.MQOO_OUTPUT);
          gmo = new MQGetMessageOptions();
          gmo.options = MQC.MQGMO_WAIT + MQC.MQGMO_FAIL_IF_QUIESCING;
          gmo.waitInterval = MQC.MQWI_UNLIMITED;
          pmo = new MQPutMessageOptions();
          pmo.options = MQC.MQPMO_FAIL_IF_QUIESCING + MQC.MQPMO_NEW_MSG_ID;
          message = new MQMessage();
       } catch (Exception e) {
          e.printStackTrace();
       }
    }
}
```

8.8 Conclusion

We have explained how to use the WebSphere MQ binding of WebSphere Enterprise Service Bus in conjunction with the WebSphere JCA adapter tooling and the CICS-MQ Bridge to build a customized mediation module to route from an SOAP/HTTP request into a WebSphere MQ transport and to transform SOAP XML syntax into CICS COBOL syntax and vice versa.

We have shown the flexibility of the WebSphere MQ binding, the use of SCA API to manipulate a service data object, and the mediation capabilities of WebSphere Enterprise Service Bus to build a robust scalable solution to link SOAP or Web clients to CICS applications in a service-oriented architecture.

The same technique is also applicable to the IMS Transactional System, as we have a WebSphere Adapter and the same JCA tooling for IMS shipped with WebSphere Enterprise Service Bus and WebSphere Integration Developer. You can also use the same technique for any application that has a COBOL copybook interface definition and is accessible over WebSphere MQ.

9

Code-free CICS integration using WebSphere MQ

In Chapter 8, “Custom CICS integration using WebSphere MQ” on page 215, we implemented 6.1, “Connection pattern 1: integration” on page 144 to solve the business problem described in 5.1.1, “Scenario 1: claimants’ access to claims information” on page 131 for the business scenario described in 5.1, “Case 1: insurance claims and call center” on page 131. In this chapter we described how to automate much of the implementation by developing an WebSphere Integration Developer plug-in and custom mediation to perform code generation for the data binding and header data binding classes.

The WebSphere MQ-CICS Bridge Plug-in is developed and supported by the Solution Analysis Laboratory in the IBM Hursley Laboratories, England, and is available for download from:

http://www-1.ibm.com/support/docview.wss?rs=171&uid=swg27009734#1

Search for BA78: IBM WebSphere Integration Developer - WebSphere MQ-CICS Bridge Plug-in.

Our primary objective for this chapter is to describe how to use the plug-in to connect to CICS transactions. This is an excellent model for you to follow should you wish to build your own plug-in, modelled on this one, to connect to IMS (for example, using the MQ-IMS bridge).

You can use the same pattern to automate connecting to other applications that have an asynchronous message or document-driven interface. The interface needs to separate application connectivity from routing requests to and from specific application functions. The CICS/MQ bridge provides this separation, as, for example, the IDOC architecture does for SAP.

An alternative pattern to connect legacy applications is 6.5, “Connection pattern 5: custom application integration” on page 160, which involves developing a new Custom WebSphere Adapter. The pro’s and con’s of the alternative approaches are discussed in Chapter 6, “Connection patterns” on page 143. There is no straightforward decision criteria: the decision is as much influenced by business factors, such as your skill base and where responsibility for providing integration components lies, as it is by technical considerations. If you have a hammer, look for a nail to join things together. If you have a screwdriver then use a screw.

To develop the working example we use the same CICS application as in the last chapter. There is no need for the WebSphere CICS JCA adapter tooling we used in the previous example.

9.1 Why use WebSphere MQ-CICS Bridge Plug-in

From Chapter 8, “Custom CICS integration using WebSphere MQ” on page 215, we learned that in order to access a CICS application through WebSphere MQ-CICS bridge, there are three essential elements that are specific to this kind of adapter. They are:

- A body binding class for the commarea-equivalent business object
- WebSphere MQ header information to invoke the CICS application
- MQCIH header binding class

An integration specialist needs to implement these elements using WebSphere Integration Developer. It would be useful to speed up the development effort, either to make the integration specialist more productive or to split the task between an integration specialist, who develops the integration framework, and more solutions or applications oriented developers who configure the integration components with application-specific information by following the standardized integration pattern.

Developing a new mediation for each application function call or transaction in the manner of Chapter 8, “Custom CICS integration using WebSphere MQ” on page 215, can be time consuming because:

- The developer has to examine each element of the schema that is generated from the commarea copybook in order to construct the data binding class. This is a lengthy task. More important, the time spent grows with the scale and complexity of the schema.
- Hand-crafting data binding code can be error-prone. As we saw, there are a lot of fields to be copied, and the business object has to be developed with the correct structure. One would soon be looking for an algorithm to map Java object attributes to business object attributes to eliminate all the hard coded field names in getXXXX and setXXXX methods.
- Comprehensive testing is needed in order to have confidence in the code. If there are variations in the message structure to be accommodated, these need to be tested for every transaction. If the mappings were automated, one could rely on the pattern followed by the code generator.
- Although the code to implement WebSphere MQ header information and header binding class is relatively static, the user has to do the task for every CICS application, and time spent is multiplied with the number of CICS transactions, or other application functions, to be accessed.

For a company that has a number of transactions to access through WebSphere Enterprise Service Bus or WebSphere Process Server, the project manager should assess building each mediation by hand, against developing a tool to automate the task and, in the longer run, saving a lot of development time and resource, and probably leading to a more robust solution.

If the requirement is to develop mediations to work with the CICS-MQ bridge, then the plug-in described in this chapter is available and supported by IBM, and the pattern it follows can be replicated to implement other plug-ins.

The WebSphere MQ-CICS Bridge Plug-in is an extension to WebSphere Integration Developer v6.0.2. It automates the implementation of data binding class, provides a mediation primitive to add WebSphere MQ header information, and provides a corresponding MQCIH header binding class to configure the WebSphere MQ binding node. The user configures the mediation using the plug-in as one of its components, and does not need to write any Java code to move header and body data between the WebSphere MQ message and the service component implementation.

9.1.1 Time savings you can expect from using the tool

Time savings is not the only (nor perhaps the major) consideration when using a tool to generate mediation components instead of crafting them by hand. By separating the task of coding the specialist integration components from the task of assembling components to develop a specific integration solution, the solution should also be more maintainable and reliable.

Unless there is already a tool to generate the mediation components available, such as the WebSphere CICS-MQ Bridge Plug-in we describe in this chapter, you need to develop one. How much effort does that take? What guidance can we give a project manager as to the trade-offs between building a mediation by hand compared with developing a WebSphere Integration Developer plug-in or a WebSphere Adapter to automate the task?

The WebSphere MQ-CICS Bridge Plug-in tool is already available from IBM at no charge. Using the tool is no more difficult than using any other supplied mediation flow component. With the help of the documentation in this book, getting familiar with the tool for the first time probably takes about a day. From then on, using the tool probably takes about thirty minutes out of the time spent developing a mediation flow

Developing the same mediation by hand took our resident, an integration specialist with some knowledge of WebSphere Enterprise Service Bus and CICS, about four weeks. With the experience gained, next time it would probably take between one and two days for them to develop and thoroughly test new mediations for CICS by hand. So the times saving between using the plug-in versus coding by hand is probably of the order of 20:1.

You may be interested in knowing how long it took us to develop this tool. It took about three or four developers, who were not experienced in plug-in or custom mediation development before starting the project, about six months to get the tool to a release standard. Next time the development effort would be much less.

9.2 WebSphere MQ-CICS Bridge Plug-in

In the following sections we describe how to use the WebSphere MQ-CICS Bridge plug-in.

First, make the WebSphere MQ-CICS Bridge Plug-in available to WebSphere Integration Developer v6.0.2. The plug-in does not work in earlier versions of WebSphere Integration Developer due to MQ binding only being available from v6.0.2.

To obtain the WebSphere MQ-CICS Bridge Plug-in, download the plug-in zip file from:

http://www-1.ibm.com/support/docview.wss?rs=171&uid=swg27009734#1

To install the plug-in from the zip file:

1. Unzip the downloaded zip file to $WID_INSTALL_ROOT/wstools/eclipse.
2. Start WebSphere Integration Developer with the -clean option in a command line window:

   ```
   $WID_INSTALL_ROOT/wid.exe -clean
   ```

To confirm that the plug-in is available in WebSphere Integration Developer for us to use, click **Help** → **About IBM WebSphere Integration Developer** → **Feature Details**. In the Feature id column, look for com.ibm.sal.mqcics. If it exists, the WebSphere MQ-CICS Bridge Plug-in is installed and available for use.

There is a readme file that lists the limitations of the plug-in in addition prerequisites and installation information. It is worth checking this file for updated information.

There is a cheat sheet available to guide you through the tooling after WebSphere MQ-CICS Bridge Plug-in is installed. To access the cheat sheet, click **Help** → **Cheat sheets** → **Enterprise Integration** → **Access CICS applications through MQ-CICS bridge** → **OK** (Figure 9-1).

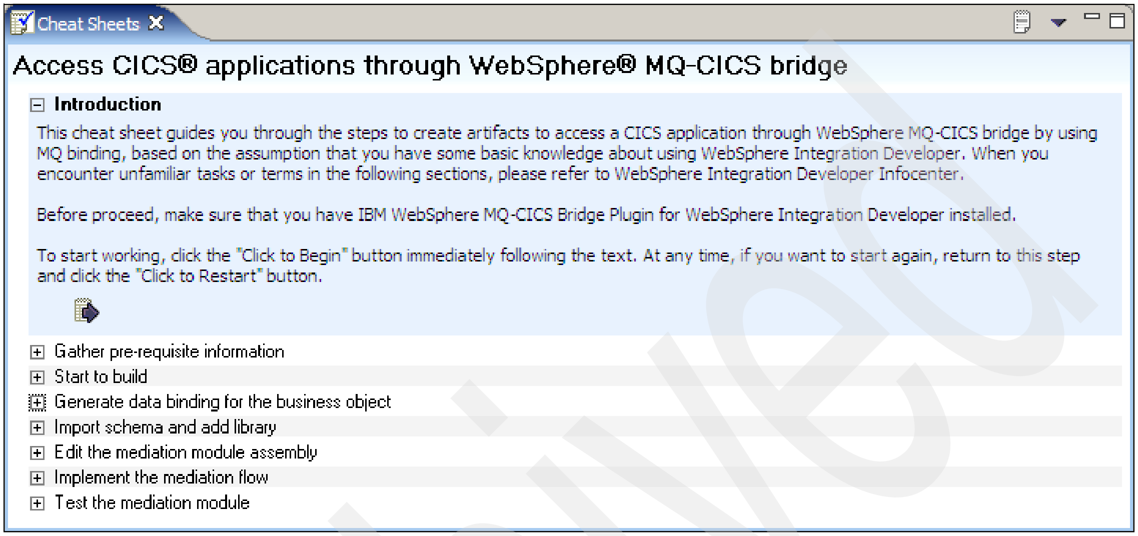


Figure 9-1 WebSphere MQ-CICS Cheat Sheet

9.3 Put into context

We base this scenario on the same scenario as the one described in Chapter 8, "Custom CICS integration using WebSphere MQ" on page 215. That is, we need to build a mediation flow that has the interfaces and elements shown in Figure 9-2.

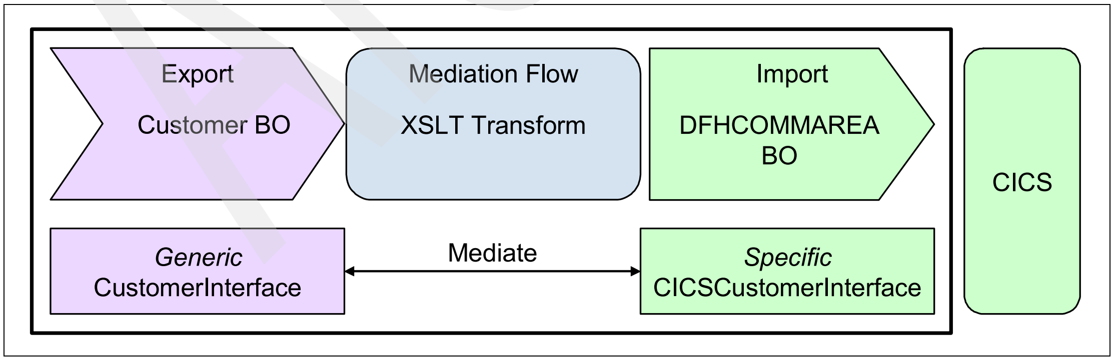


Figure 9-2 Business objects and interfaces of the mediation module

Figure 9-3 shows us the tasks we need to perform to configure the components of the solution.

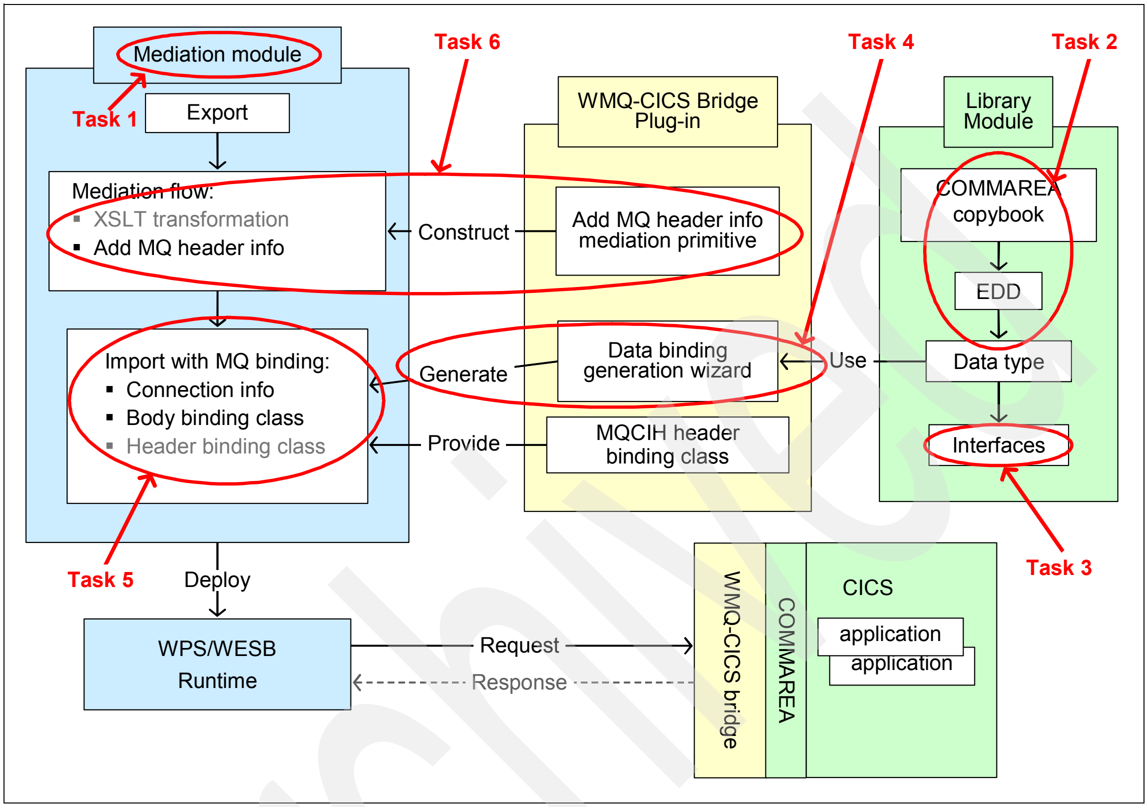


Figure 9-3 Interaction between WebSphere MQ-CICS Bridge Plug-in and modules

These tasks are explained in the following sections:

- Task 1 - create mediation module on page 306.
- Task 2 - define business objects on page 306.
- Task 3 - create interfaces on page 308.
- Task 4 - generate data binding class on page 309.
- Task 5 - configure assembly diagram on page 311.
- Task 6 - add WebSphere MQ header information on page 317.

9.4 Task 1 - create mediation module

To build the example, we create a new mediation module called CICSMQPlugin:

1. Click **File** → **New** → **Mediation module**. Name it `CICSMQPlugin` and uncheck Create mediation flow component (Figure 9-4). Click **Next**.

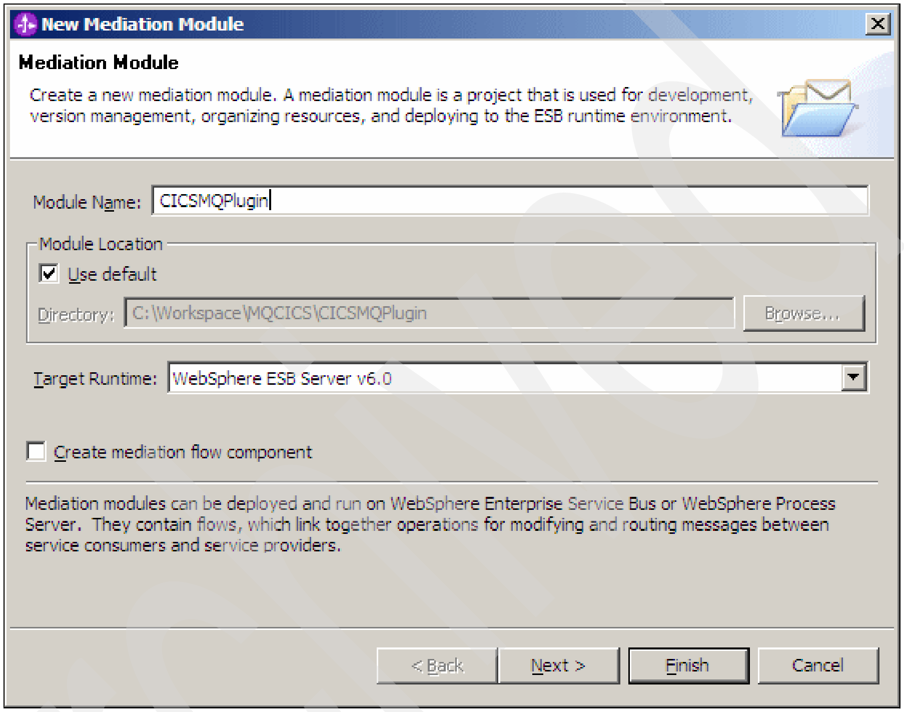


Figure 9-4 Create CICSMQPlugin

2. Check **CICSLibrary**[1] and click **Finish**.

9.5 Task 2 - define business objects

Before starting to build the solution you need to collect information from the CICS application owner and the WebSphere MQ administrator, which will be used to define the business objects used in the mediation.

[1] To create CICSLibrary, follow the instructions in 8.3, “Mediation module design” on page 219.

From the CICS application owner:

- CICS application name
- Corresponding COMMAREA COBOL copybook or C structure that is specific to the interface to be accessed

From the WebSphere MQ administrator:

1. The request queue manager name for the CICS application
2. The send destination queue for the request message
3. The receive destination queue for the response message if the mediation flow expects a reply
4. The host name for the queue manager, port number, and server channel name
5. The CCSID (coded character-set ID) and encoding for the operating system that the CICS application runs
6. The format of the COMMAREA COBOL copybook

9.5.1 Define CICS-specific business object

To generate the CICS-specific business object from the commarea COBOL copybook, we use the Enterprise Data Discovery tool. We follow the same steps as in 8.3.1, “Application-specific DFHCOMMAREA business object” on page 221.

If you are reusing the CICSLibrary created in Chapter 8, “Custom CICS integration using WebSphere MQ” on page 215, you do not need to do anything else here.

After we have done that, our project tree should look like Figure 9-5.

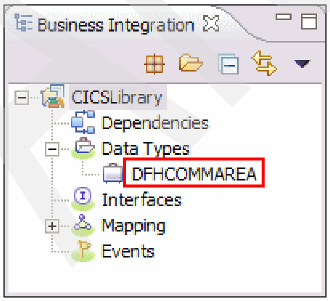


Figure 9-5 The CICS-specific business object appearing in the module

9.5.2 Define generic business object

We follow the same steps as in 8.3.2, “Generic customer business object” on page 225, to define the generic business object named customer that is based on the CICS-specific business object DFHCOMMAREA.

Again, there do nothing if you are reusing the same CICSLibrary module.

9.6 Task 3 - create interfaces

We follow the same steps as in 8.4, “CICS interfaces” on page 228, to create two interfaces, named CICSCustomerInterface and CustomerInterface. CICSCustomerInterface serves as the import interface to the CICS application, and CustomerInterface serves as the export interface that the to-be-created mediation module exposes.

Again, do nothing if you are reusing the same CICSLibrary module.

9.7 Task 4 - generate data binding class

After the CICS-specific business object (in our case, DFHCOMMAREA) is defined, we can generate the custom body data binding class for it with the use of the WebSphere MQ-CICS Bridge Plug-in.

1. In the Business Integration perspective, expand **Data Types** → **DFHCOMMAREA** → **Generate Data Binding class** → **MQ-CICS**. This menu is provided by the WebSphere MQ-CICS Bridge Plug-in (Figure 9-6).

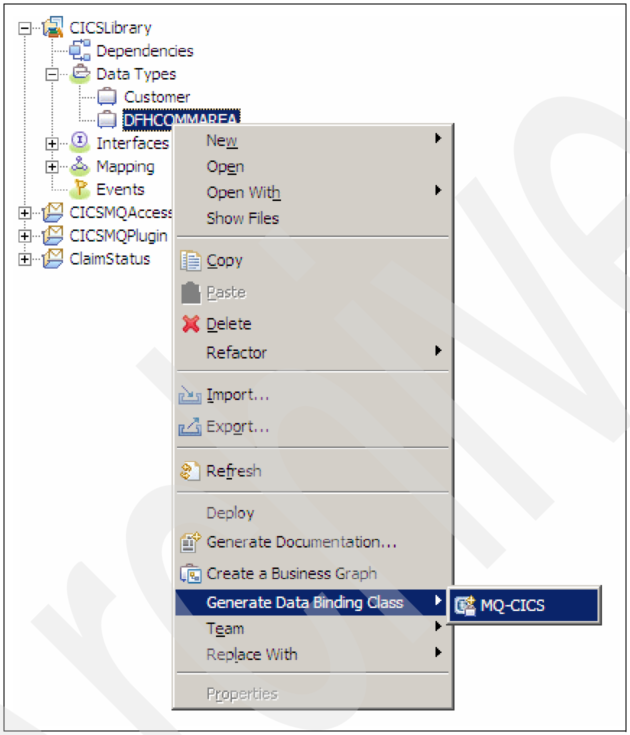


Figure 9-6 Access the data binding generation tool

2. A window is opened (Figure 9-7). Choose a module from the drop-down list. Optionally, change the name for the data binding class. Click **Finish**.

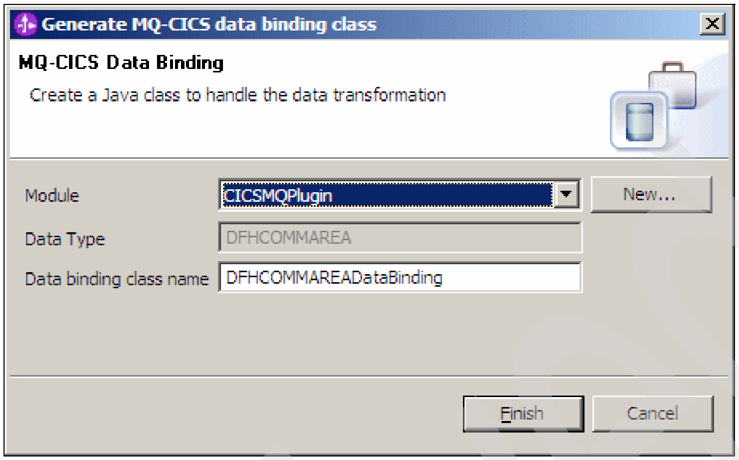


Figure 9-7 GUI of data binding generation tool

3. Right-click the **MQCICSPlugin** module and click **Refresh**.

> **Tip:** You must click **Refresh** to see the files, and again later for the import to work.

4. Right-click the **MQCICSPlugin** module and select **Show Files**. Respond **OK** if you get a window titled Physical Resources View.

   A file named DFHCOMMAREADataBinding.java and corresponding class DFHCOMMAREADataBinding.class appear in the Physical Resources View.

Look at the generated code and compare that with the custom data binding classes in Figure 8-41 on page 254 and Figure 8-42 on page 255[2]. The generated data binding class is more thorough, having trace and logging code, and testing return codes.

> **Tip:** If your CICS application has different input and output copybook structure, perform the steps in 9.5.1, “Define CICS-specific business object” on page 307, for input and output structure separately. You will then get two data binding classes, one for input business object and one for output business object.

[2] If you are curious to see the generated code, and are not doing these examples in WebSphere Integration Developer, then you will find a .zip file in Appendix A, “Additional material” on page 499, which contains all the code and projects.

We use this class to configure the MQ binding Import node later.

9.8 Task 5 - configure assembly diagram

Next we need to build essentially the same assembly diagram as we did in 8.5, “Mediation module” on page 231, implement the mediation flow, and add the exports and imports.

9.8.1 Create assembly diagram

Return to the Business Integration view.

1. Double-click **CICSMQPlugin** and select **Assembly Diagram** to open the CICSMQPlugin assembly diagram in the editor.
2. Expand **CICSLibrary** module, expand the **Interfaces** folder, and drop the CustomerInterface into the assembly diagram editor.
3. In the pop-up Component Creation window (Figure 9-8) select **Export with Web Service Binding** → **OK**.

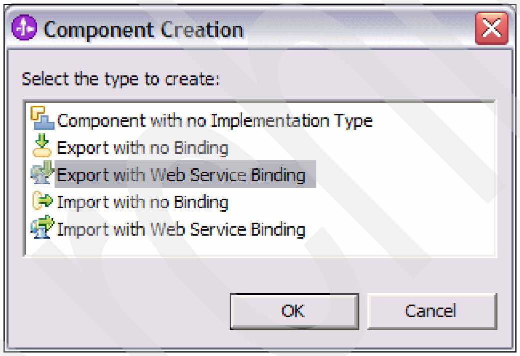


Figure 9-8 Select type in the Component Creation window

4. In the pop-up Transport Selection window (Figure 9-9) select **soap/http**.

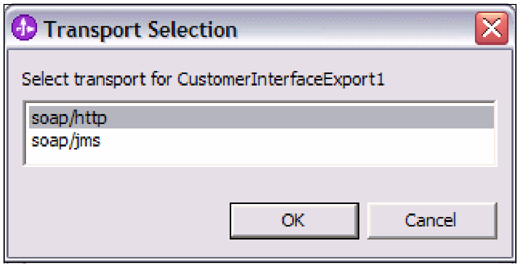


Figure 9-9 Select transport type

An export named CustomerInterfaceExport1 is placed in the assembly diagram.

5. Drag and drop **CICSCustomerInterface** into the assembly diagram editor.
6. In the pop-up Component Creation window (Figure 9-8 on page 311) select **Import with no Binding** → **OK**.

   An import named CICSCustomerInterfaceImport1 is placed in the assembly diagram.
7. Drop a new mediation from the assembly diagram palette onto the editor and name it CICSMediation. Drop the CustomerInterface onto the mediation.
8. Right-click **CICSMediation** and click **Add Reference** → **CICSCustomerInterface** → **OK**.
9. Right-click **CICSMediation** in the assembly diagram and select **Wire to existing** → **File** → **Save All**.

   This connects the CICSMediation flow to the CICSCustomerInterfaceImport1 and CustomerInterfaceExport1 components (Figure 9-10).

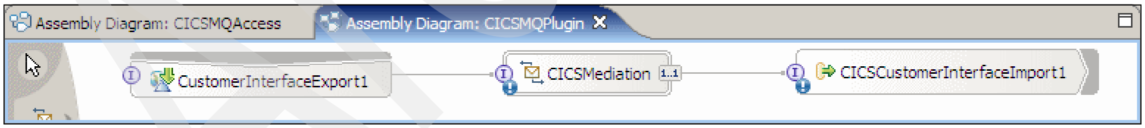


Figure 9-10 CICSMQPlugin mediation module

10. Right-click **CICSCustomerInterfaceImport1** and select **Generate Binding** → **Messaging Binding** → **MQ Binding**. A window titled MQ Import Binding is opened.
11. In the section End-point Configuration, fill in details obtained in 9.5, “Task 2 - define business objects” on page 306.

12. In the Data format section select **Request Serialization type** → **User Supplied** → **Browse** → **Select type** → **DFHCOMMAREDataBinding** (Figure 9-11) → **OK**.

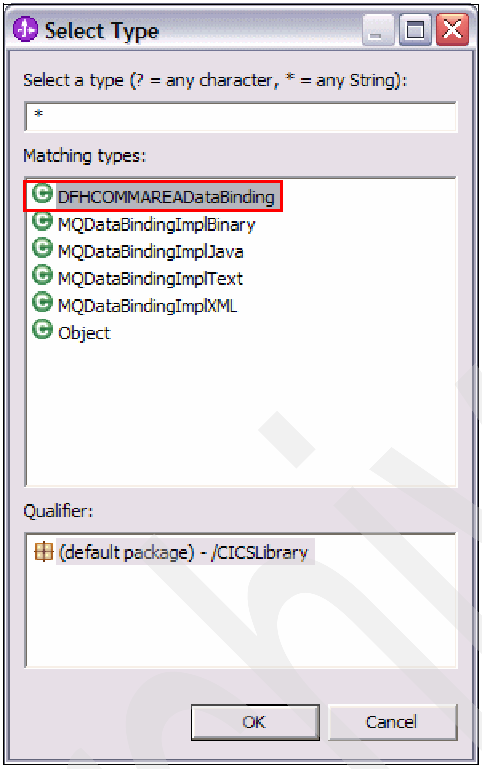


Figure 9-11 Select a data binding class

13. Repeat for response serialization type.

We have now coupled the generated data binding class to the import component. Figure 9-12 shows the details.

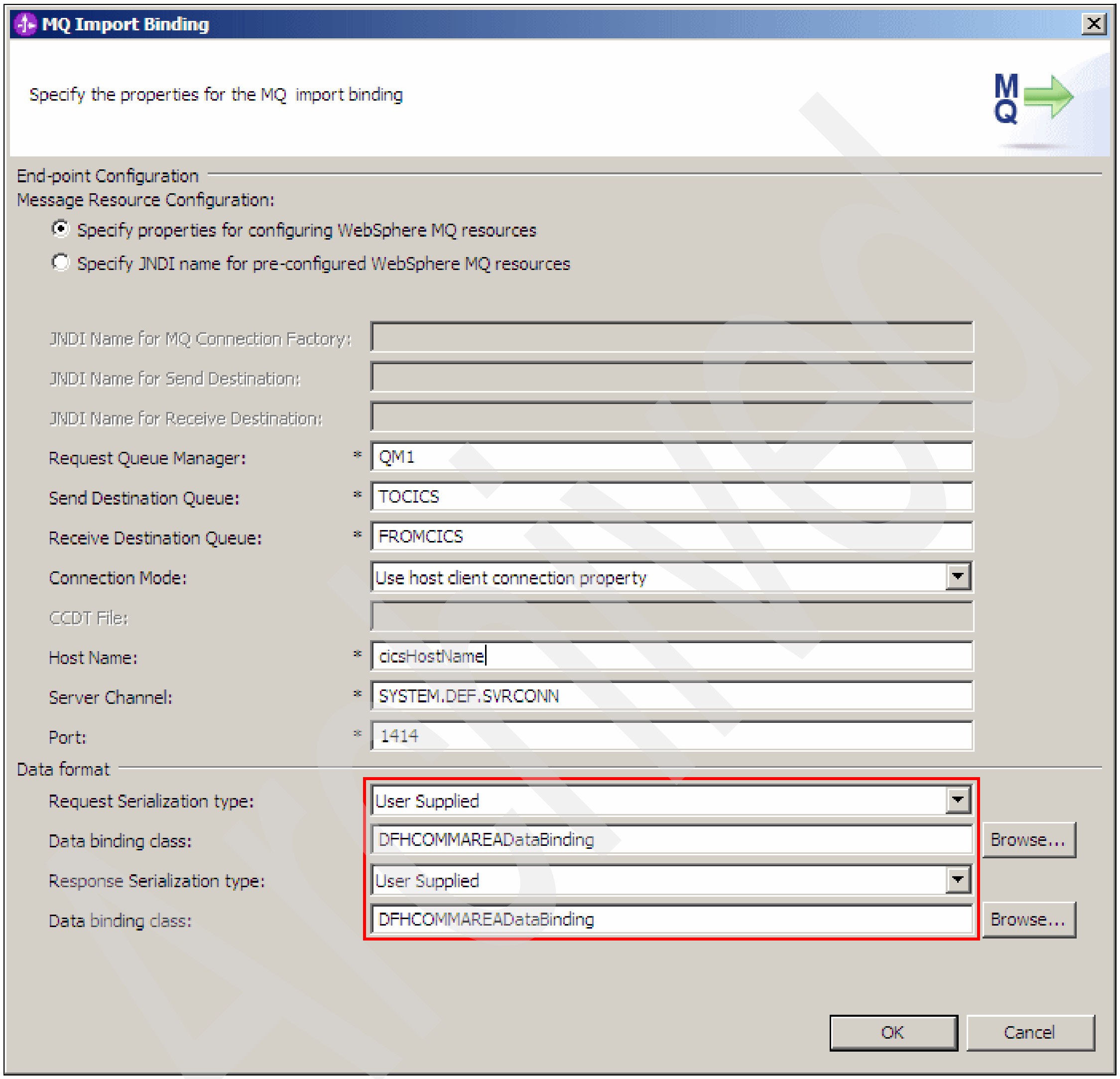


Figure 9-12 MQ import binding details

We have finished editing the assembly diagram (Figure 9-13).

The next few steps are to implement the CICSMediation mediation flow component.

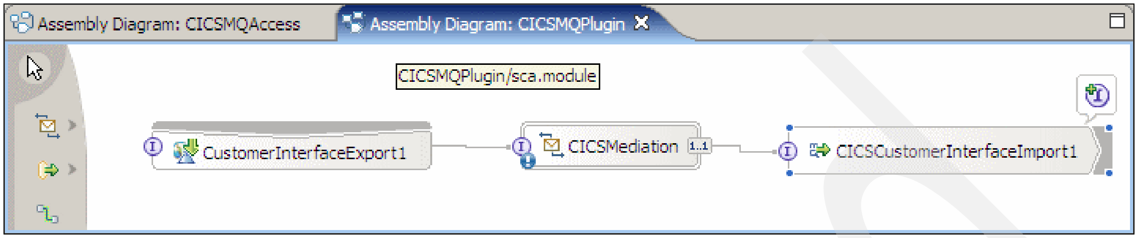


Figure 9-13 CICSMQAccess mediation module assembly diagram

9.8.2 Create XSLT transformation

You can follow the instructions in Mediation flow implementation on page 234 to create XSLT transformations from CustomerInterface to CICSCustomerInterface and vice versa in response flow. You will need to disconnect from the XSLT to the destination in the request flow after creating the XSLTs so that we can add another mediation primitive to create the MQCIH.

If you have already created the XSLTs in the CICSMQAccess project, copy them from the Mapping → XML maps folder to the corresponding folder in the CICSMQPlugin folder. Then in the Details tab for each XSLT click **Browse → Mapping File Selection** (Figure 9-14) and select the corresponding map.

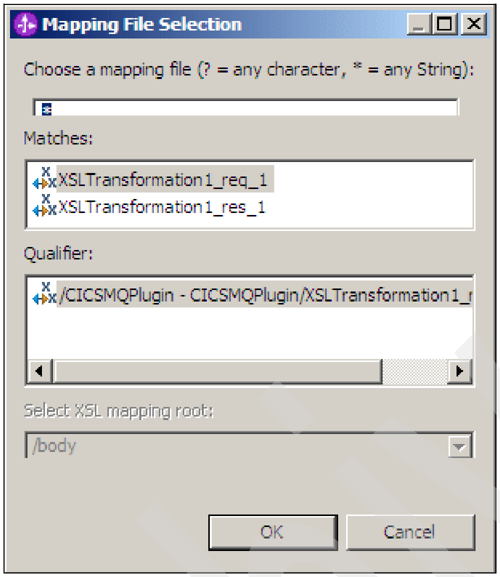


Figure 9-14 Mapping file selection

Alternatively, rather than linking the flow all the way through (Figure 9-15), you can specify in a panel what the target data type of the request XSLT is using a panel as explained below.

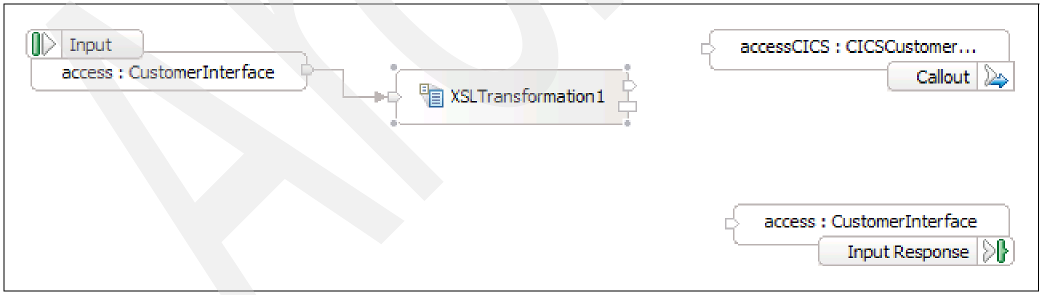


Figure 9-15 Mediation request flow with XSLT transformation not connected

In the Terminal tab of XSLTTransformation1 select the message type for the output of XSLTTransformation1 to the one that accessCICS expects. This is illustrated in Figure 9-16.

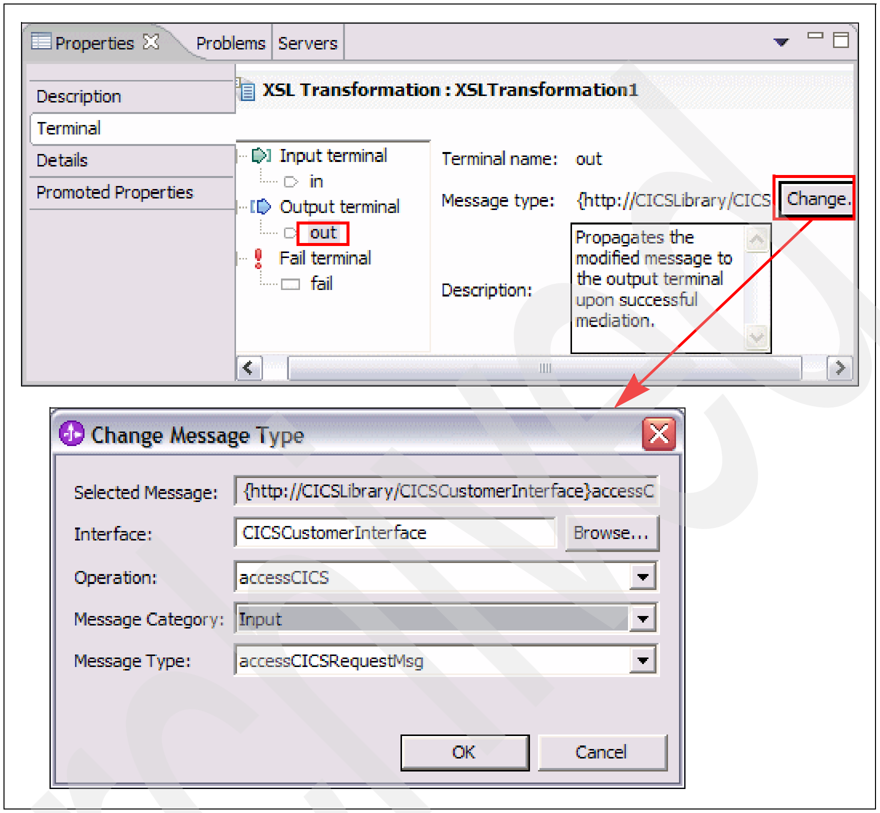


Figure 9-16 Specify terminal type for XSLT transformation

9.9 Task 6 - add WebSphere MQ header information

In order for the request message to able to be picked up by WebSphere MQ-CICS bridge and processed, certain WebSphere MQ header information has to present in the request message.

Figure 8-2 on page 218 illustrates what the WebSphere MQ message looks like with the presence of MQCIH header. But there is a case in which the user may want to send a WebSphere MQ message without a MQCIH header. Below we illustrate the development tasks for both cases using the WebSphere MQ-CICS Bridge Plug-in.

9.9.1 Add required resources to module

First of all we need to import two required resources to the CICSLibrary module:

1. Right-click **CICSLibrary** and select **Import** → **File system** → **Next**. Browse to $WID INSTALL ROOT/wstools/eclipse/plugins/com.ibm.sal.mqcics_6.0.2/schemas. Tick the box for CIHHeader.xsd and click **Finish**.

   This imports a schema that includes some MQCIH field elements and a program name element. Figure 9-17 and Figure 9-18 on page 319 show the windows for import and the resulted in module after that.

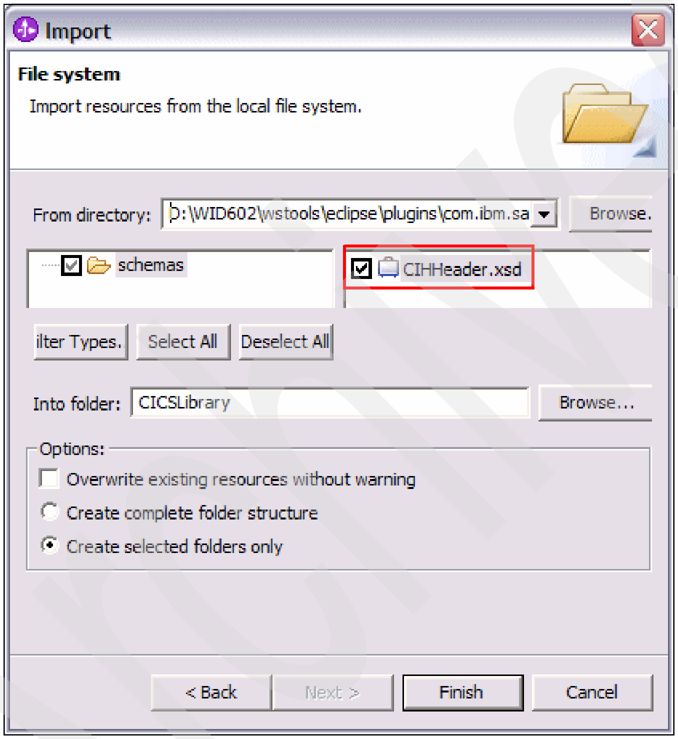


Figure 9-17 Import CIHHeader.xsd schema file

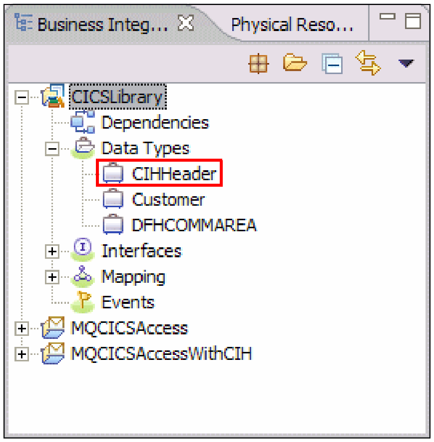


Figure 9-18 Module structure showing CIHHeader as a data type

2. Right-click the **CICSMQPlugin** mediation module and select **Properties** → **Java Build Path** → **Libraries** → **Add External Jars**. Browse to $WID INSTALL ROOT/wstools/eclipse/com.ibm.sal.mqcics_6.0.2, choose **mqcicsmediation.jar**, and click **Open** → **OK**.

Tip: You must have Java capabilities enabled to set the Java Build Path. Switch to the Java Perspective as a quick way to enable Java capabilities.

Note: mqcicsmediation.jar contains the data binding class for the MQCIH header and the implementation class for the mediation primitives this plug-in provides.

9.9.2 Configure MQImport with MQCIH header binding class

Now we connect the MQCIH header binding class to CICSCustomerInterfaceImport1 so that a SMO message with this kind of header structure can be translated to WebSphere MQ message format.

1. Open the CICSMQplugin assembly diagram. Click **CICSCustomerInterfaceImport1** → **Properties** → **Binding**. Locate the WebSphere MQ Header section (Figure 9-19).

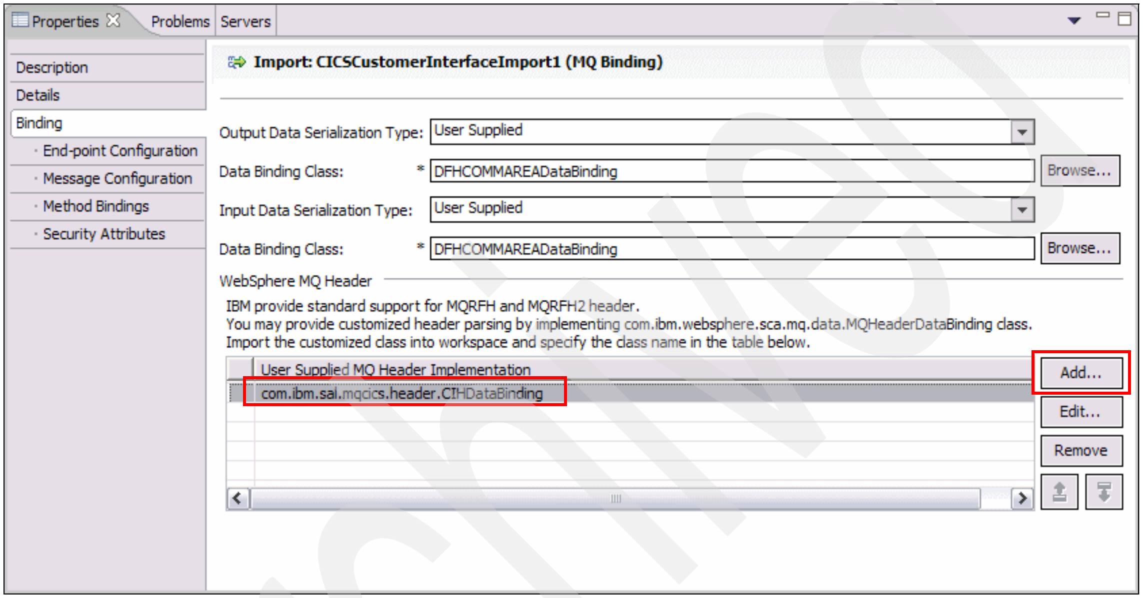


Figure 9-19 MQ import binding configuration with MQCIH header binding class specified

2. Click **Add** → **CIHDataBinding** → **OK** (Figure 9-20).

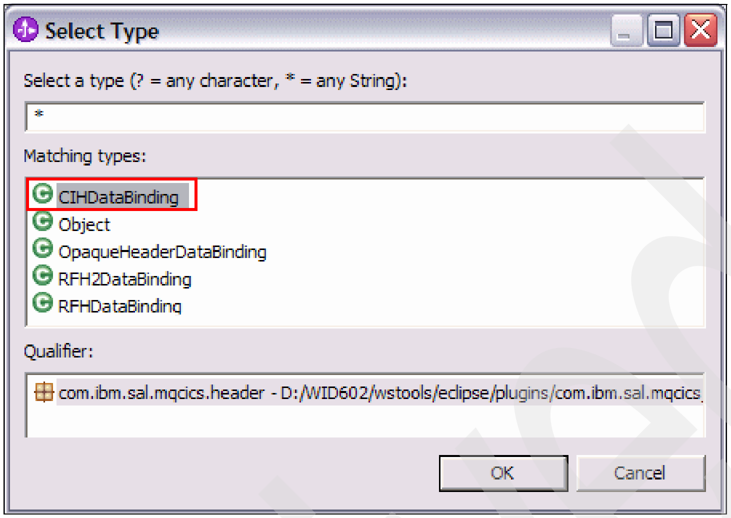


Figure 9-20 Select MQCIH binding class

Tip: The MQCIH header binding class needs to be configured even in the case when no MQCIH header is sent in the request message. This is because if any error occurs then MQ-CICS bridge returns a response message that has a MQCIH header to indicate the error codes.

9.9.3 For the case without MQCIH header

In the case without MQCIH header, the WebSphere MQ message structure will be MQMD followed by ProgramName+Commarea data, as shown in Figure 9-21.

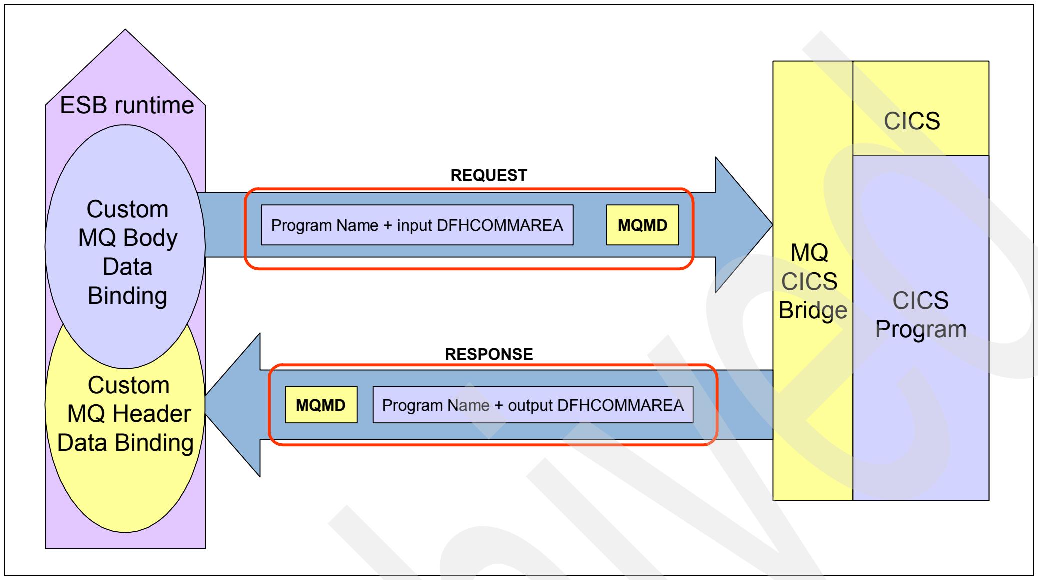


Figure 9-21 WebSphere MQ message structure for the case without MQCIH header

> **Tip:** Messages without an MQCIH header are shorter. You may gain some performance by not including an MQCIH header. However, without an MQCIH header, you lose the flexibility of setting the fields of MQCIH. Therefore, you can only invoke the CICS application in the default way. The next section shows you what these fields are.

To add related WebSphere MQ header information to the message, we use the addMQInfo primitive that is provided by the WebSphere MQ-CICS Bridge Plug-in:

1. Open the request flow of CICSMediation, and from the mediation flow editor palette (Figure 9-22), drag and drop the primitive named **AddMQInfo** to the canvas.
2. Connect the out terminal of XSLTTransform1 to the in terminal of AddMQInfo1, and the out terminal of AddMQInfo1 to accessCICS. The finished mediation flow should look like Figure 9-22.

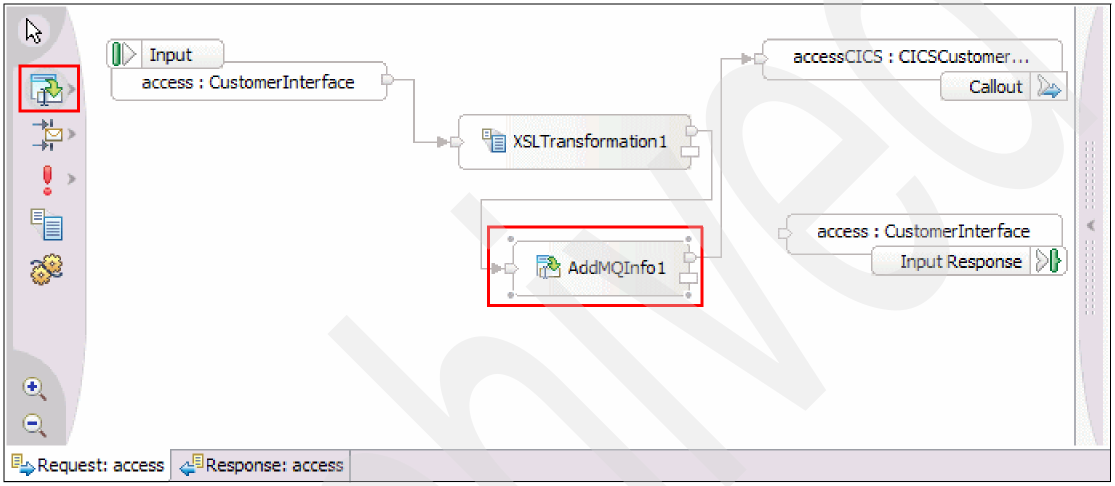


Figure 9-22 Mediation flow that has AddMQInfo mediation primitive

3. Click **AddMQInfo1** → **Properties** → **Details**. In the **Required** tab choose the **CCSID** and **Encoding** for the operating system at which the CICS application runs. Type in the name of the CICS application (LGICUS01 in our example) and choose **MQSTR** or overtype a value for `MQMD.Format`.

4. Uncheck the box beside “Add MQCIH header to the message” to leave the MQCIH header out of the message. Save and close the mediation flow. The completed Details page is shown in Figure 9-23.

Tip: The MQMD.Format field indicates the format of the data that follows MQMD.

Attention: Since we are not adding an MQCIH header to the message, there is no need for us to specify the fields on the MQCIH fields tab of the mediation primitive.

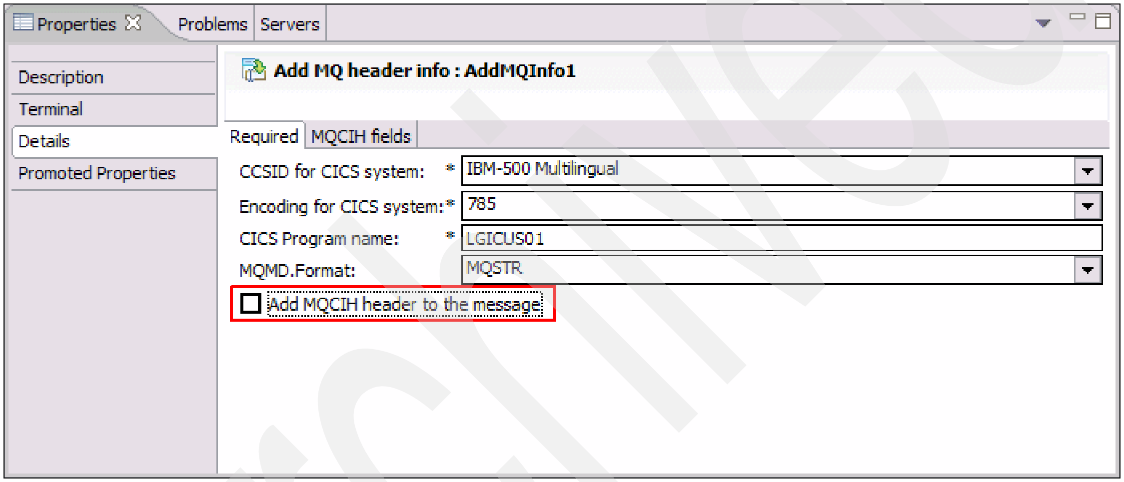


Figure 9-23 Properties details page of AddMQInfo mediation primitive for case without MQCIH header

We have finished developing the mediation module. The module can be deployed to the runtime of WebSphere Enterprise Service Bus or WebSphere Process Server.

Attention: Enable J2EE capabilities and do a clean rebuild before deploying. The plug-in does not enable J2EE capabilities itself, and this appears to be the cause of a runtime failure to load the mediation module.

Note: The scaffolded MQCICServer program does not handle the no-MQCIH case. Using it, you can only test the outgoing flow.

> **Important:** Before running the mediation module, copy mqcicsmediation.jar located in $WID INSTALL ROOT/wstools/eclipse/com.ibm.sal.mqcics_6.0.2/ to $RUNTIME ROOT/lib/ext. Then restart the runtime server to load the resource.

9.9.4 For the case of with MQCIH header

To add WebSphere MQ header information and MQCIH header into the message, we use the AddMQInfo mediation primitive. We modify the same CICSMQPlugin project to convert to the with-MQCIH case.

The next step is to develop the mediation flow so that we can add an MQCIH header to the SMO message.

1. Continuing from the previous step (Figure 9-23 on page 324), check the box beside **Add MQCIH header to the message**. The finished page looks like Figure 9-24.

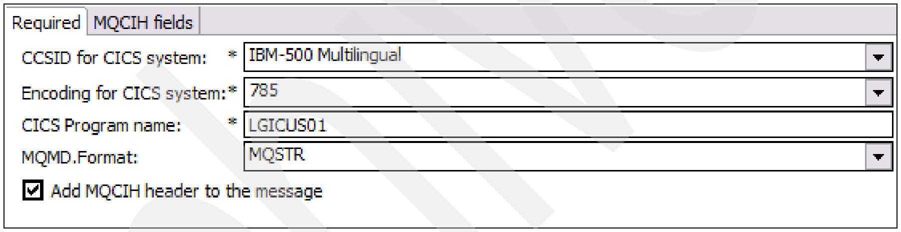


Figure 9-24 Details of Required tab for the case of with-MQCIH header

2. Click the **MQCIH fields** tab and specify the fields as shown in Figure 9-25. Save and close the mediation flow.

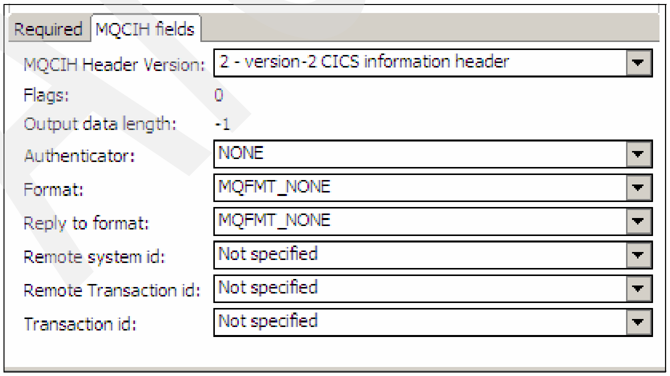


Figure 9-25 Details of MQCIH fields tab in AddMQInfo primitive for the case with MQCIH header

CICSMQPlugin is now ready to be deployed and tested as before. The MQCICServer.java program bounces the request back to test the response flow. Run it from the CICSMQAccess project, as described in 8.7, “Test the mediation” on page 284.

Remember to deploy mqcicsmediation.jar located in $WID INSTALL ROOT/wstools/eclipse/com.ibm.sal.mqcics_6.0.2/ to $RUNTIME ROOT/lib/ext. Then restart the runtime server to load the resource.

Tracing output

The output trace from the plug-in is shown in Figure 9-26. To get this trace you need to enable tracing for the CICS-MQ plug-in and look for the trace.log file, which is in the <WESB Install Root>\pf\esb\logs\server1 directory.

```
CICSMQPlugin/CICSCustomerInterfaceImport1_MQ_RECEIVE_D
[20/06/07 15:04:20:328 BST] 00000014 ApplicationMg A   WSVR0221I: Application started: CICSMQPluginApp
[20/06/07 15:08:00:375 BST] 0000009d AddCIHMediati 3 com.ibm.sal.mqcics.mediation.AddCIHMediation mediate $Revision: 
571 $ Starting add MQ info ......
[20/06/07 15:08:00:625 BST] 0000009d AddCIHMediati 3 com.ibm.sal.mqcics.mediation.AddCIHMediation mediate $Revision: 
571 $ MQCIH header is created successfully.
[20/06/07 15:08:00:781 BST] 0000009d CIHDataBindin 3 com.ibm.sal.mqcics.header.CIHDataBinding mqbinding $Revision: 572 
$ Starting to write MQCIH header ......
[20/06/07 15:08:00:781 BST] 0000009d CIHDataBindin 3 com.ibm.sal.mqcics.header.CIHDataBinding mqbinding $Revision: 572 
$ MQCIH header writes OK.
[20/06/07 15:08:13:703 BST] 0000009a CIHDataBindin 3 com.ibm.sal.mqcics.header.CIHDataBinding mqbinding $Revision: 572 
$ Starting to read MQCIH header ......
[20/06/07 15:08:13:703 BST] 0000009a CIHDataBindin 3 com.ibm.sal.mqcics.header.CIHDataBinding mqbinding $Revision: 572 
$ MQCIH header reads OK.
```

Figure 9-26 Trace from CICS-MQ plugin

To enable trace:

1. Open the Administration Console and select **Troubleshooting** → **Logs and trace** → **server1** → **Change log details levels**. Click **com.ibm.sal.*** in the list of packages and select **all** → **Apply** → **Save** → **Save**.
2. Make sure that **Logging and tracing** → **server1** → **Diagnostic Trace Service** → **Enable log** are checked.

   This panel also allows you to change the location of the trace file and specify its format.

Detach CIH

Optionally, you can modify the response flow to detach the MQCIH header from the SMO message. The detached MQCIH header contains the response fields from CICS system so that you may check fields such as return code for error determination. If you are modifying an existing solution then it may be prudent to remove the MQCIH header from the SMO using this primitive, so that any existing code using the SMO is not upset by the presence of a new header field.

(There is no reason why well-written code should be bothered, but you may want to remove any traces of the header from the SMO anyway.)

For more information about the MQCIH fields, refer to the WebSphere MQ infocenter:

http://publib.boulder.ibm.com/infocenter/wmqv6/v6r0/index.jsp

Important: The DetachCIH mediation primitive can only be used in a response flow when the AddMQInfo mediation primitive is used in the corresponding request flow.

To detach the MQCIH header:

1. Open CICSMediation to edit the mediation flow. Click the **Response** tab, drag and drop the **DetachCIH** mediation primitive into the flow, connect the out terminal of XSLTTransform1 to the in terminal of DetachCIH1, connect the out terminal of DetachCIH1 to access, drag and drop **Stop** from the palette, and connect the header terminal of DetachCIH1 to the in terminal of Stop.
2. Save the flow.

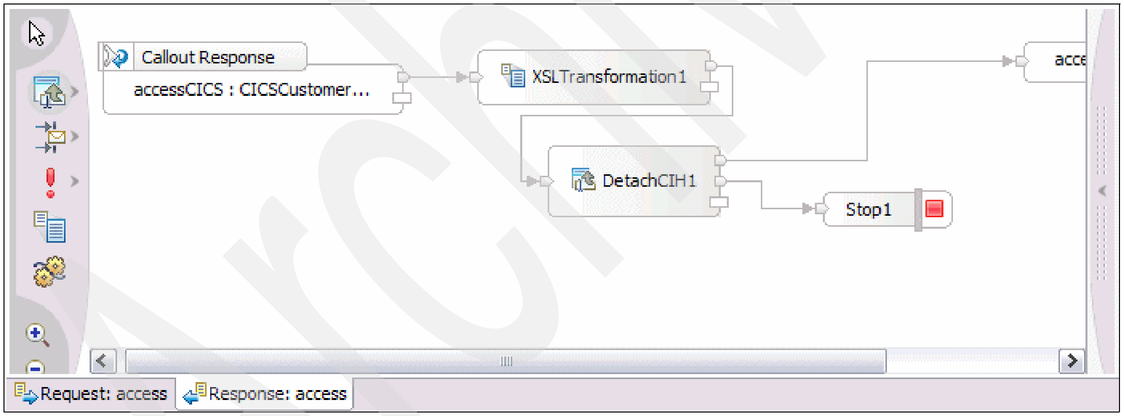


Figure 9-27 Response mediation flow with DetachCIH mediation primitive

Figure 9-27 shows such a flow with the DetachCIH mediation primitive.

9.10 Developing an Eclipse plug-in

From previous sections, we learned the function provided by the WebSphere MQ-CICS Bridge Plug-in. This plug-in is based on WebSphere Integration Developer v6.0.2, which is in turn based on Eclipse.

Plug-ins are the building blocks of Eclipse. Each plug-in can extend other plug-ins and also can provide extension points for others to extend. Plug-in Development Environment (PDE) is a tool that is based on Eclipse for developing plug-ins. A good introduction to plug-in development is found at:

http://www.eclipse.org/articles/Article-PDE-does-plugins/PDE-intro.html

As you can imagine, one of the most important tasks for developing a plug-in is to obtain the correct extension points for the to-be-developed plug-in to extend. Our experience is to look for the extension points by searching the manifest files of existing plug-ins that provide similar features.

Once you find the correct extension points, add them to the plug-in manifest file, that is, plugin.xml. The PDE tool will automatically add those plug-ins that provide these extension points as dependency to the one under development.

Figure 9-28 shows the structure of the WebSphere MQ-CICS Bridge Plug-in and associated extension points.

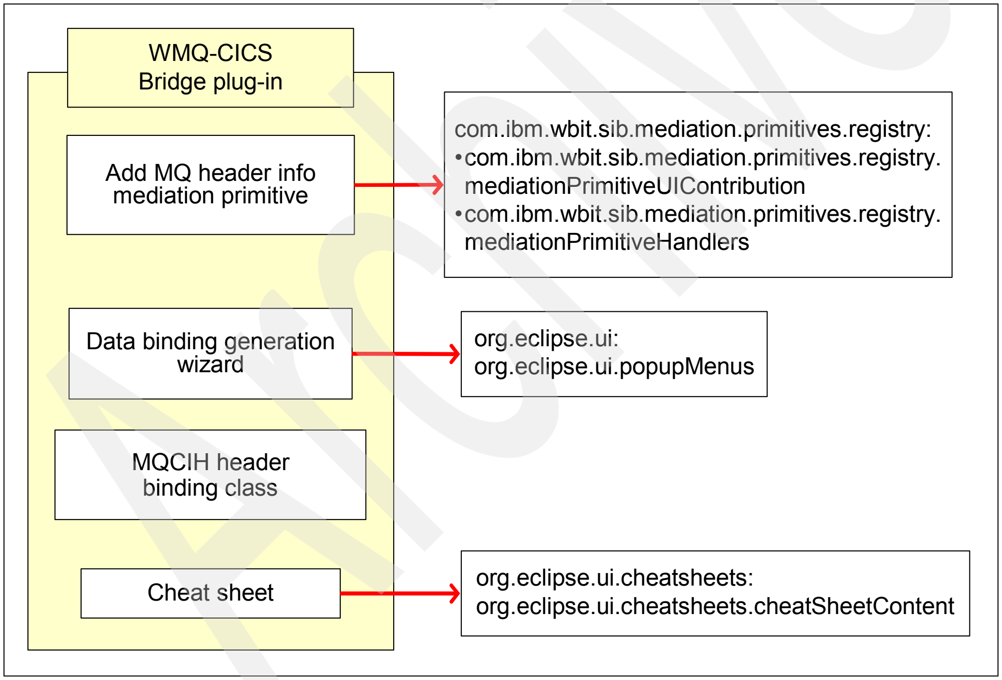


Figure 9-28 WebSphere MQ-CICS Bridge Plug-in structure

An Eclipse environment that has this plug-in installed loads the plug-in by reading plugin.xml and performing the specified function by these extension points.

9.10.1 Extension points used for mediation primitive

Two extension points are used to create the mediation primitive. They are:

- com.ibm.wbit.sib.mediation.primitives.registry.mediationPrimitiveUIContribution
- com.ibm.wbit.sib.mediation.primitives.registry.mediationPrimitiveHandlers

These extension points are provided by the plug-in of com.ibm.wbit.sib.mediation.primitives.registry.

By extending mediationPrimitiveUIContribution, our mediation primitive appears in the palette of the mediation flow editor so that the user can drag and drop it to construct the mediation flow. Example 9-1 shows the statements in plugin.xml that declare the extension for the AddMQInfo mediation primitive.

Example 9-1 Extending mediationPrimitiveUIContribution

```
<extension
point="com.ibm.wbit.sib.mediation.primitives.registry.mediationPrimitiv
eUIContribution">
      <mediationPrimitiveUIContribution
            helpContextId="salmed_addcih"
            largeIcon="icons/addcih_pal.gif"
            mediationPrimitiveTypeName="AddMQInfo"
mediationPrimitiveTypeNamespace="mednode://mednodes/AddMQInfo.mednode"
            paletteCategory="/MQ"
            smallIcon="icons/addcih_dgm.gif">
         add MQ header info
      </mediationPrimitiveUIContribution>
</extension>
```

By extending mediationPrimitiveHandlers, the logic performed by our mediation primitive is invoked by the WebSphere Enterprise Service Bus runtime engine, in the context of a mediation flow that contains an instance of the primitive. Example 9-2 extracts the implementation from plugin.xml.

Example 9-2 Extending mediationPrimitiveHandlers

```
<extension
point="com.ibm.wbit.sib.mediation.primitives.registry.mediationPrimitiv
eHandlers">
      <mediationPrimitiveHandler
implementationClass="com.ibm.sal.mqcics.mediation.AddCIHMediation"
            isTerminalTypeDeducible="false"
```

```
            propertyDefinition="properties/AddCIHPropertyGroup.xml"
            typeName="AddMQInfo"
          typeNamespace="mednode://mednodes/AddMQInfo.mednode">
         <shortDescription>
            Add MQ header info
         </shortDescription>
         <longDescription>
            This primitive is for adding MQ header information to the
message.
         </longDescription>
         <terminalCategory>
               isDynamic="false"
               name="in"
               type="input"/>
         <terminalCategory>
               isDynamic="false"
               name="out"
               type="output"/>
         <terminalCategory>
               isDynamic="false"
               name="fail"
               type="fail"/>
      </mediationPrimitiveHandler>
</extension>
```

For more detailed information, refer to the WebSphere Integration Developer v6.0.2 infocenter, which dedicates a section on the topic of contributing your own mediation primitive plug-in. It can be found at the following link:

http://publib.boulder.ibm.com/infocenter/dmndhelp/v6rxmx/index.jsp?topic=/com.ibm.wbit.help.sib.mediation.ui.doc/topics/rcustspi.html

9.10.2 Extension point used for data binding menu

Another extension point used in WebSphere MQ-CICS Bridge Plug-in is org.eclipse.ui.popupMenus, which is provided by the plug-in of org.eclipse.ui.

By extending it, we get the menu shown in Figure 9-6 on page 309. The logic behind this menu is implemented in a customized class that is specified as part of extending the extension point. The logic is then developed according to our needs.

Example 9-3 shows the implementation for this extension point in plugin.xml.

Example 9-3 Extending popupMenus

```
<extension point="org.eclipse.ui.popupMenus">
      <objectContribution

objectClass="com.ibm.wbit.ui.logicalview.model.BusinessObjectArtifact"
            nameFilter="*"
            id="com.ibm.sal.mqcics.contribution1">
         <menu
               id="com.ibm.sal.mqcics.dbgen.menu1"
               label="Generate Data Binding Class"
               path="additions">
            <separator name="group1">
            </separator>
         </menu>
         <action
               id="com.ibm.sal.mqcics.dbgen.runWizard"
               label="MQ-CICS"

class="com.ibm.sal.mqcics.dbgen.wizard.DataBindingWizardAction"
               enablesFor="1"
               icon="icons/mqcics_db.gif"
               menubarPath="com.ibm.sal.mqcics.dbgen.menu1/group1">
         </action>
      </objectContribution>
   </extension>
```

9.10.3 Extension point used for cheat sheet

Eclipse provides a quick way to document information about the under-developed plug-in. The cheat sheet is the one that is used in the WebSphere MQ-CICS Bridge Plug-in to achieve that purpose.

The extension point is org.eclipse.ui.cheatsheets.cheatSheetContent and is provided by org.eclipse.ui.cheatsheets plug-in. The content of the information is specified in an xml file. Which name and location are declared in the WebSphere MQ-CICS Bridge Plug-in manifest file.

9.11 Conclusion

In this chapter, we described the scenario of using the WebSphere MQ-CICS Bridge Plug-in to simplify integrating CICS applications with an SOA implemented using the WebSphere Enterprise Service Bus. We also briefly discussed the structure of the WebSphere MQ-CICS Bridge Plug-in and plug-in development.

If you compare the user experience of integrating a CICS transaction using the plug-in to the procedure followed in Chapter 8, “Custom CICS integration using WebSphere MQ” on page 215, you can see the amount of coding that has been eliminated.

Writing a plug-in to extend the WebSphere Enterprise Service Bus tooling is an alternative to hand-crafting mediation code, and to writing a WebSphere Adapter.

10

Custom application integration using JMS

In 5.3, “Case 3: automotive spare parts” on page 138, ITSO_Auto has a manual purchase order system that it wants to replace. ITSO-Auto found the purchase order system to be inefficient and time consuming. The new purchase order system needs to find spare parts prices in real-time from the suppliers, select the most competitive price, and reduce manual overheads (Figure 10-1).

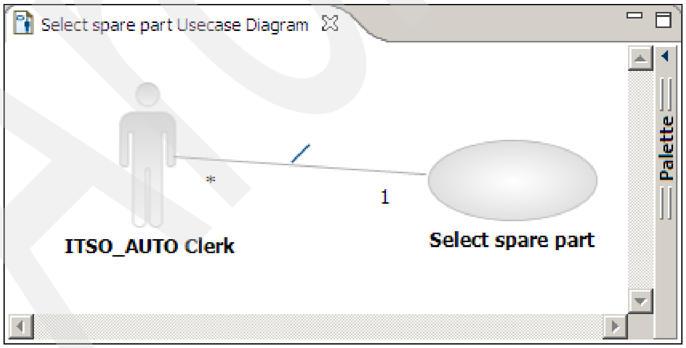


Figure 10-1 Select spare part use case diagram

10.1 Solution design

In this scenario we assume that both the new purchasing system and the connections to the dealer network are to be modified. Essentially, there is little constraint on the choice of how to connect to the systems, as the vendor who will do the integration can choose and modify the purchasing system and the connections to the dealer network. The connection pattern is described in 6.5, “Connection pattern 5: custom application integration” on page 160. The architecture pattern figure is repeated in Figure 10-2 for convenience.

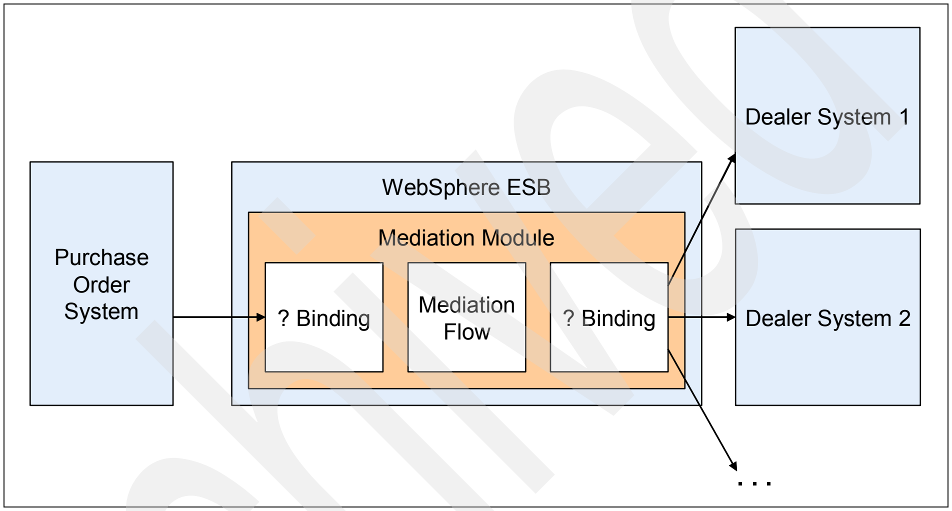


Figure 10-2 Integrating the purchase order system and dealership systems

The vendor responsible for building the solution has decided to connect to the purchase order system using XML over JMS, and to the spare parts system using SOAP over http. In Chapter 13, “Lightweight Web service integration using http” on page 465, an alternative connection technology to the dealer network is built using JSON-RPC. The proposed architecture for the business scenario is show in Figure 10-3.

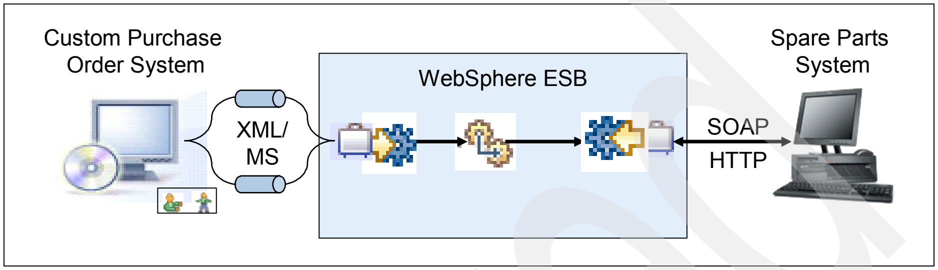


Figure 10-3 Solution architecture overview

The technical assumptions behind the business scenario are:

- The spare parts suppliers expose their purchase order system as a SOAP/http Web service. On reception of a well-formatted SOAP message, the supplier system is able to provide the price quote as a response.
- The custom ISV purchase order system provides reliable synchronous and asynchronous communication using XML over JMS.

The example in this chapter shows how to:

- Use the WebSphere Enterprise Service Bus JMS/XML binding, and how to set up the JMS communication.
- Configure the WebSphere Enterprise Service Bus SOAP/HTTP transport binding to build and send a well-formed SOAP message.
- How protocol switching and data transformation is performed by WebSphere ESB.

10.1.1 Service component design

The design uses the following components,

- Purchase-order system connected JMS/XML
- Supplier order management services connected using SOAP/ HTTP
- PurchasePrderMediation mediation module
- SOAP/HTTP binding
- JMS data binding

Component interactions

The ITSO-Auto user will enter a spare parts ID and quantity to find the competitive prices. A JSP™ generates an XML document and places it on a JMS input queue. The Export JMS binding listens for messages on the queue and picks up the message from the JMS input queue, validates it for the format, and passes the request to Mediation implementation. In the Mediation implementation the request would normally be processed using XSL transformations to convert the PurchaseOrder XML business object into a SpareParts business object and then to invoke the order method on the spare parts system using SOAP. In the simple example in this book the format of the two objects is the same and no transformation is necessary.

On the reply flow the JMS binding constructs an XML message from the business object and places it on the JMS output queue to be received by the JSP and formatted to output onto the Web page.

10.2 Create PurchaseOrder JMS client

This section describes creating a JSP client that has been configured to send an order and receive a confirmation using JMS. This provides us with a simple browser interface for testing purposes.

1. Open WebSphere Integration Developer. Close the welcome pane. Switch to the Web perspective.

2. Click **File** → **New** → **Dynamic Web Project**. Name it `ITSO-AUTOPOJMSClient`. Click **Show Advanced**. Set the target server to **WebSphere ESB Server v6.0** (Figure 10-4) and click **Finish**.

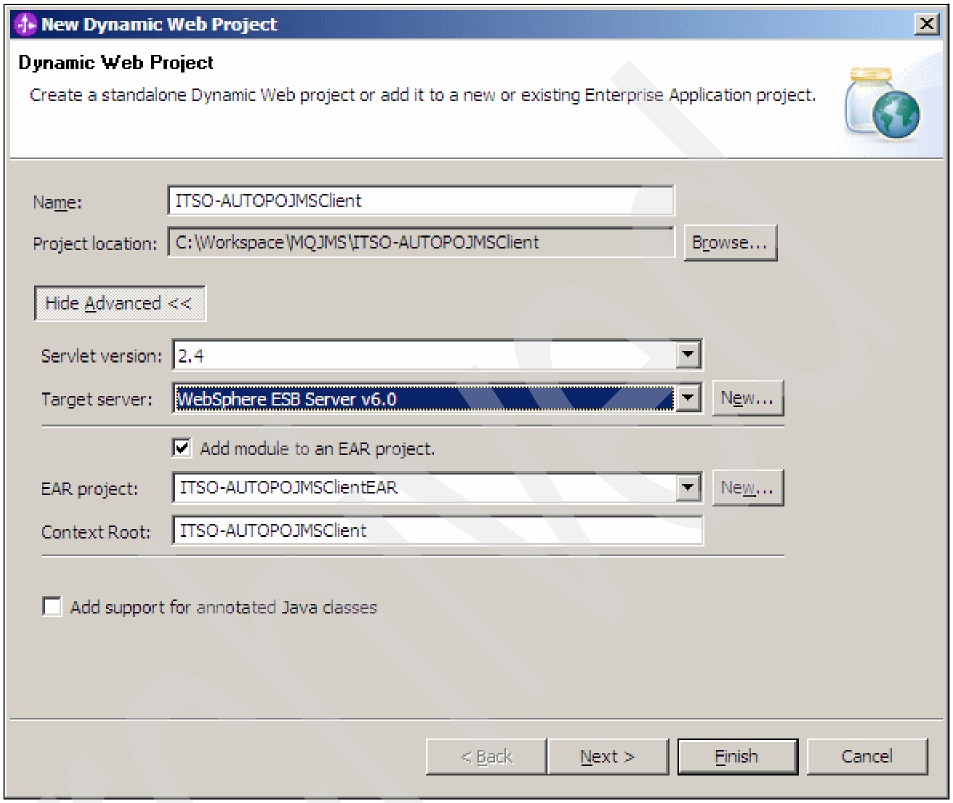


Figure 10-4 Dynamic Web Project creation wizard with Advanced Options

3. Right-click the **ITSO-AUTOPOJMSClient** Web project and select **New** → **JSP file**. Name it `ITSO-AutoFindPartsPrice` (Figure 10-5) and click **Finish**.

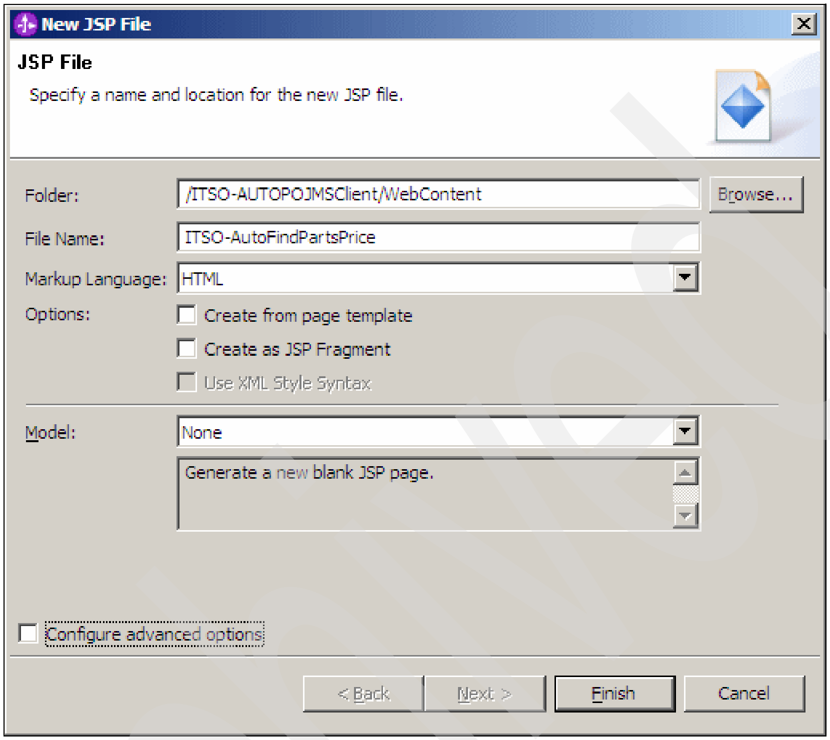


Figure 10-5 Create ITSO-AutoFindPartsPrice

4. In the JSP editor switch to the **Source** tab (Figure 10-6).

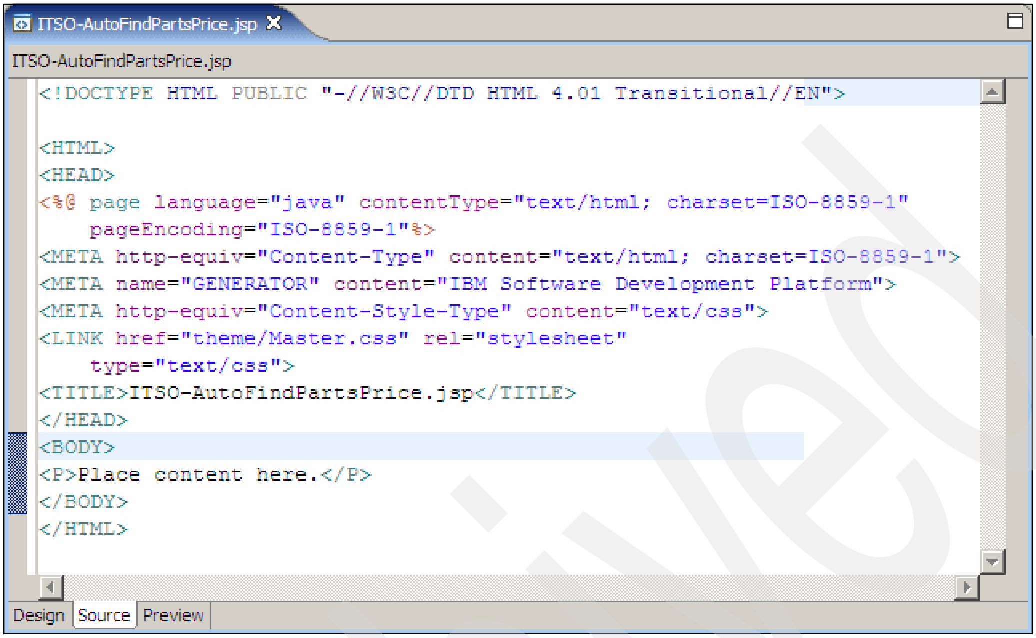


Figure 10-6 Default source for ITSO-AutoFindPartsPrice

5. Replace the contents of ITSO-AutoFindPartsPrice.jsp with that shown in Example B-2 on page 509. The details of the JSP are described in “Design of ITSO-AutoFindPartsPrice.jsp” on page 340 below.

6. To display the Redbooks logo, copy the redbooklogo.gif file from the additional materials into the WebContent folder of the project (Figure 10-7).

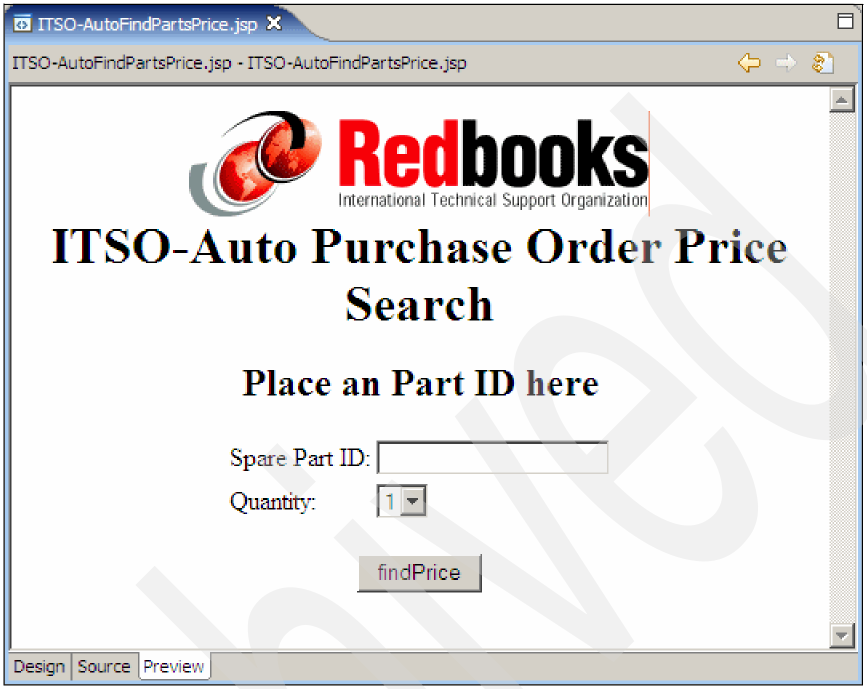


Figure 10-7 Preview ITSO-AutoFindPartsPrice.jsp

Design of ITSO-AutoFindPartsPrice.jsp

The static boilerplate html to format the browser window is shown in Figure 10-9 on page 342. The JSP Scriplet is listed in Figure 10-11 on page 344 and Figure 10-12 on page 345.

The XML document containing the part request is described by two schemas, a type definition for the request, called PartRequest, in the schema file PartRequest.xsd; and the document definition containing the root element, requestParts, typed as a PartRequest, in the file requestPart.xsd. See Figure 10-8 on page 341. Using the XML wizard to create an XML document from a schema, you can generate a template for the scriplet that exactly matches the schemas. We later import the PartRequest type into the mediation that receives the XML document, and use the requestParts root element to define the mediation interface. The JMS business object from JMSText message binding uses the mediation interface to parse the incoming XML document, and by using the document schema definition to define the interface we can be more confident that the incoming XML message matches the generated binding parser.

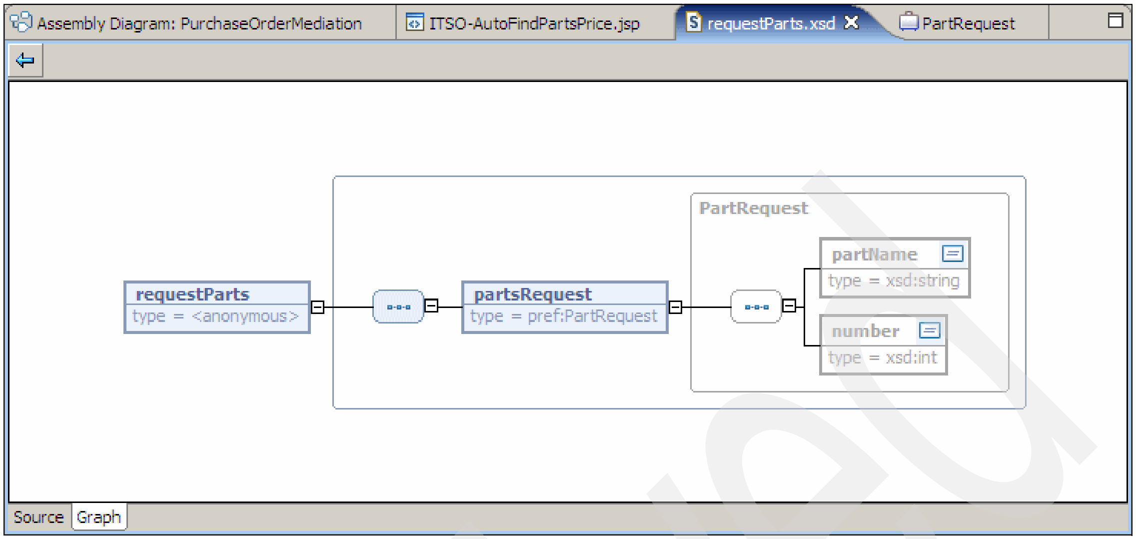


Figure 10-8 PartRequest.xsd and requestParts.xsd

When we come to build the mediation interface, the root element name of the document becomes the name of the method, and the namespace of the root document element schema becomes the namespace of the interface. We also need to set the JMSFunctionType of the JMS message to the root document element name.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
   pageEncoding="ISO-8859-1"%>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<meta name="GENERATOR" content="IBM Software Development Platform">
<title>ITSO-AutoFindPartsPrice.jsp</title>
</head>
<body>
<div style="text-align: center">
<h1><IMG border="0" src="redbooklogo.gif" width="288" height="66" alt=""><BR>
ITSO-Auto Purchase Order Price Search</h1>
<h2>Place an Part ID here</h2>
<form method="get" action="ITSO-AutoFindPartsPrice.jsp">
<table align="center">
   <tr>
      <td>Spare Part ID:</td>
      <td><input type="text" name="productid" /></td>
   </tr>
   <tr>
      <td>Quantity:</td>
      <td><select name="quantity">
         <option>1</option>
         <option>2</option>
         <option>3</option>
         <option>4</option>
         <option>5</option>
      </select></td>
   </tr>
</table>
<br />
<input name="findPrice" type="submit" value="findPrice" /></form>
<%... jsp scriplet here ... %>
</div>
</body>
</html>
```

Figure 10-9 Boilerplate html in ITSO-AutoFindPartsPrice

Figure 10-10 shows the constants to be used by the servlet. You need to look up the port for the Provider URL for your system. It is normally 2809.

Find the Provider URL by opening the Administration console and selecting **Servers** → **Application servers** → **Communications** → **Ports** (Figure 10-10).

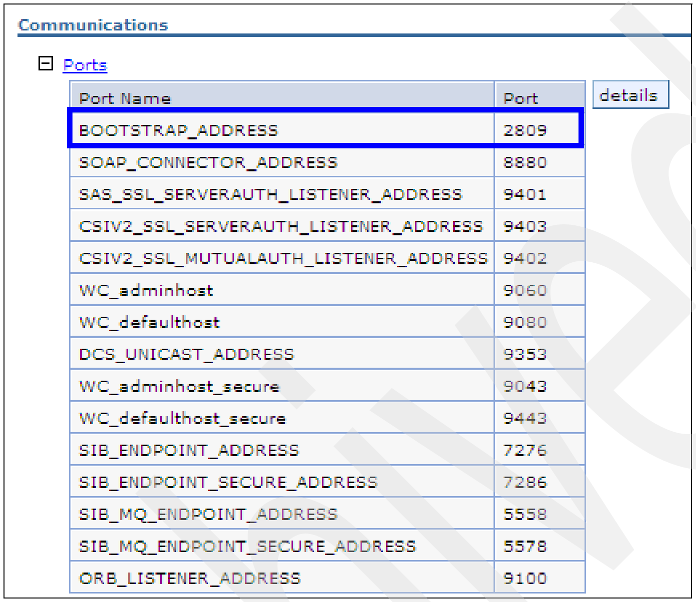


Figure 10-10 Bootstrap address

The JMS names ned are configured in 10.4, “Define WebSphere ESB server resources” on page 356.

```
if (request.getParameter("productid") != null) {
// The Initial Context Factory
    String icf = "com.ibm.websphere.naming.WsnInitialContextFactory";
// the Provider URL (This port number may be different depending on your install)
    String url = "iiop://localhost:2809/";
// The Queue Connection Factory used to connect to the bus
    String sampleQCF = "jms/purchaseOrderQCF";
// The Queue used to send requests to the mediation module
    String sampleSendQueue = "jms/purchaseOrderInputQueue";
// The Queue used to receive responses from the mediation module
    String sampleReceiveQueue = "jms/purchaseOrderOutputQueue";
// The XML representation of an Order Business Object required by the
// placeOrder operation on the OrderService Interface
    String message = "<?xml version=\"1.0\" encoding=\"UTF-8\"?>";
    message += "<tns:requestParts xmlns:p=\"http://PurchaseOrderMediation\" ";
    message += "xmlns:tns=\"http://PurchaseOrderMediation/RequestForParts\">";
    message += "<partsRequest><partName>";
    message += request.getParameter("productid");
    message += "</partName><number>";
    message += request.getParameter("quantity");
    message += "</number></partsRequest></tns:requestParts>";
    System.out.println("message is " + message);
```

Figure 10-11 Set up constants for the jsp scriplet

```
    try {
// Create the Initial Context
        java.util.Hashtable env = new java.util.Hashtable();
        env.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY, icf);
        env.put(javax.naming.Context.PROVIDER_URL, url);
        javax.naming.Context ctx = new javax.naming.directory.InitialDirContext(env);
// Lookup the ConnectionFactory
        javax.jms.ConnectionFactory factory = (javax.jms.ConnectionFactory) ctx.lookup(sampleQCF);
// Create a Connection
        javax.jms.Connection connection = factory
                .createConnection();
// Start the Connection
        connection.start();
// Create a Session
        javax.jms.Session jmsSession = connection.createSession(
                false, javax.jms.Session.AUTO_ACKNOWLEDGE);
// Lookup the send Destination
        javax.jms.Destination sendQueue = (javax.jms.Destination) ctx.lookup(sampleSendQueue);
// Create a MessageProducer
        javax.jms.MessageProducer producer = jmsSession.createProducer(sendQueue);
// Create the TextMessage that will hold our Order as text
        javax.jms.TextMessage sendMessage = jmsSession.createTextMessage();
// Set the content of the message to be the XML defined Order
        sendMessage.setText(message);
// Set the operation to call on the OrderService interface to be placeOrder
        sendMessage.setStringProperty("TargetFunctionName","requestParts");
// Send the message
        producer.send(sendMessage);
// Lookup the receive Destination
        javax.jms.Destination receiveQueue = (javax.jms.Destination) ctx.lookup(sampleReceiveQueue);
// Create a MessageConsumer
        javax.jms.MessageConsumer consumer = jmsSession.createConsumer(receiveQueue);
// Wait 15 seconds to receive the response
        javax.jms.TextMessage receiveMessage = (javax.jms.TextMessage) consumer.receive(15000);
// If we receive a response print the contents of the message to the screen
            String confirmation = "Price Search failed.";
        if (receiveMessage != null) {
// Print the contents of the message.
            confirmation = "Price Search Successfull.<br/>Price per Unit: $" + receiveMessage.getText();
        }
        out.println("<p>" + confirmation + "</p>");
// Close the Connection
        connection.close();
    } catch (Exception e) {
        out.println(e);
    }
}
```

Figure 10-12 JSP scriptlet

10.3 Create spare parts server

The spare parts server is simulated by a simple Java class running in the J2EE Web container.

1. Right-click **Dynamic Web Projects** and select **New** → **Dynamic Web Project**. Name it `PartsPrice`. Select **WebSphere Enterprise Service Bus v6.0** as the target server (Figure 10-13) and click **Finish**.

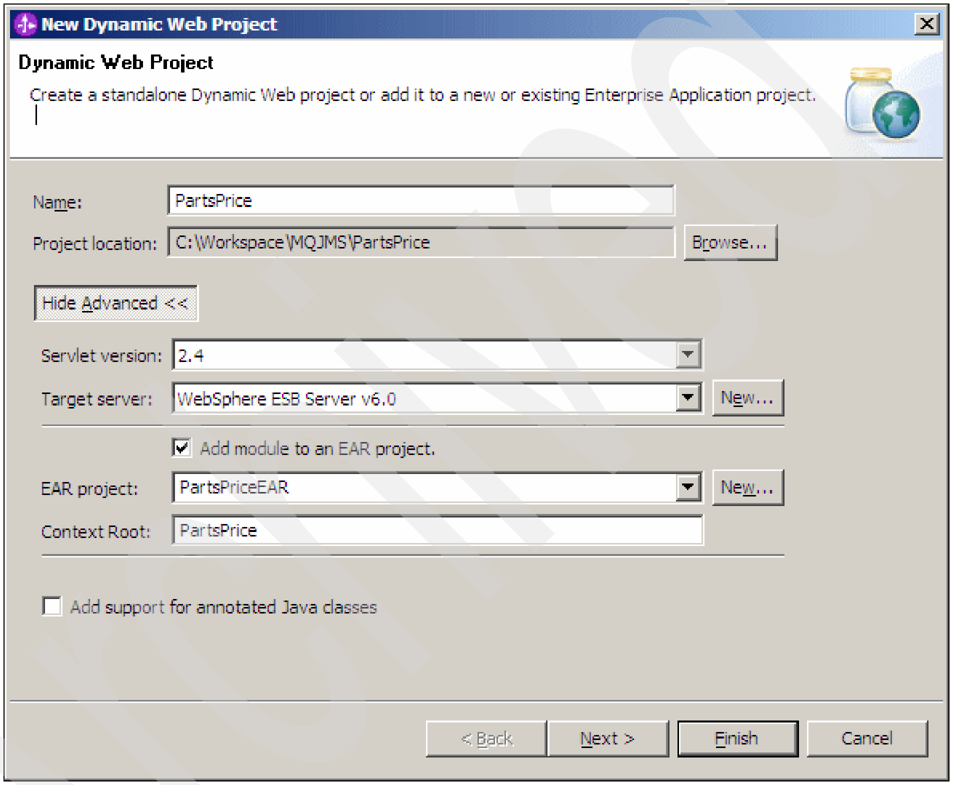


Figure 10-13 Create PartsPrice Web project

2. Select **Dynamic Web Projects** → **PartsPrice** → **Java Resources** → **Java Source** → **New** → **Class**. Name the package `com.ibm.itsospare.purchaseorder`. Name the class `PartsPrice` (Figure 10-14) and click **Finish**.

   The Java editor opens with the template PartsPrice.java.

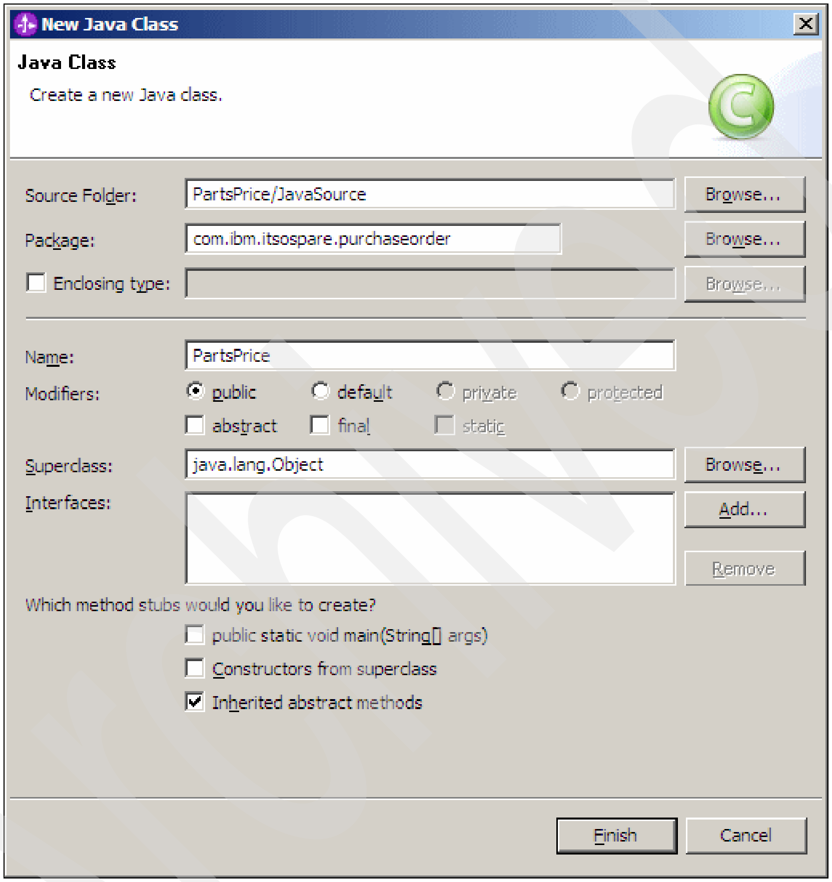


Figure 10-14 Creation of PartsPrice Java class

3. Add the order(PurchaseOrder pOrder) method to the class (Figure 10-15).

```
package com.ibm.itsospare.purchaseorder;
import java.util.Random;
public class PartsPrice {
   public OrderQuote order(PurchaseOrder pOrder) {
      OrderQuote oQuote = new OrderQuote();
      if (!pOrder.getId().equals(null) && pOrder.getQuantity() != 0) {
         oQuote.setPrice(Integer.toString(new Random().nextInt(1000)));
         System.out.println("Part Id= " + pOrder.getId() + " Quantity= "
            + pOrder.getQuantity() + " Quote= " + oQuote.getPrice());
      }
      return oQuote;
   }
}
```

Figure 10-15   order(PurchaseOrder pOrder) method

4. Next add the two classes required by the order(PurchaseOrder pOrder) to create the package contents shown in Figure 10-16.

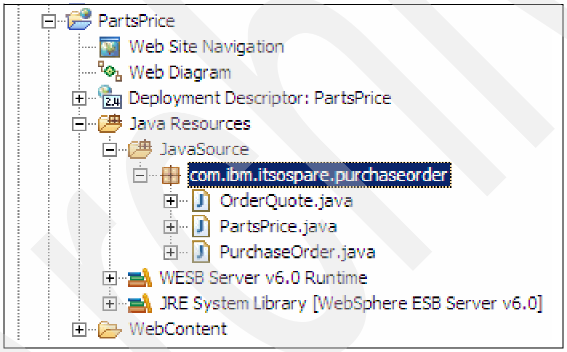


Figure 10-16   com.ibm.itsospare.purchaseorder package

5. Right-click the package **com.ibm.itsospare.purchaseorder** and select **New** → **Class**. Name the class `PurchaseOrder` (Figure 10-17).

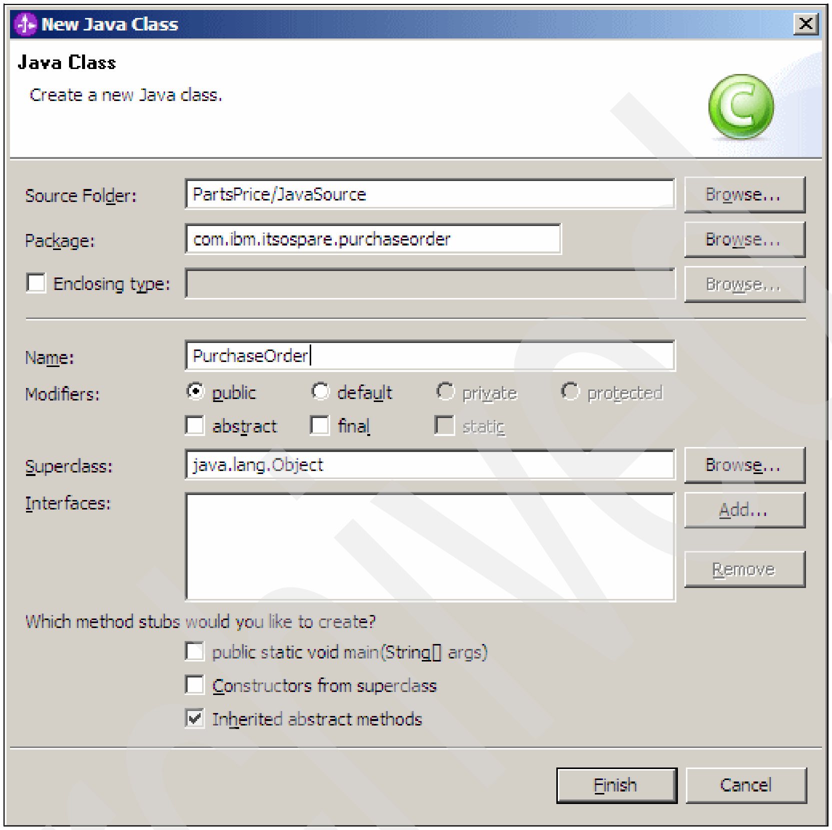


Figure 10-17 Create PurchaseOrderClass

The Java editor opens with the template PurchaseOrder.java.

6. Replace the class with the code from Figure 10-18.

```
package com.ibm.itsospare.purchaseorder;
public class PurchaseOrder {
   private String id;
   private int quantity;
   public PurchaseOrder() {}
   public String getId() {return id;}
   public void setId(String id) { this.id = id; }
   public int getQuantity() { return quantity; }
   public void setQuantity(int quantity) { this.quantity = quantity; }
}
```

Figure 10-18   PurchaseOrder class

7. Right-click the package **com.ibm.itsospare.purchaseorder** and select **New** → **Class**. Name the class `OrderQuote` (Figure 10-19).

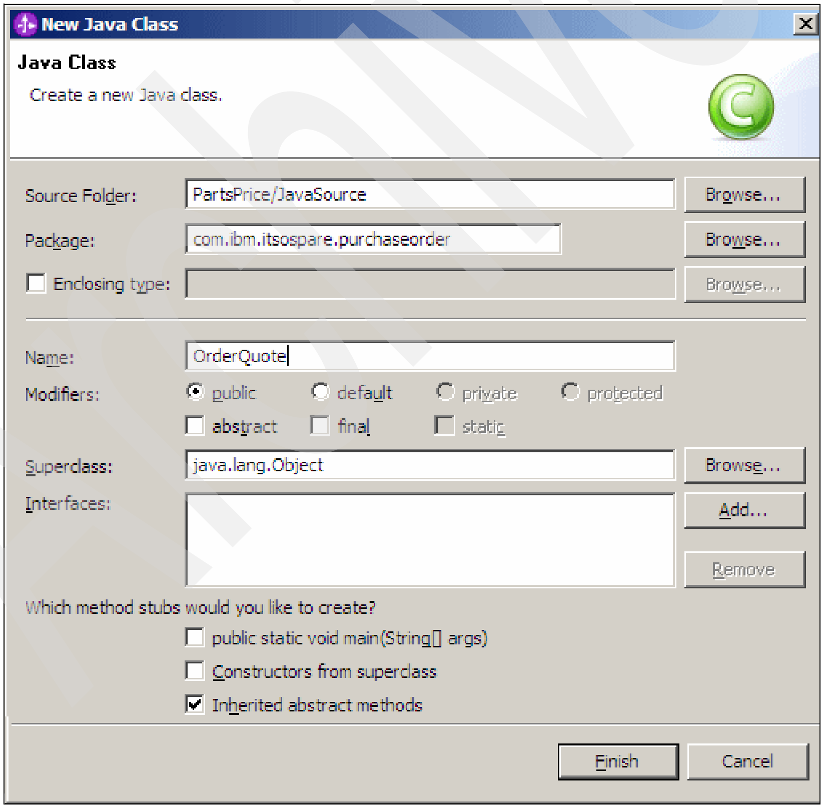


Figure 10-19   OrderQuote class

8. Replace the class with the code from Figure 10-20 and click **Save All**.

```
package com.ibm.itsospare.purchaseorder;
public class OrderQuote {
   private String price;
   public OrderQuote() {}
   public String getPrice() { return price; }
   public void setPrice(String price) { this.price = price; }
}
```

Figure 10-20 OrderQuote class

There should be no errors left in the Problems view.

10.3.1 Expose spare parts as a Web service

To create the Web service we need to enable the Web service Developer Capability, and, for good measure, the XML Developer.

1. Select **Window** → **Preferences** → **Workbench** → **Capabilities**. Check **Web Service Developer** and **XML Developer** (Figure 10-21) and click **OK**.

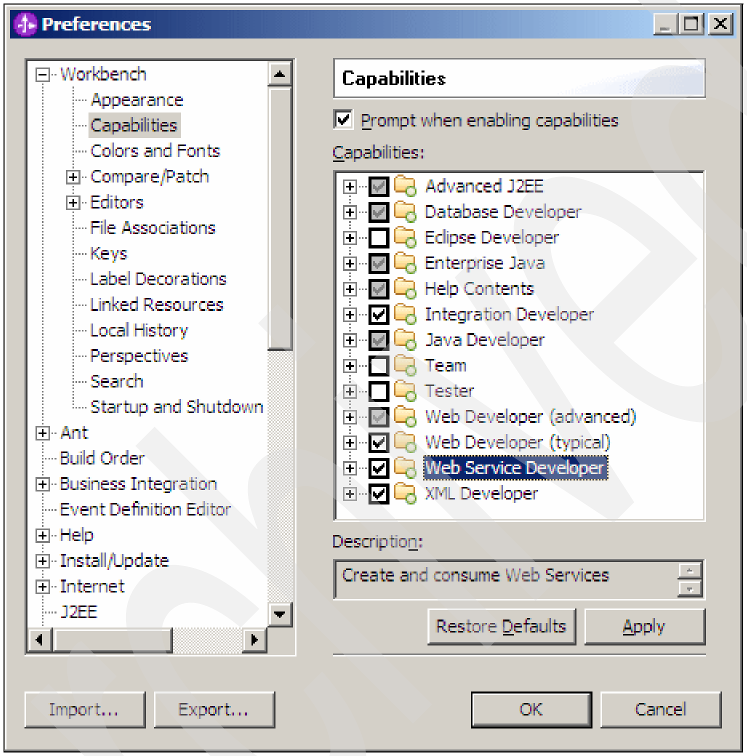


Figure 10-21 Enable Web service and XML developers capabilities

2. Right-click **PartsPrice.java** and select **Web services** → **Create Web service** → **Uncheck**. Start the Web service in a Web project (Figure 10-22) and click **Next** → **Next**.

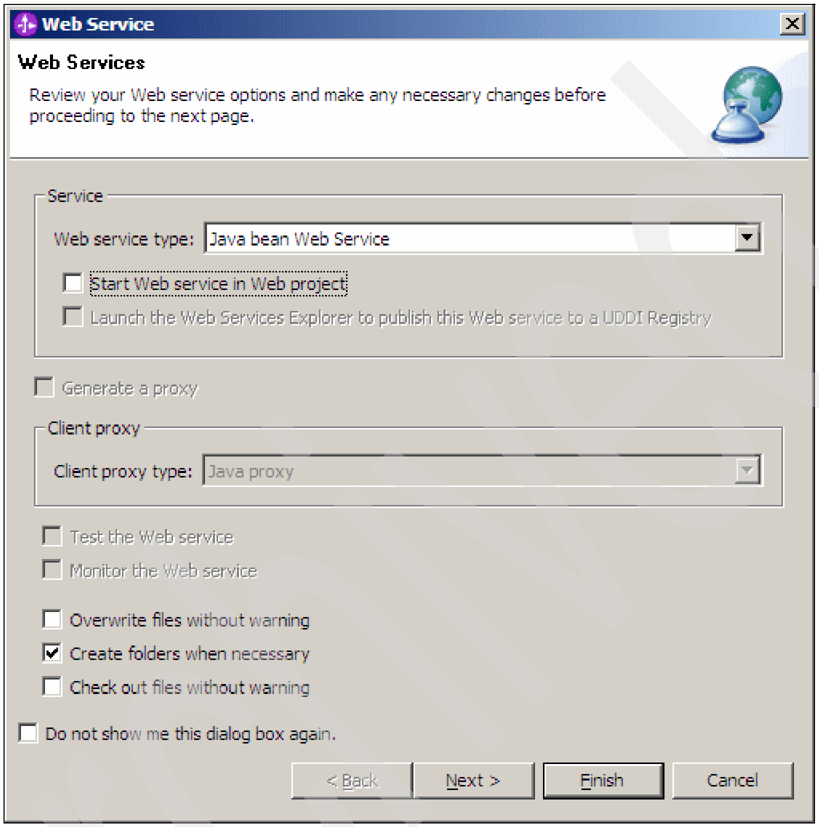


Figure 10-22 Creating Web service from PartsPriceService Java class

3. Check that the Service Deployment Configuration panel settings match those in Figure 10-23.

   You may have to change the Web service runtime to IBM WebSphere and the J2EE version to 1.4, depending what the preferences set up for the Dynamic Web Project are. Click the **Edit** button to change the settings.

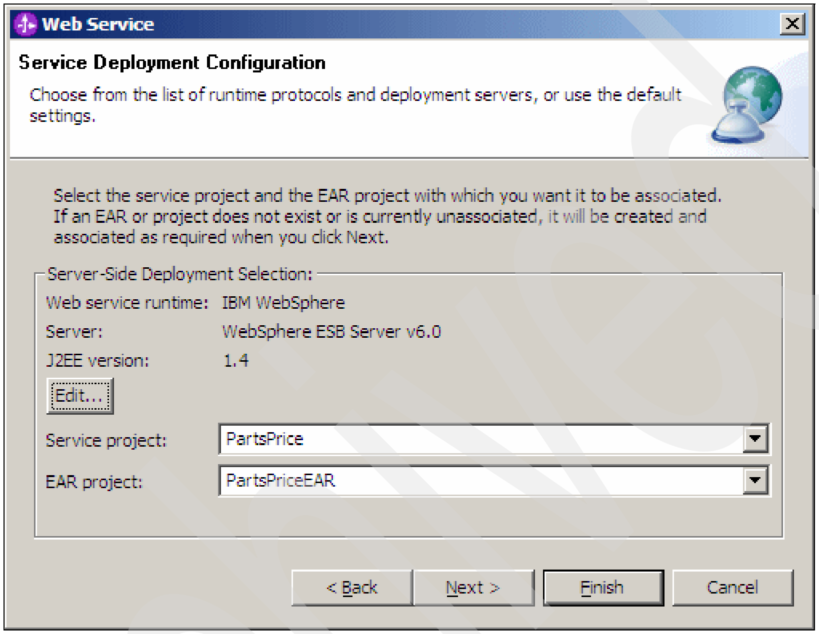


Figure 10-23 Service Deployment Configuration Panel

4. Clicking **Next** displays the Service Endpoint Interface Selection panel.
5. If WebSphere Enterprise Service Bus v6.0 is not running, at this point it will be started, so this step may take some time.
6. Clicking **Next** displays the Web service Java Bean Identity panel. Click **Finish**.

   This generates several Java files and a WSDL file describing the Web service.

10.3.2 Test the Web service

Test the Web service using the Web Services Explorer. It is worth re-publishing the service before doing the test by clicking the publish icon in the Servers view.

1. Launch the Web Services Explorer from the icon in the menu bar, or right-click the **PartsPrice** wsdl file and select **Web Servers** → **Test with Web**

Services Explorer, or open the Web Services folder in the Project explorer and click **Services** → **PartsPriceService** → **Test with Web Services Explorer**.

2. Expand the Navigator tree, select **order**, enter some parameters, and click **Go**.

The results are displayed in Figure 10-24.

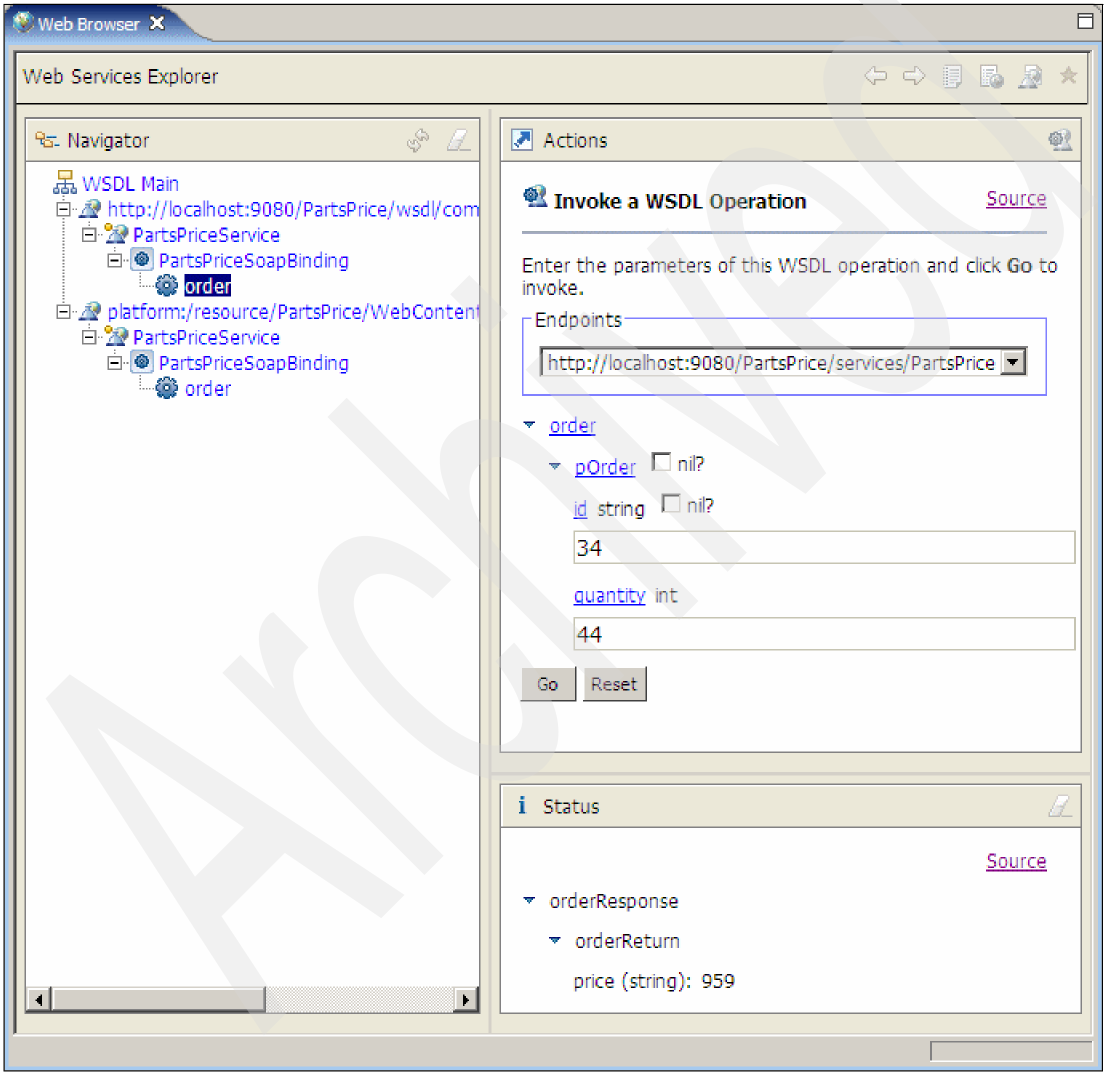


Figure 10-24 Test PartsPrice.order()

10.4 Define WebSphere ESB server resources

This section describes the JMS resources needed for the PurchaseOrder JMS client and the mediation module to communicate. These are listed in Table 10-1. The following section has instructions on how to apply these values to WebSphere Enterprise Service Bus. Rather than have the tooling generate the configuration automatically (see the option in Figure 10-48 on page 370), we show how to create these values manually. Omit this selection if you want to proceed more quickly and choose **Configure new messaging provider resources** in the JMS Export binding wizard in 10.5.6, “Creating a JMS Export binding” on page 369.

Table 10-1 JMS and service integration bus resources

Attribute	Value	Description
Service integration bus	SCA.APPLICATION.esbCell.Bus	Default application service bus
JMS Queue connection factory	jms/purchaseOrderQCF	SCA.APPLICATION.esbCell.Bus
Bus Input destination queue	OrderInputQueue	Stores request messages
Bus Output destination queue	OrderOutputQueue	Stores response messages
JMS Input destination	jms/purchaseOrderInputQueue	OrderInputQueue
JMS Output destination	jms/purchaseOrderOutputQueue	OrderOutputQueue
JMS activation specification	jms/purchaseOrderAS	Used by the mediation module to register queue with message endpoint listener
purchaseOrderAS JNDI name	jms/purchaseOrderAS	Attribute of activation specification
purchaseOrderAS destination JNDI name	jms/purchaseOrderInputQueue	Attribute of activation specification

Start a browser with the URL http://localhost:9060/ibm/console/ to open the Administration console.

> **Note:** You may need to find the task filtering selector on the welcome page and set the filter to **All** → **Apply**.

10.4.1 Create the input and output bus destination queues

Create the physical queues on the default Service Integration Bus.

1. In the navigator click **Service Integration** → **Buses** (Figure 10-25 on page 357).

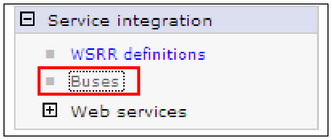


Figure 10-25 Select Buses in the navigator

2. Click **Buses**. A list of buses is displayed (Figure 10-28 on page 358). Click **SCA.APPLICATION.esbCell.Bus**.

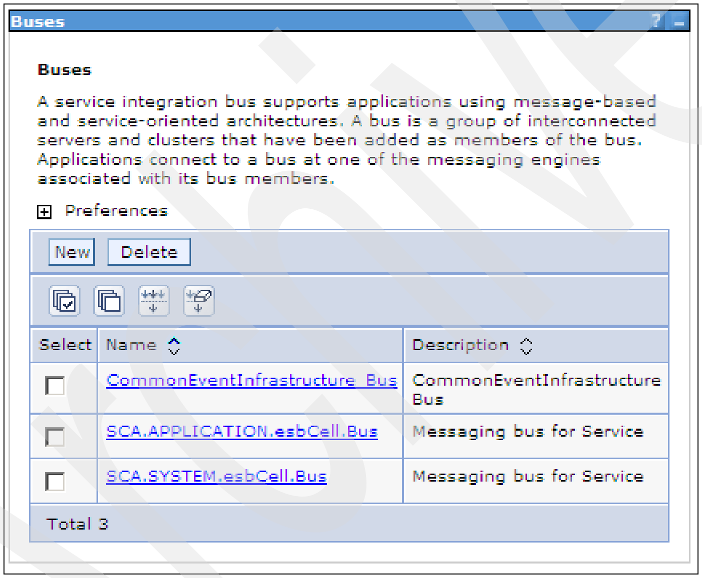


Figure 10-26 Service buses

3. In the list of Destination resources (Figure 10-27), select **Destinations** (Figure 10-28 on page 358).

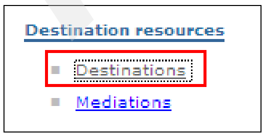


Figure 10-27 Destination resources

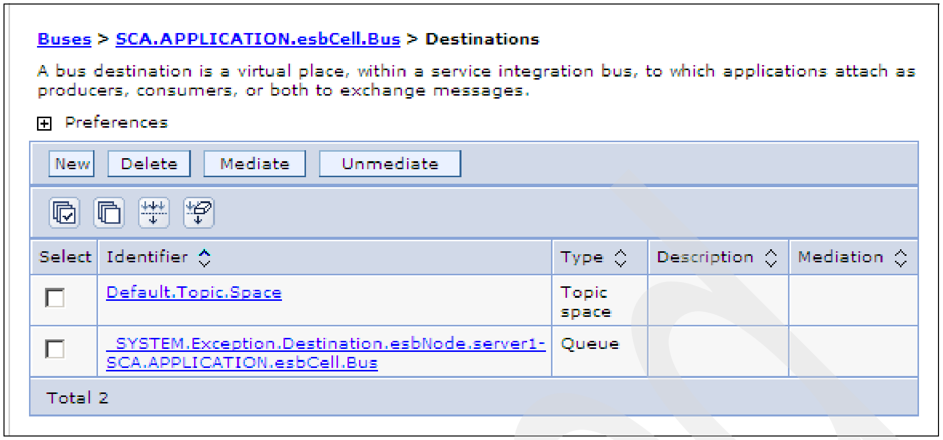


Figure 10-28 List of destinations

4. Click **New** → **Queue** (Figure 10-29) → **Next**.

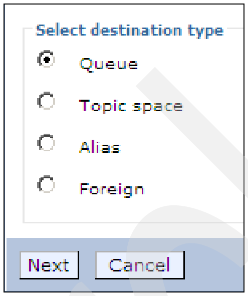


Figure 10-29 Destination types

5. Set the Identifier field to `OrderInputQueue` (Figure 10-30) and click **Next** → **Next** → **Finish**.

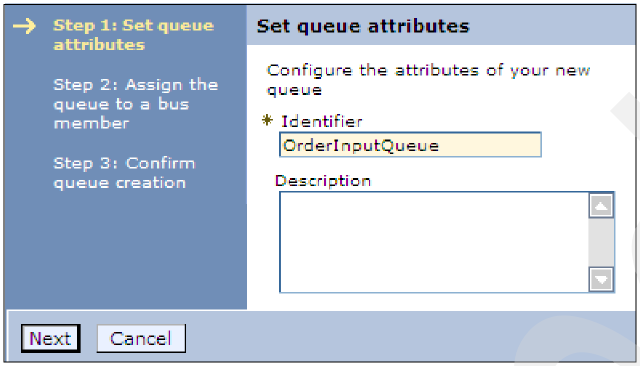


Figure 10-30 Destination queue creation

6. Repeat the steps to create another destination queue called `OrderOutputQueue` (Figure 10-31), then click **Save** → **Save**.

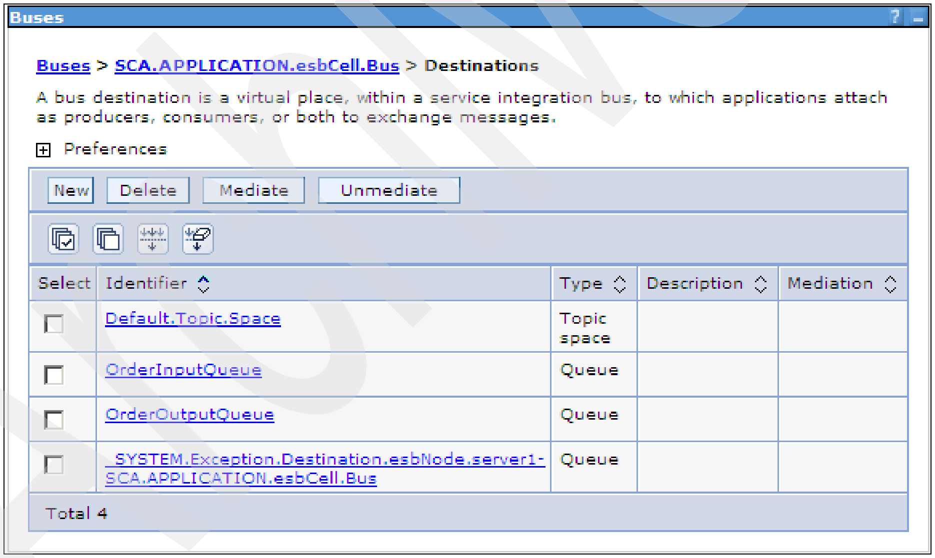


Figure 10-31 List of destination queues

10.4.2 Create JMS queue destinations

The JMS queues are stored in JNDI and describe how to access the physical queues. The JMS queue destinations are configured to point at the Service Integration Bus OrderInputQueue and OrderOutputQueue queue destinations.

1. In the navigator select **Resources** → **JMS Providers** → **Default Messaging** (Figure 10-32).

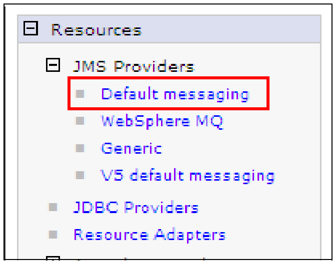


Figure 10-32 JMS Default messaging provider

2. In the Destinations click **JMS queue** (Figure 10-33) → **New**

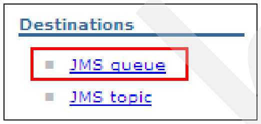


Figure 10-33 JMS queue

3. For bus name select **SCA.APPLICATION.esbCell.Bus**. For queue name select **OrderInputQueue**. For name type `purchaseOrderInputQueue`. For JNDI name type `jms/ purchaseOrderInputQueue` (Figure 10-34). Click **OK**.

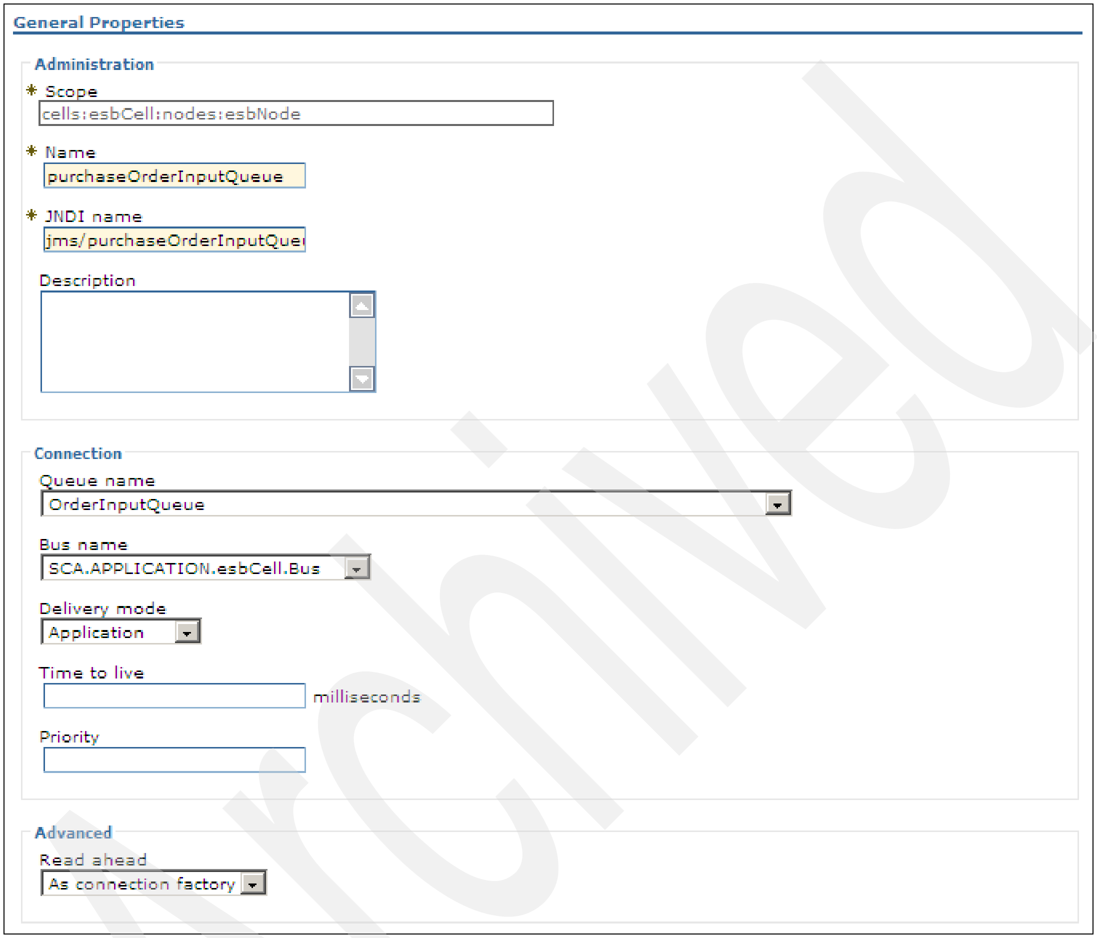


Figure 10-34 purchaseOrderInputQueue destination

4. Repeat these steps to define the JMS queue destination purchaseOrderOutputQueue, selecting OrderOutputQueue as the connection queue and defining jms/purchaseOrderOutputQueue as the JNDI name (Figure 10-35). Click **OK** → **Save** → **Save**.

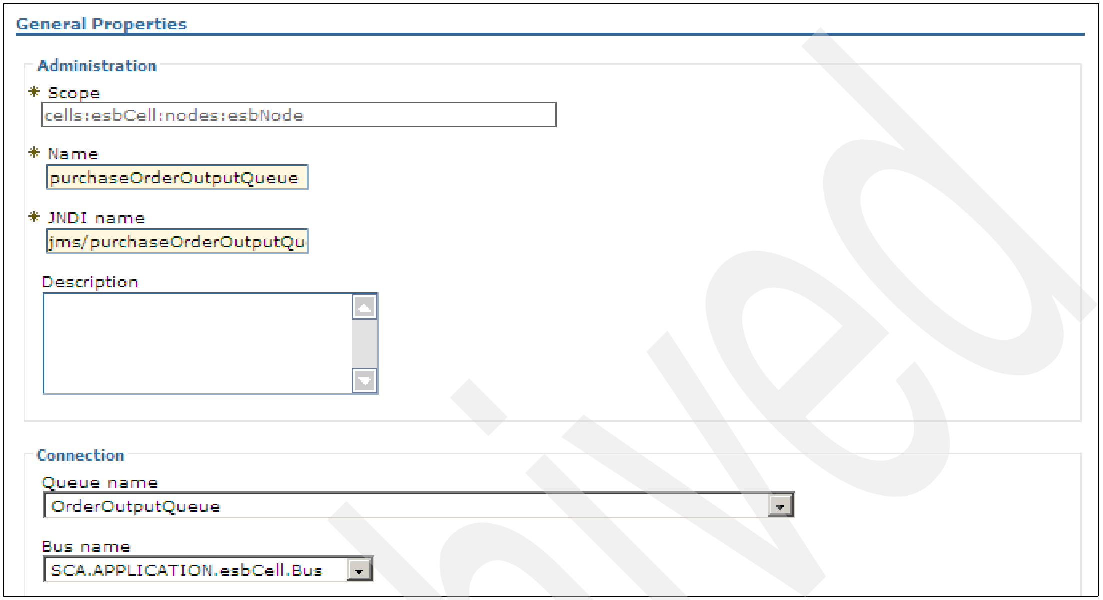


Figure 10-35 JMS purchaseOrderOutputQueue destination

5. The JMS destination queues are listed in Figure 10-36.

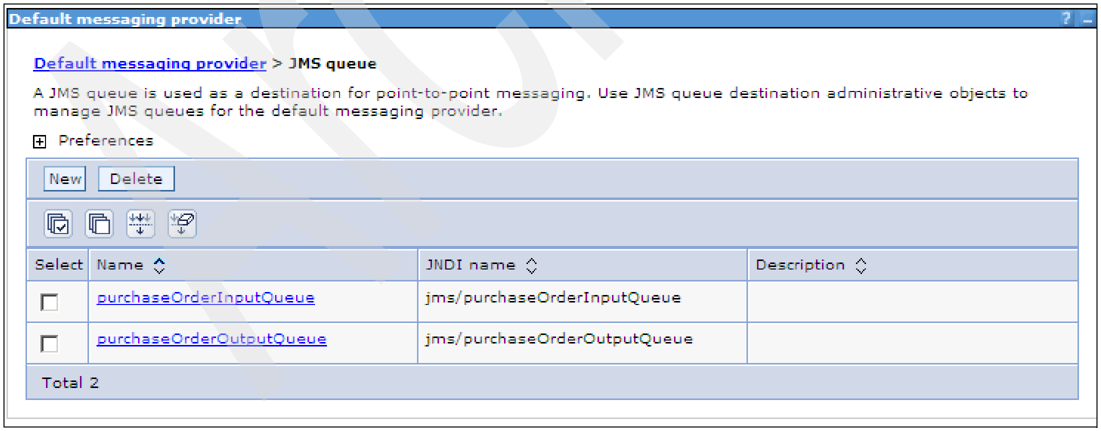


Figure 10-36 JMS destination queues

10.4.3 Create the JMS QueueConnectionFactory

The queue connection factory is stored in JNDI and describes how to create a connection to the messaging engine. The messaging engine is available through the SCA.APPLICATION.esbCell.Bus service integration bus.

1. In the navigator select **Resources** → **JMS Providers** → **Default Messaging** (Figure 10-32 on page 360).
2. In the resulting page select **JMS queue connection factory** (Figure 10-37) → **New**.

Figure 10-37 JMS queue connection factory

3. For bus name select **SCA.APPICATION.esbCell.Bus** from the drop-down list. Set name to `purchaseOrderQCF`. Set JNDI name to `jms/purchaseOrderQCF` (Figure 10-38) and click **OK** → **Save** → **Save**.

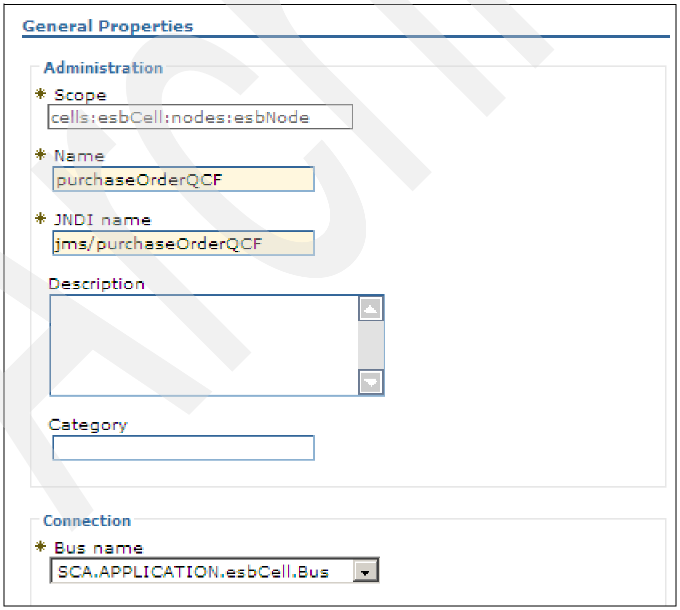


Figure 10-38 purchaseOrderQCF queue connection factory

10.4.4 Create the JMS activation specification

The activation specification is stored in JNDI and registers a queue with the mediation module message listener. Our activation specification is configured to use the jms purchaseOrderInputQueue.

1. In the navigator click **Resources** → **JMS Providers** → **Default Messaging** (Figure 10-32 on page 360).
2. Click **JMS activation specification** (Figure 10-39) → **New**.

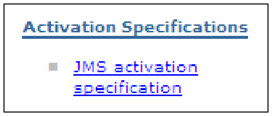


Figure 10-39 JMS Queue activation specification

3. For bus name select **SCA.APPLICATION.esbCell.Bus** from the drop-down list. Set name to `purchaseOrderAS`. Set JNDI name to `jms/purchaseOrderAS`. Set Destination JNDI name to `jms/purchaseOrderInputQueueS` (Figure 10-38 on page 363). Click **OK** → **Save** → **Save**.

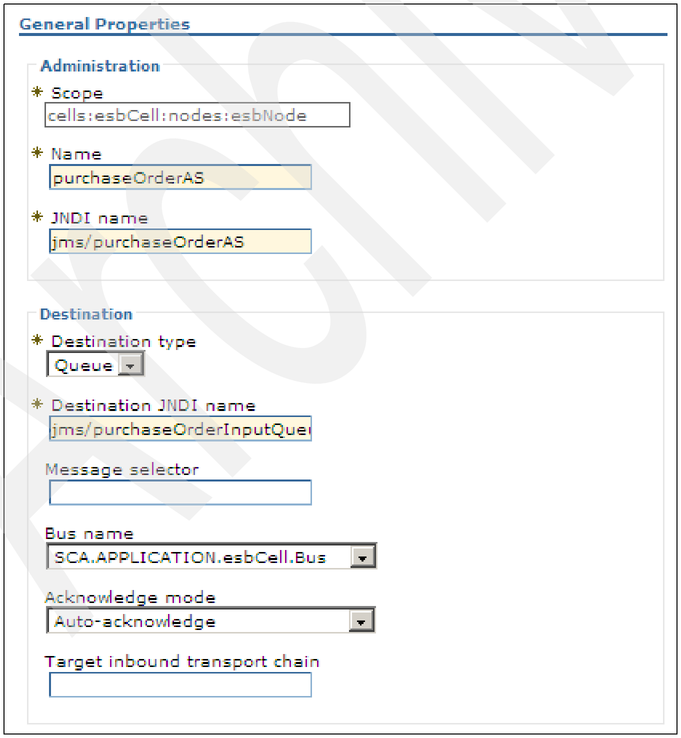


Figure 10-40 JMS activation specification

4. Close the Administrative Console.

10.5 Build the mediation module

The mediation module performs two functions:

- Protocol switches a JMS request to a SOAP/HTTP request.
- Maps the PartsPriceService to an interface that is called the JMS client.

First we create the mediation module. Then we build the assembly diagram with imports, exports, and the mediation component.

10.5.1 Create the mediation module

The mediation module contains the logic to connect a service requester to a service provider.

1. Switch to the Business Integration view and click **File** → **New** → **Other** → **Mediation Module** → **Next**. Type `PurchaseOrderMediation` for the module name. Ensure that the target runtime is set to **WebSphere ESB Server v6.0** (Figure 10-41) and click **Finish**.

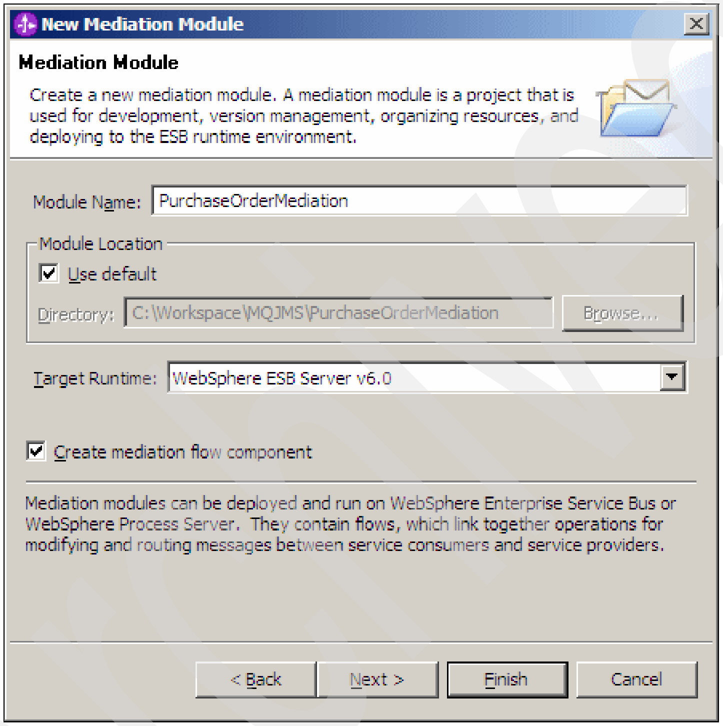


Figure 10-41 Create PurchaseOrderMediation module

2. Expand the **PurchaseOrderMediation** project and double-click **Assembly Diagram** to open the Assembly Editor. Rename the mediation component Mediation1 `PurchaseOrderMediation` (Figure 10-42).

Figure 10-42 PurchaseOrderMediation component

10.5.2 Import the PartsPrice service

The mediation calls the PartsPrice service. We need to create an import and reference it from the mediation.

1. Click the **PurchaseOrderMediation** project → **Show Files**. Respond **YES** to the prompt. This opens the Physical Resources view.
2. Find the **PartsPrice** project → **WebContent** → **WEB-INF** → **wsdl**. Copy **PartsPrice.wsdl**. Switch to the Business Integration perspective. Paste PartsPrice.wsdl into the PurchaseOrderMediation project.

   This populates the project with a Web service port, an interface, and two datatypes.
3. Open the PurchaseOrderMediation assembly diagram. Drag **PartsPrice** from the Web Service Ports folder onto the Assembly editor. Select **Import with Web Service Binding** → **OK** (Figure 10-43) → **Save All**.

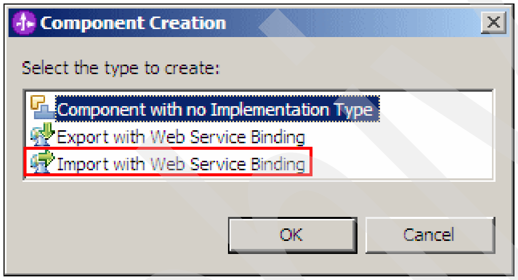


Figure 10-43 Import creation Binding options

10.5.3 Import the PartRequest data type

The PartRequest.xsd file contains the PartRequest datatype, which was used to build the XML document to be received by PurchaseOrderMediation.

Click the **PurchaseOrderMediation** project → **Import** → **File system**. Find the PartRequest.xsd file and click **OK**. This creates the business object PartRequest in the data types folder.

10.5.4 Create PartsRequest interface

Before creating the export for mediation to receive the JMS message from the ITSO-AUTOPOJMSClient,we need to define the interface to be used.

1. Select the **Interfaces** folder → **New** → **Interface**. Name the interface `PartsRequest`.

2. Select the icon to create a new request/reply operation and define the input and output parameters as shown in Figure 10-44.

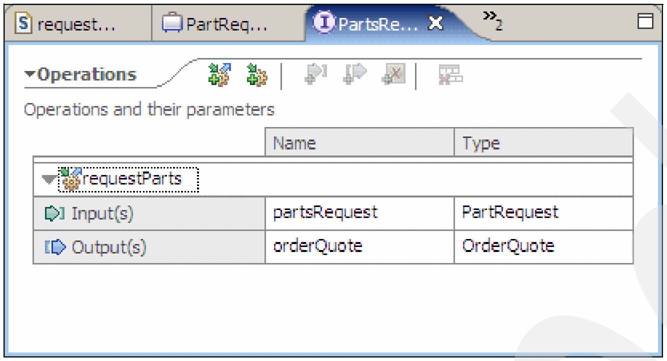


Figure 10-44 requestparts interface

> **Note:** Make sure that the namespace of the interface matches the namespace of the root element in the XML document to be received, and the name of the method is the name of the root element. In our case the namespace needs to be changed to http://PurchaseOrderMediation/RequestForParts.

10.5.5 Add the PartsRequest interface to PurchaseOrderMediation

Drag the **PartsPrice** interface from the Interface folder and drop it on the PurchaseOrderMediation in the Assembly diagram (Figure 10-45). Click **Save All**.

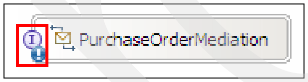


Figure 10-45 Added PartsPrice interface to PurchaseOrderMediation

10.5.6 Creating a JMS Export binding

Now create the export that describes how the mediation module will receive JMS request messages.

1. Right-click **PurchaseOrderMediation** and select **Generate Export** → **Message Binding** → **JMS Binding** (Figure 10-46).

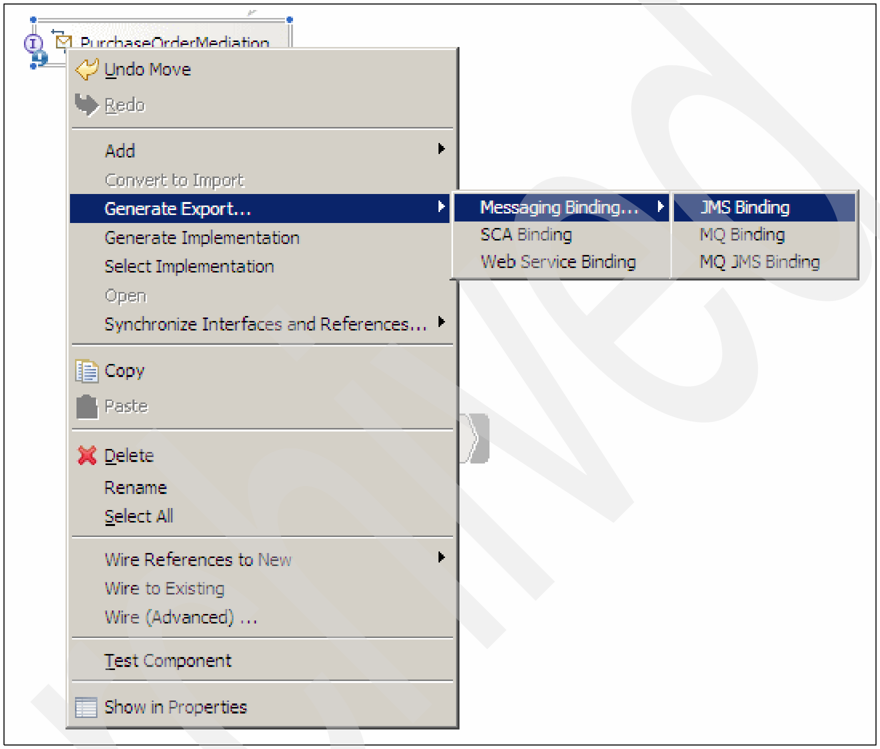


Figure 10-46 Generate JMS Messaging binding

The JMS Export Binding dialog opens.

2. If you followed the manual steps in 10.4, “Define WebSphere ESB server resources” on page 356, change the End-point Configuration radio button to Use pre-configured messaging provider resources. Otherwise leave the wizard to define the resources automatically. For manual configuration provide the three JNDI fields from Table 10-1 on page 356.

3. In Serialization, select Business Object **XML** using **JMSTextMessage** → **OK** (Figure 10-48). The export is wired to the mediation in the assembly diagram (Figure 10-47).

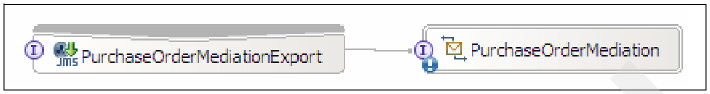


Figure 10-47 Export wired to mediation

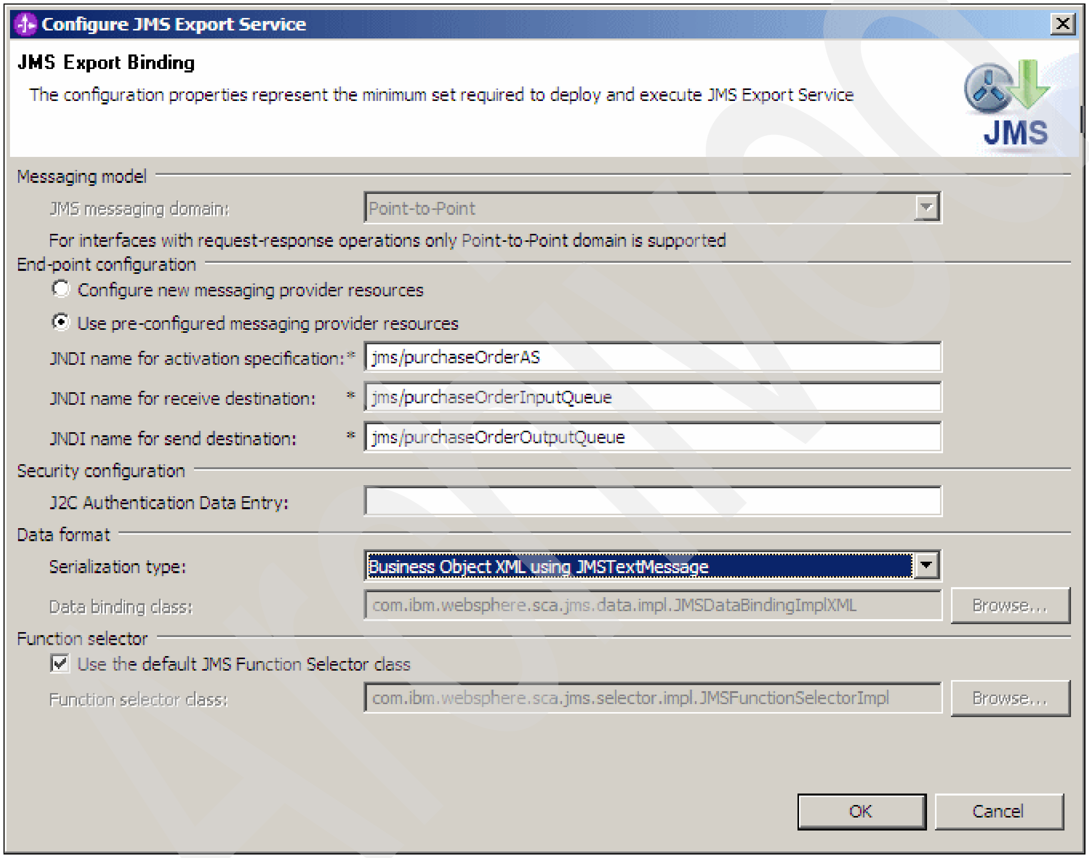


Figure 10-48 JMS export binding with manual configuration of the messaging provider

4. Open the Response Connection tab. In Select Configuration View Option select the **Specify JNDI Name for preconfigured Messaging Provider Resources** radio button and enter `jms/purchaseOrderQCF` (Figure 10-49). Click **Save All**.

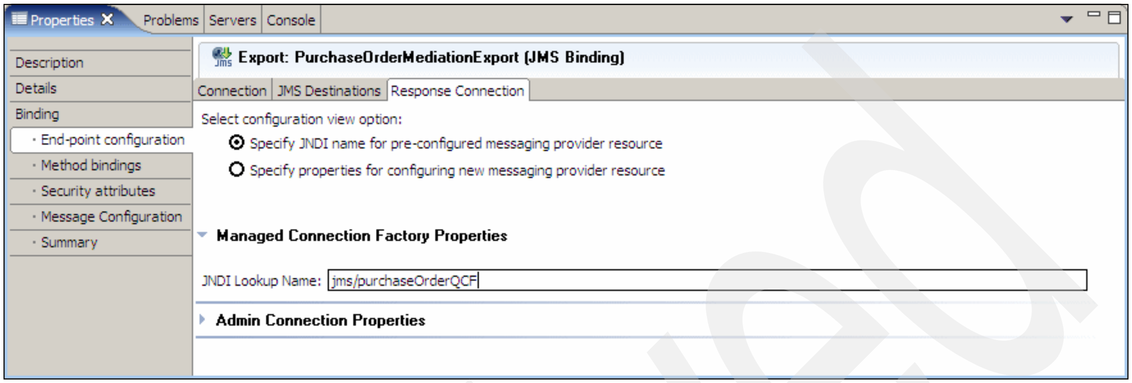


Figure 10-49 JMS Response Connection tab

10.5.7 Complete the component wiring

Connect the PartPriceImport to the PurchaseOrdermediation by creating a wire from the mediation to the import and generating a reference to the import in the mediation.

Move the mouse over the edge of PurchaseOrderMediation and grab the yellow handle to connect it to the PartPriceImport. Respond **OK** to create the matching reference (Figure 10-50).

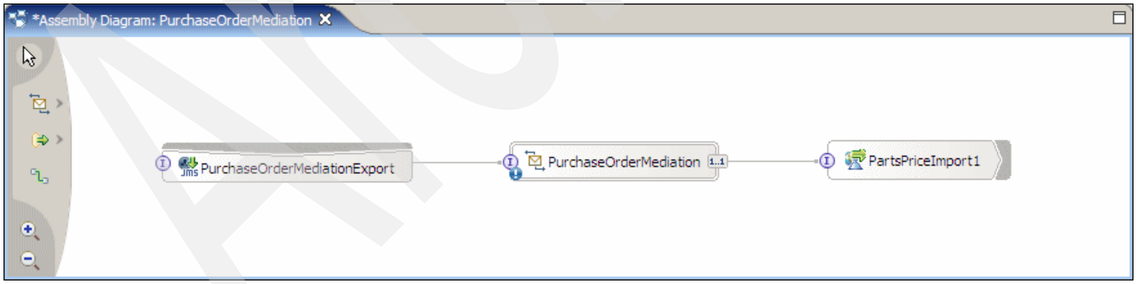


Figure 10-50 Wired Assembly diagram

10.5.8 Create the mediation flows

To do this:

1. Right-click **PurchaseOrderMediation** and select **Generate Implementation** → **OK** to store the mediation flow in the default folder.
2. In the Operation connections panel, wire requestParts to order() (Figure 10-51).

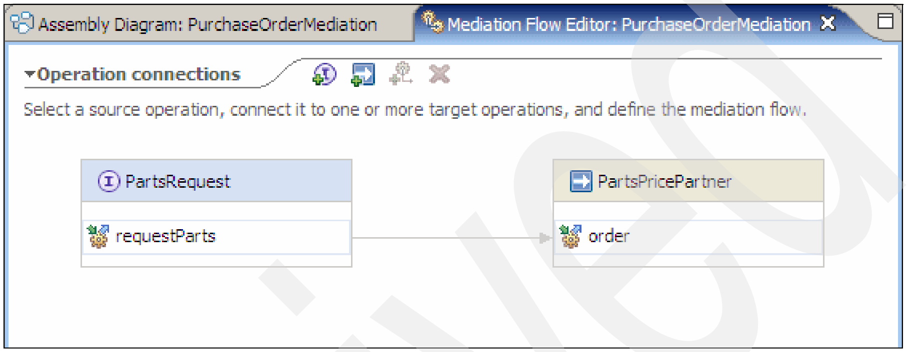


Figure 10-51 Mediation Operation connections

3. In the Request tab wire an XML style sheet between the Input and the Callout tab (Figure 10-52).

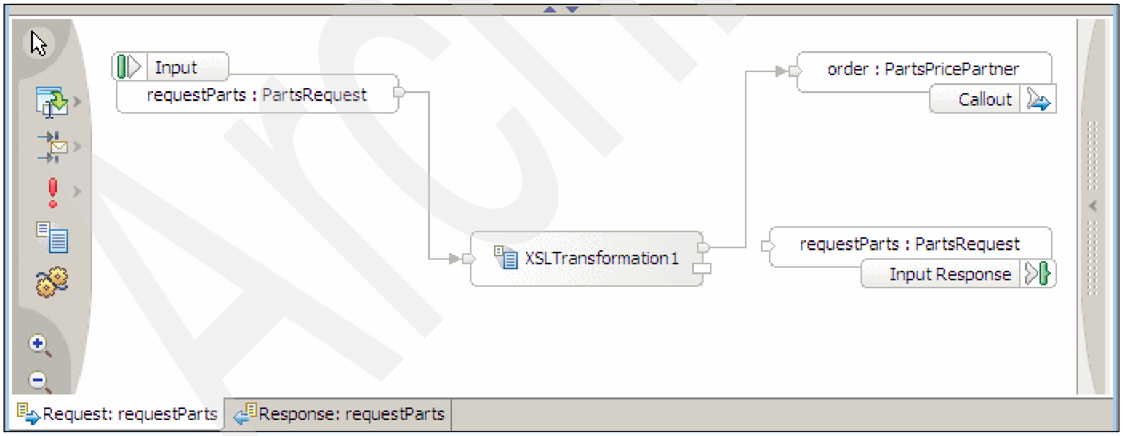


Figure 10-52 Wire Operation connection request

4. On the Response tab wire an XML stylesheet between the Callout Response and the Input Response. Click **Save All**.

10.5.9 Create the style sheet transformations

In the request flow, select **XSLTransformation1**. In its Properties view select the **Details** tab → **New**. In the new style sheet wire up the partName and number as shown in Figure 10-53.

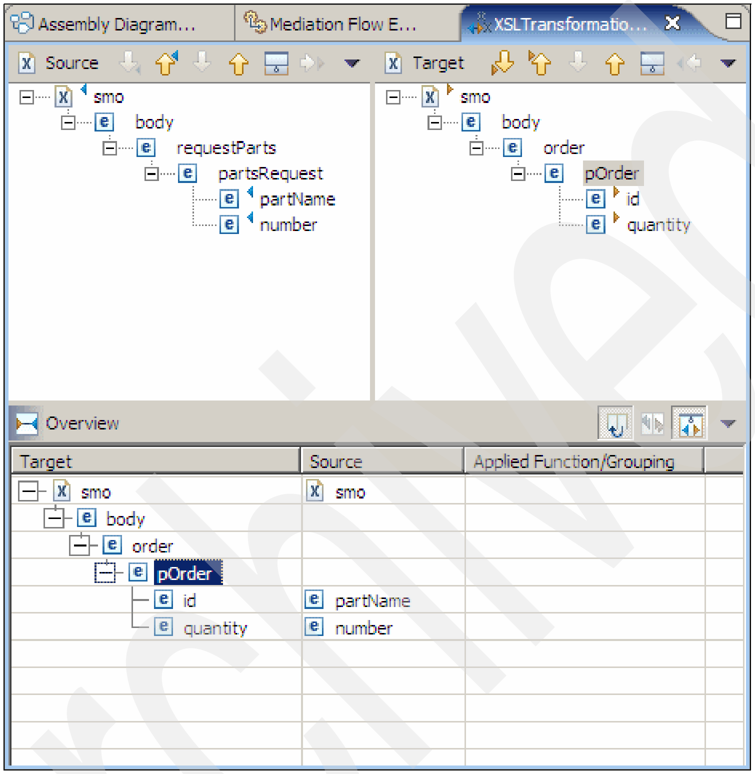


Figure 10-53 requestParts stylesheet

Repeat this procedure to wire up the price in the response flow and click **Save All**.

10.6 Test the solution

Now we have the three components required to test the JMS to SOAP Web service invocation.

In the Servers view right-click **WebSphere ESB Server v6.0** and select **Add and Remove Projects**. Add the three projects shown in Figure 10-54, and click **Finish** → **Publish** by clicking the publish icon.

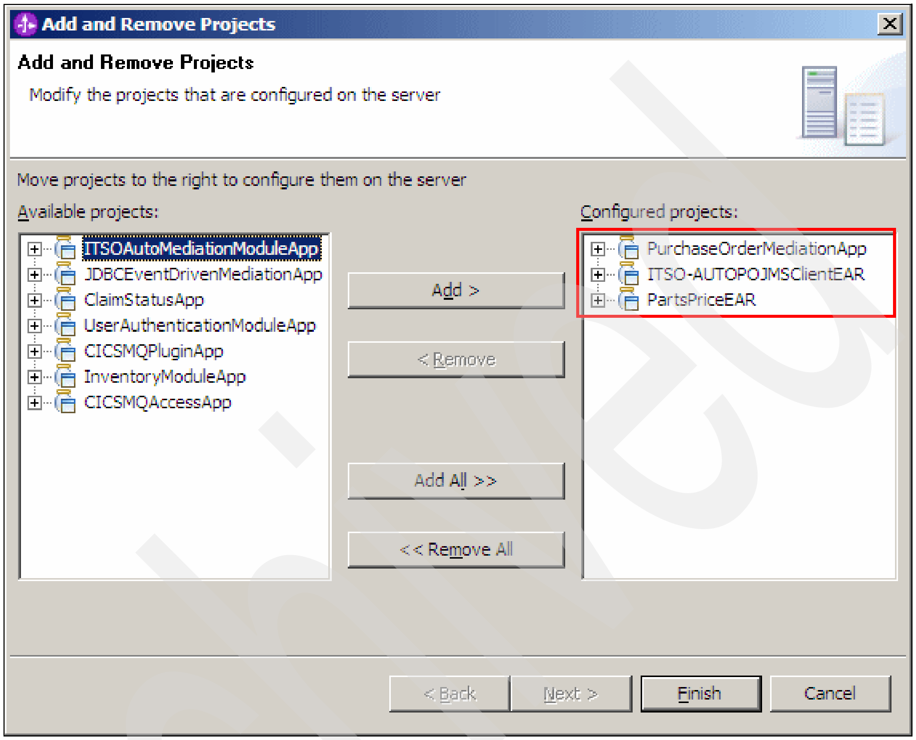


Figure 10-54 Add and remove projects to WebSphere ESB Server

10.6.1 Unit test the Web service and mediation

It is a good idea to test the Web service and the mediation before testing the JSP client. This does not test the JMS binding, but it verifies that the rest of the flow is working.

1. In the Assembly diagram click in the white space → **Test Module**.

2. Select the interface you wish to test and enter valid data (Figure 10-55).

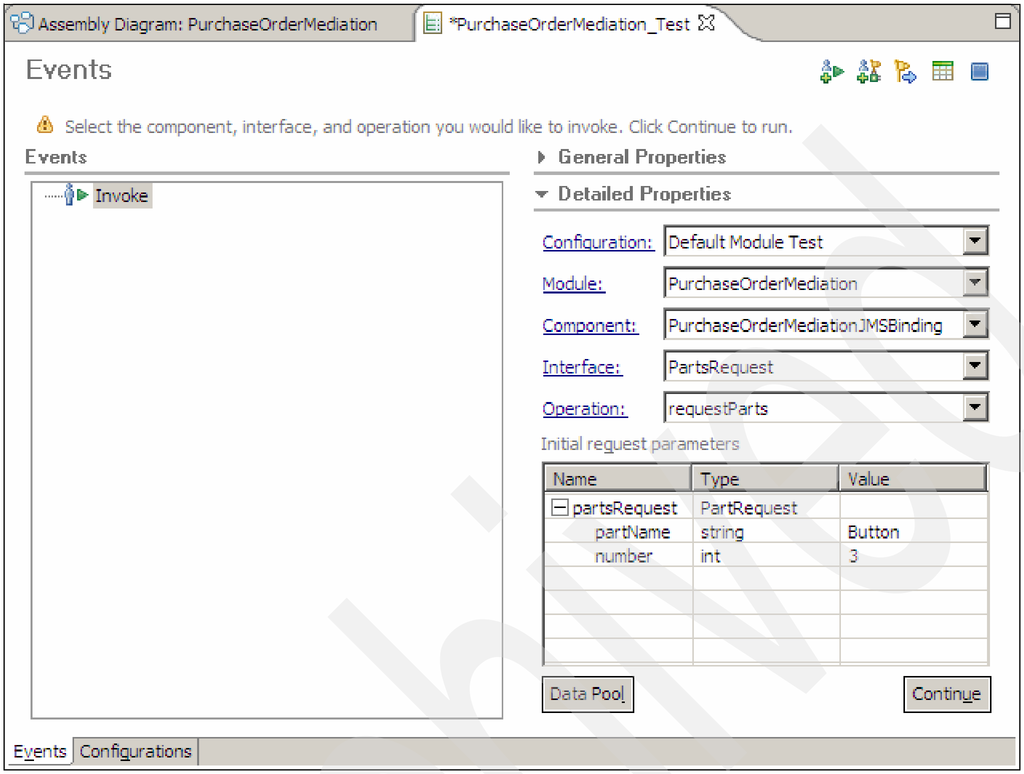


Figure 10-55 Test module

3. Click **Continue** to see the results (Figure 10-56).

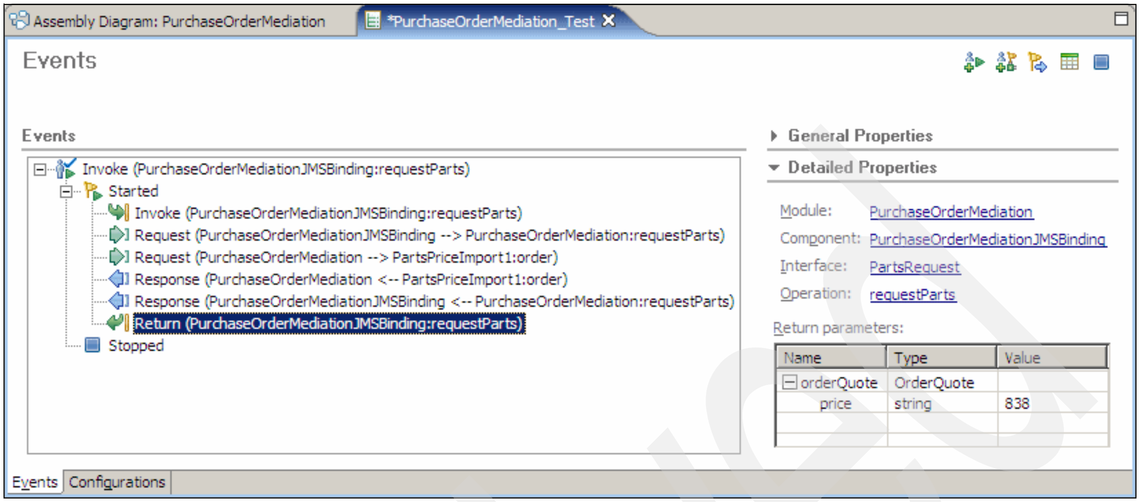


Figure 10-56 Results from Test Module

10.6.2 Test using the JSP and JMS message binding

To do this:

1. Restart the WebSphere ESB server to ensure that the activation specification is registered with the mediation module.
2. Open a Web browser and enter the URL:

   `http://localhost:9080/ITSO-AUTOPOJMSClient/ITSO-AutoFindPartsPrice.jsp.`

ITSO-AutoFindPartsPrice.jsp will be displayed (Figure 10-57).

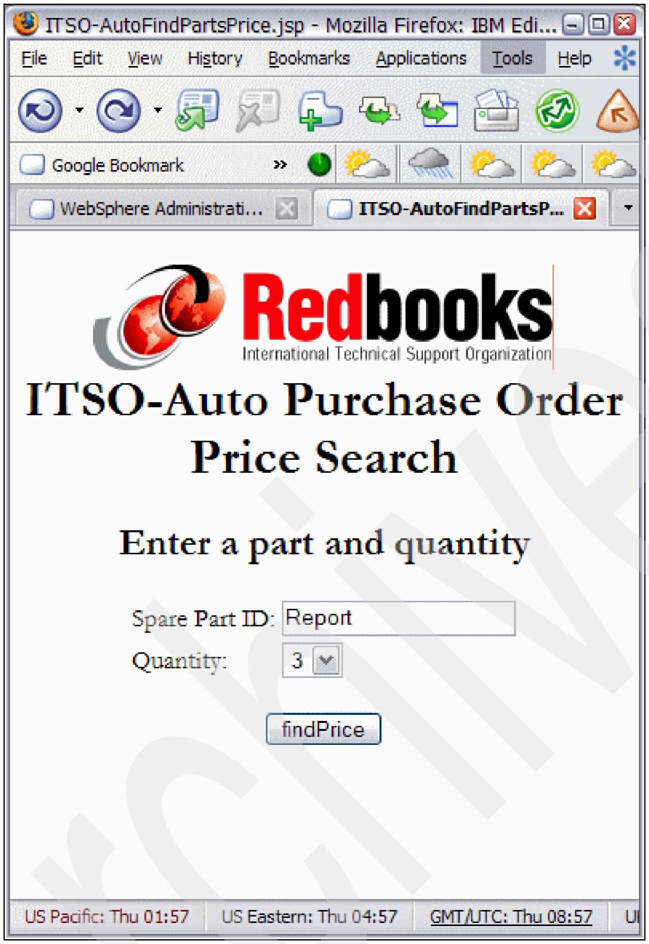


Figure 10-57 ITSO-AutoFindPartsPrice.jsp request page

3. Enter a spare part ID. Select a quantity and click **findPrice**.

   If the Web service was invoked correctly, a success message with the price of the part is displayed (Figure 10-58).

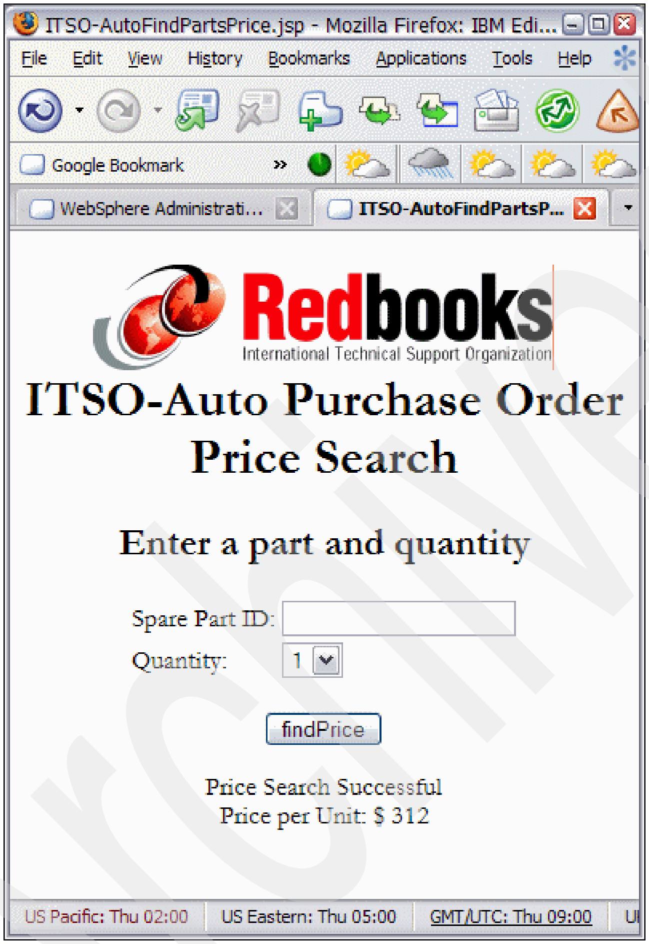


Figure 10-58 ITSO-AutoFindPartsPrice.jsp response with results

10.6.3 Conclusion

In this chapter we showed how custom applications can be integrated using JMS/XML and SOAP/http as transports. The JMS/XML message contained an XML document, which was transformed by the WebSphere Enterprise Service Bus mediation in to a SOAP/Http request.

11

Event-driven integration using a JDBC adapter

In 5.2, “Case 2: banking and fraud management” on page 134, we introduced the requirements of ITSO-BANK to integrate its in-house transaction system with a recently introduced fraud management system. The bank implemented a procedure for writing suspicious transactions to a database table.

In 6.4, “Connection pattern 4: event-driven integration” on page 157, we explored the requirements and described the WebSphere adapter for JDBC.

This chapter follows on from Chapter 6, “Connection patterns” on page 143, and guides you through the process for setting up a development and test environment for the WebSphere Adapter for JDBC for process event-driven service requests.

We cover the following topics:

- Overview of the development environment.
- Prepare the development environment.
- Create the ATT database and fraud table.
- Import metadata from the database.
- Configure the mediation module.
- Test the application.

11.1 Overview of the development environment

In Chapter 5, “Business scenarios” on page 129, we described a typical integration scenario where a bank, ITSO-BANK, is integrating a new Fraud Management System (FMS) with its existing core banking system (Figure 11-1).

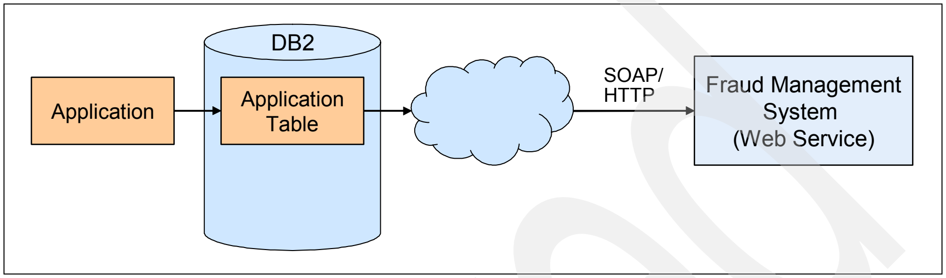


Figure 11-1 Connection required for scenario

In the FMS, events are triggered by records being written to a database table (in our case the fraud table within the Automatic Transaction Tracking (ATT) database). In Chapter 6, “Connection patterns” on page 143, we described the architecture of a solution based on the WebSphere Enterprise Service Bus and the WebSphere adapter for JDBC (Figure 11-2).

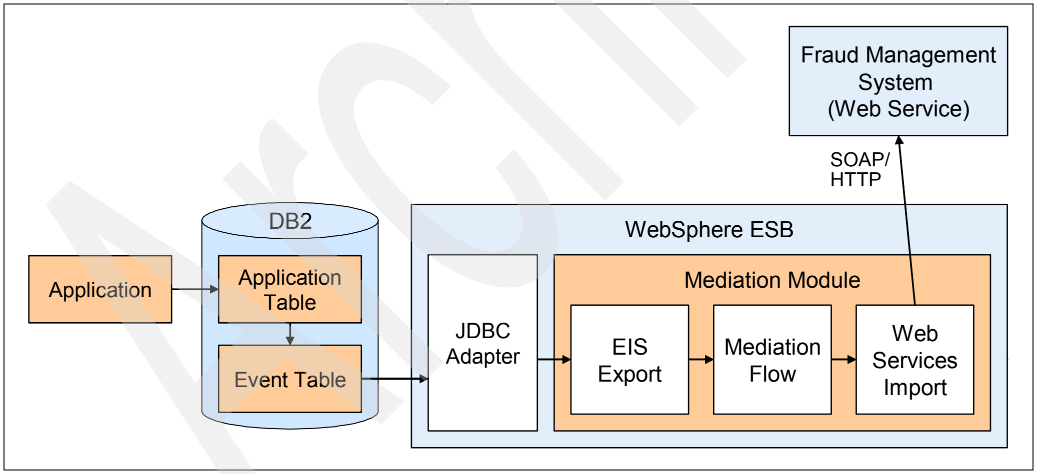


Figure 11-2 Triggering service requests using the JDBC Adapter

The development environment to build the solution is shown in Figure 11-3.

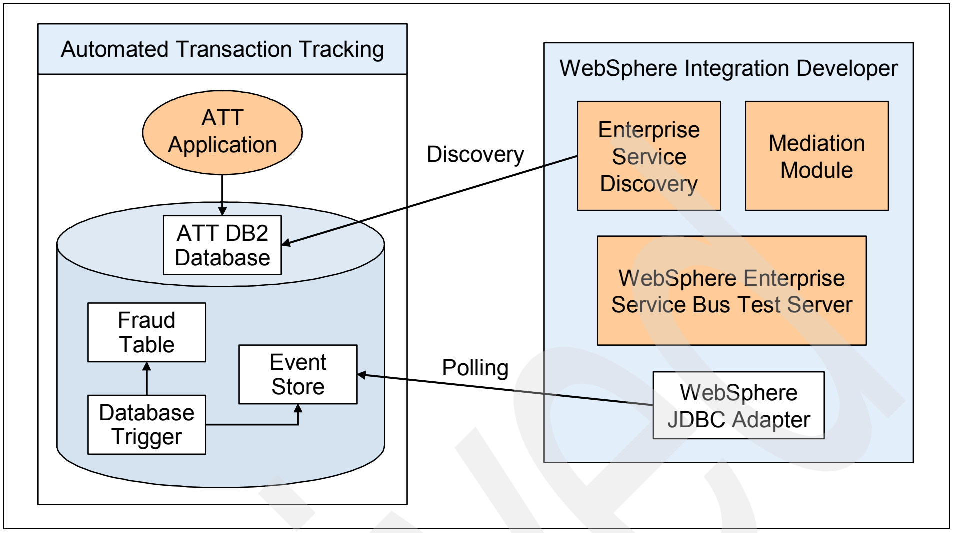


Figure 11-3 Development environment

WebSphere Integration Developer v6.0.2 provides an integrated WebSphere Enterprise Service Bus integrated test environment for testing our composed applications. We have also utilized the Enterprise Service Discovery within WebSphere Integration Developer for discovering enterprise data and creating business objects.

We have used WebSphere Adapter for JDBC v6.0.2 to process events triggered by database updates.

Note: Although the WebSphere Adapter for JDBC is bi-directional between EIS and applications, we only discussing inbound processing in this chapter.

11.2 Prepare the development environment

This section describes how to create the database, database tables, and associated triggers. Throughout this section reference is made to locations with variable substitutions. Table 11-1 lists the variables directories and their associated directories.

> **Note:** When installing the software packages, the default install directory is usually C:\Program Files\IBM.... For our installation we have always installed directly under C:\IBM.

Table 11-1 Variables and their locations

Variable	Windows® location
<JDBC_ADAPTER_HOME>	C:\IBM\ResourceAdapters\JDBC
<DB2_INSTALL_ROOT>	C:\IBM\SQLLIB
<WID_INSTALL_ROOT>	C:\IBM\WebSphere\WID\602

11.2.1 Software prerequisites

The following list gives details of the Windows products used for the development and testing in our environment:

- DB2 UDB Universal Database 8.2 with FixPak 5.
- WebSphere Integration Developer v6.0.2 with latest fixes. Included in the install is the integrated WebSphere Enterprise Service Bus Test Environment.
- WebSphere Adapter for JDBC 6.0.2.

11.2.2 Set up the development environment

This section details the processes involved in setting up the WebSphere Integration Developer, which we use for development.

Start the WebSphere Integration Developer and when prompted set a workspace (Figure 11-4).

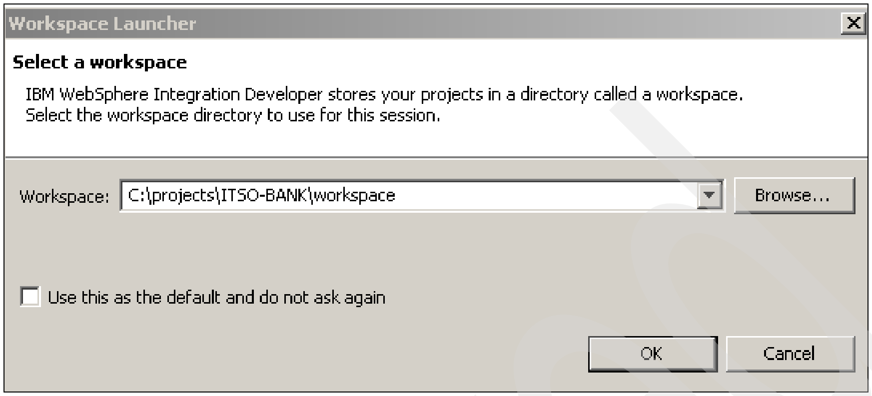


Figure 11-4 Starting WebSphere Integration Developer

To enable the Enterprise service Discovery facility, import the Resource Adapter Resource Archive (RAR) file into our development environment.

Select **File** → **Import**. Select **RAR** file → **Next**. On the Connector Import panel browse to <WID_INSTALL_ROOT> \Resource Adapters\JDBC\deploy. Select **CWYBC_JDBC RAR** → **OK**. Deselect the **Add module to EAR project** and ensure that **WebSphere ESB Server v6.0** is selected as the target server (Figure 11-5). Click **Finish** and respond **Yes** to change perspective.

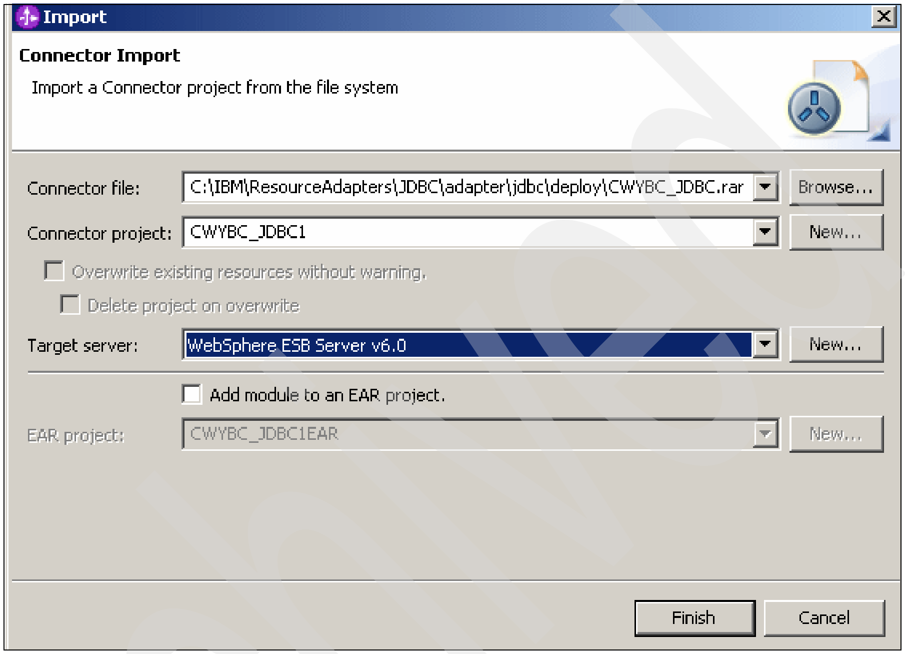


Figure 11-5 Importing the resource archive file

There are also two external jar files that are required by the adapter project for enabling connections to the relevant database. For our project we use a DB2 database so that the external files are specific to that database.

Select **Project Explorer** → **Connector Projects**. Right-click **CWYBC_JDBC** and select **Properties** → **Java Build Path** → **Libraries** tab → **Add External Jars**. Select **db2jcc.jar** and **db2jcc_license_cu.jar** in <DB2_INSTALL_ROOT>\java\, and click **Open** → **OK** (Figure 11-5 on page 384).

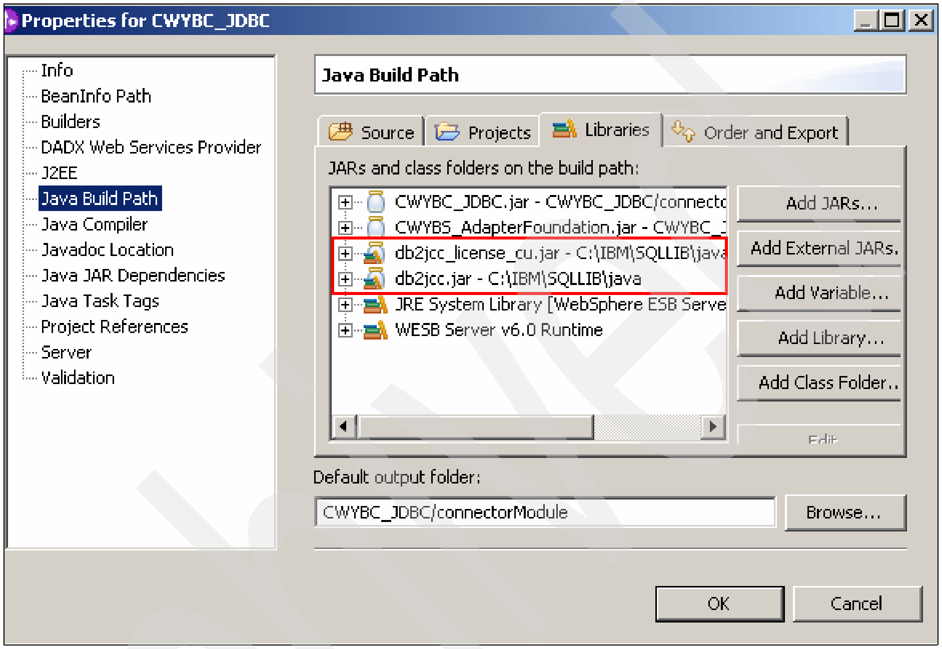


Figure 11-6 Adding the external files into the development project

11.2.3 Summary

In this section we configured the JDBC resource adapter and added the Java libraries needed to connect to the selected DB2 database middleware.

11.3 Create the ATT database and fraud table

This section shows you how to create the DB2 database for the Fraud solution using the scripts to:

- Create a fraud table that we will use for holding suspicious transactions.
- Add the event store table, used for holding updates to the database.
- Include triggers onto the fraud table that will fire whenever a record is added, updated, or deleted.

> **Note:** The object_name field in the script should correspond with the application-specific business graph created in the mediation.

The scripts have all be collected in one file and are run using the DB2 command-line processor, or from the DB/2 Control center.

To create a database we must first ensure that the DB2 Administration server is started. Click **Start** → **All Programs** → **IBM DB2** → **Command Line Tools** → **Command window**. Type `db2admin start`.

Before creating any database objects you must connect to the database using the user name and password that you specified when installing DB2. For our example we specified admin for the user name and itso4you as the password.

```
db2 create database attdb
db2 connect to attdb user Admin using itso4you
```

Either use the createAllObjects.sql script (instructions can be found in Appendix A, “Additional material” on page 499) or paste the text in Example 11-1 into a file.

Change to the directory where you have saved the script and issue the command:

```
db2 -tvf createAllObjects.sql
```

There will be errors the first time the script runs, and reference is made to objects that have not been created before.

Example 11-1 Create the database objects

```
SET SCHEMA Admin;
DROP TABLE fraud;
CREATE TABLE fraud (
   fraudid VARCHAR(10) NOT NULL PRIMARY KEY,
   transid VARCHAR(20),
```

```
   account VARCHAR(25),
   status VARCHAR(10),
   event_time TIMESTAMP default CURRENT TIMESTAMP NOT NULL );

DROP TABLE wbia_jdbc_eventstore;
CREATE TABLE WBIA_JDBC_EventStore (
   event_id INTEGER NOT NULL GENERATED ALWAYS AS IDENTITY
   (START WITH 1, INCREMENT BY 1, NO CACHE ) PRIMARY KEY,
   xid VARCHAR(200),
   object_key VARCHAR(80) NOT NULL,
   object_name VARCHAR(40) NOT NULL,
   object_functionVARCHAR(40) NOT NULL,
   event_priority INTEGER NOT NULL,
   event_time TIMESTAMP default CURRENT TIMESTAMP NOT NULL,
   event_status INTEGER NOT NULL,
   event_comment VARCHAR(100) );

DROP TRIGGER EVENT_CREATE;
DROP TRIGGER EVENT_UPDATE;
DROP TRIGGER EVENT_DELETE;

CREATE TRIGGER event_create
AFTER INSERT ON FRAUD REFERENCING NEW AS N
FOR EACH ROW MODE DB2SQL
INSERT INTO wbia_jdbc_eventstore
   (object_key, object_name, object_function, event_priority, event_status)
   VALUES (N.fraudid, 'InAdminFraudBG', 'Create', 1, 0);

CREATE TRIGGER event_udpate
AFTER UPDATE ON FRAUD REFERENCING NEW AS N
FOR EACH ROW MODE DB2SQL
INSERT INTO wbia_jdbc_eventstore
   (object_key, object_name, object_function, event_priority, event_status)
   VALUES (N.fraudid, 'InAdminFraudBG', 'Update', 1, 0);

CREATE TRIGGER event_delete
AFTER DELETE ON FRAUD REFERENCING OLD AS O
FOR EACH ROW MODE DB2SQL
INSERT INTO wbia_jdbc_eventstore
   (object_key, object_name, object_function, event_priority, event_status)
   VALUES (O.fraudid, 'InAdminFraudBG', 'Delete', 1, 0);
```

To view the results type `db2cc` in the command window to start the control center and click **OK** to accept the Advanced view default.

Select **All Databases** → **ATTDB** → **Tables** → **FRAUD** (Figure 11-7).

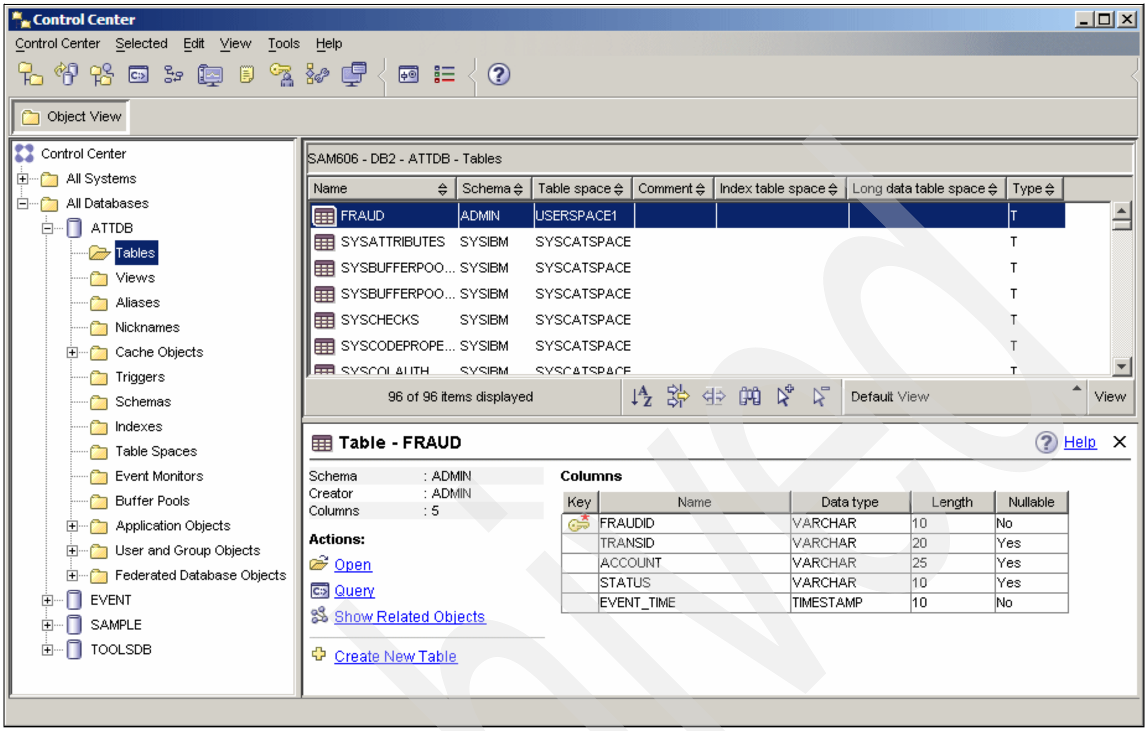


Figure 11-7 FRAUD table

Scroll down to view the WBIA_JDBC_EVENTSTORE (Figure 11-8).

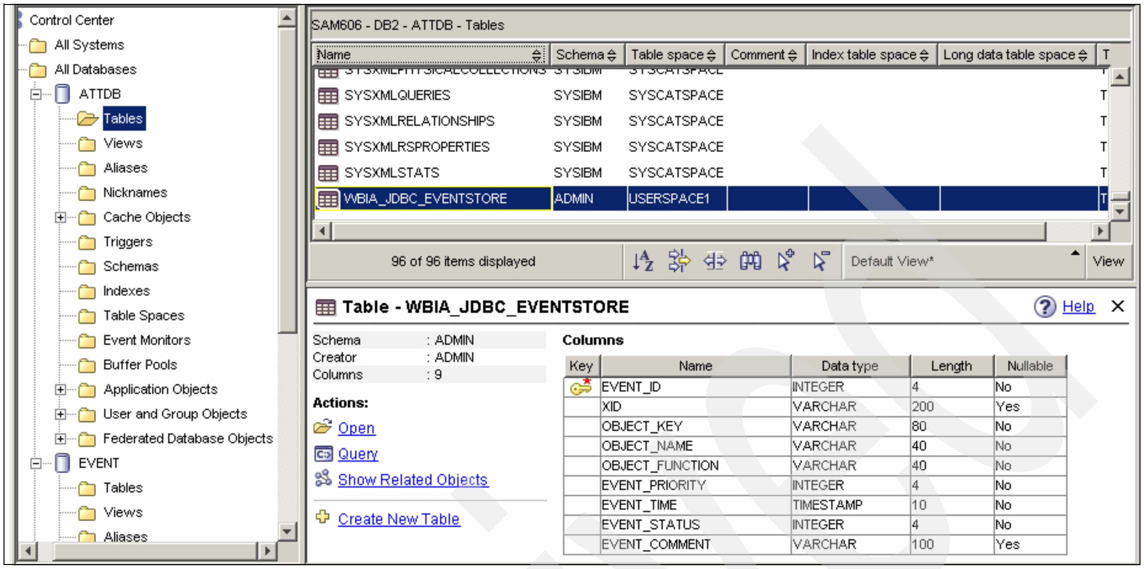


Figure 11-8 WBIA_JDBC_EVENTSTORE

Click the **Triggers** folder → **EVENT_CREATE** (Figure 11-9).

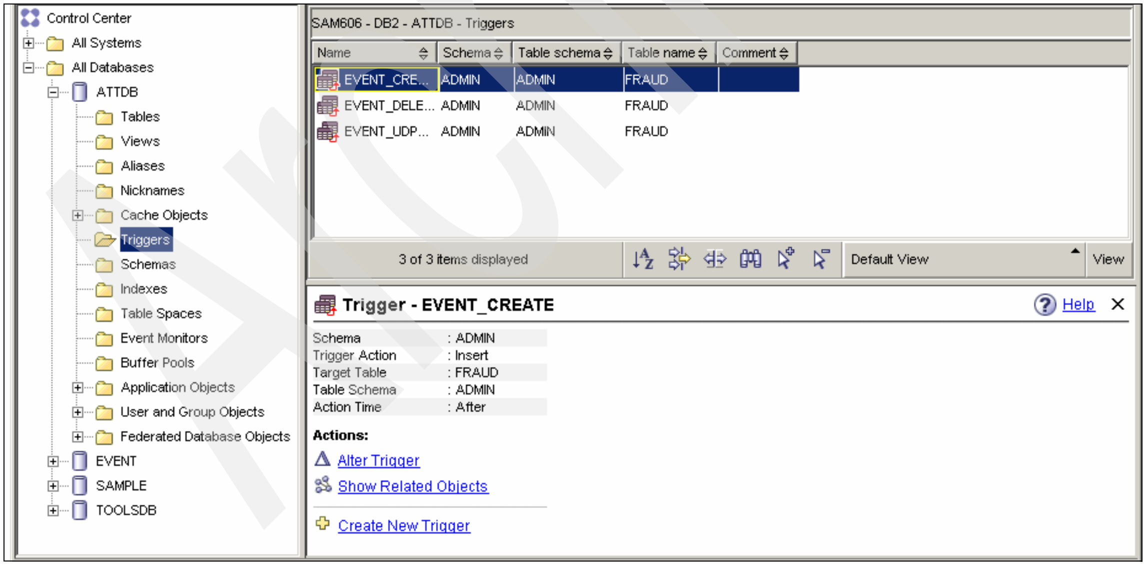


Figure 11-9 EVENT_CREATION trigger

11.3.1 Summary

In this section we created the database ATTDB, the FRAUD and WBIA_JDBC_EVENTSTORE tables, and three trigger events. The WBIA_JDBC_EVENTSTORE table is used to manage passing the events to the WebSphere Enterprise Service Bus.

11.4 Import metadata from the database

In this section we use the Enterprise Service Discovery utility to select the database objects to import as business objects. To do this we set the connection properties in the tool and then run a query. From the results of the query we select and configure the database objects to convert into business objects.

Switch to the Business Integration perspective and click **File** → **New** → **Enterprise Service Discovery**. Highlight the **JDBC EMD Adapter** we imported in the setup section (Figure 11-10) and click **Next**.

> **Note:** If the JDBC resource adapter is not displayed, use the Import Resource Adapter button to import the RAR file.

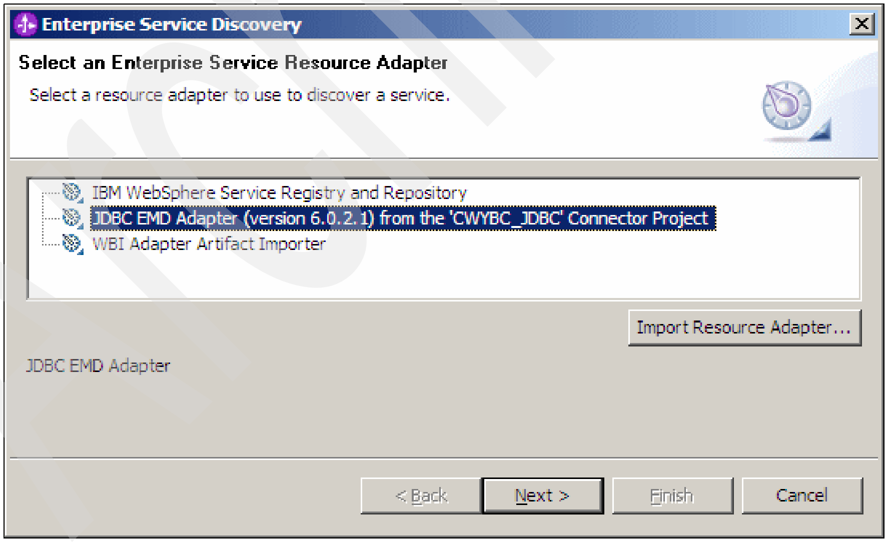


Figure 11-10 JDBC Resource Adapter

The Configure Settings for Discovery Agent dialog appears (Figure 11-11.)

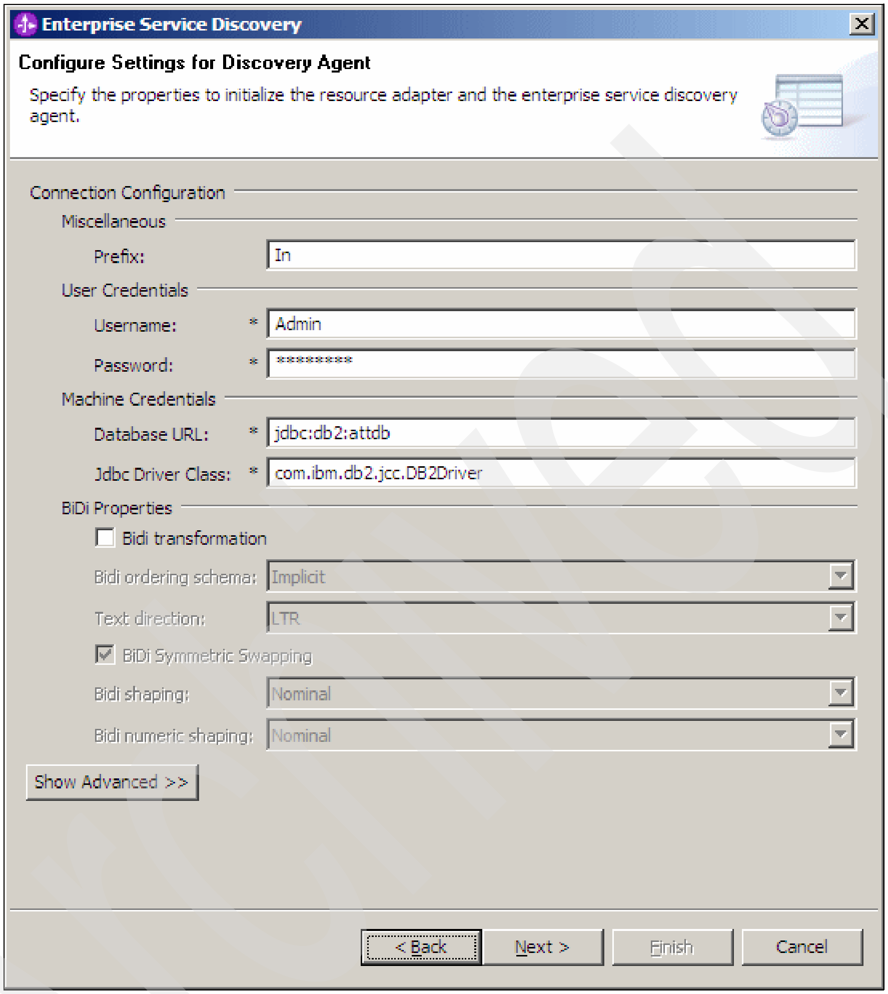


Figure 11-11 Connection Configuration settings

Enter the details shown in Table 11-2 and click **Next**.

Table 11-2 Attributes for discovery agent

Attribute	Value
Prefix	In
Username	Admin
Password	itso4you

Attribute	Value
Database URL:	jdbc.db2.attrb
Jdbc Driver Class	com.ibm.db2.jcc.DB2Drive r

With the connection established, invoke a query to find and discover the services we need. In a typical database scenario there are often a large number of databases and tables created under different schemas.

1. In the Find and Discover Enterprise service dialog click **Edit Query**.
2. In the Schema Name Filter enter the database schema name for your table. We used ADMIN. (Ensure that you use uppercase.). Click **OK**.

3. Click **Execute Query**, expand **ADMIN** → **Tables**. Highlight the **FRAUD** table and select **Add to import list** (Figure 11-12). Click **Next**.

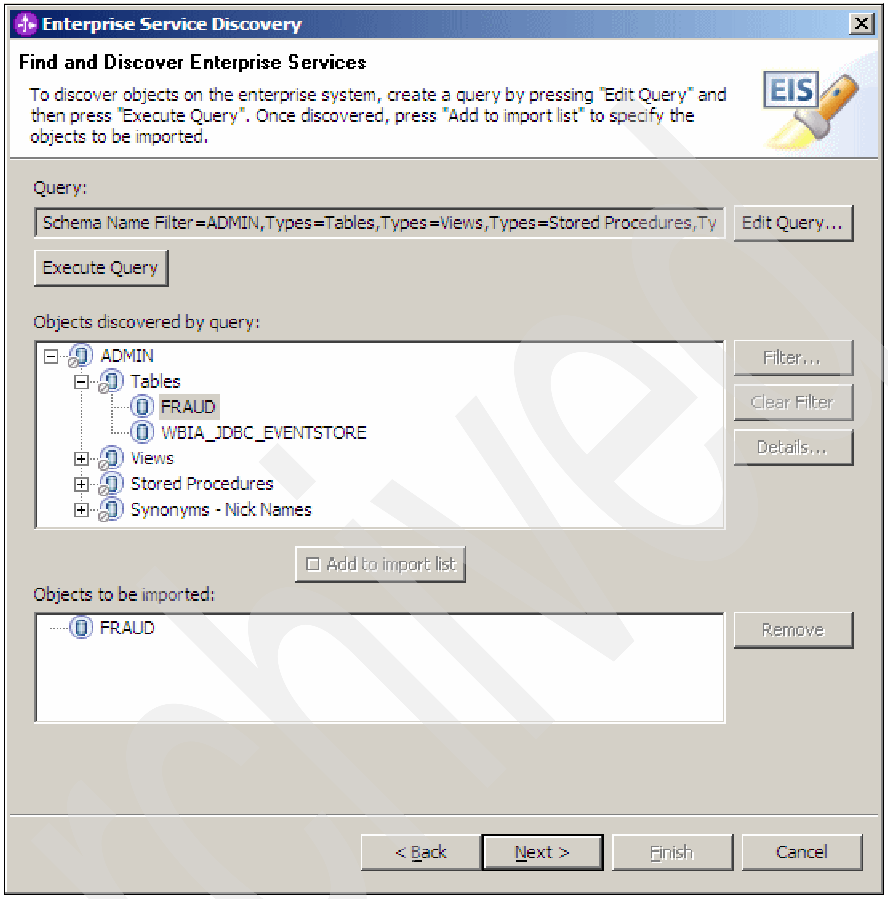


Figure 11-12 Importing the fraud object

4. In the Configure Objects pop-up dialog enter the business object location as `com/ibm/dataobjects` (Figure 11-13) and click **Next**.

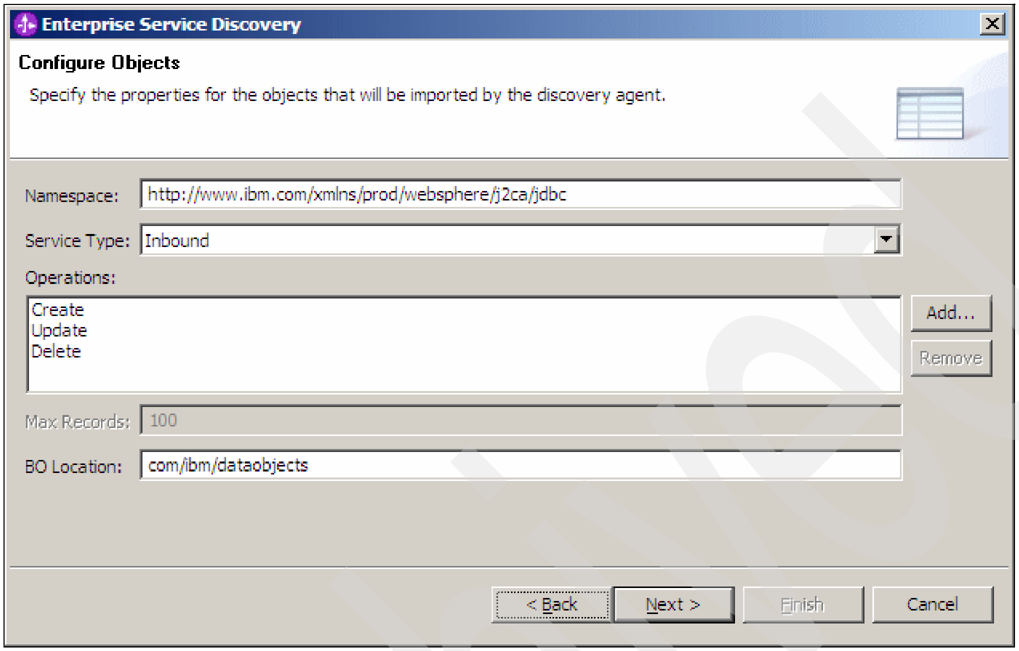


Figure 11-13 Configure Objects

5. In the Generate Artifacts window create a new Mediation project and module by selecting **New**. Check **Create a Mediation module project** and click **Next**. Enter `JDBCEventDrivenMediation` for the module name. The target runtime is **WebSphere ESB Server v6.0**. Click Finish (Figure 11-14).

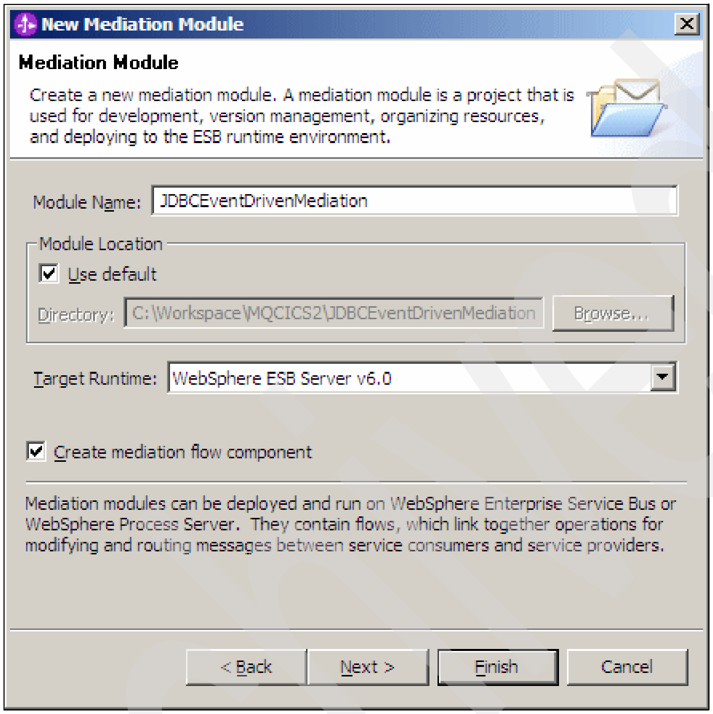


Figure 11-14 Create new Mediation project

6. Using the modified values from Table 11-3, generate the remaining artifacts (Figure 11-15 on page 396 and Figure 11-16 on page 397).

Table 11-3 Modified attributes in Generate Artifacts dialog

Attribute	Value
J2C Authentication Data Entry	esbNode/fraudAuth
Password	itso4you
Database Vendor	DB2

Note: The default name for the event table is WBIA_JDBC_EventStore. This is the name we specified in the SQL script for creating the event table. If you have created the event table with a different name, substitute your table name.

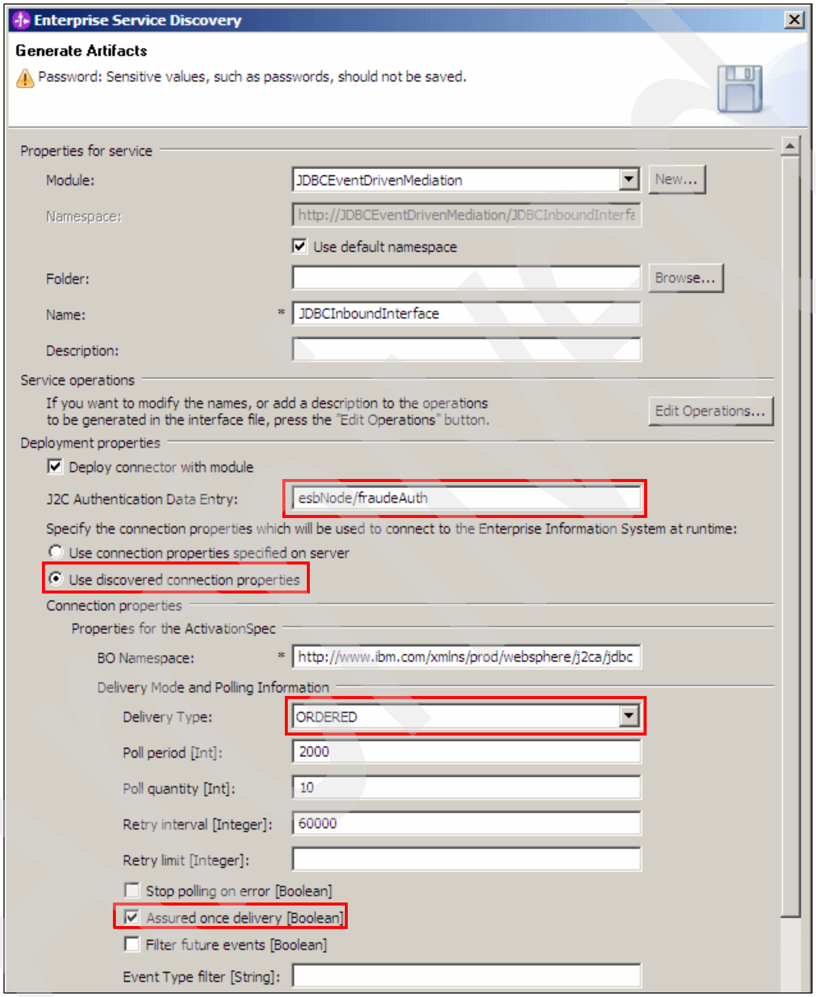


Figure 11-15 Generate Artifacts - part 1

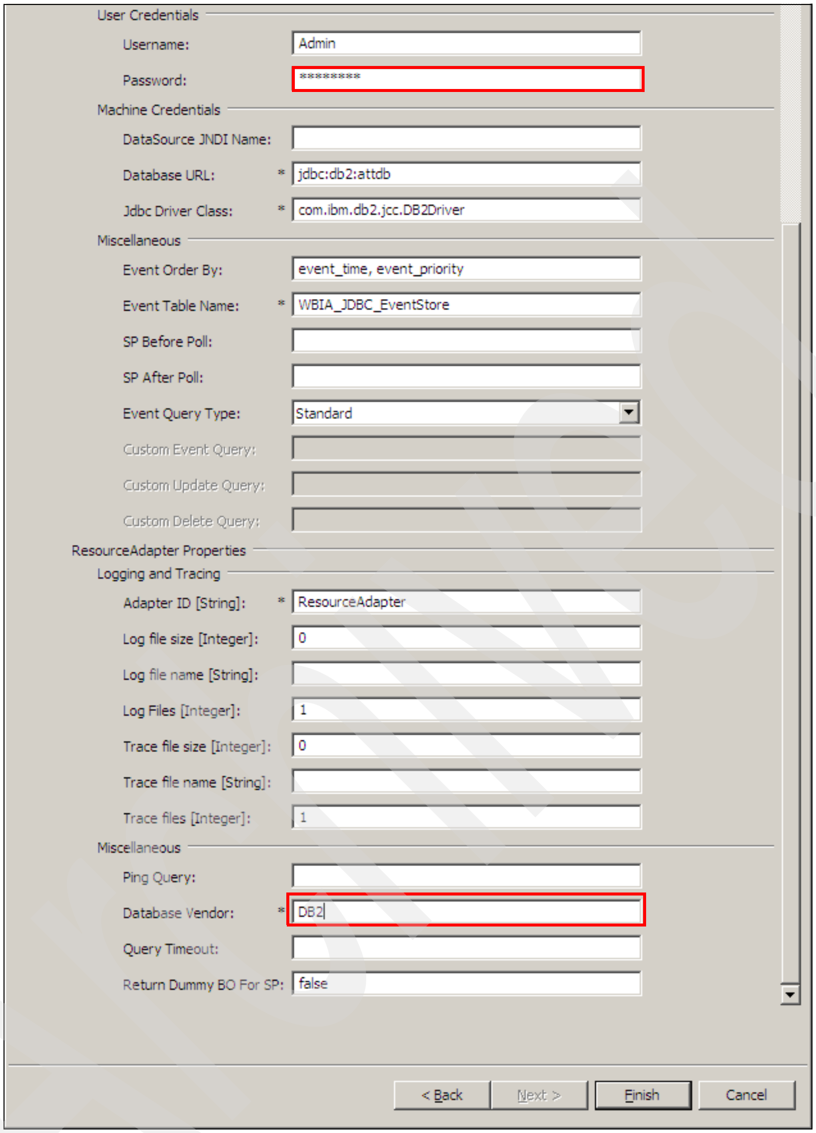


Figure 11-16 Generate attributes - part 2

Find the generated artifacts in the new JDBCEventDrivenMediation project.

Under the assembly diagram there is an EIS import (see 3.5.6, “EIS bindings” on page 97).

Under Datatypes there is a Business graph (inAdminFraudBG) and a business object (InAdminFraud) (Figure 11-17).

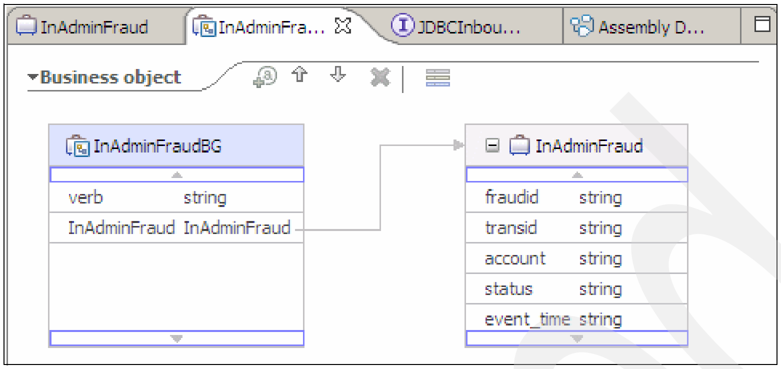


Figure 11-17 Generated business objects

Under Interfaces there is one interface (JDBCInboundInterface) with three operations (Figure 11-18).

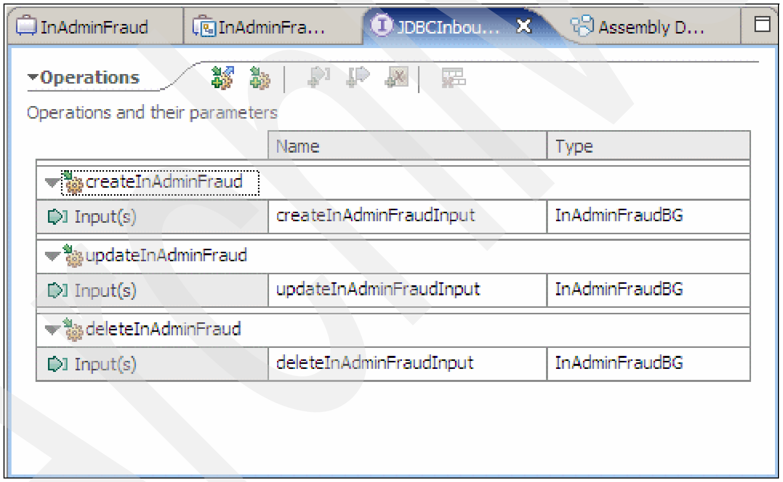


Figure 11-18 JDBCInboundInterface

During the configuration we checked the Assured Once Delivery flag. By checking this the activation specification property AssuredOnceDelivery is set to true, and an xid (transaction ID) value is set for each event in the event store. After an event is obtained for processing, the xid value for that event is updated in the event table. The event is then delivered to its corresponding endpoint, and subsequently deleted from the event table. If the database connection is lost or the application is stopped before the event can be delivered to the endpoint, the event may not be processed completely. In this case, the xid column ensures that the event is reprocessed and sent to the endpoint. Once the database connection is re-established or the adapter starts again, the adapter checks for events in the event table that have a value in the xid column.

11.4.1 Summary

In this section we used the Enterprise Service Discovery to connect to the database and create our service objects. Next we build our mediation modules to act on the generated business object.

11.5 Configure the mediation module

The mediation module JDBCEventDrivenMediation responds to the inbound events from the database.

1. Double-click the Assembly Diagram to open it (Figure Figure 11-19).

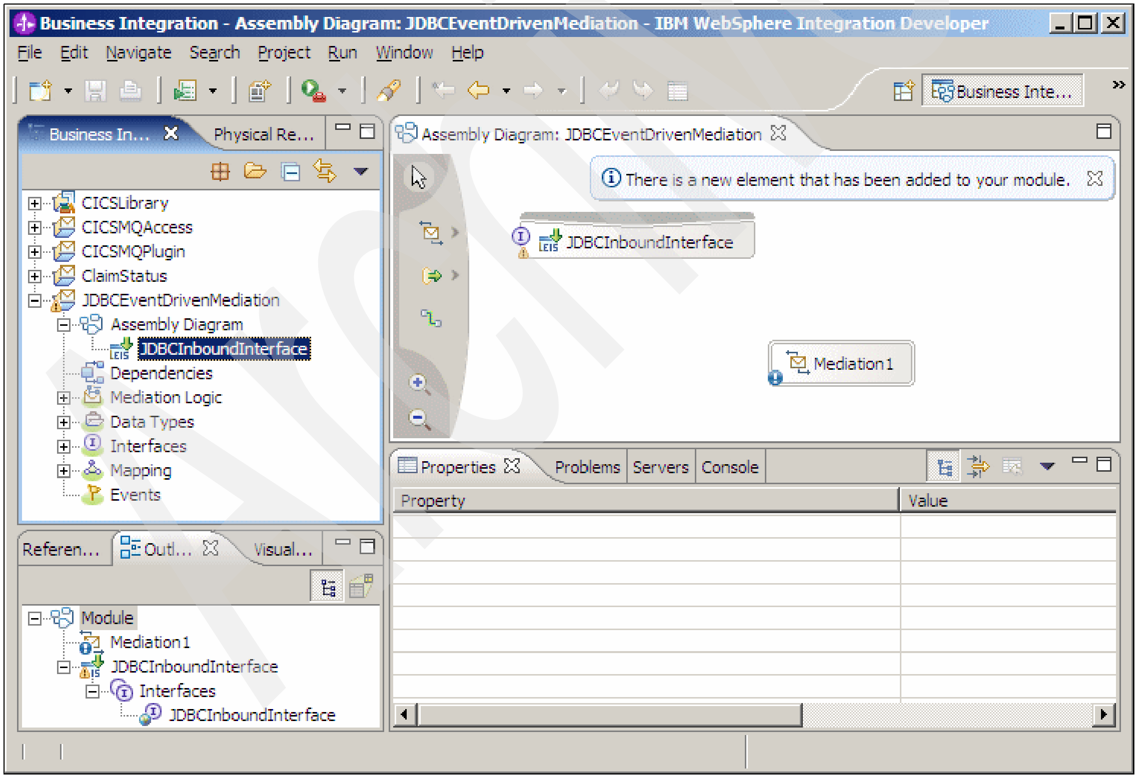


Figure 11-19 Assembly Diagram

2. Click **Mediation1** → **Rename** to **JDBCEventDrivenMediation**. Alternatively, open the **Properties** tab and amend the name property in the Name Field.

We add an interface to the mediation by using the same interface specified for the JDBC Adapter. By doing this, any operations that are defined for the adapter interface will be automatically implemented in our mediation.

Right-click **JDBCEventDrivenMediation** and click **Add** → **Interface** → **JDBCInboundInterface** (Figure 11-20) → **OK**.

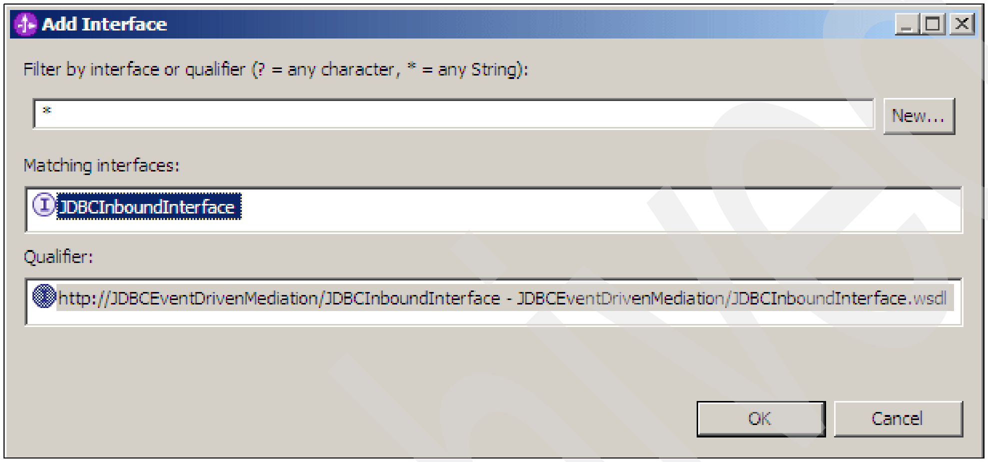


Figure 11-20 Specifying the Interface for the mediation

Wire together the components to form a mediation flow:

1. Right-click **JDBCEventDrivenMediation** and select **Wire to existing** → **OK**.

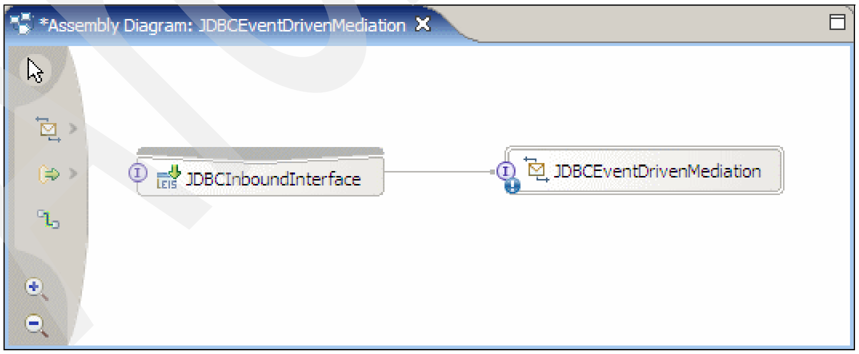


Figure 11-21 Wiring the components together

At this stage, although we have created the mediation, it is not implemented yet.

2. Double-click **JDBCEventDrivenMediation** and select **Yes** when asked to implement the component in the pop-up window.

   The generate Implementation Box will pop up asking for a folder for the Mediation Flow.

3. Expand **com\ibm** → **New Folder** → **mediations** (Figure 11-22) → **OK** → **OK**.

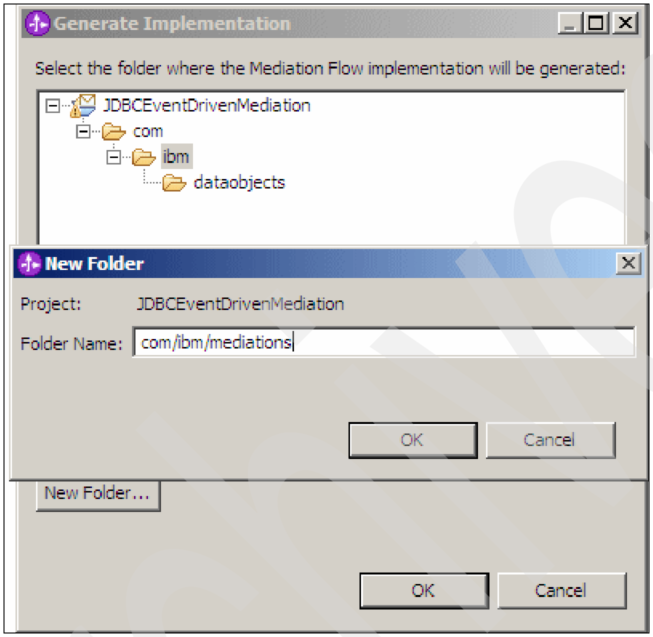


Figure 11-22 Folder specification for mediation

The Mediation flow editor opens for create, update, and delete operations. Select any of the flows.

For our demonstration project we do not do any further mediation, so wire the stop mediation primitive to stop the flow.

Ensure that you are in the top Operation connections window of the Mediation Flow Editor:

1. Click **JDBCInboundInterface** → **createInDb2AdminFraud**.

   The input for the interface is shown in the second window

2. From the palette in the second window drag the **Custom Mediation** button and drop it on the canvas (Figure 11-23).

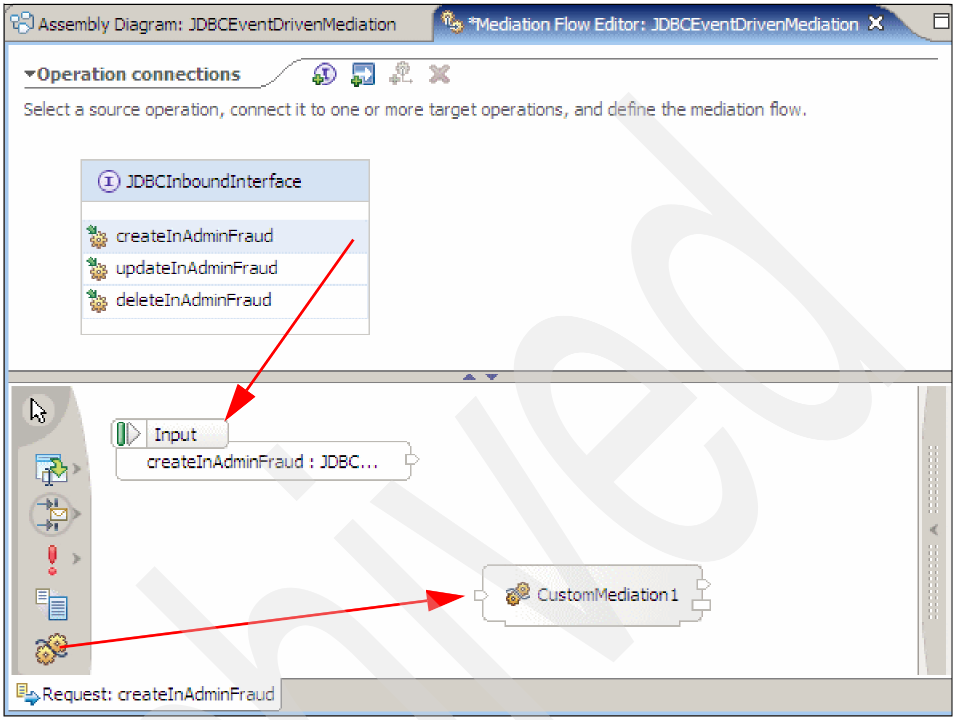


Figure 11-23 Drop CustomMediation primitive

3. From the palette in the second window click the **Fail** button. This will open two further buttons, Fail and Stop. Drag the **Stop** button from the left window and drop it on the canvas.
4. Rename the custom mediation to `CreateCustomMediation` and rename the Stop node `STOPCreate`.

5. Connect the *out* terminal of the Input node createInAdminFraud to the *in* terminal of CreateCustomMediation. Connect the out terminal of CreateCustomMediation to the in terminal of STOPCreate (Figure 11-24).

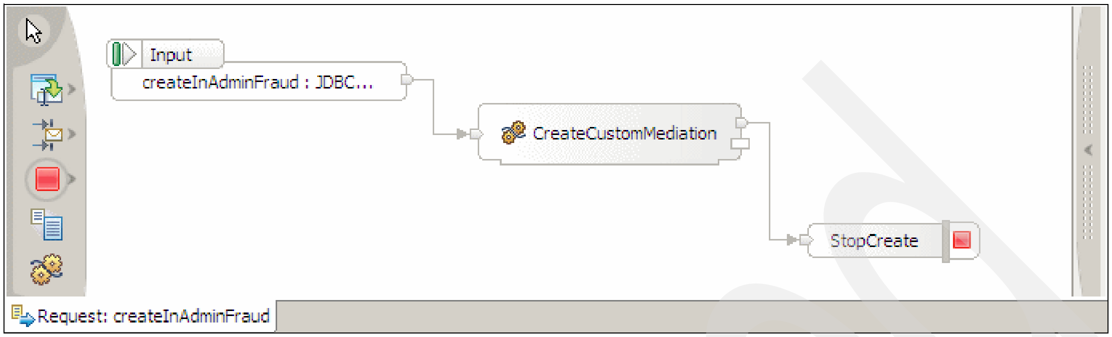


Figure 11-24 Operation connections

For our demonstration we simply display the contents of the inbound message using the CreateCustomMediation primitive.

1. Click the **CreateCustomMediation** primitive. Open its **Details** tab in the Properties view. Click **Java radio**. Select **/body** as the root.
2. Insert the mediation code from Example 11-2 and click **Save**.

Example 11-2 Code for create event mediation

```
System.out.println("Output of Create Event");

commonj.sdo.DataObject createAdminFraud = input1.getDataObject("createInAdminFraud");
commonj.sdo.DataObject createAdminFraudInput =
createAdminFraud.getDataObject("createInAdminFraudInput");
System.out.println("Verb is : " + createAdminFraudInput.getString("verb"));

commonj.sdo.DataObject inAdminFraud =
createAdminFraudInput.getDataObject("InAdminFraud");

System.out.println("fraudid is : " + inAdminFraud.getString("fraudid"));
System.out.println("transid is : " + inAdminFraud.getString("transid"));
System.out.println("account is : " + inAdminFraud.getString("account"));
System.out.println("status  is : " + inAdminFraud.getString("status"));
System.out.println("event_time  is : " + inAdminFraud.getString("event_time"));
return input1;
```

3. Repeat the steps for updateInAdminFraud, naming the mediation `UpdateCustomMediation` and the stop mediation `StopUpdate`. Insert the mediation code from Example 11-3.

Example 11-3 Code for update event mediation

```
System.out.println("Output of Update Event");
commonj.sdo.DataObject updateInAdminFraud =
input1.getDataObject("updateInAdminFraud");
commonj.sdo.DataObject updateInAdminFraudInput =
updateInAdminFraud.getDataObject("updateInAdminFraudInput");

System.out.println("Verb is : " + updateInAdminFraudInput.getString("verb"));

commonj.sdo.DataObject inAdminFraud =
updateInAdminFraudInput.getDataObject("InAdminFraud");
System.out.println("fraudid is : " + inAdminFraud.getString("fraudid"));
System.out.println("transid is : " + inAdminFraud.getString("transid"));
System.out.println("account is : " + inAdminFraud.getString("account"));
System.out.println("status  is : " + inAdminFraud.getString("status"));
System.out.println("event_time  is : " + inAdminFraud.getString("event_time"));
return input1;
```

4. Repeat the steps for deleteInAdminFraud, naming the mediation `DeleteCustomMediation` and the stop mediation `StopDelete`. Insert the mediation code from Example 11-4.

Example 11-4 Code for delete event mediation

```
System.out.println("Output of Delete Event");
commonj.sdo.DataObject deleteInAdminFraud =
input1.getDataObject("deleteInAdminFraud");
commonj.sdo.DataObject deleteInAdminFraudInput =
deleteInAdminFraud.getDataObject("deleteInAdminFraudInput");
System.out.println("Verb is : " + deleteInAdminFraudInput.getString("verb"));
commonj.sdo.DataObject inAdminFraud =
deleteInAdminFraudInput.getDataObject("InAdminFraud");
System.out.println("fraudid is : " + inAdminFraud.getString("fraudid"));
return input1;
```

11.5.1 Summary

In this section we added a simple mediation flow to handle the three inbound operations corresponding to the three events defined on the FRAUD table in the ATTDB database.

11.6 Test the application

In the previous sections we configured the necessary artifacts for the WebSphere Adapter for JDBC. We also created the mediations and interfaces within the development environment.

In this section we:

- Configure the J2C Authentication Alias.
- Deploy to WebSphere Enterprise Service Bus.
- Test the solution.
- View the output of our generated events.

11.6.1 Configure the J2C Authentication Alias

To do this:

1. In the Business Integration view close all the editors. Open the **Server** view, right-click **WebSphere ESB server v6.0**, and click **Start**.

   Check that the server starts without errors by monitoring the output of the Console view.

2. Right-click **WebSphere ESB server v6.0** and click **Run Administration console**. Or Open a browser window and enter `http://localhost:9060/ibm/console`.

3. Enter any user name and log in. Select **All** in the Available task filters from the left menu and click **Apply**.

4. Expand the **Security** tab and click **Global Security**. Expand the **JAAS Configuration** tab under Authentication (Figure 11-25).

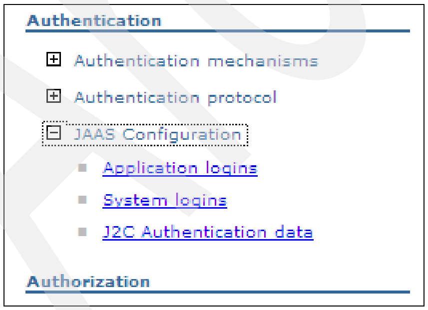


Figure 11-25 Select J2C Authentication data

5. Click **J2C Authentication data** and wait for the Global security box to appear. Click **New** (Figure 11-26).

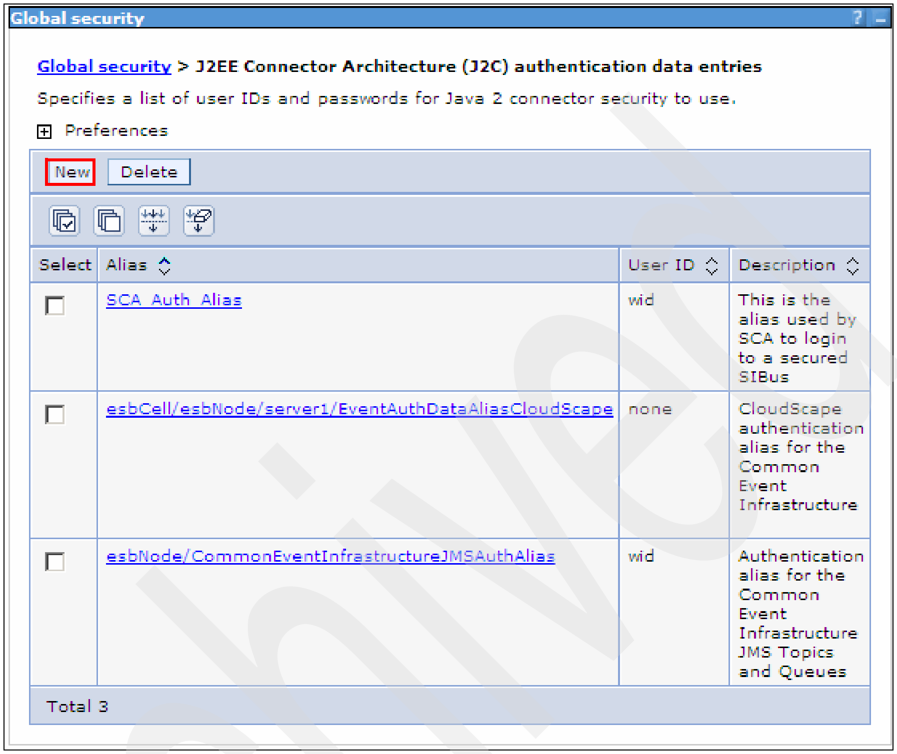


Figure 11-26 Add new authentication alias

6. Add the authentication values from Table 11-4 and click **OK** (Figure 11-27 on page 407) → **Save** → **Save**.

Table 11-4 JAAS authentication values

Attribute	Value
Alias	fraudAuth
User Id	Admin
Password	itso4you

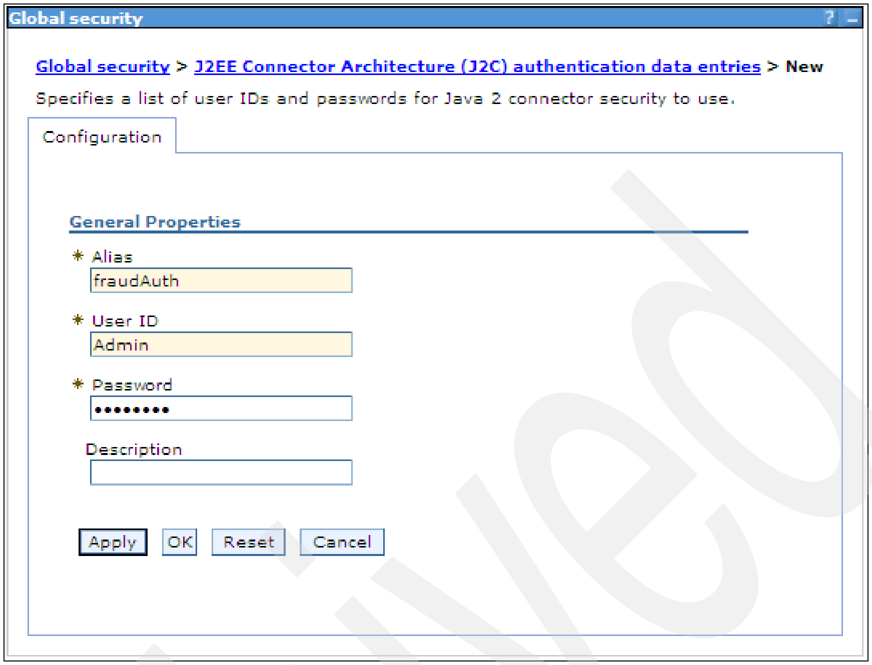


Figure 11-27 Authentication alias values

The Global Security windows appears with the alias esbNode/fraudAuth added (Figure 11-29 on page 409).

> **Tip:** Check the J2C authentication alias on the server matches the alias defined on the Security Attributes tab of the JDBCInboundInterface export.

7. On the Global Security panel click **User Registries** → **Local OS** and provide the server user ID and password (Figure 11-28).

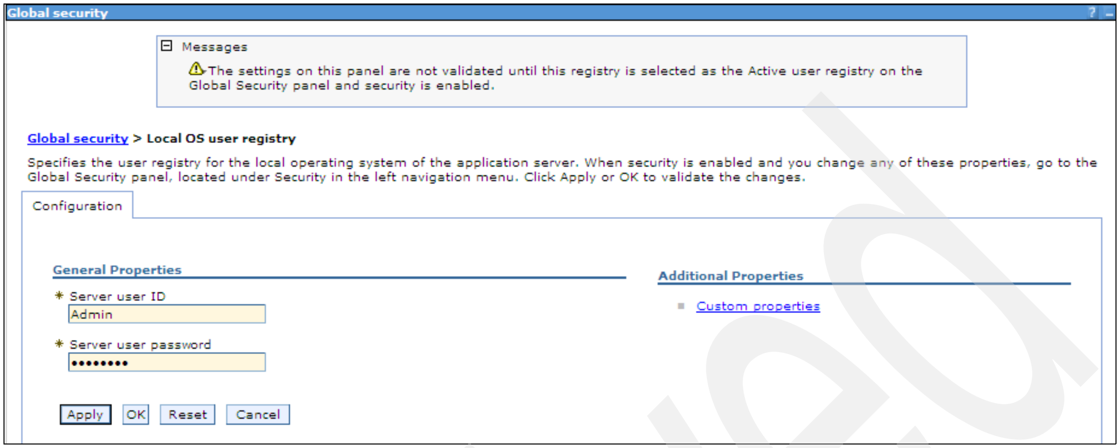


Figure 11-28 Supply user ID and password for local OS registry

8. Log out of the Administration Console.

11.6.2 Deploy to WebSphere Enterprise Service Bus

To do this:

1. Copy the db2 driver .jars (db2cc.jar and db2jcc_license_cu.jar), which were referenced on the external jar build path in 11.2.2, “Set up the development environment” on page 382, into the <WID_INSTALL_ROOT>\runtimes\bi_v6\lib\ext directory (the same place where we copied mqcicsmediation.jar) from <DB2_INSTALL_ROOT>\java.

> **Note:** Alternatively, you can copy the DB2 .jar files into the connector project and point the Java external library path at them there. They will be deployed in the .ear file. See “Getting connected with WebSphere Integration Developer adapters: Part 3, An introduction to the WebSphere Adapter for JDBC,” found at:
>
> `http://www.ibm.com/developerworks/websphere/techjournal/0706_gregory/0706_gregory.html`

2. Restart the server.

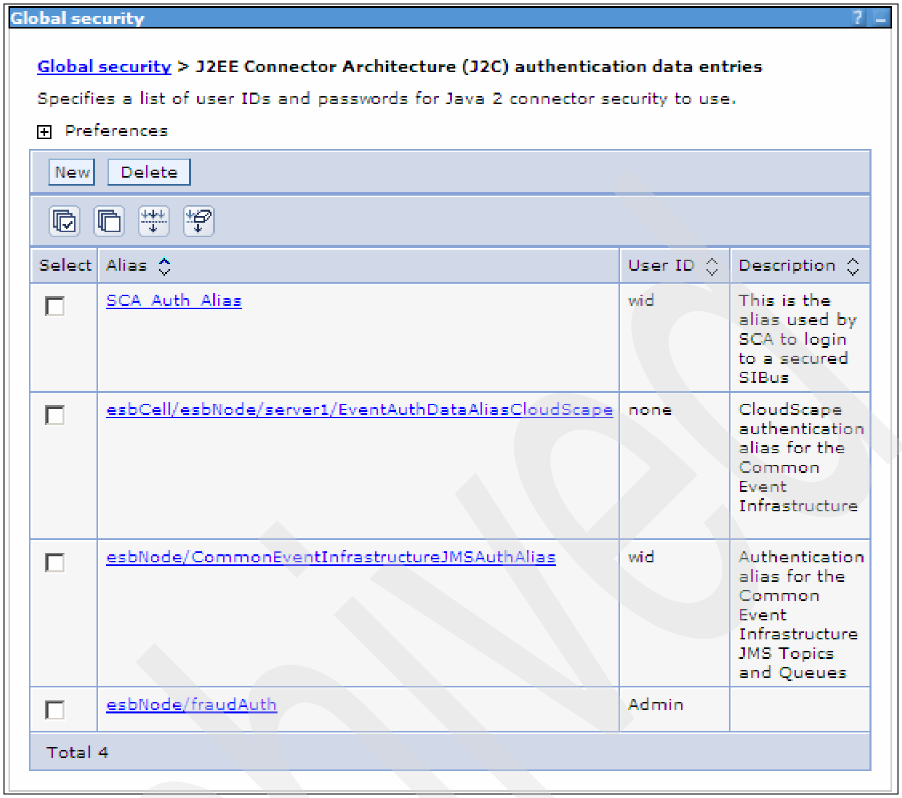


Figure 11-29 FraudAuth alias added to Global Security

3. Right-click **WebSphere ESB server v6.0** and select **Add and remove projects** → **JDBCEventDrivenMediationApp** → **Add** (Figure 11-30) → **Finish**.
4. Click the Publish icon. (This is strictly unnecessary after an Add, but good practice, as you must do this when you update a project that is already deployed and automatic publishing is not enabled.)

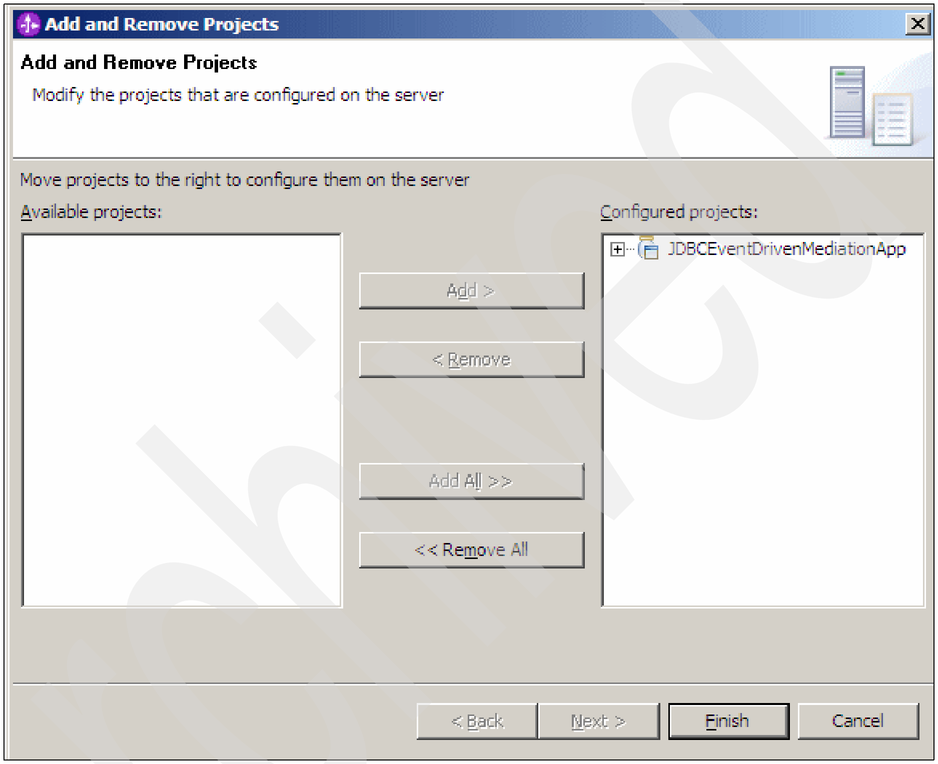


Figure 11-30 Adding our project to the server

11.6.3 Test the solution

Test the project by inserting a record into the fraud table:

1. Select **Start** → **All Programs** → **IBM DB2** → **General Administration Tools** → **Control Center**. Accept the Advanced View and click **OK**.

> **Tip:** You can also start the control center from the task status in the Windows task bar.

2. Expand the **All Databases** tab and click **ATTDB** → **Tables**. Double-click the **Fraud** table to open it (Figure 11-31).

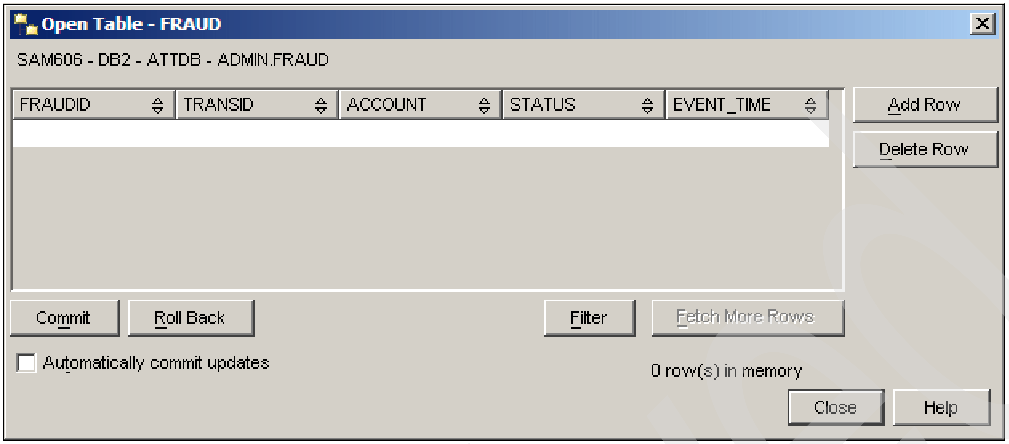


Figure 11-31 FRAUD table

3. Click **Add Row**. Enter test data (timestamp in yyyy-mm-dd hh:mm:ss.mmm) and click **Commit**.

We can view the Server Console in the WebSphere Integration Developer and view the output. Figure 11-32 illustrates the output of the create event.

```
O Output of Create Event
O Verb is : Create
O fraudid is : Peter
O transid is : Car
O account is : Transport
O status  is : Moving
O event_time  is : 2007-03-07 17:50:00.5
```

Figure 11-32 Output from create event

4. Amend the record and change the status. Click **Commit** (Figure 11-33).

```
O Output of Update Event
O Verb is : Update
O fraudid is : Peter
O transid is : Car
O account is : Transport
O status  is : Stopped
O event_time  is : 2007-03-07 17:50:00.5
```

Figure 11-33 Output from update event

5. Delete the record and click **Commit** ().

```
O Output of Delete Event
O Verb is : Delete
O fraudid is : Peter
```

Figure 11-34 Delete event

A fuller trace is shown in Example 11-5. Ignore the E-level trace lines created by WebSphere Enterprise Service Bus. They should be W level. This is a known bug.

Example 11-5 Full trace

```
[21/06/07 17:54:29:938 BST] 00000052 Core          I   SCA_STARTED_APP_MODULE
[21/06/07 17:54:30:594 BST] 00000052 ResourceMgrIm I   WSVR0049I: Binding
javax.resource.cci.ConnectionFactory as eis/javax.resource.cci.ConnectionFactory
[21/06/07 17:54:30:609 BST] 00000052 EJBContainerI I   WSVR0207I: Preparing to start
EJB jar: JDBCEventDrivenMediationEJB.jar
[21/06/07 17:54:30:625 BST] 00000052 EJBContainerI I   WSVR0037I: Starting EJB jar:
JDBCEventDrivenMediationEJB.jar
[21/06/07 17:54:30:641 BST] 00000052 ActivationSpe E   J2CA0306I: Application
JDBCEventDrivenMediationApp#JDBCEventDrivenMediationEJB.jar#ServiceSIBusMessageBean
has provided no <activation-config-property> for the ActivationSpec class
com.ibm.ws.sib.ra.inbound.impl.SibRaActivationSpecImpl of ResourceAdapter
cells/esbCell/nodes/esbNode/resources.xml#J2CResourceAdapter_1181846832234. This
may have undesirable effects.

[21/06/07 17:54:30:672 BST] 00000052 ActivationSpe E   J2CA0306I: Application
JDBCEventDrivenMediationApp#JDBCEventDrivenMediationEJB.jar#export.JDBCInboundInterfa
ceEIS has provided no <activation-config-property> for the
```

```
ActivationSpec class com.ibm.j2ca.jdbc.inbound.JDBCActivationSpecWithXid of
ResourceAdapter cells/esbCell/applications/JDBCEventDrivenMediationApp.ear/
deployments/JDBCEventDrivenMediationApp/deployment.xml#J2CResourceAdapter_11824448572
66. This may have undesirable effects.
[21/06/07 17:54:30:828 BST] 00000052 WebGroup      A   SRVE0169I: Loading Web Module:
JDBCEventDrivenMediationWeb.
[21/06/07 17:54:30:859 BST] 00000052 VirtualHost   I   SRVE0250I: Web Module
JDBCEventDrivenMediationWeb has been bound to default_host[*:9080,*:80,*:9443].
[21/06/07 17:54:30:875 BST] 00000052 ApplicationMg A   WSVR0221I: Application
started: JDBCEventDrivenMediationApp
[21/06/07 17:54:31:234 BST] 000000ae ResourceAdapt I PollEventManagerWorker run()
CWYBS0011I: Polling has started.  UserAction=No action is required.
[21/06/07 17:56:47:719 BST] 0000007d ResourceAdapt A
com.ibm.j2ca.extension.eventmanagement.internal.EventSender deliverEvent()
CWYBS0505I: Logging event before delivering to the endpoint.  Explanation=Logging
event before delivering to the endpoint.  UserAction=No action is required.
[21/06/07 17:56:48:094 BST] 0000007d ResourceAdapt I
com.ibm.j2ca.extension.eventmanagement.internal.EventSender sendEvent() CWYBS0505I:
The event has been delivered.  Explanation=An event was delivered.  UserAction=No
action is required.
[21/06/07 17:56:48:562 BST] 00000080 SystemOut     O Output of Create Event
[21/06/07 17:56:48:562 BST] 00000080 SystemOut     O Verb is : Create
[21/06/07 17:56:48:562 BST] 00000080 SystemOut     O fraudid is : Peter
[21/06/07 17:56:48:562 BST] 00000080 SystemOut     O transid is : Car
[21/06/07 17:56:48:562 BST] 00000080 SystemOut     O account is : Transport
[21/06/07 17:56:48:562 BST] 00000080 SystemOut     O status  is : Moving
[21/06/07 17:56:48:562 BST] 00000080 SystemOut     O event_time  is : 2007-03-07
17:50:00.5
[21/06/07 17:56:59:453 BST] 0000007d ResourceAdapt A
com.ibm.j2ca.extension.eventmanagement.internal.EventSender deliverEvent()
CWYBS0505I: Logging event before delivering to the endpoint.  Explanation=Logging
event before delivering to the endpoint.  UserAction=No action is required.
[21/06/07 17:56:59:578 BST] 00000080 SystemOut     O Output of Update Event
[21/06/07 17:56:59:578 BST] 00000080 SystemOut     O Verb is : Update
[21/06/07 17:56:59:578 BST] 00000080 SystemOut     O fraudid is : Peter
[21/06/07 17:56:59:578 BST] 00000080 SystemOut     O transid is : Car
[21/06/07 17:56:59:578 BST] 00000080 SystemOut     O account is : Transport
[21/06/07 17:56:59:578 BST] 00000080 SystemOut     O status  is : Stopped
[21/06/07 17:56:59:578 BST] 00000080 SystemOut     O event_time  is : 2007-03-07
17:50:00.5
[21/06/07 17:56:59:578 BST] 0000007d ResourceAdapt I
com.ibm.j2ca.extension.eventmanagement.internal.EventSender sendEvent() CWYBS0505I:
The event has been
delivered.  Explanation=An event was delivered.  UserAction=No action is required.
```

```
[21/06/07 17:57:07:031 BST] 00000080 ResourceAdapt A
com.ibm.j2ca.extension.eventmanagement.internal.EventSender deliverEvent()
CWYBS0505I: Logging event before delivering to the endpoint.  Explanation=Logging
event before delivering to the endpoint.  UserAction=No action is required.
[21/06/07 17:57:07:172 BST] 0000007d SystemOut     O Output of Delete Event
[21/06/07 17:57:07:172 BST] 0000007d SystemOut     O Verb is : Delete
[21/06/07 17:57:07:172 BST] 0000007d SystemOut     O fraudid is : Peter
[21/06/07 17:57:07:188 BST] 00000080 ResourceAdapt I
com.ibm.j2ca.extension.eventmanagement.internal.EventSender sendEvent() CWYBS0505I:
The event has been delivered.  Explanation=An event was delivered.  UserAction=No
action is required.
```

11.7 Summary

In this chapter we implemented 6.4, “Connection pattern 4: event-driven integration” on page 157. The example was kept simple. We did not do anything more with the events once they were propagated into WebSphere Enterprise Service Bus. The essential steps in driving events into the bus from a database using the JDBC adapter are easily followed.

12

Lightweight Web client integration using http

In 5.3, “Case 3: automotive spare parts” on page 138 we describe a parts distributor that has a particular problem with integrating a parts supplier who only supplied parts in response to direct orders over the Web (5.3.2, “Scenario 6: integration of a third-party supplier” on page 141). The technical solution to this requirement is described in 6.6, “Connection pattern 6: Web integration” on page 164.

In this chapter we describe how the WebSphere Enterprise Service Bus allows lightweight JSON RPC-based clients (that use non SOAP data) to communicate to the standard SOAP-based Web services. In the next chapter we describe how the WebSphere Enterprise Service Bus can be used to connect to lightweight services on the Internet (outbound connections).

This chapter describes the design and implementation of an inbound scenario that demonstrates how WebSphere Enterprise Service Bus can be used to allow Web 2.0 style HTTP clients that communicate using non-SOAP data (such as plain XML, JSON) to connect to standard SOAP-based Web services. This is an inbound connection scenario relative to WebSphere Enterprise Service Bus.

Even though the inbound scenario in this chapter makes use of JSON RPC as the communication mechanism to demonstrate lightweight Web clients, it can easily be replaced with a similar lightweight RPC protocol such as XML-RPC.

Furthermore, some of the design and architecture concepts presented in this chapter will hold true for most HTTP communication using ESB. For more details on alternative Internet technologies refer back to 1.6, “Web 2.0 - next generation Web services” on page 43.

Note: We use WebSphere Integration Developer 6.0.2 to develop the scenarios. They were tested using the integrated test environment on WebSphere Integration Developer 6.0.2 and a FireFox browser.

12.1 Auto company business scenario revisited

In this section we briefly revisit the business scenario described in 5.3, “Case 3: automotive spare parts” on page 138 that serves as a business driver for connecting to a non SOAP service on the Internet.

12.1.1 Exposing SOAP to JSON RPC clients - inbound scenario

The auto company described in 5.3, “Case 3: automotive spare parts” on page 138 wants to expose its spare parts ordering system to its dealers through a Web front end. This front end should allow the dealers to log in, select a particular model, and see the list of spare parts for this model. ITSO-AUTO wants the front end to be very lightweight and dynamic. It should be able to show the spare parts without refreshing the page.

The auto company decides to create this dynamic Web front end using JSON RPC. This Web front end will need to make use of their user authentication and traditional inventory services that are part of the spare parts ordering system and are implemented as standard Web services using SOAP messages.

There is very limited support available for SOAP handling in AJAX/JSON RPC front ends. ITSO-AUTO wanted a middleware solution that integrates their dynamic JSON RPC Web clients with other existing SOAP-based Web services in the company. They would also like to log the details of each request in a database. They wanted to investigate whether the WebSphere Enterprise Service Bus can be used to support this inbound HTTP communication using JSON RPC. This is shown in Figure 12-1.

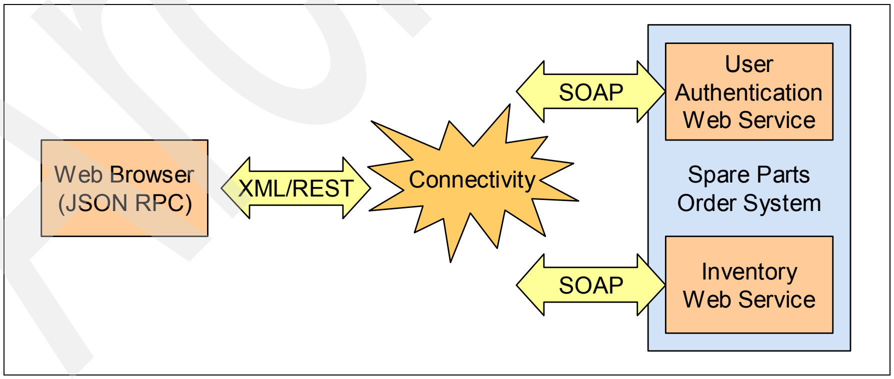


Figure 12-1 Exposing SOAP-based Web services to JSON RPC Web clients - inbound scenario

12.2 Architecture patterns revisited

WebSphere Enterprise Service Bus currently does not support a direct HTTP binding that can be used with non SOAP data. This means that various architecture patterns have to be used in order to support inbound and outbound communication using non SOAP data over HTTP.

As discussed in 6.6, “Connection pattern 6: Web integration” on page 164, the WebSphere Enterprise Service Bus can be used in three different architectural solutions for connecting to a non SOAP service or exposing a service to a non SOAP client over HTTP. We list them here again:

- “Solution 1 - custom JCA HTTP Adapter” on page 166
- “Solution 2 - invocation of the target service via a Web service proxy” on page 167
- “Solution 3 - Java SCA component acting as a target protocol client” on page 168

The preferred solution depends on the relative QOS characteristics (transaction support, connection pooling, thread pooling, and so on) of each, ease of development, and the specific scenario.

In most cases, solution 1 is not recommended, and solution 2 is overkill, as transactional support and connection pooling are not relevant to HTTP (mainly due to its connectionless nature). Solutions 3 and 4 are much the same except that solution 3 requires an extra RPC. In summary, we recommend that you look at solution 4 as the most straightforward architecture, and this is what we implement in the next few sections.

12.3 Implementing inbound communication

As discussed above, the challenge here is to enable JSON RPC-based browser clients that make use of JSON RPC to invoke SOAP-based Web services.

The inbound scenario consists of the dealers logging in to a user authentication service using the JSP-based front end and requesting a list of cars for a customer. The login ID and password are sent to the server using JSON RPC.

- The user authentication service is a Web service that expects SOAP messages. It returns a response code and a customer ID.
- The inventory service is a Web service that expects SOAP messages. It takes in a customer ID and authentication response code and returns the

authentication response code, customer ID, and a list of car models for that customer ID.

12.3.1 Architecture pattern

A high-level schematic diagram of the architecture pattern to support inbound HTTP communication using non SOAP data is shown in Figure 12-2.

An important feature of this architecture is the use of a standalone reference for non-SCA components to invoke a SCA component. The JSP and InBound Adapter object that handles the JSON RPC method calls reside inside a Web module and call the SCA mediation flow component using a Standalone reference.

It is not necessary to convert JSON to SOAP explicitly to call the user authentication and inventory SOAP services. It is done implicitly by creating business objects in the inbound adapter and then by using the service bus. The mediation module includes standard mediation primitives (if required) as well as imports to invoke other services. In our scenario, we make use of the Message Logger mediation primitive to log request and response messages, and two imports to invoke the SOAP-based services.

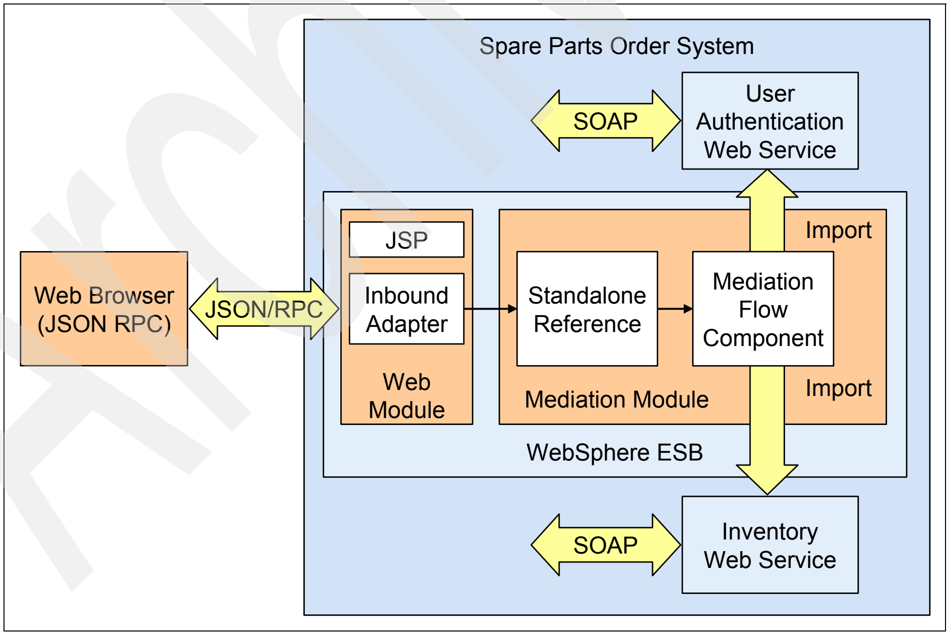


Figure 12-2 High-level architecture pattern to support inbound HTTP communication

12.3.2 Web module

The Web module consists of:

- A JSP page (login.jsp)
- An InBoundAdapter class (ITSOAutoIboundAdapter)
- Server-side JSON and JSONRPC libraries (json.jar and json-rpc.jar)
- Client-side JSONRPC javascript libraries (jsonrpc.js)

  The two JSON RPC classes (packaged in json-rpc.jar) that handle all incoming JSON RPC method calls are JSONRPCServlet and JSONRPCBridge.

A typical interaction involves the following steps:

1. The Web browser makes the initial HTTP GET to login.jsp.
2. The JSP page creates an instance of the InBoundAdapter class as well as the JSONRPCBridge class with a session scope.
3. The JSP page registers the InBoundAdapter object with the JSONRPCBridge instance.
4. The JSP page includes a reference to the JSONRPC client-side Javascript library (jsonrpc.js) as well as all other Javascript functions that invoke methods directly on the InBoundAdapter object using the client-side library.
5. When the user interacts with the Web page in the browser (by clicking a button or making a selection in a list), the client-side library routes all method invocations on the InBoundAdapter object to JSONRPCServlet.
6. JSONRPCServlet, in turn, redirects the calls to JSONRPCBridge.
7. JSONRPCBridge has a direct reference to the InBoundAdapter object and invokes the given method on it.
8. The InBoundAdapter creates the required business objects and makes the method calls to the Web services through the SCA service manager.

The components and steps involved are shown in Figure 12-3.

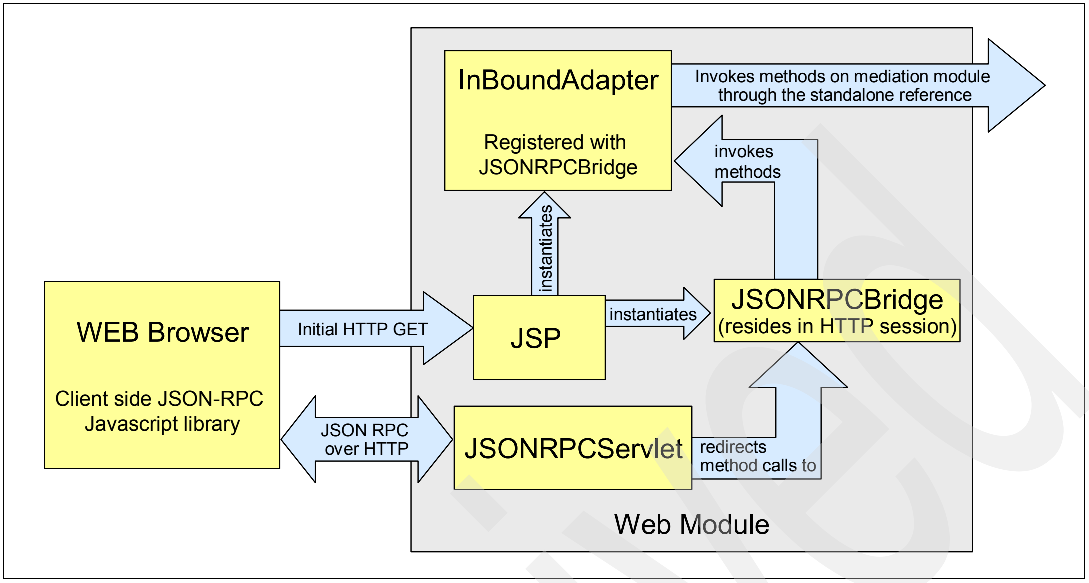


Figure 12-3 Web Module internal details for handling inbound JSON RPC communication

A discussion about design alternatives for adapters to perform HTTP communication with EBS in a RESTful manner can be found at:

http://www-128.ibm.com/developerworks/websphere/library/techarticles/0611_flurry/0611_flurry.html

12.3.3 Solution outline

The overall solution for inbound communication consists of the following:

- A mediation module comprising:
  - Standalone reference
  - Mediation flow component
  - UserAuthentication import with Web service binding
  - Inventory import with Web service binding
- WSDL and schema files for the two Web services
- A Web module that consists of:
  - JSP page (login.jsp)
  - InBoundAdapter class (ITSOAutoInboundAdapter)
  - JSONRPC and JSON libraries (json-rpc.jar and json.jar)
  - JSONRPC client side Javascript library (jsonrpc.js)

The steps for implementing the inbound scenario are:

1. Create the mediation module project.
2. Import interfaces and data types.
3. Create the imports.
4. Create the ITSOAutoInterface.
5. Create the standalone reference.
6. Implement ITSOAutoMediationFlowComponent.
   - Request flow
     - Request flow - insert the message logger primitive
     - Request flow - insert the XSL transformation primitive
     - Request flow - define XSL transformation
   - Response flow
     - Response flow - custom mediation
     - Response flow - add and wire up primitives
     - Response flow - type the XSLT terminals
     - Response flow - message logging
     - Response flow - XSL transformation
7. Import the Web Module.
8. Create the InBoundAdapter class.
9. Deploy and test.

12.4 Create the mediation module project

Begin by creating the mediation module:

1. Start WebSphere Integration Developer 6.0.2. Specify a workspace directory. Switch to the Business Integration Perspective.
2. Click **File** → **New** → **Project** → **Mediation Module** → **Next**.

3. Name the mediation module `ITSOAutoMediationModule`. Specify **WebSphere ESB Server 6.0** as the target runtime. Check the box for **Create mediation flow component** (Figure 12-4) and click **Finish**.

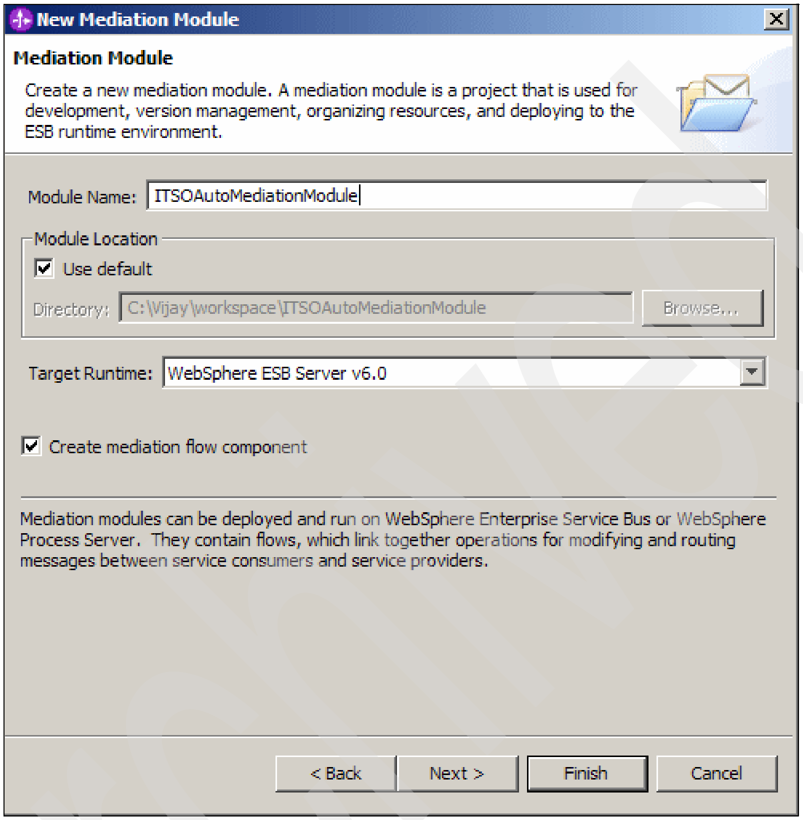


Figure 12-4 Creating a new mediation module

12.4.1 Import interfaces and data types

Next import the user authentication Web service, the inventory Web service interface (WSDL), and the data type schemas. Unzip the WSDLs.zip file (from Appendix A, “Additional material” on page 499).

1. Click **File** → **Import** → **WSDL/Interface**.
2. In the Import from selection panel select the **WSDLs** directory created by unzipping the WSDLs.zip file. Check the **WSDL** directory in the left-hand pane. This checks the xsd-includes sub-directory automatically, and all the WSDL files on the right-hand side are checked, too.

3. Leave the check in the Import dependent resources box (Figure 12-5) and click **Finish**.

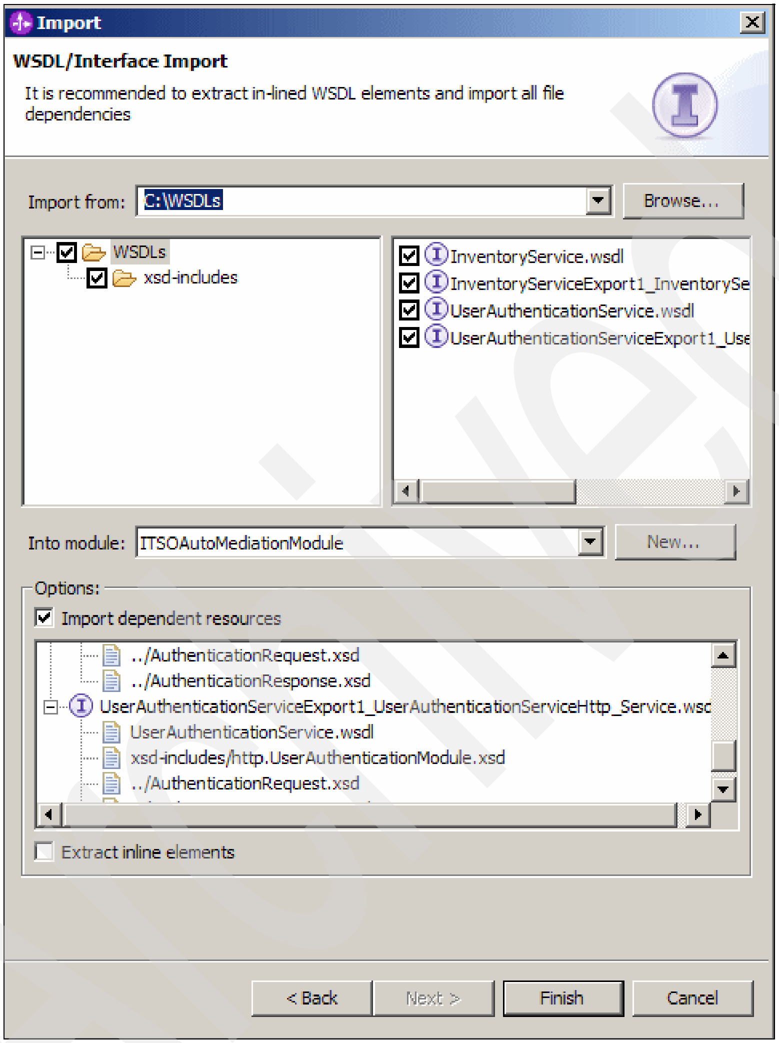


Figure 12-5 Importing WSDLs

In the Business Integration window, under ITSOAutoMediationModule, there are three new data types (Figure 12-6 on page 425):

- AuthenticationRequest
- AuthenticationResponse
- InventoryResponse

There are two new interfaces:

- InventoryService
- UserAuthentication Service

And two new Web service ports:

- InventoryServiceExport1_InventoryServiceHttpPort
- UserAuthenticationServiceExport1_UserAuthenticationServiceHttpPort

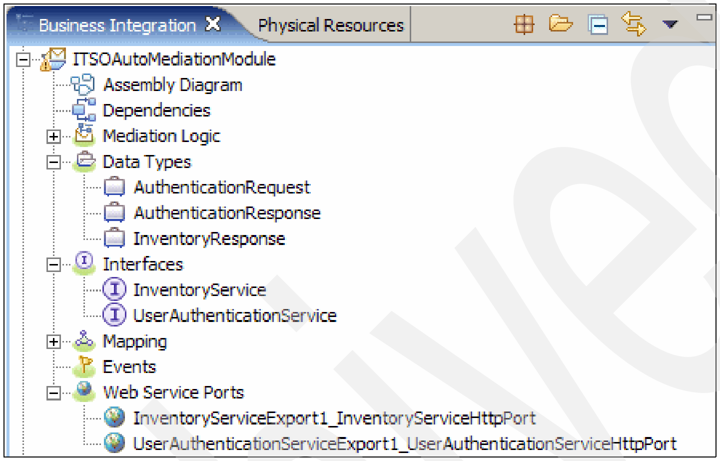


Figure 12-6 Business Integration view after importing the user authentication and inventory Web services

In this example, for simplicity, we deploy the Web services in the same runtime environment as the mediation module.

12.4.2 Create the imports

In this section we define the mediation interfaces and imports:

1. Open the Assembly diagram for the ITSOAutoMediationModule. Rename the mediation flow component Mediation1 to `ITSOAutoMediationFlowComponent`.
2. In the Business Integration explorer click **Web Service Ports** and drag the **UserAuthenticationServiceExport1_UserAuthenticationServiceHttpPort** Web service port onto the Assembly diagram white space.

3. Specify **Import with Web service Binding** in the component creation dialog box that appears (Figure 12-7).

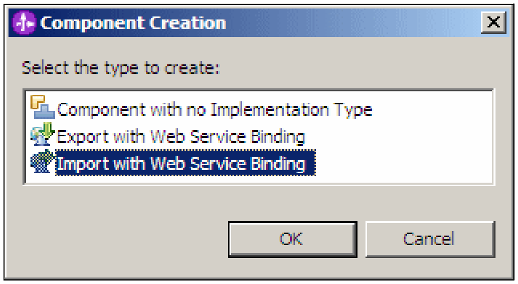


Figure 12-7 Importing with Web service binding

This creates an import called UserAuthenticationSericeImport1.

4. Wire the ITSOAutoMediationFlowComponent to UserAuthenticationSericeImport1 and click **OK** in response to the dialog box that appears.
5. Repeat the steps for the InventoryService Web Service Port (Figure 12-8).

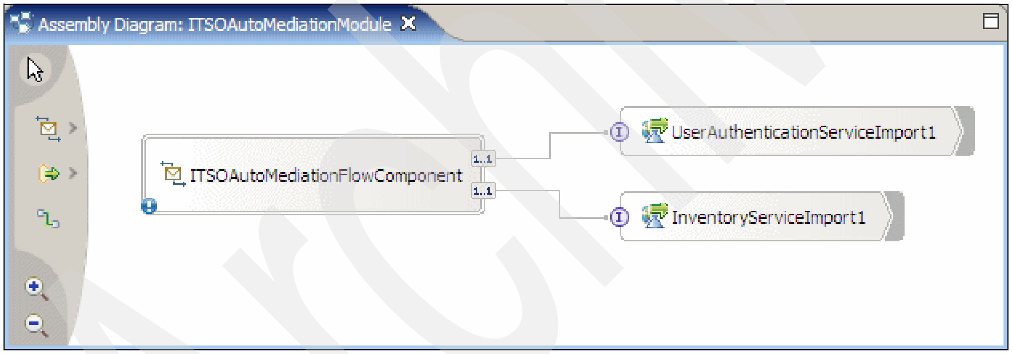


Figure 12-8 Mediation module assembly diagram

12.4.3 Create the ITSOAutoInterface

The ITSOAutoInterface is used by ITSOAutoInboundAdapter to connect to the ITSOAutoMediationModule through a standalone reference. In this section we describe the steps to create the ITSOAutoInterface.

Operations

The ITSOAutoInterface consists of two operations:

- logIn

  logIn takes the authentication parameters (customerId and password) as input, authenticates the user with the authentication service, invokes the inventory service to get a list of auto models for a given customer, and returns three values:

  – Customer ID
  – Response string
  – List of auto models for the given customer ID

  All request/response flows should be logged.

  We use this operation in the example to illustrate the inbound communication scenario.

- getSpareParts

  getSpareParts invokes a third-party spare parts service. It takes in an auto model name and returns a list of available spare parts for that model.

  We use this operation in the example to illustrate an outbound communication scenario.

Business objects

Two business objects are required in the logIn operation:

- AuthenticationRequest

  AuthenticationRequest has a customerId string and a password string (Figure 12-9).

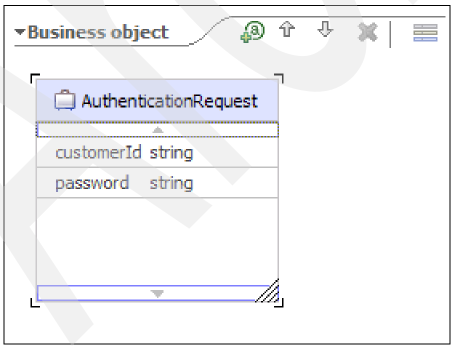


Figure 12-9 AuthenticationRequest business object

- AuthenticationResponse

  AuthenticationResponse has a response string and a customerId string (Figure 12-10).

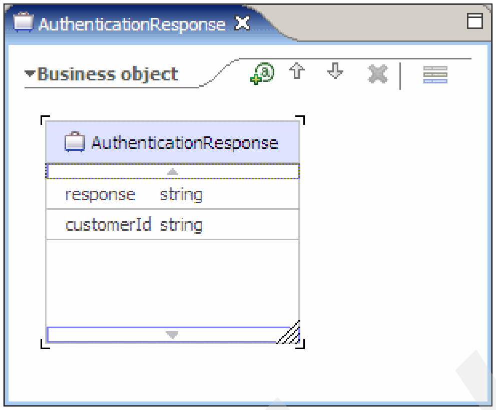


Figure 12-10 AuthenticationRequest business object

In the getSpareParts operation we make use of the AuthenticationResponse business object and a third business object.

- InventoryResponse

  InventoryResponse has a customerId string, a response string, and an auto model list in the form of a JSON RPC string (Figure 12-11).

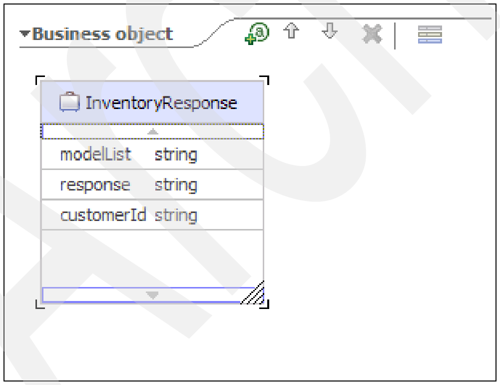


Figure 12-11 InventoryResponse business object

Create the interfaces

To do this:

1. Right-click **Interfaces** in the Business Integration window and select **New** → **Interface**. Name it `ITSOAutoInterface` and click **Finish**.
2. In the Interface editor, click the Add Request Response Operation icon. Create a new operation and name it `logIn`.
3. Rename input1 `authenticationRequest` and output1 `inventoryResponse`. Type the input and output using the pull-down box AuthenticationRequest and InventoryReponse (Figure 12-12).

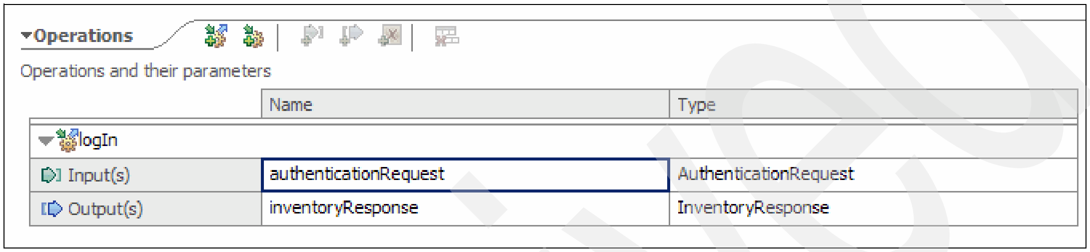


Figure 12-12 LogIn operation

Add ITSOAutoInterface to ITSOAutoMediationFlowComponent

Drag **ITSOAutoInterface** from Interfaces and drop it on to the ITSOAutoMediationFlowComponent and click **Save**.

An interface icon is added to the left side of the mediation component (Figure 12-13).

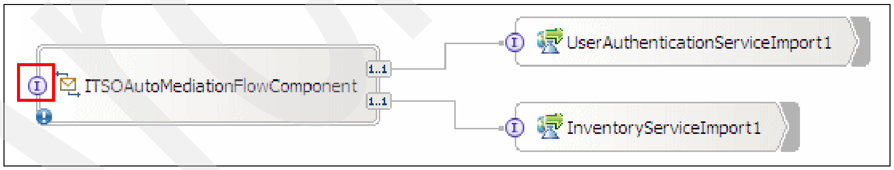


Figure 12-13 Added ITSOAutoInterface to ITSOAutoMediationFlowComponent

12.4.4 Create the standalone reference

To call the ITSOAutoInterface interface from a SCA component we need to create a standalone reference:

1. Select the standalone reference widget from the palette and drag it onto the Assembly Diagram Editor.
2. Wire the standalone reference to ITSOAutoMediationFlowComponent.

3. The Add Wire dialog box appears and asks you whether you want to convert the WSDL interface used by the reference to a Java Interface. Since we invoke the mediation flow component from a JSP locally, we need Java interfaces. Click **Yes** (Figure 12-14).

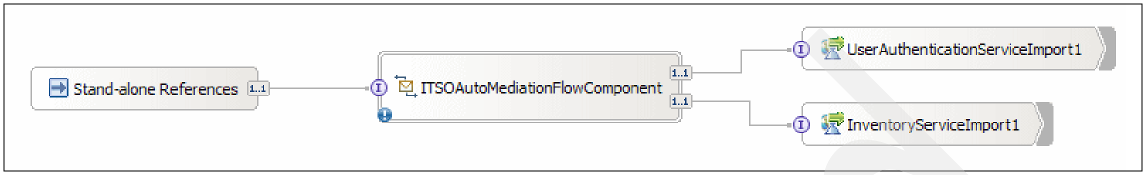


Figure 12-14 Assembly Diagram Editor after creating the imports and a standalone reference, and wiring them together with the mediation flow component

12.5 Implement ITSOAutoMediationFlowComponent

With the component assembly design in place, the next step is to implement the mediation flow component.

1. Right-click **ITSOAutoMediationFlowComponent** in the Assembly diagram and click **Generate Implementation**. Respond **OK** to accept the default ITSOAutoMediationModule folder to store the generated mediation flow component.

This opens up the mediation flow editor. The top panel is the Operation connections panel. It has the ITSOAutoInterface and the logIn operation on the left and the target operations on the right. The bottom panel is the Mediation flow panel and is empty in the beginning (Figure 12-15).

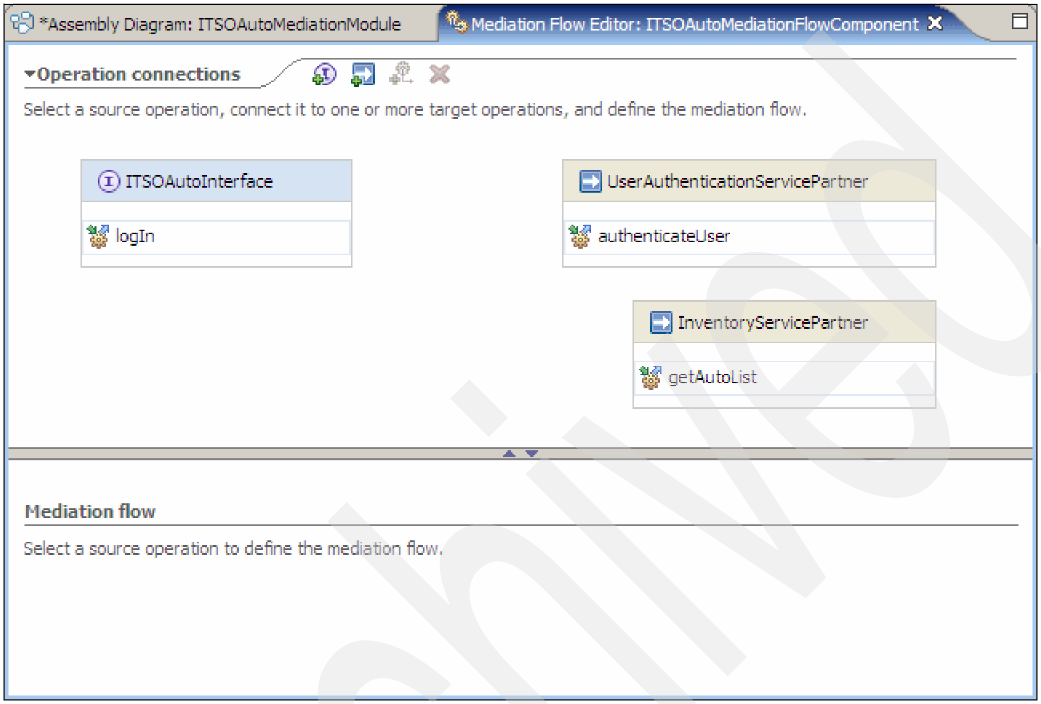


Figure 12-15 Mediation flow editor for logIn operation

2. Connect the logIn operation in ITSOAutoInterface with the authenticateUser target Operation (Figure 12-16 on page 432).

   The Mediation flow panel has two tabs — one for the request flow and one for the response flow.

12.5.1 Request flow

The request flow tab shows the Input node (logIn operation), the Callout node (to the authenticateUser operation in UserAuthentication Web service), and the Input Response node (Figure 12-16).

We need to connect the Input node to the Callout node and insert the message logging and XSLT transformation primitives in the flow.

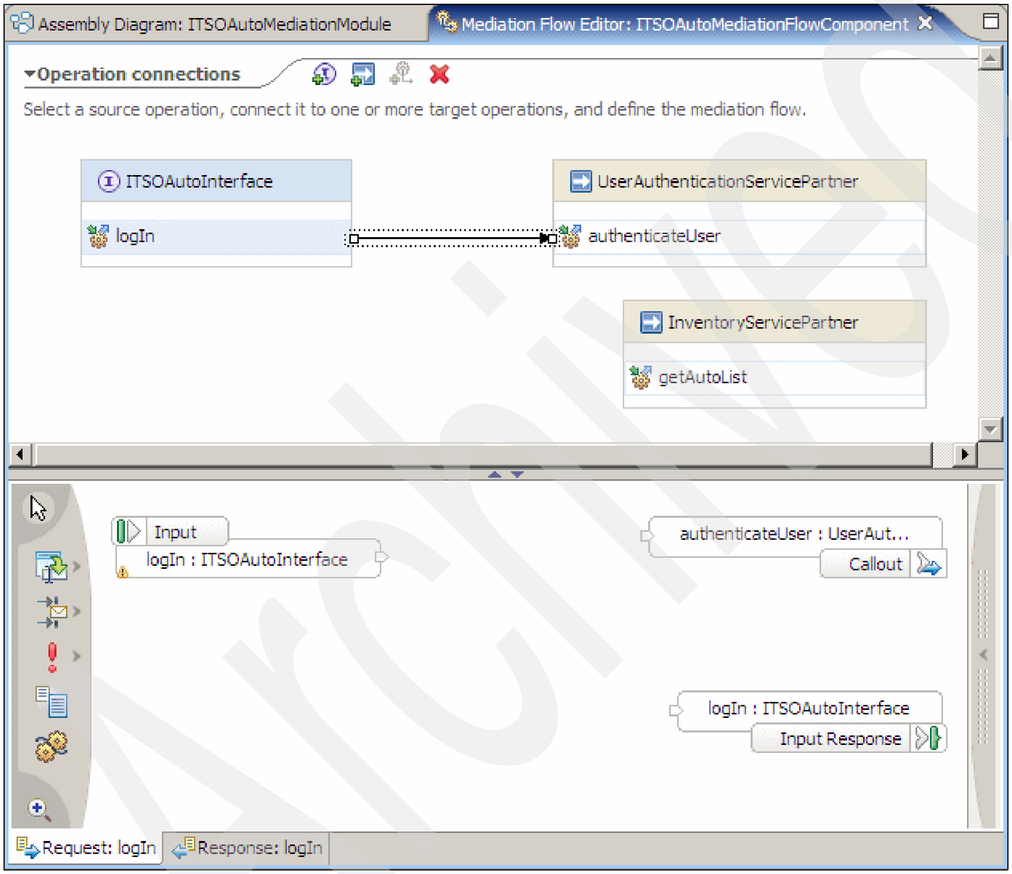


Figure 12-16 Wiring logIn operation to authenticateUser target operation

Request flow - insert the message logger primitive

To do this:

1. Drag the MessageLogger primitive from the palette  and drop it onto the Mediation flow panel. Rename it `logInRequestLogger`.

Since we log only the body of messages, we accept the defaults for the message logger primitive. This is shown in Figure 12-17.

2. Wire the Input Node to the logInRequestLogger primitive.

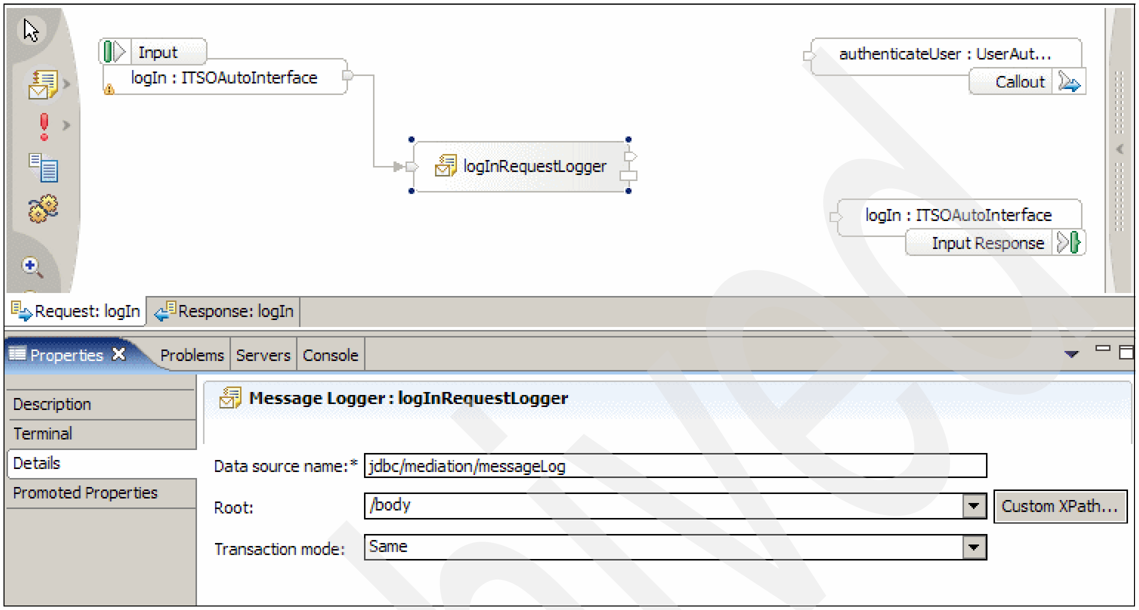


Figure 12-17 Default properties for message logger primitive

Request flow - insert the XSL transformation primitive

Output from the logInRequestLogger is a message of type logInRequestMsg, whereas the CallOut node expects an input message of type authenticateUserRequestMsg. In order to map the elements of a logInRequestMsg to those of a authenticateUserRequestMsg, we make use of an XSL Transformation primitive.

1. Drag XSL Transformation primitive from the palette  and drop it onto the mediation flow. Rename it `logInRequestXSLTransformation`.
2. Wire the output terminal of logInRequestLogger to the input terminal of logInRequestXSLTransformation.

3. Wire the output terminal of logInRequestXSLTransformation to the CallOut node (Figure 12-18).

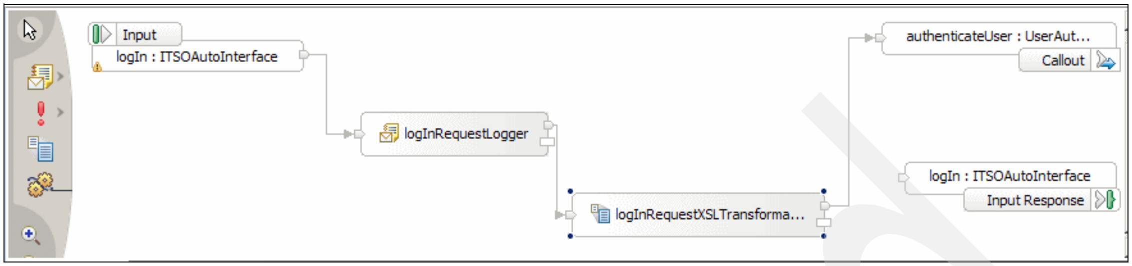


Figure 12-18 Mediation flow panel illustrating request flow for logIn operation

Request flow - define XSL transformation

Next map the elements of logInRequestMsg to those in authenticateUserRequestMsg. The XSL transformation does this for us, but we must create a new mapping file to achieve this.

1. Click **logInRequestXSLTransformation**, select the **Details** tab in the Properties view, and click **New**.

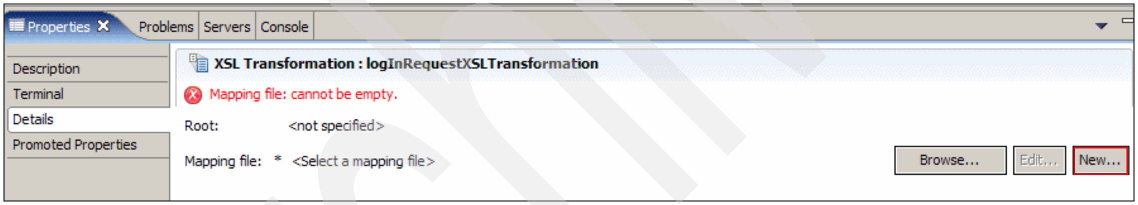


Figure 12-19 Creating a new mapping file for XSL Transformation primitive

2. Accept the defaults on the Specify Message Types dialog box (Figure 12-20) and click **Finish**.

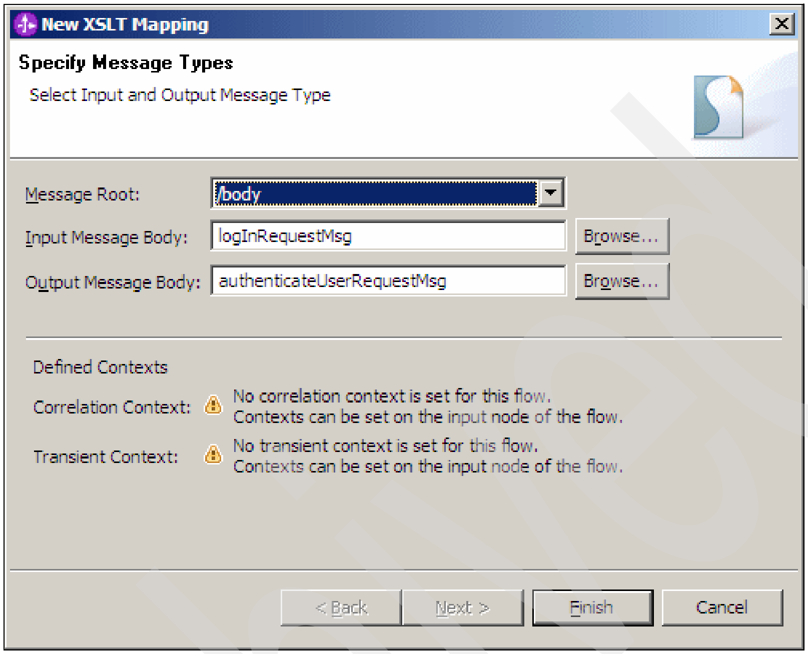


Figure 12-20 Specify message types for XSL Transformation mapping

3. In the XML Transformation mapping file editor select **SystemAuthenticatioRequest** in Source and **authenticationRequest** in Target. Right-click **Match Mapping** (Figure 12-21).

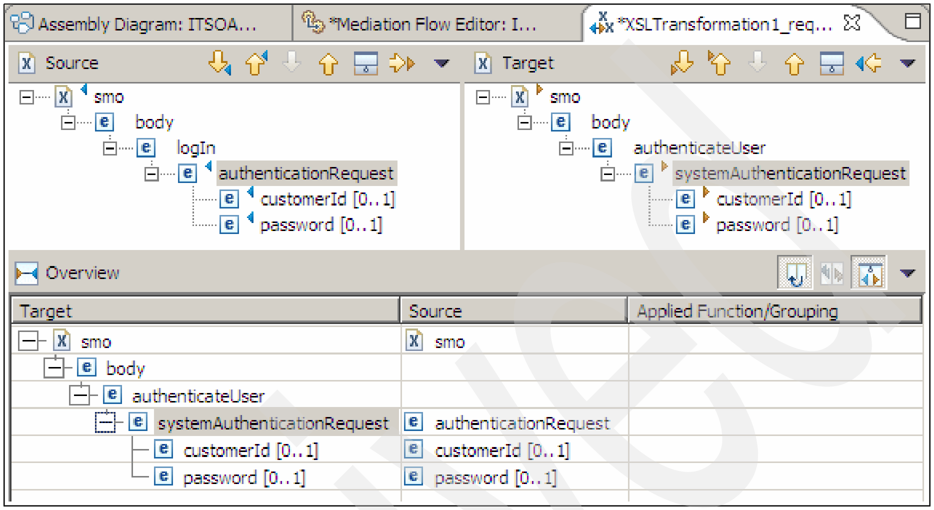


Figure 12-21 Creating a mapping between source and target message types

12.5.2 Response flow

Click the **Response** tab in the Mediation editor.

The response flow tab shows the Callout Response node (for the authenticateUser operation in UserAuthentication Web service) and the Input Response node (Figure 12-22).

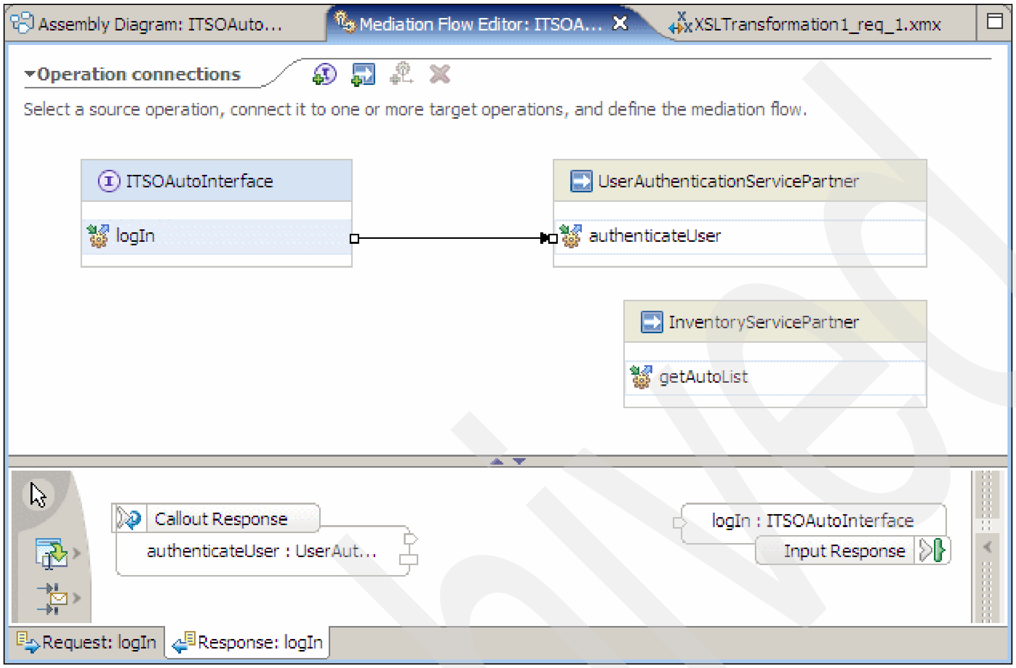


Figure 12-22 Response flow in the mediation flow panel

We make use of a custom mediation to invoke the inventory Web service to get the valid list of auto models for the given customer ID. We connect the Callout Response node with the Input Response node, but insert the message logging, XSLT transformation, and custom mediation primitives in the flow.

Response flow - custom mediation

For custom mediation:

1. Drag the **CustomMediation** primitive from the palette and drop it onto the Mediation flow panel. Rename it `InventoryCustomMediation`.
2. Wire the *out* terminal of the Callout Response (authenticateUser) to the *in* terminal of InventoryCustomMediation.

3. In the Details panel of InventoryCustomMediation in its Properties view, select **Invoke** as the Implementation, **InventoryServicePartner** in the Reference drop-down list, and **getAutoList** in the Operation drop-down list (shown in Figure 12-23).

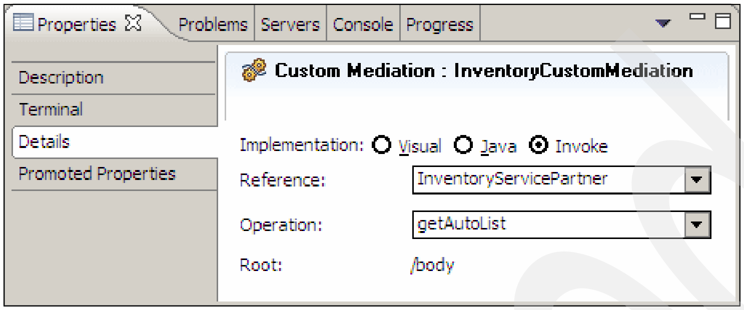


Figure 12-23 Custom mediation details

Response flow - add and wire up primitives

The flow consists of the custom mediation, two logging mediations, and two XSLTs.

Drop the four additional primitives into the mediation flow. Name the logging mediations `logInResponseLogger` and `InventoryResponseLogger`, and the XSLTs `logInResponseXSLTransformation` and `InventoryResponseXSLTransformation`, and wire them up as shown in Figure 12-24. Click **Save**.

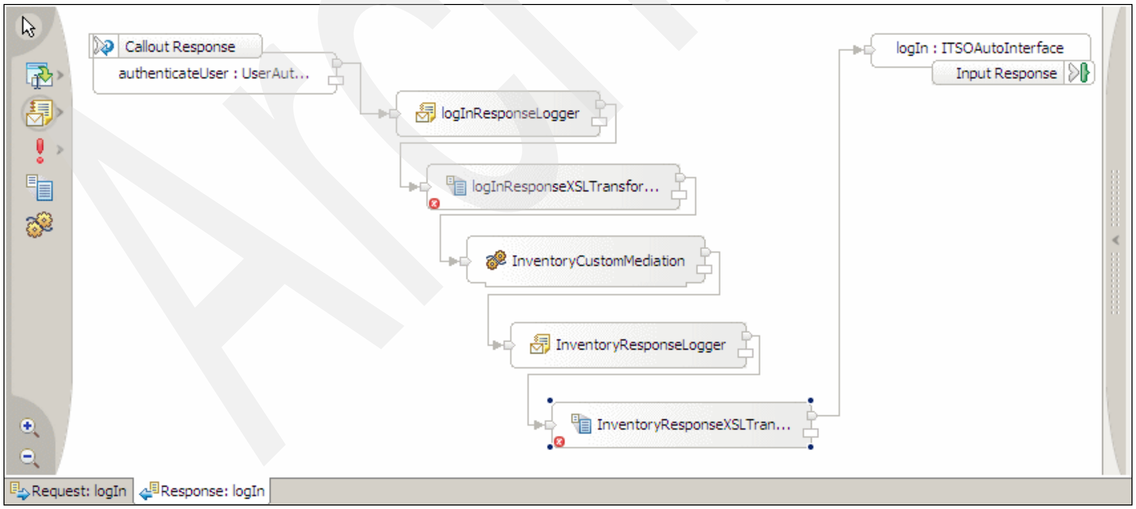


Figure 12-24 Response flow

There are errors in the XSLTs. Four are due to not yet having defined the transformations, and the others are due to the correct data types not being inferred for all of the in and out terminals in the flow. We correct this next.

Response flow - type the XSLT terminals

First we type the **in** terminal of the last XSLT, InventoryResponseXSLTransformation.

1. Click the **in** terminal on the Terminal tab of the Properties view of the logInResponseXSLTransformation primitive (Figure 12-25) and select change **Change**.

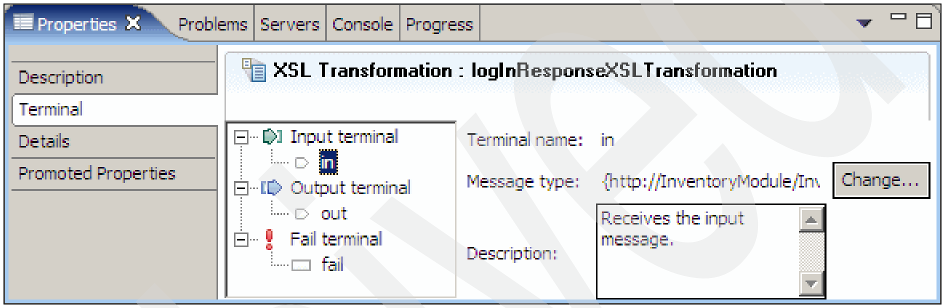


Figure 12-25 Change the type of the in terminal in logInResponseXSLTransformation

2. In the Change Message Type dialog box select **Browse**. Select the **InventoryService** Interface, **getAutoList** Operation, **Output** Message Category, and **getAutoListResponseMsg** Message Type (Figure 12-26). Click **OK** → **Save All**.

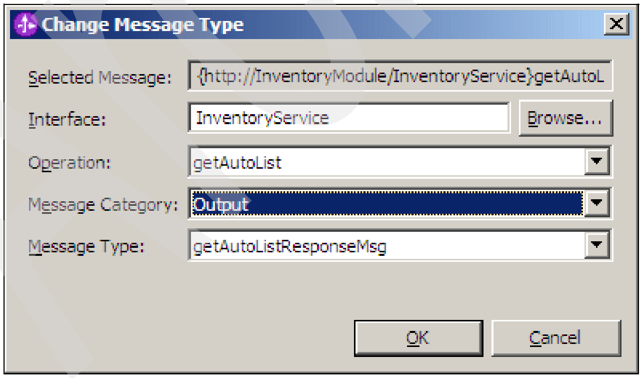


Figure 12-26 Type the in terminal of the XSLT

Now we type the *out* terminal of the first XSLT the logInResponseXSLTransformation.

3. Click the **out** terminal in the Terminal tab of the Properties view of the InventoryResponseXSLTransformation primitive (Figure 12-27). Select **Change**.

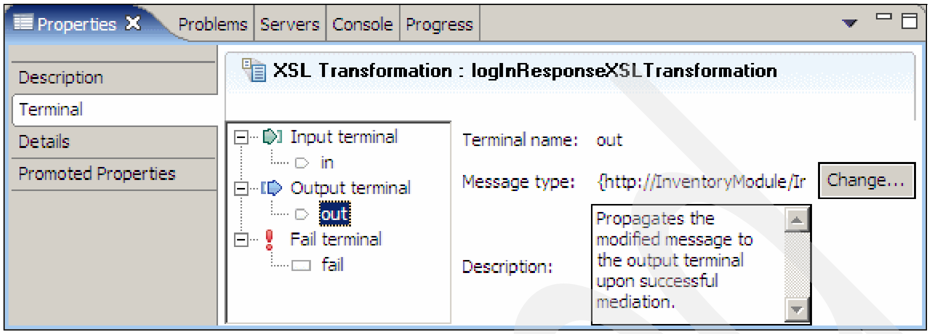


Figure 12-27 Select the out terminal of the logInResponseTransformation

4. In the Change Message Type dialog box click **Browse**. Select the **InventoryService** Interface, **getAutoList** Operation, **Input** Message Category, and **getAutoList*Request*Msg** Message Type (Figure 12-26 on page 439). Click **OK** → **Save All**.

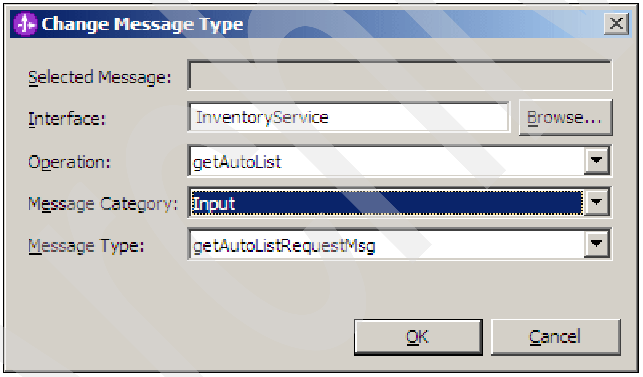


Figure 12-28 Type the XSLT out message

All the in and out terminals in the entire flow are now typed. Check that this is true.

Response flow - message logging

We do not need to change the default properties of the message loggers.

Response flow - XSL transformation

Each of the XSL transformations needs a map defined.

1. Click the **Details** tab of the InventoryResponseXSLTransformation. Click **New** → **Finish**.
2. Select **getAutoResponse** in the Source pane and **logInResponse** in the Target pane. Right-click **Match Mapping** and select **Save** (Figure 12-29).

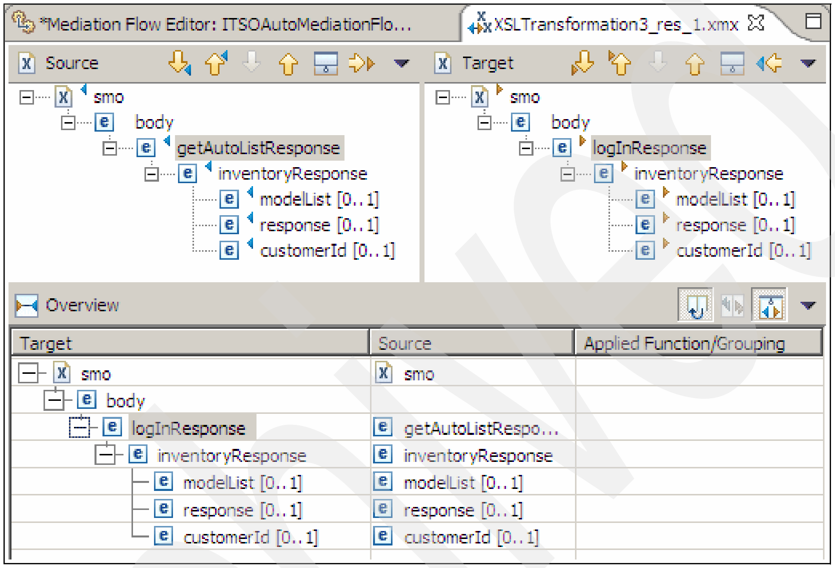


Figure 12-29 InventoryResponseXSLTransformation mapping

3. Click the **Details** tab of the logInResponseXSLTransformation and select **New** → **Finish**.

4. Select **systemAuthenticationResponse** in the Source pane and **authResponse** in the Target pane. Right-click **Match Mapping** and click **Save** (Figure 12-30).
5. Click **Project** → **Clean** → **Clean All Projects** → **Build automatically**.

   All the errors in the workspace are fixed.

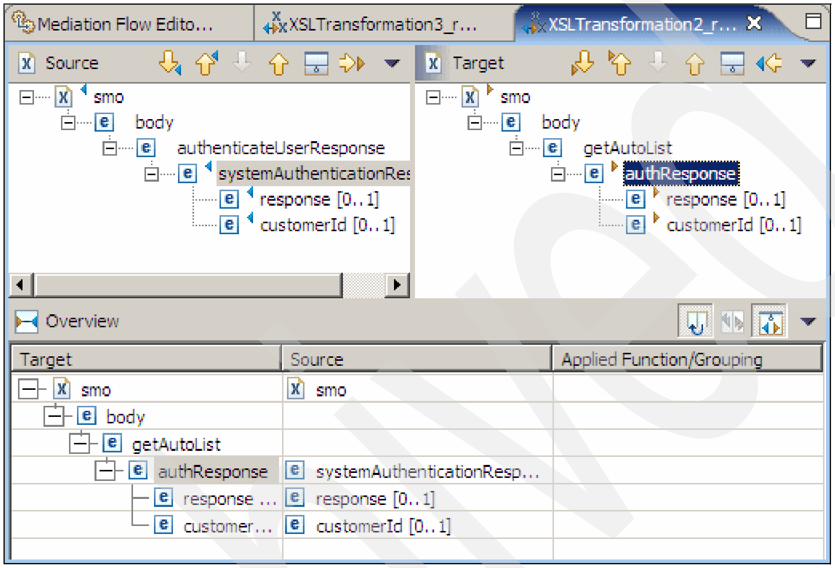


Figure 12-30 logInResponseXSLTransformation mapping

12.6 Import the Web Module

With a standalone reference, we can now invoke SCA components from non-SCA components in the same module. Since a SCA module is realized by an EAR file, one can add additional J2EE components to the SCA module. In our example we import an existing WAR file that has a JSP file. The WAR file will be part of the underlying EAR file.

1. Select **File** → **Import** to bring up the Import wizard. Select **WAR file** and click **Next** (Figure 12-31).

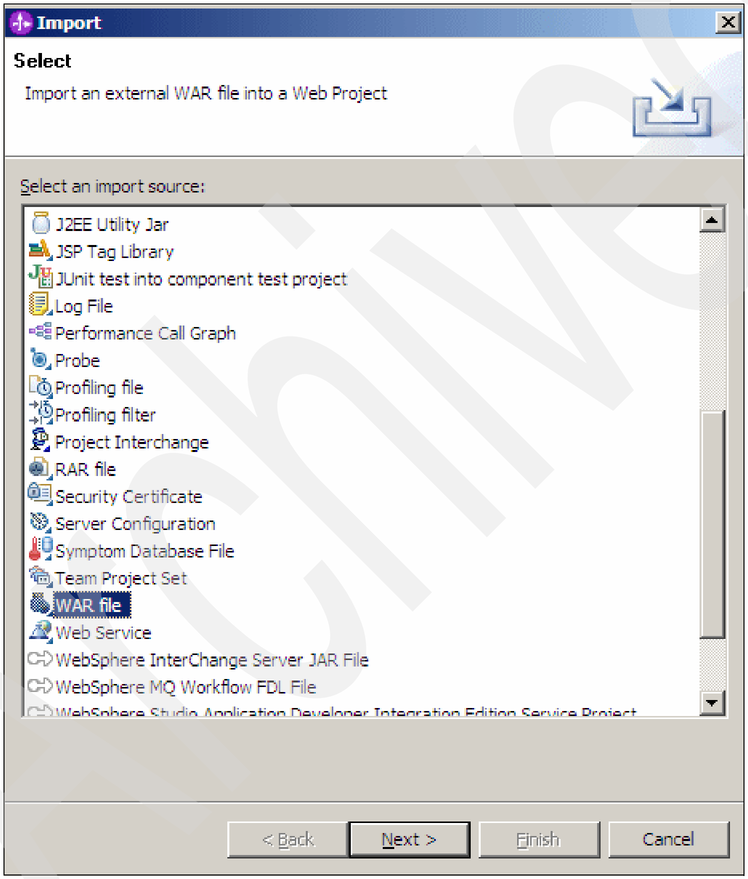


Figure 12-31 Importing a WAR file

2. In the WAR Import dialog box select the **AutoCompanyLoginWeb.war** file (which is part of Appendix A, “Additional material” on page 499). Accept the default AutoCompanyLoginWeb Web project and specify **WebSphere ESB**

Server v6.0 as the target server. Deselect **Add module to an EAR Project** (Figure 12-32). Select **Yes** to switch to the Web Perspective.

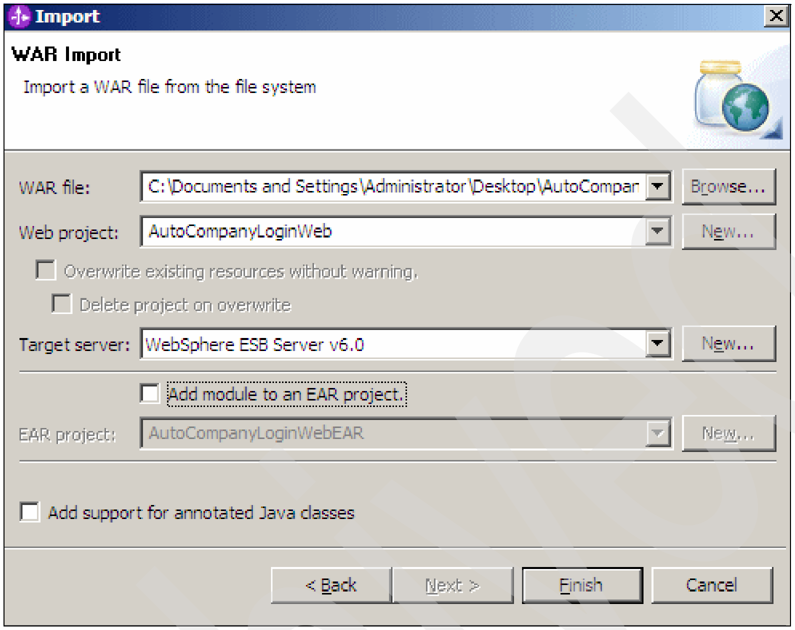


Figure 12-32 Settings for importing the WAR file

3. Open the JSP file to examine the code. It is under the Dynamic Web Projects directory (Figure 12-33).

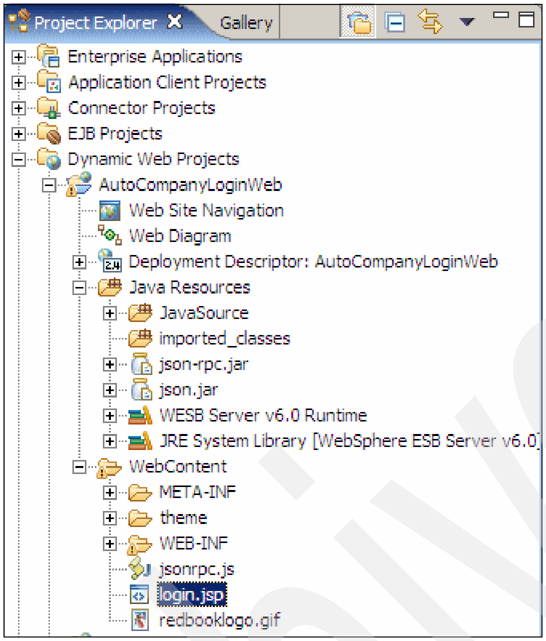


Figure 12-33 Contents of the Web Module in the AutoCompanyLoginWeb project

Figure 12-33 shows the JSON jar, JSON RPC server-side jar, and the JSONRPC client-side Javascript library (jsonrpc.js). The JSP is called login.jsp. The JSP page is used for rendering the HTML, as well as making Javascript function calls for invoking method calls on the InBoundAdapter object (which is created and added to the Web module in 12.7, “Create the InBoundAdapter class” on page 450).

12.6.1 Add the .war project as a dependency

WebSphere Integration Developer has a Dependency Editor to add different packages to the integration .ear file. We need to add the AutoCompanyLoginWeb project.

1. Switch to the Business Integration perspective. Right-click **ITSOAutoMediationModule** and click **Open Dependencies** (Figure 12-34).

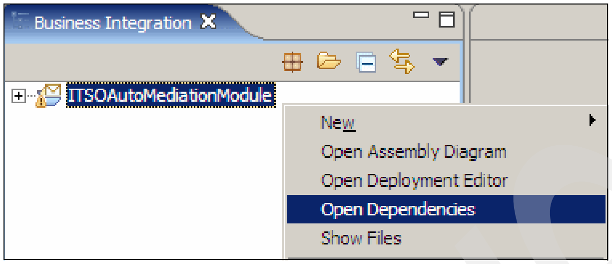


Figure 12-34 Opening the dependencies editor for ITSOAutoMediationModule

2. Expand the **J2EE** section and click **Add** → **AutoCompanyLoginWeb** (Figure 12-35). This makes the WAR file part of the EAR file.

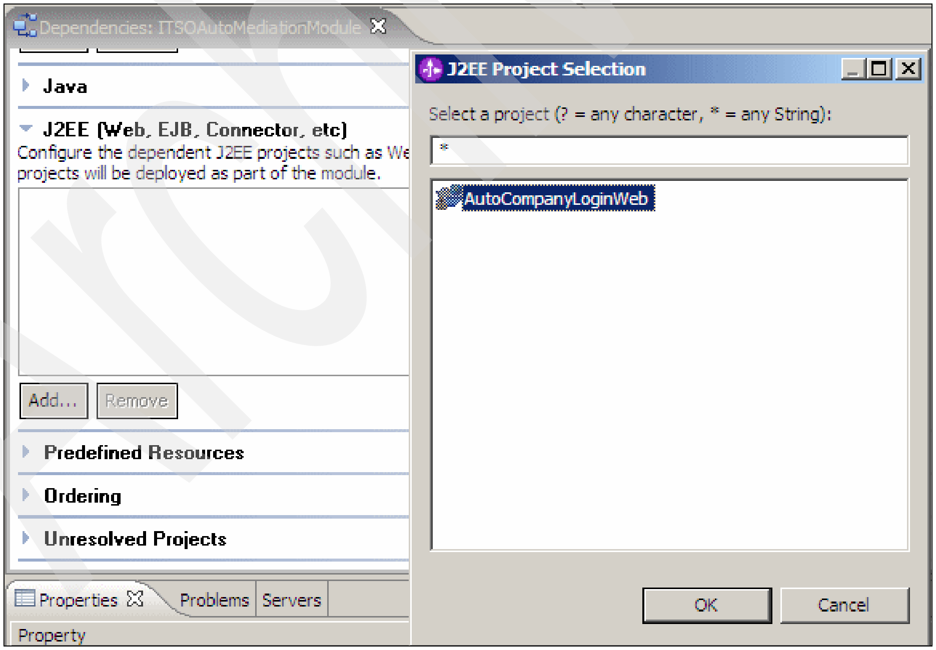


Figure 12-35 Adding AutoCompanyLoginWeb as a J2EE module into ITSOAutoMediationModule

3. Select **AutoCompanyLoginWeb**. Uncheck **On Build Path** (Figure 12-36) and click **Save**.

   Build Path adds this component to the classpath of the SCA module. In our case, we need to make the WAR file depend on the SCA module.

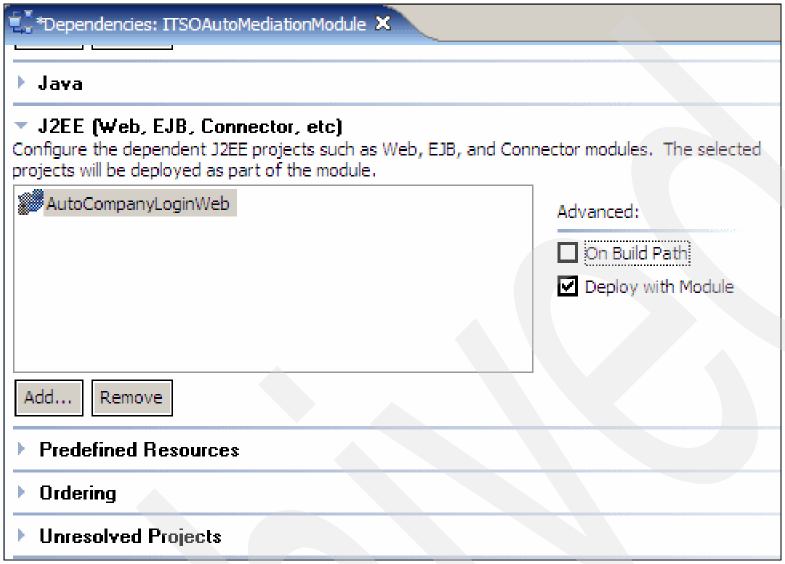


Figure 12-36 Settings for adding AutoCompanyLoginWeb project as a J2EE module

4. For the Web project to find the SCA Java interface, switch to the Web perspective, right-click the **AutoCompanyLoginWeb** project, and select **Properties**.

5. Click **Java JAR Dependencies**. Select **Use EJB client JARs** and check **ITSOAutoMediationModuleEJB.jar**, which is the generated EJB client JAR file (shown in Figure 12-37).

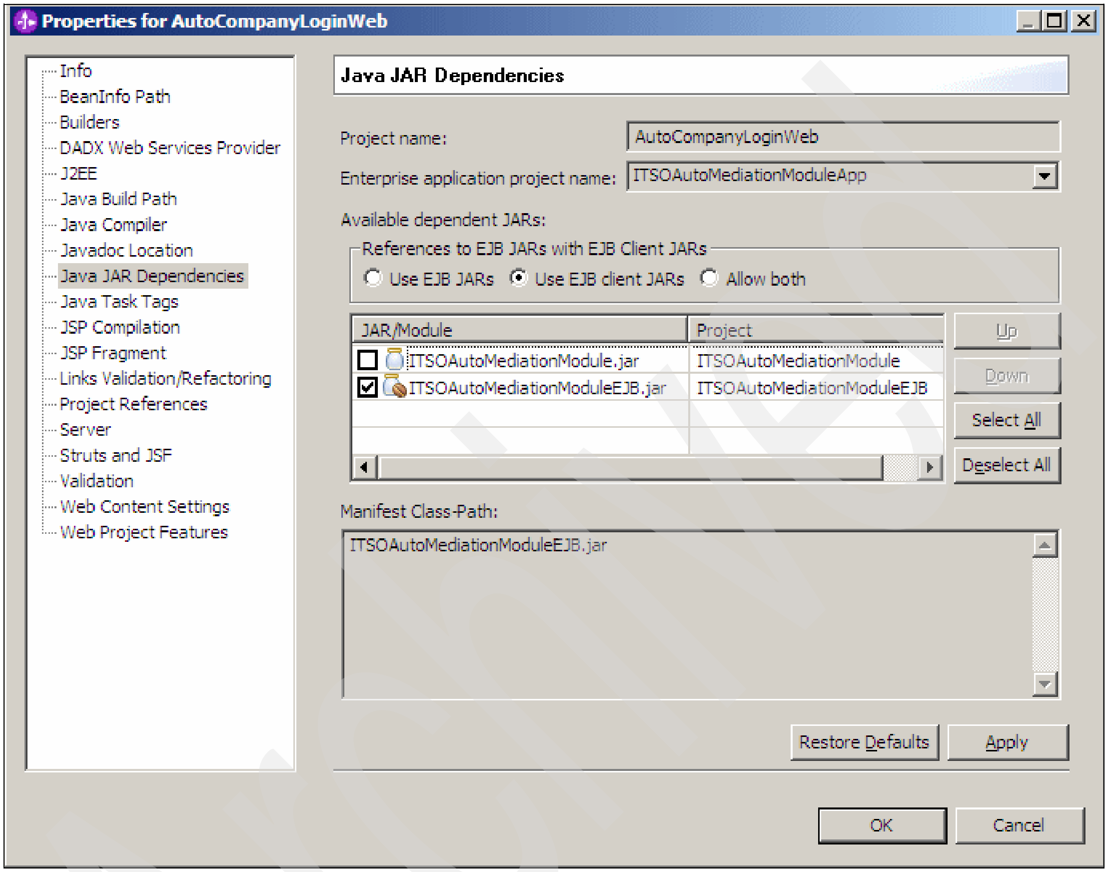


Figure 12-37 Adding ITSOAutoMediationModuleEJB.jar as dependency to AutoCompanyLoginWeb project

12.6.2 Browse the Web deployment descriptor file (web.xml)

The Web deployment descriptor (web.xml) for this Web module has the following entries (Example 12-1) for JSONRPCServlet.

Example 12-1 web.xml entries for JSONRPCServlet

```
<servlet>
<servlet-name>com.metaparadigm.jsonrpc.JSONRPCServlet</servlet-name>
<servlet-class>com.metaparadigm.jsonrpc.JSONRPCServlet</servlet-class>
</servlet>
....
```

```
<servlet-mapping>
<servlet-name>com.metaparadigm.jsonrpc.JSONRPCServlet</servlet-name>
    <url-pattern>/JSON-RPC</url-pattern>
</servlet-mapping>
```

12.6.3 Browse the code in the JSP page

The JSP page (login.jsp) has all the HTML that it needs to render as well as all the JavaScript function to make JSON RPC method calls and use the response to modify the page contents.

Instantiation of JSONRPCBridge and InBoundAdapter class

The following code snippet (Example 12-2) shows the code that is used to instantiate JSONRPCBridge and the ITOSAutoInBoundAdapter class and then register the ITOSAutoInBoundAdapter instance with the JSONRPCBridge instance.

Example 12-2 instantiate and register JSONBridge and ITSOInboundAdapter

```
<jsp:useBean id="JSONRPCBridge" scope="session"
   class="com.metaparadigm.jsonrpc.JSONRPCBridge" />
<jsp:useBean id="itsoAutoObject" scope="session"
   class="com.ibm.itso.autocompany.ITSOAutoInBoundAdapter" />
<%JSONRPCBridge.registerObject("itsoAutoService", itsoAutoObject);%>
```

Including the client-side JavaScript library for JSONRPC

The client-side JavaScript library for JSONRPC, jsonrpc.js, is included in the page as follows:

```
<script type="text/javascript" src="jsonrpc.js">
</script>
```

Invocation of InBoundAdapter object methods from JavaScript

Instantiation of the JSONRPCClient (passing it the JSONRPC servlet URL) and using it to invoke methods on the Inbound Adapter object on the server is shown in Example 12-3.

Example 12-3 Invoking methods on the InBound Adapter object from the client

```
jsonrpc = new JSONRpcClient("/AutoCompanyLoginWeb/JSON-RPC");
// Call a Java method on the server
var result = jsonrpc.itsoAutoService.logIn(customerId, passwd);
```

```
var inventoryResponse = eval( "(" +  result + ")" );
...
jsonrpc = new JSONRpcClient("/AutoCompanyLoginWeb/JSON-RPC");
var resultObj = jsonrpc.itsoAutoService.getSpareParts(modelName);
var result = eval( "(" +  resultObj + ")" );
```

12.7 Create the InBoundAdapter class

Now that we have ensured that the Web project can find the SCA Java interface for the mediation module, we create the InBoundAdapter class that receives the JSON RPC method calls made by the Web client. the InBoundAdapter then invokes the Web services using the SCA API and SCA service manager. It also creates the business objects required for the Web service invocation.

1. Switch to **Web perspective** → **Dynamic Web projects** → **AutoCompanyLoginWeb** → **Java Resources** → **New** → **Class**.

2. Browse to AutoCompanyLoginWeb/JavaSource as the source folder and type `com.ibm.itso.autocompany` as the package name and `ITSOAutoInBoundAdapter` as the name of the class (Figure 12-38). Click **Finish**.

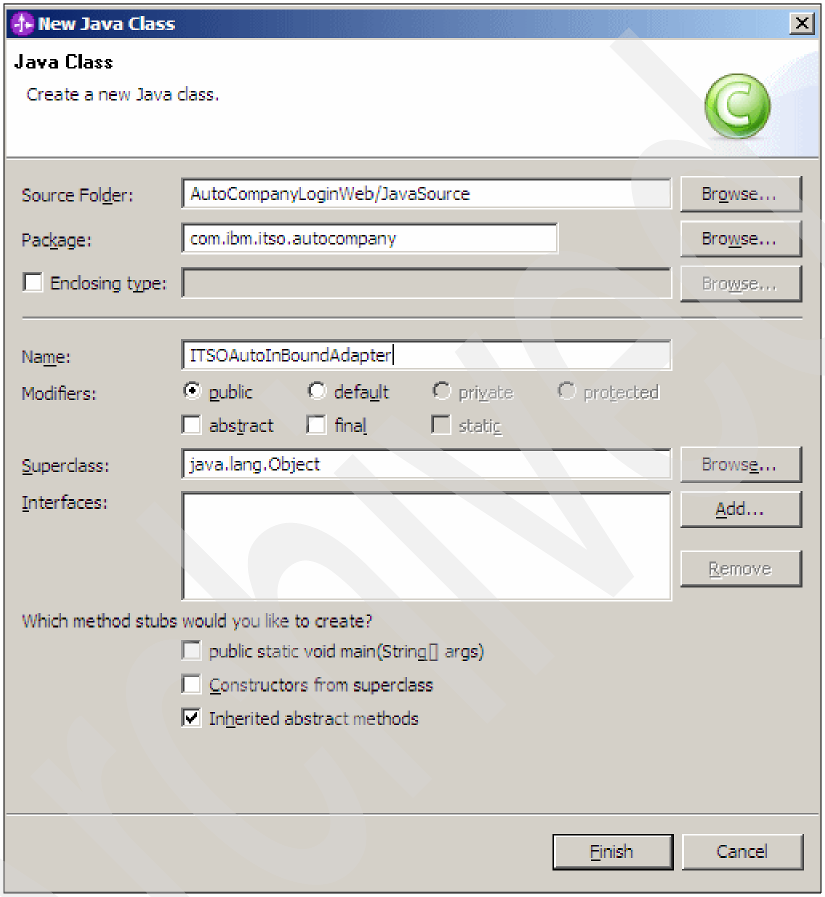


Figure 12-38 Creating the ITSOAutoInBoundAdapter class

The new Java file is opened in the editor. Replace the code template with the code in Example B-1 on page 507.

The important details of the code are shown in the following snippets.

The code uses the SCA API to create a ServiceManager instance and locate a BOFactory (Figure 12-39).

```
ServiceManager serviceManager = new ServiceManager();
BOFactory bof = (BOFactory)
serviceManager.locateService("com/ibm/websphere/bo/BOFactory");
```

Figure 12-39 Create ServiceManager and find BusinessObject Factory

It then creates the AuthenticationRequest business object and populates it (Figure 12-40).

```
DataObject authenticationRequest = bof.create(
      "http://UserAuthenticationModule", "AuthenticationRequest");
authenticationRequest.setString("customerId", customerId);
authenticationRequest.setString("password", passwd);
```

Figure 12-40 Create an AuthenticationRequest BusinessObject

It then locates the ITSOAutoInterface service and invokes the method logIn on it, passing it the created AuthenticationRequest business object (Figure 12-41).

```
ITSOAutoInterface autoService = (ITSOAutoInterface)
        serviceManager.locateService("ITSOAutoInterfacePartner");
DataObject inventoryResponse = (DataObject)
        autoService.logIn(authenticationRequest);
```

Figure 12-41 Invoke login on ITSOAutoInterfacePartner

The response DataObject is converted to XML by invoking the writeDataObject method on the BOXMLSerializer. The returned XML (in the form of a byte array) is converted to a JSONObject by invoking the static method toJSONObject(String) on the XML class (which is part of the json library, json.jar). JSONObject is then converted to its string representation (using toString( )) and returned (Figure 12-42).

```
ByteArrayOutputStream bos = new ByteArrayOutputStream();
BOXMLSerializer xmlSerializerService = (BOXMLSerializer) new ServiceManager()
        .locateService("com/ibm/websphere/bo/BOXMLSerializer");
xmlSerializerService.writeDataObject(inventoryResponse, null,
        "InventoryResponse", bos);
JSONObject jo = XML.toJSONObject(bos.toString());
return jo.toString();
```

Figure 12-42 Returning a JSON object by converting the BusinessObject to XML and then to JSONObject

12.8 Deploy and test

In order to deploy and test the entire scenario, we make use of the integrated test environment on WebSphere Integration Developer v6.0.2.

There are two Web service projects to import from the additional materials (UserAuthentication and InventoryService). Then deploy the mediation and Web service projects to the WebSphere Enterprise Service Bus test server and test the adapter by launching the login.jsp in a browser. Finally, make sure that everything works by checking the message log database.

12.8.1 Import the Web services

To do this:

1. Click **File** → **Import** → **Project Interchange** → **Browse** and select **UserAuthenticationModuleApp.zip**. Leave the current workspace directory as the Project location root (Figure 12-43). Click **Select All** → **Finish**.
2. Repeat the import for InventoryModuleApp.zip.

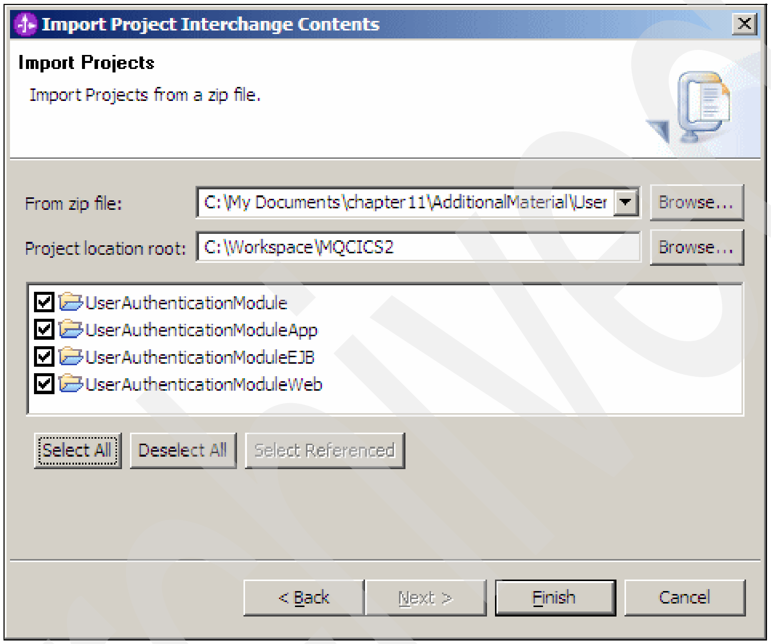


Figure 12-43 Importing the UserAuthentication Web service project interchange file

12.8.2 Create a test server and deploy the applications

To do this:

1. If you do not have a server created already, then click **File** → **New** → **Other** → **Server**. Select **localhost** as the host name and **WebSphere ESB Server 6.0** (under IBM), and click **Next** (shown in Figure 12-44). Accept all default settings in the next dialog box and click **Next** (Figure 12-44).

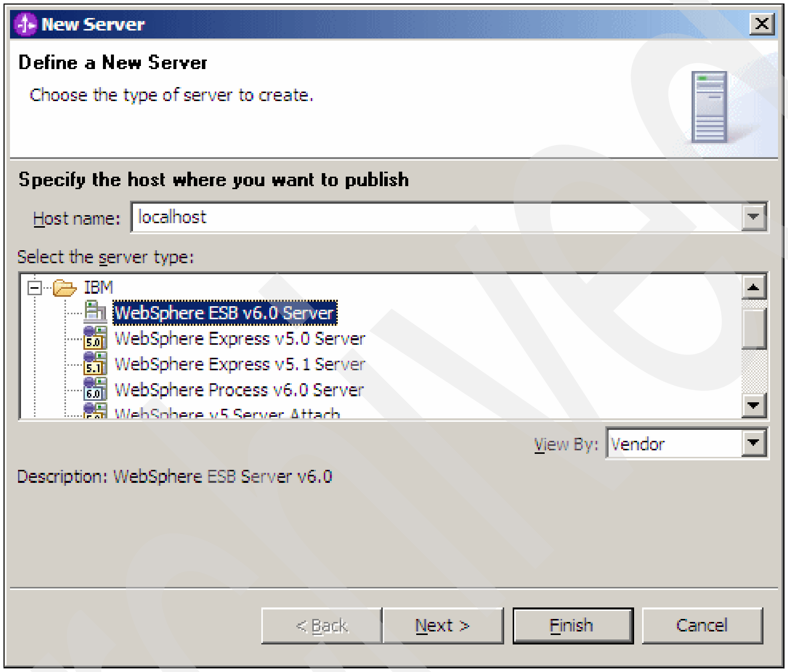


Figure 12-44 Creating a new test server

2. If you have a server created and running, right-click the server and click **Add and Remove Projects**.
3. Click **Add** to add each of the three projects in the Configured projects pane:
   - ITSOAutoMediationModule
   - UserAuthenticationModule
   - InventoryModuleApp

4. Click **Finish**. Wait until the console finishes tracing the deployment and the applications are running.

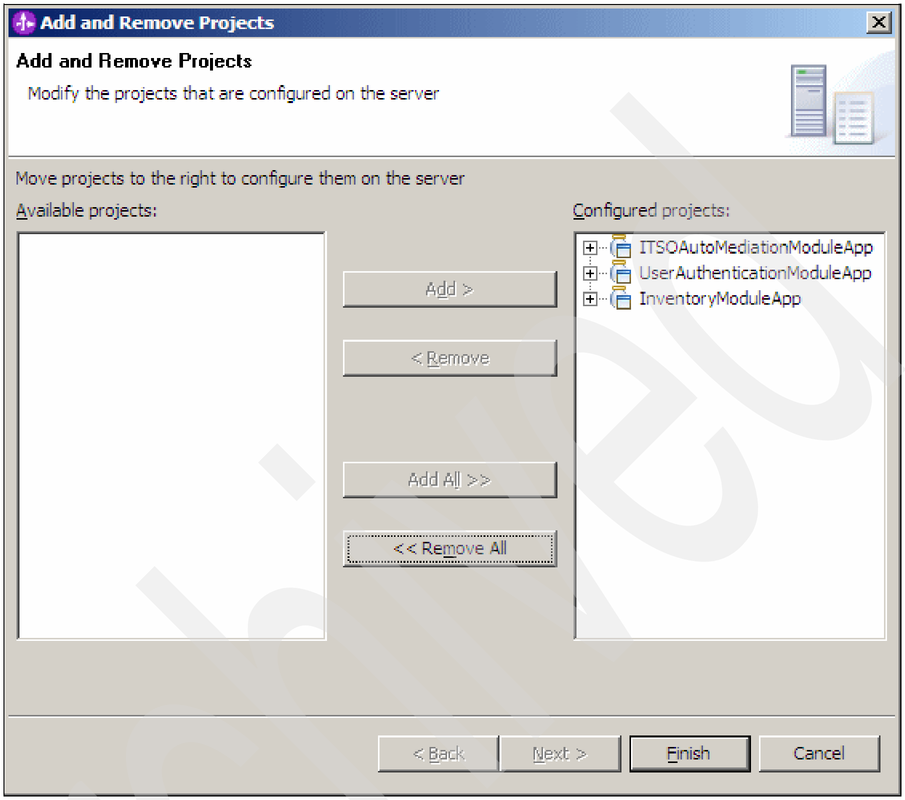


Figure 12-45 Adding projects to the test server

Test the adapter

To do this:

1. To test the adapter do one of the following:
   - Open a browser window and point it to:

     `http://localhost:9080/AutoCompanyLoginWeb/login.jsp`

   - Right-click **logon.jsp** in the AutoCompanyLoginWeb project and click **Run** → **Run on Server**.

   This opens up the login.jsp page.

2. Enter customerID `customerId_3` and password `customerId_3` and click **Signin**.

The sample UserAuthentication Web service used in this scenario is configured to authenticate successfully any customer ID as long as the password is the same as the customer ID.

Similarly, the InventoryService Web service is configured to return valid car models for customer IDs (customerId_1 .. customerID_9). For each valid customer ID it returns a randomly generated list of 20 car models.

This sends a JSON RPC request to the InBoundAdapter object that invokes the logIn operation on the mediation module.

The logIn operation on the mediation module will:

1. Record the logIn request in the message logger.
2. Authenticate with the UserAuthentication Web service.
3. Record the logIn response in the message logger.
4. Invoke the Inventory Web service.
5. Record the inventory response in the message logger.
6. Return the response back to the InBoundAdapter object that serializes the DataObject into XML and then into JSON, which is returned to the client, and the JavaScript code on the client uses this JSON to populate the list of car models.

This is shown in Figure 12-46.

Figure 12-46 Testing the application

A subset of the console trace is shown in Example 12-4, which may be useful in understanding the code fragments.

Example 12-4 Subset of console trace

```
[22/06/07 22:14:12:672 BST] 00000066 ServletWrappe A   SRVE0242I: [ITSOAutoMediationModuleApp] [/AutoCompanyLoginWeb]
[/login.jsp]: Initialization successful.
[22/06/07 22:14:13:391 BST] 00000066 JSONRPCBridge I   registered local arg resolver
com.metaparadigm.jsonrpc.HttpServletRequestArgResolver for local class javax.servlet.http.HttpServletRequest with context
javax.servlet.http.HttpServletRequest
[22/06/07 22:14:13:391 BST] 00000066 JSONRPCBridge I   registered local arg resolver
com.metaparadigm.jsonrpc.HttpSessionArgResolver for local class javax.servlet.http.HttpSession with context
javax.servlet.http.HttpServletRequest
```

```
[22/06/07 22:14:13:406 BST] 00000066 JSONRPCBridge I   registered local arg resolver
com.metaparadigm.jsonrpc.JSONRPCBridgeServletArgResolver for local class com.metaparadigm.jsonrpc.JSONRPCBridge with
context javax.servlet.http.HttpServletRequest
[22/06/07 22:14:42:812 BST] 00000066 JSONRPCServle I   auto_session_bridge=true, keepalive=true
[22/06/07 22:14:42:828 BST] 00000066 ServletWrappe A   SRVE0242I: [ITSOAutoMediationModuleApp] [/AutoCompanyLoginWeb]
[com.metaparadigm.jsonrpc.JSONRPCServlet]: Initialization successful.
[22/06/07 22:14:42:906 BST] 00000066 JSONRPCBridge I   analyzing com.ibm.itso.autocompany.ITSOAutoInBoundAdapter
[22/06/07 22:14:42:953 BST] 00000066 SystemOut     O ITSOAuto::logIn ************ creating service manager
[22/06/07 22:14:43:188 BST] 00000066 SystemOut     O ITSOAuto::logIn ************ BOFactory created
[22/06/07 22:14:43:188 BST] 00000066 SystemOut     O ITSOAuto::logIn ************ Authentication Request object created
[22/06/07 22:14:43:188 BST] 00000066 SystemOut     O ITSOAuto::logIn ************ locating ITSO Auto service
[22/06/07 22:14:43:250 BST] 00000066 SystemOut     O ITSOAuto::logIn ************ logging in to ITSO Auto service
...
[22/06/07 22:14:50:516 BST] 00000067 ServletWrappe A   SRVE0242I: [UserAuthenticationModuleApp]
[/UserAuthenticationModuleWeb] [UserAuthenticationServiceExport1_UserAuthenticationServiceHttpPort]: Initialization
successful.
[22/06/07 22:14:51:094 BST] 00000067 SystemOut     O UserAuthenicationComponentImpl::authenticateUser +++++++++ User
Authentication Service invoked
[22/06/07 22:14:51:109 BST] 00000067 SystemOut     O <?xml version="1.0" encoding="UTF-8"?>
<_:test2 xsi:type="user:AuthenticationRequest" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:user="http://UserAuthenticationModule" xmlns:_="test1">
  <customerId>customerId_3</customerId>
  <password>customerId_3</password>
</_:test2>
[22/06/07 22:14:51:109 BST] 00000067 SystemOut     O UserAuthenicationComponentImpl::authenticateUser +++++++++
customerId=customerId_3, password=customerId_3
[22/06/07 22:14:51:109 BST] 00000067 SystemOut     O UserAuthenicationComponentImpl::authenticateUser +++++++++ BOFactory
created
[22/06/07 22:14:51:109 BST] 00000067 SystemOut     O UserAuthenicationComponentImpl::authenticateUser +++++++++
Authentication Response object created
[22/06/07 22:14:51:109 BST] 00000067 SystemOut     O UserAuthenicationComponentImpl::authenticateUser +++++++++ returning
authentication response object
...
[22/06/07 22:14:51:438 BST] 00000067 ServletWrappe A   SRVE0242I: [InventoryModuleApp] [/InventoryModuleWeb]
[InventoryServiceExport1_InventoryServiceHttpPort]: Initialization successful.
[22/06/07 22:14:51:703 BST] 00000067 SystemOut     O InventoryComponentImpl::getAutoList -------- Inventory Service
invoked
[22/06/07 22:14:51:703 BST] 00000067 SystemOut     O <?xml version="1.0" encoding="UTF-8"?>
<_:test2 xsi:type="user:AuthenticationResponse" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:user="http://UserAuthenticationModule" xmlns:_="test1">
  <response>success</response>
  <customerId>customerId_3</customerId>
</_:test2>
[22/06/07 22:14:51:703 BST] 00000067 SystemOut     O InventoryComponentImpl::getAutoList -------- customerId=customerId_3,
authResponse=success
[22/06/07 22:14:51:703 BST] 00000067 SystemOut     O InventoryComponentImpl::getAutoList -------- BOFactory created
[22/06/07 22:14:51:703 BST] 00000067 SystemOut     O InventoryComponentImpl::getAutoList -------- Inventory Response
object created
[22/06/07 22:14:51:703 BST] 00000067 SystemOut     O InventoryComponentImpl::getAutoList -------- returning inventory
response object
[22/06/07 22:14:51:750 BST] 00000066 SystemOut     O <?xml version="1.0" encoding="UTF-8"?>
<InventoryResponse xsi:type="inventory:InventoryResponse" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:inventory="http://InventoryModule">

<modelList>&lt;AutoList>&lt;Car>Car_1&lt;/Car>&lt;Car>Car_0&lt;/Car>&lt;Car>Car_0&lt;/Car>&lt;Car>Car_22&lt;/Car>&lt;Car>C
ar_26&lt;/Car>&lt;Car>Car_28&lt;/Car>&lt;Car>Car_6&lt;/Car>&lt;Car>Car_3&lt;/Car>&lt;Car>Car_7&lt;/Car>&lt;Car>Car_20&lt;/
Car>&lt;Car>Car_6&lt;/Car>&lt;Car>Car_7&lt;/Car>&lt;Car>Car_5&lt;/Car>&lt;Car>Car_3&lt;/Car>&lt;Car>Car_10&lt;/Car>&lt;Car
>Car_1&lt;/Car>&lt;Car>Car_10&lt;/Car>&lt;Car>Car_26&lt;/Car>&lt;Car>Car_0&lt;/Car>&lt;Car>Car_17&lt;/Car>&lt;/AutoList></
modelList>
  <response>success</response>
  <customerId>customerId_3</customerId>
</InventoryResponse>
[22/06/07 22:14:51:766 BST] 00000066 SystemOut     O ITSOAuto::logIn ************ ITSOAuto:: JSON String
{"InventoryResponse":{"modelList":"<AutoList><Car>Car_1<\/Car><Car>Car_0<\/Car><Car>Car_0<\/Car><Car>Car_22<\/Car><Car>Car
_26<\/Car><Car>Car_28<\/Car><Car>Car_6<\/Car><Car>Car_3<\/Car><Car>Car_7<\/Car><Car>Car_20<\/Car><Car>Car_6<\/Car><Car>Car
```

```
_7<\/Car><Car>Car_5<\/Car><Car>Car_3<\/Car><Car>Car_10<\/Car><Car>Car_1<\/Car><Car>Car_10<\/Car><Car>Car_26<\/Car><Car>Car
_0<\/Car><Car>Car_17<\/Car><\/AutoList>","xmlns:inventory":"http://InventoryModule","xmlns:xsi":"http://www.w3.org/2001/XM
LSchema-instance","xsi:type":"inventory:InventoryResponse","response":"success","customerId":"customerId_3"}}
```


The list of spare parts is empty right now. In Chapter 13 on page 465 we add the code to populate it in the OutBound scenario.

Checking the message log

Normally, the Message Logger primitive logs everything in the Cloudscape™ database that comes with WebSphere Integration Developer, or in the DB/2 database if that is configured for the runtime. It is easier to inspect the logs in DB/2 than it is in Cloudscape, because DB/2 has a more advanced control center. This account is only for Cloudscape and is relevant to the development environment.

1. In the Servers view click the red button to stop the server, as the server holds a lock on the Cloudscape database while it is running.
2. Run:

```
<WebSphere Integration Developer Install
Directory>\runtimes\bi_v6\cloudscape\bin\embedded\cview.bat
```

3. To access the GUI interface to the database (Figure 12-47), click **File** → **Open** and select **<WebSphere Integration Developer Install Directory>\pf\esb\databases\EsbLogMedDB**.

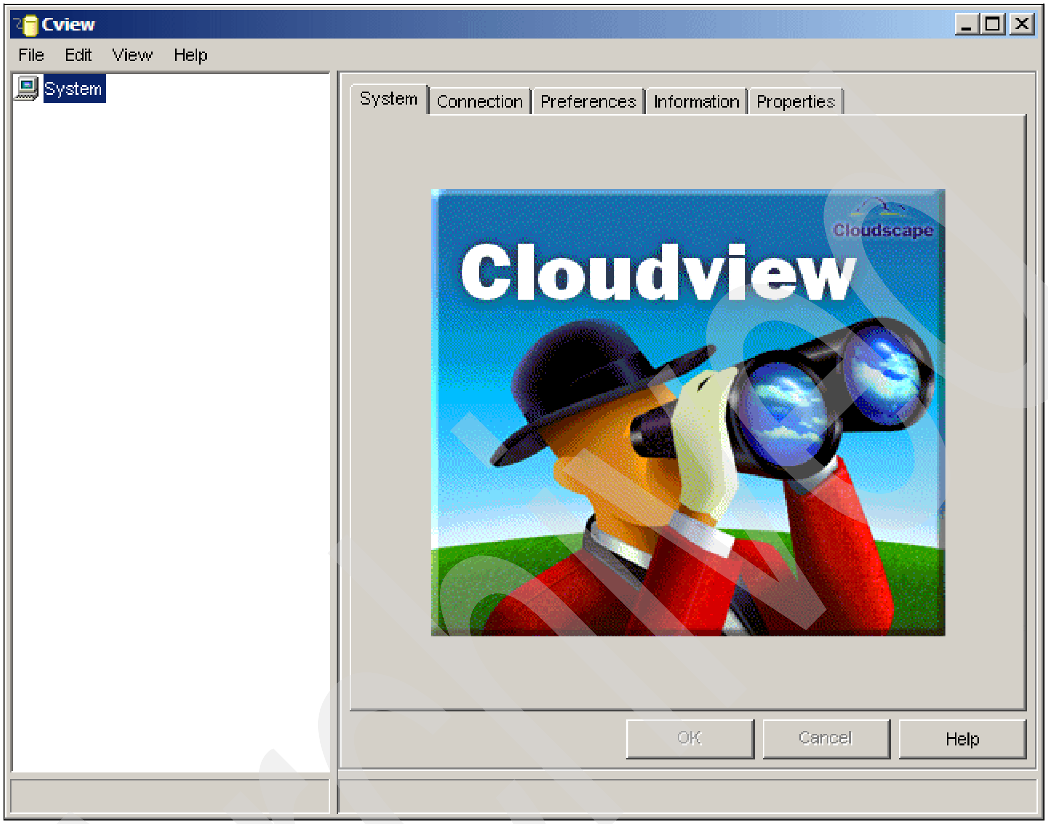


Figure 12-47 Cloudscape GUI

4. Expand **Tables** in the side menu and click **MSGLOG**, which is the default table used to store the message logs (Figure 12-48).
5. Click the **Data** tab in the right-hand panel. It stores the time stamp, a message ID, the primitive who logged the message, mediation where this primitive resides, and the message contents.

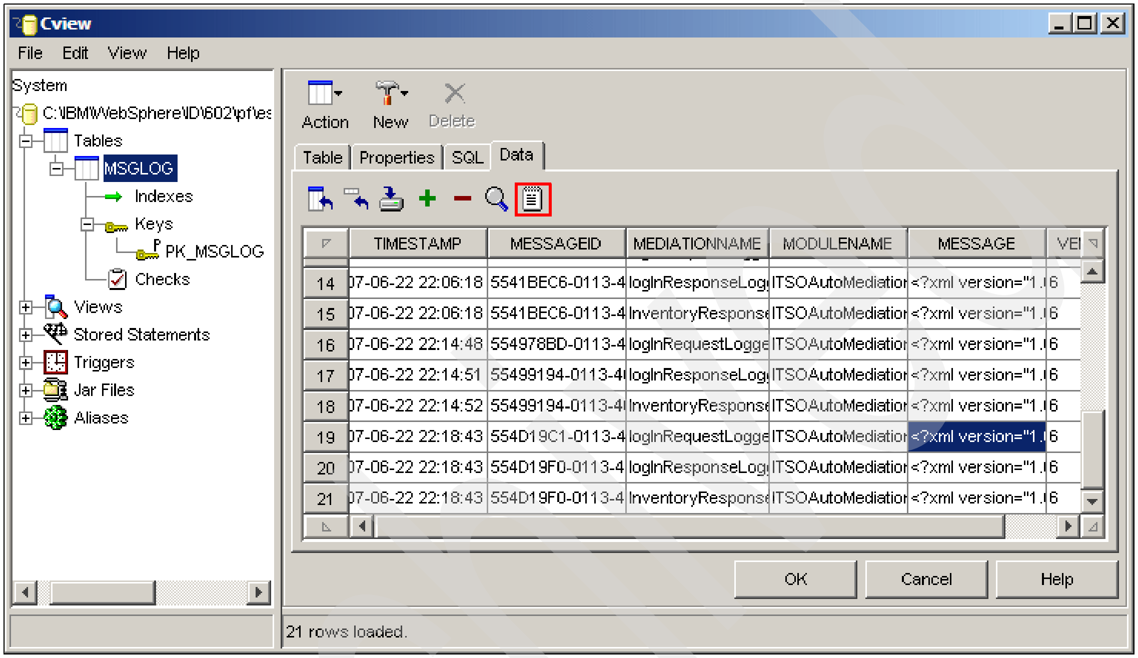


Figure 12-48 EsbLogMedDB database

6. Select one of the messages there (for example, the one logged by logInRequestLogger) and click the Text Editor icon to display the message in another editor (shown in Figure 12-49).

   Since we chose the default Message Logger settings, only the message body is logged in the table.

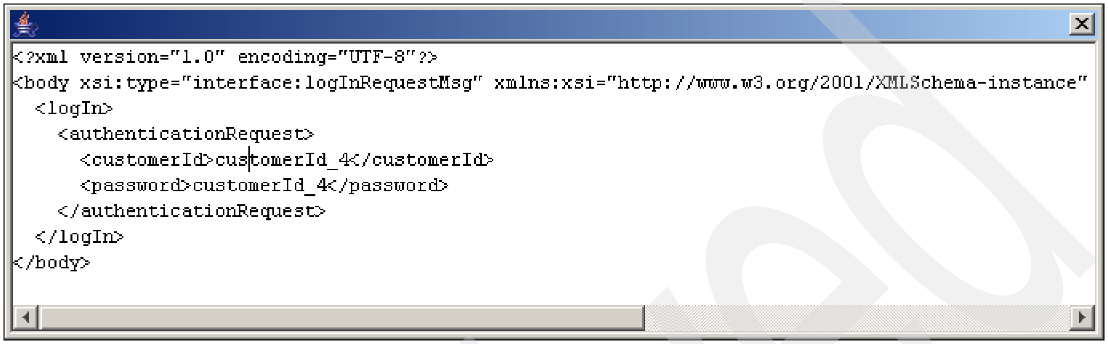


Figure 12-49 Checking the message log table in Cloudscape

12.9 Summary

The inbound scenario provides a pattern for invoking a mediation flow of WebSphere Enterprise Service Bus using a standalone reference. Our example showed how to use the pattern in the context of a JSON RPC Web client. The same pattern could be used to invoke a service directly from any JSP, Servlet, or EJB.

13

Lightweight Web service integration using http

In this chapter we describe how the WebSphere Enterprise Service Bus can be used to connect to lightweight services on the Internet (outbound connections). In the previous chapter we described how to allow lightweight JSON RPC-based clients (that use non-SOAP data) to communicate to the standard SOAP-based Web services.

This chapter describes the design and implementation of an outbound scenario that demonstrates how the WebSphere Enterprise Service Bus can be used to connect to an existing service on the Internet that accepts non-SOAP data over HTTP. This constitutes an outbound connection (relative to ESB) scenario.

Even though the outbound scenario in this chapter makes use of JSON RPC as the communication mechanism, it can easily be replaced with a similar lightweight RPC protocol such as XML-RPC. Furthermore, some of the design and architecture concepts presented in this chapter will hold true for most HTTP communication using ESB. For more details on alternative Internet technologies refer to 1.6, “Web 2.0 - next generation Web services” on page 43.

13.1 Auto company business scenario revisited

At some point, the auto company decided to out source the manufacturing of some of their spare parts to a different third-party company. The third-party company already had a list of lightweight services, as most of its clients were Web-based *Web 2.0* style clients. It exposed its services over JSON RPC.

The spare parts ordering system now needs to integrate this third-party spare parts system with its existing infrastructure so that its dealers can order all spare parts through the same Web-based front end. This is relatively easy, as the Web clients were already JSON RPC-based (Figure 13-1).

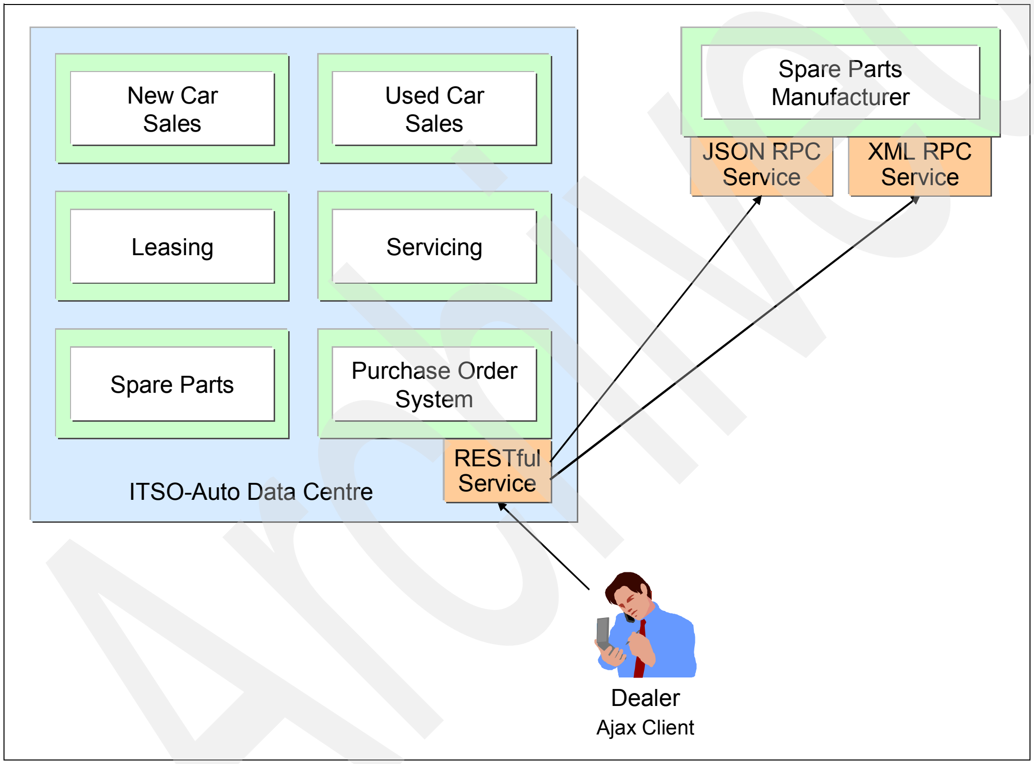


Figure 13-1 Connecting services to the Internet

However, they would like to trace all requests going to this third-party service and would like to log them in their database. They want to investigate whether the WebSphere Enterprise Service Bus can be used to integrate these third-party services into their existing infrastructure. This is shown in Figure 13-2.

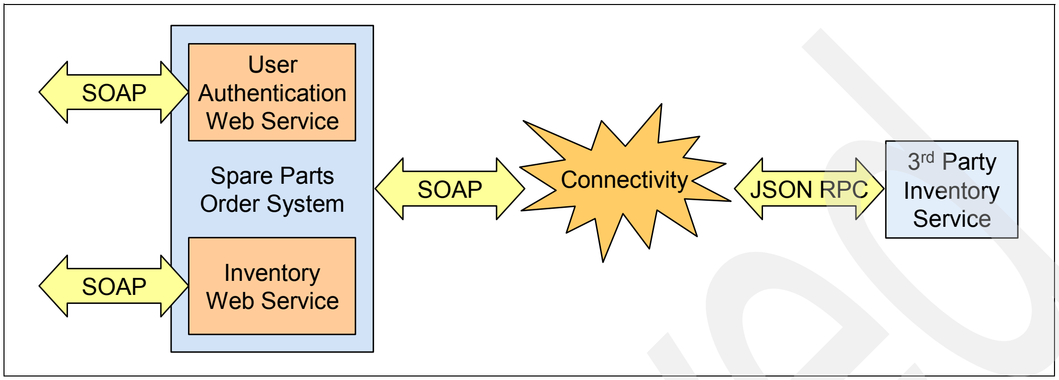


Figure 13-2 Connecting to non-SOAP based services over HTTP - outbound scenario

13.2 Implementing outbound communication

As discussed in 12.2, “Architecture patterns revisited” on page 418, the technical objective of the scenario is to provide Web service SOAP clients or Web services themselves with a means to invoke a service over HTTP that does not use SOAP. Instead, it might use XML-RPC, JSON RPC, or just plain text in a RESTful fashion.

In 5.3.2, “Scenario 6: integration of a third-party supplier” on page 141 the auto company needs to invoke the third-party spare parts service over JSON RPC so that it can log all requests to its enterprise database, while its dealers can get the full list of spare parts through the JSON RPC front end.

13.2.1 Architecture pattern

A high-level schematic diagram of the architecture pattern to support outbound HTTP communication using non-SOAP protocols is shown in Figure 13-3.

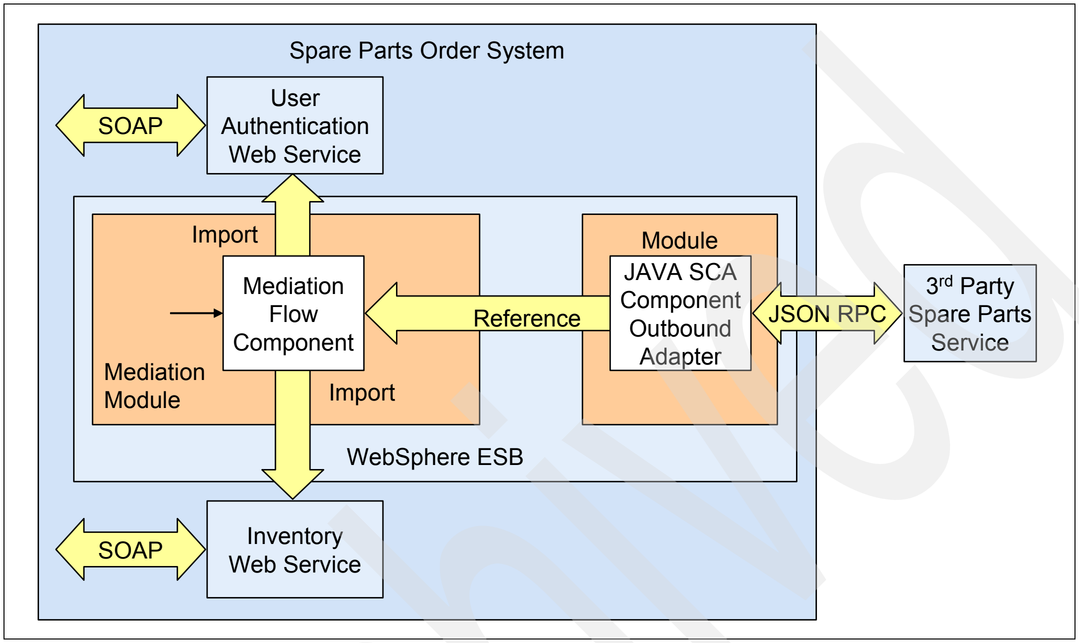


Figure 13-3 High-level architecture pattern to support outbound HTTP communication

The key feature of this architecture is the use of a Java SCA component that acts as an OutBound adapter. It receives method calls from the mediation flow component and in turn makes JSON RPC calls over http to the third-party spare parts service.

In this architecture we kept the Java SCA Component in the same mediation module as the mediation flow, to keep it simple. We could have implemented the Java component in its own module and then accessed it in the mediation module using an import.

A stimulating discussion about design alternatives for http adapters (for example, http communication in a RESTful manner) appears in the following article:

http://www-128.ibm.com/developerworks/websphere/library/techarticles/0611_flurry/0611_flurry.html

When combined with the inbound communication architecture pattern, our two examples are an end-to-end architecture for JSON RPC-based Web clients to access both standard SOAP services (the user authentication service and the inventory service) and JSON RPC services (the third-party spare parts service). This is shown in Figure 13-4.

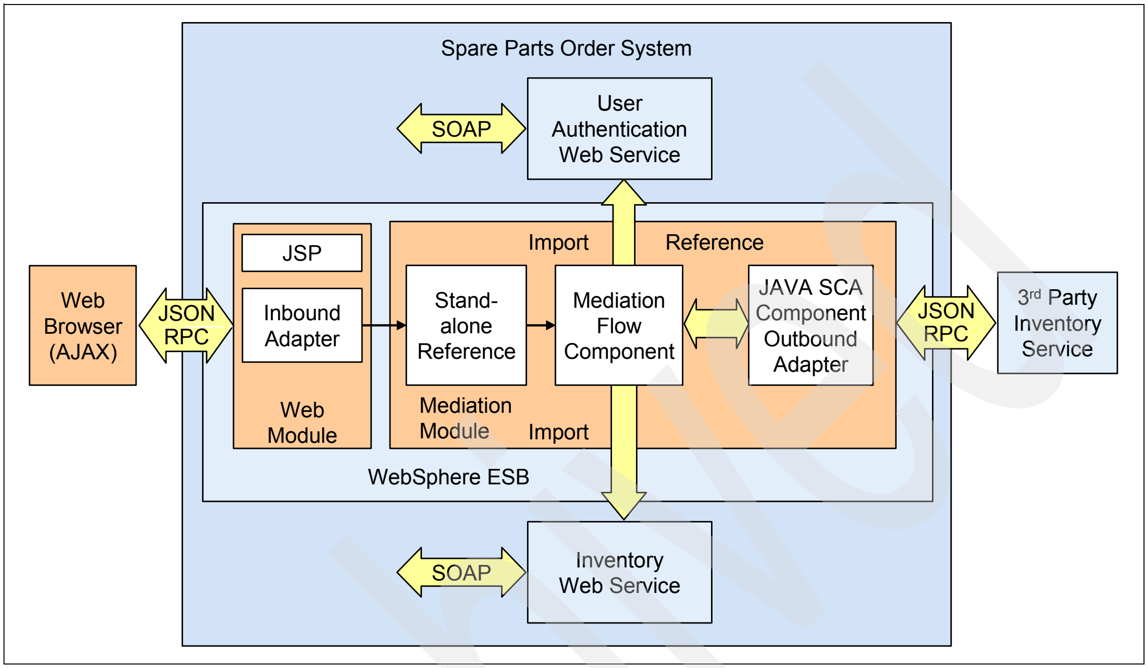


Figure 13-4 High-level end-to-end architecture pattern for inbound/outbound HTTP communication

13.2.2 Solution outline

To demonstrate the outbound scenario we extend the inbound scenario to include an outbound communication with a third-party spare parts service that uses JSON RPC, and implement an end-to-end JSON RPC architecture mediated by the WebSphere Enterprise Service Bus.

The steps consist of the following:

1. We first enhance ITSOAutoInterface to include another operation called “String getSpareParts(String)” that takes in a car model name and returns a list of spare parts for that model.
2. Since ITSOAutoInterface gets modified, we remove the existing wiring between the standalone reference and the mediation flow component and rewire it.
3. We then create a JAVA SCA component that acts as the OutBound Adapter.

4. We also create an interface for the JAVA SCA component and wire it to the mediation flow component.
5. As the JAVA SCA component invokes a JSON RPC service and deals with data in JSON format, we add the required JSON library (json.jar) to its build.path.
6. We then complete the request flow and response flow (through the mediation module) for the new operation -getSpareParts.

   During this process, we add the desired message logging primitives and required XSL transformation primitives to create the request and response flow.
7. Finally, we enhance the InBoundAdapter class to include this new operation so that the JSP page can make calls to this method over JSON RPC, and it gets routed through the mediation flow component to the OutBound Adapter component.

We need not make any changes to the JSP page, as it already has all the JavaScript code required to invoke the getSpareParts method (upon selecting a particular car model in the drop-down list).

13.3 Add getSpareParts to ITSOAutoInterface

After adding the operation to the interface, we need to regenerate the standalone reference to ITSOAutoInterface to pick up the new operation.

1. Switch to the **Business Integration** perspective → **ITSOAutoMediationModule** → **Interfaces** and open the ITSOAutoInterface in the interface editor.

2. Click **Add Request Response Operation** to create a new operation and name it `getSpareParts`. Change input1 to `modelName`. Change output1 to `sparePartsList`. Leave both types as string (Figure 13-5).

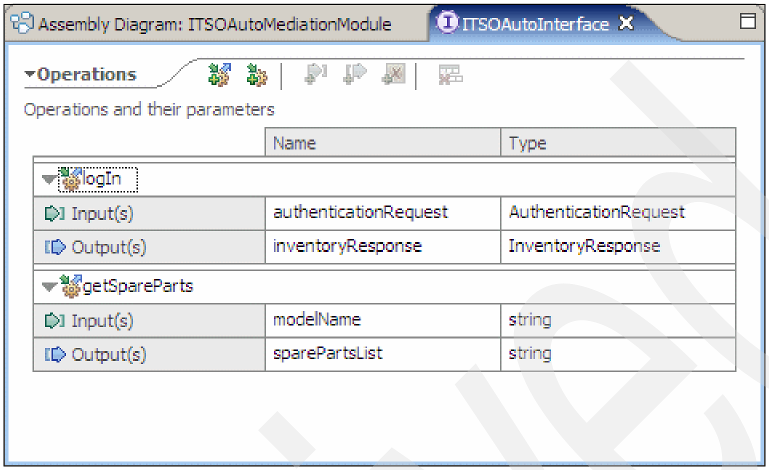


Figure 13-5 Add getSpareParts operation to ITSOAutoInterface

13.3.1 Rewire the standalone reference to pick up the new operation

Deleting the reference in the following procedure removes the previously generated Java code that converts the WSDL interface to a Java invocation. Adding the reference back again regenerates a new Java adapter to the SCA component.

1. Open the assembly diagram for ITSOAutoMediationModule. Delete the standalone reference to the ITSOAutoMediationFlowComponent by selecting the square box on the right-hand edge of the stand-alone reference and deleting it (Figure 13-6).

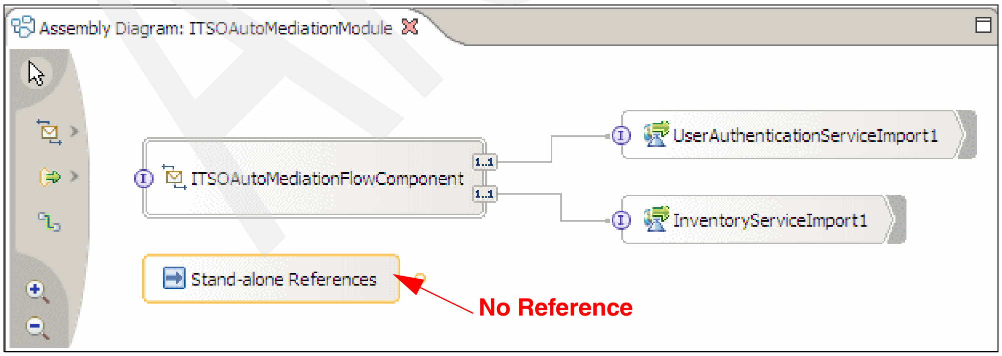


Figure 13-6 Assembly editor after removing the wiring and Java reference on the standalone reference

2. Re-wire the standalone reference to ITSOAutoMediationFlowComponent.

   The Add Wire dialog box appears and asks you to convert the WSDL interface used by the ITSOAutoMediationFlowComponent's references to a Java Interface. Respond **Yes**. Since we invoke the mediation flow component from a JSP locally, we only need Java interfaces and not WSDL interfaces.

13.3.2 Create the Java SparePartsOutbound Adapter component

Click the JAVA component in the palette  and drop it on the assembly diagram. Rename the component `SparePartsOutBoundAdapter`.

Create an interface to the SparePartsOutBoundAdapter

To do this:

1. Right-click the **Interfaces** folder (under ITSOAutoMediationModule in the business integration window) and select **New** → **Interface**. Specify `SparePartsOutBoundAdapterInterface` as the interface name (shown in Figure 13-7) and click **Finish**.

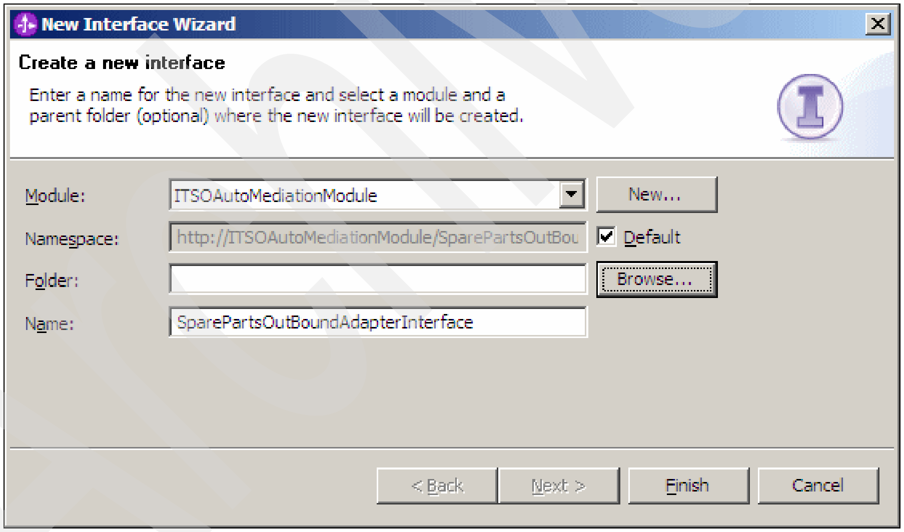


Figure 13-7 Add SparePartsOutBoundAdapterInterface

   This opens up the interface editor.

2. Click **Add Request Response Operation** to create a new operation and name it `getSpareParts`. Change input1 to `modelName`. Change output1 to `sparePartsList`. Leave both types as string (Figure 13-8).

   Earlier we added the same operation to ITSOAutoInterface for the standalone reference to invoke. Now we add it to the SparePartsOutBoundAdapter for the ITSOAutoMediationFlowComponent to invoke on SparePartsOutBoundAdapter.

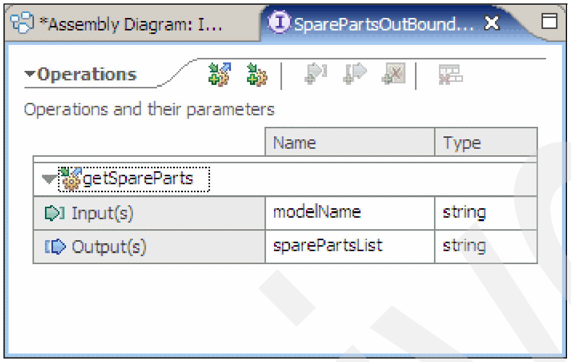


Figure 13-8 Add getSpareParts to SparePartsOutBoundAdapterInterface

Add the interface to SparePartsOutBoundAdapter

Next we add the SparePartsOutBoundAdapterInterface to the SparePartsOutBoundAdapter component.

Drag the **SparePartsOutBoundAdapterInterface** from the Interfaces folder and drop it on SparePartsOutBoundAdapter in the assembly diagram. Click **Save All**.

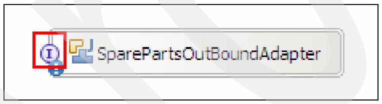


Figure 13-9 Add SparePartsOutBoundAdapterInterface to the component

Wire the SparePartsOutBoundAdapter into the flow

In the assembly editor wire the ITSOAutoMediationFlowComponent to the SparePartsOutBoundAdapter component. A dialog box appears informing you that a matching reference will be created on the source. Click **OK** (Figure 13-10).

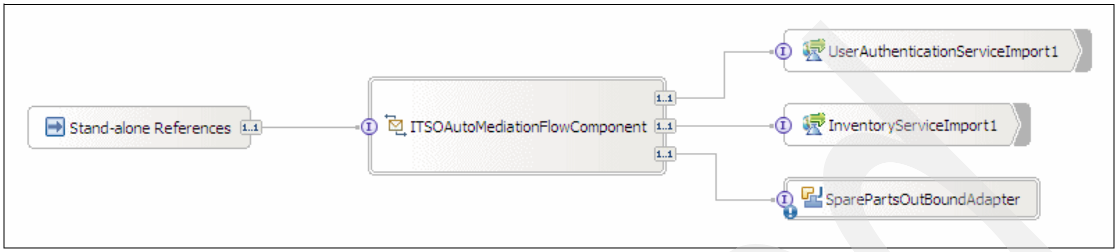


Figure 13-10 Assembly editor after wiring the Java component and the mediation flow component

13.4 Generate SparePartsOutBoundAdapter implementation

The next step is to write the code for the adapter. We do this by generating its implementation in the assembly diagram. Then we resolve the library path to include the JSON libraries to get a clean build.

In the code we first create a JSON RPC request (consisting of an ID, a method name, and the parameters to be passed in, as discussed in 1.6.6, "JSON RPC" on page 48) in JSON format. Then we do an HTTP Post to the JSON RPC Servlet on the server where the ThirdPartySparePartsService is deployed (localhost in this case), and read the response from the server (also in JSON format). The response for JSON RPC requests consists of three fields:

- Result (the result JSON string)
- ID (which is used to match responses with requests)
- Error (optional)

Next we create a JSON object out of the response, retrieve just the result JSON string, and return it.

We do not return the ID, as the JSON RPC servlet communicating with the InBoundAdapter object that issued this request to the OutBound Adapter via the mediation module inserts the ID sent by the client.

13.4.1 Write the code

Right-click **SparePartsOutBoundAdapter** and select **Generate Implementation**. In the Generate Implementation dialog box select **module.mediation.auto.itso.itso.auto** as the package name and click **OK**.

This creates SparePartsOutBoundAdapterImpl.java and opens it in the Java editor.

Find the getSpareParts(String modelName) method and replace it with the code in Figure 13-11. Click **Save**. You need to add:

```
import java.io.InputStream;
```

```
public String getSpareParts(String modelName) {
   /* URL for JSON-RPCServlet */
   String urlString = new String(
         "http://localhost:9080/ThirdPartySparePartsService/JSON-RPC");
   /* JSON-RPC request format, we use an id of 0 for all messages */
   String request = new String(
   "{\"id\": 0, \"method\": \"sparePartsService.getSpareParts\", \"params\": [\""
         + modelName + "\"]}");
   InputStream is = interactHTTP(urlString, "POST", request.getBytes());
   return getJSONResponse(is);
}
```

Figure 13-11 getSpareParts(String modelName)

The method builds a JSON RPC request using the car modelName as a parameter, calls the interactHTTP method to do the post, and returns the response from getJSONResponse. We need to add the interactHTTP and getJSONResponse methods to the class.

Copy the interactHTTP method from Figure 13-12 into the class and select **Source → Organize imports → Save**.

```
private InputStream interactHTTP(String urlString, String method, byte[] out) {
   InputStream response = null;
   try {
      URL url = new URL(urlString);
      URLConnection conn = url.openConnection();
      HttpURLConnection hConn = (HttpURLConnection) conn;
      hConn.setRequestMethod(method);
      if (out != null) {
         // send output
         conn.setDoOutput(true);
         OutputStream os = hConn.getOutputStream();
         os.write(out);
         os.close();
      }
      // get response
      response = conn.getInputStream();
   } catch (IOException ex) {
      ex.printStackTrace();
   return null;
   }
   return response;
}
```

Figure 13-12 interactHTTP method

The imports are shown in Figure 13-13.

```
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.HttpURLConnection;
import java.net.URL;
import java.net.URLConnection;
```

Figure 13-13 Imports

Finally, we add the getJSONResponse code from Figure 13-14. You need to add another import:

```
import java.io.BufferedReader
```

```
private String getJSONResponse(InputStream is) {
   String code = null;
   BufferedReader br = new BufferedReader(new InputStreamReader(is));
   try {
      code = br.readLine();
      br.close();
      is.close();
   } catch (IOException e) {
      e.printStackTrace();
   }
   System.out.println("SparePartsAdapterComponentImpl:: JSON response is "
         + code);
   JSONObject jo;
   String jsonResultString = new String("");
   try {
      jo = new JSONObject(code);
      jsonResultString = jo.getString("result");
   } catch (JSONException e1) {
      e1.printStackTrace();
   }
   System.out
         .println("SparePartsAdapterComponentImpl:: JSON result string is "
               + jsonResultString);
   return jsonResultString;
}
```

Figure 13-14 getJSONResponse(InputStream is)

There are three problems, as we have not added the JSON libraries to the build path yet.

> **TIP:** The dependency between the AutoCompanyLoginWeb project and the ITSOAutoMediation project seems to get lost at times, resulting in an error. To fix the error redo the task illustrated in Figure 12-37 on page 448, clean the workspace, and rebuild.

13.4.2 Adding JSON library (json.jar) to the build path

The SparePartsOutBoundAdapter component parses the response from the third-party spare parts service and creates a JSON object from it. It needs the JSON library (json.jar) in its build path.

1. Right-click **ITSOAutoMediationModule** and select **Properties** → **Java Build Path** → **Libraries** tab → **Add JARs**. Select **json.jar** from the AutoCompanyLoginWeb project in the JAR Selection dialog box (Figure 13-15) and click **OK**.

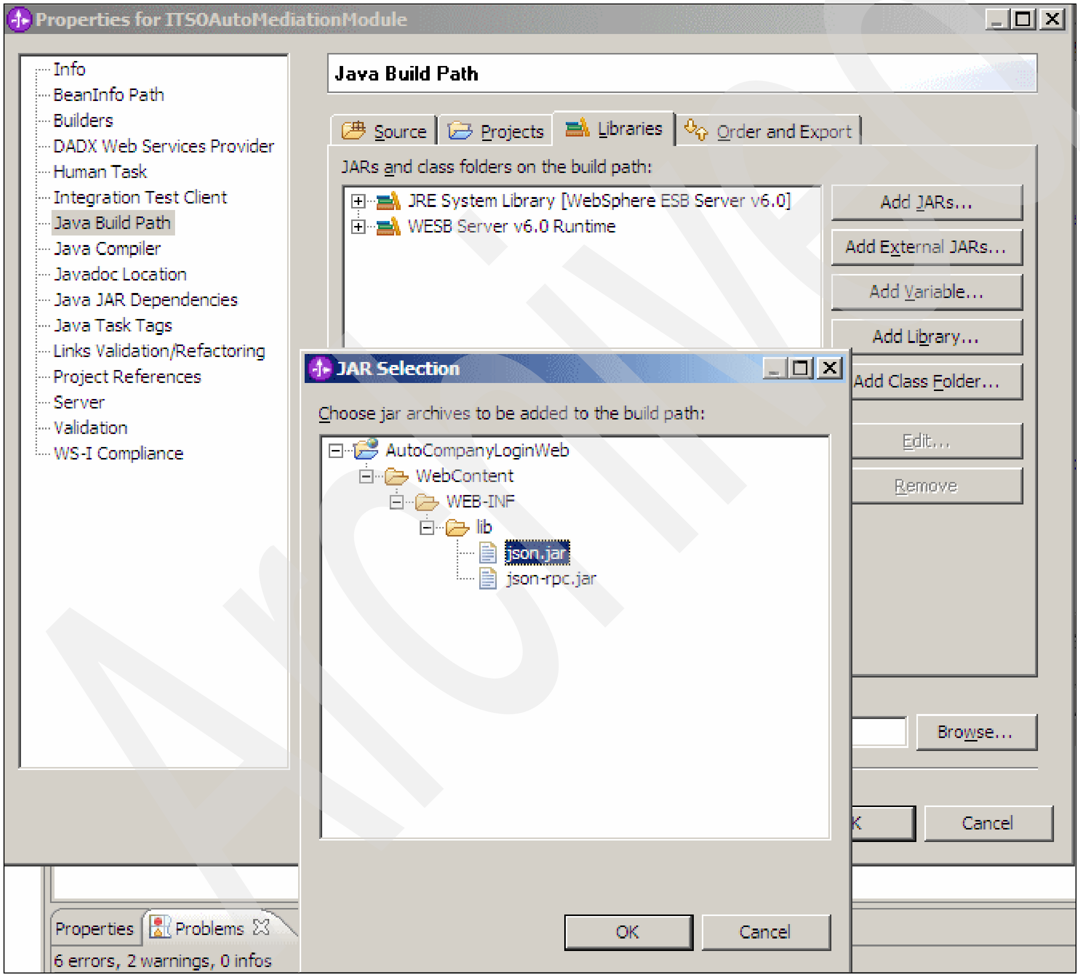


Figure 13-15 Selecting JSON library and adding it to the mediation module build path

2. Click **Source** → **Organize imports** → **Save**.

This fixes the remaining errors in the SparePartsOutBoundAdapterImpl.java file. You now need to synchronize the interface with the reference in the mediation component and create the mediation flow to invoke the adapter.

13.5 Create the getSpareParts mediation flow

First we resynchronize the reference to the newly implemented SparePartsOutBoundAdapter to get the new reference propagated to the mediation implementation, and then we create the mediation flow.

> **Tip:** WebSphere Integration Developer is a bit lazy (buggy) here, and the new Servicepartner reference does not always appear on the right-hand side of the Operation connections editor. Right-click in the Operation connection editor, click **Add Reference**, choose **SparePartsOutBoundAdapterInterfacePartner**.

1. Right-click **ITSOAutoMediationFlowComponent** in the assembly editor. Click **Synchronize Interfaces and References to Implementation** to add the new interface (and its operations) to the mediation flow component.

2. Double-click **ITSOAutoMediationFlowComponent** to open the mediation flow editor. Wire the getSpareParts operation in ITSOAutoInterface to the getSparePartsOperation in SparePartsOutBoundAdapterInterfacePartner (13.5.1, “Create request and response flows for getSpareParts” on page 480).

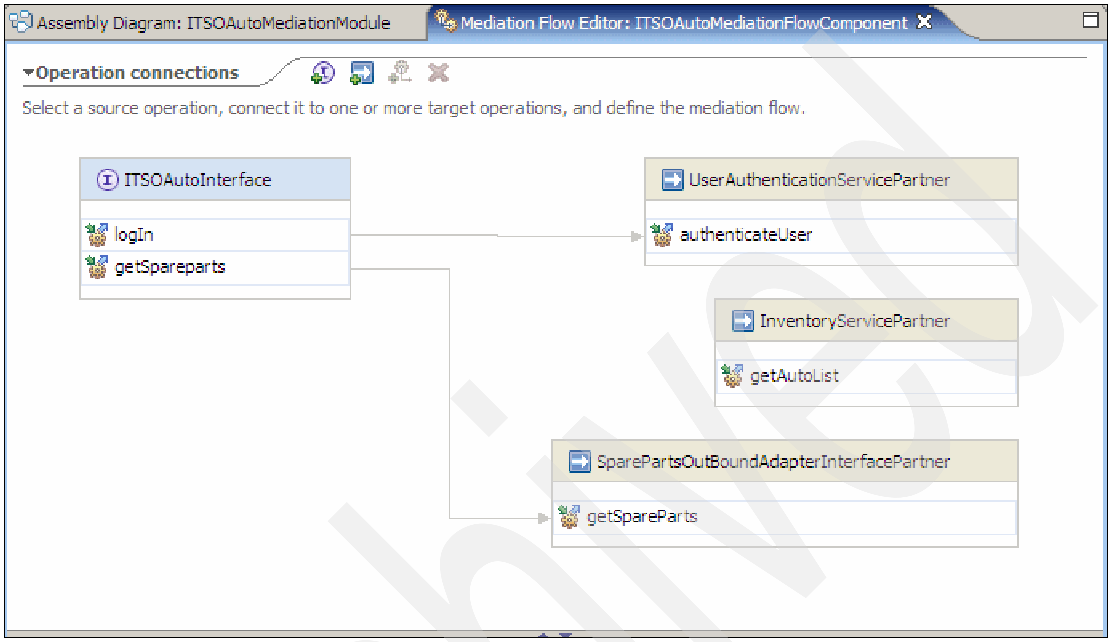


Figure 13-16 Wiring getSpareParts operation in the mediation flow editor

13.5.1 Create request and response flows for getSpareParts

The requirement is to log all of the requests and responses to the third-party spare parts service. To do this we insert a message logging and XSL transformation primitives in both the request and the response flows.

Create the request flow for getSpareParts operation

To do this:

1. In the Request tab in the lower panel of the mediation flow editor drag the **MessageLogger** from the palette and drop it into the Mediation flow. Rename it `getSparePartsRequestLogger` and accept the default properties.
2. Click the **XSLTransformation** primitive in the palette and drop it onto the mediation flow panel. Rename it `getSparePartsRequestTransformer`.
3. Wire the *out* of input to the *in* of getSparePartsRequestLogger, the out of getSparePartsRequestLogger to the in of getSparePartsRequestTransformer, and the out of

getSparePartsRequestTransformer to the in of the Callout node (Figure 13-17).

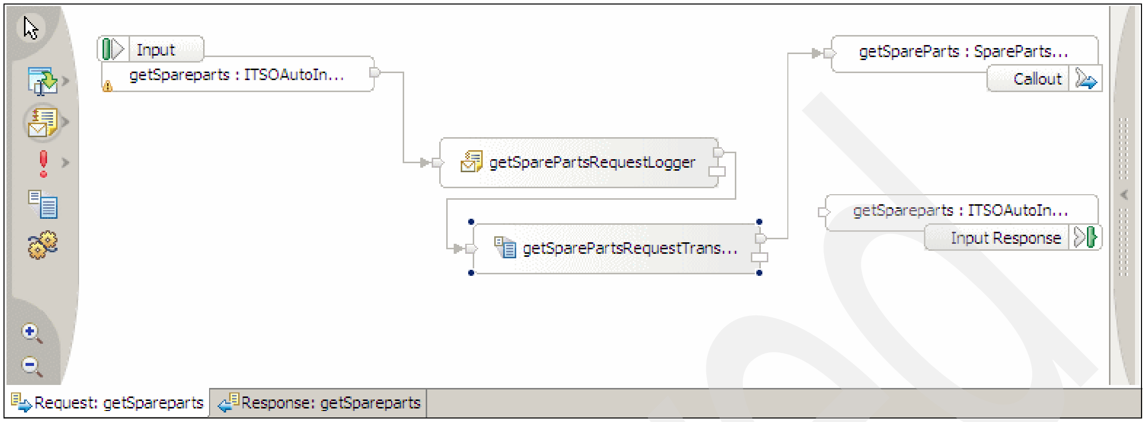


Figure 13-17 Request flow for getSpareParts operation

Next we create an XSL transformation mapping file:

1. In the getSparePartsRequestTransformer Properties view click **Details** → **New**. Accept the defaults in the Specify Message Types dialog box (Figure 13-18) and click **Finish**.

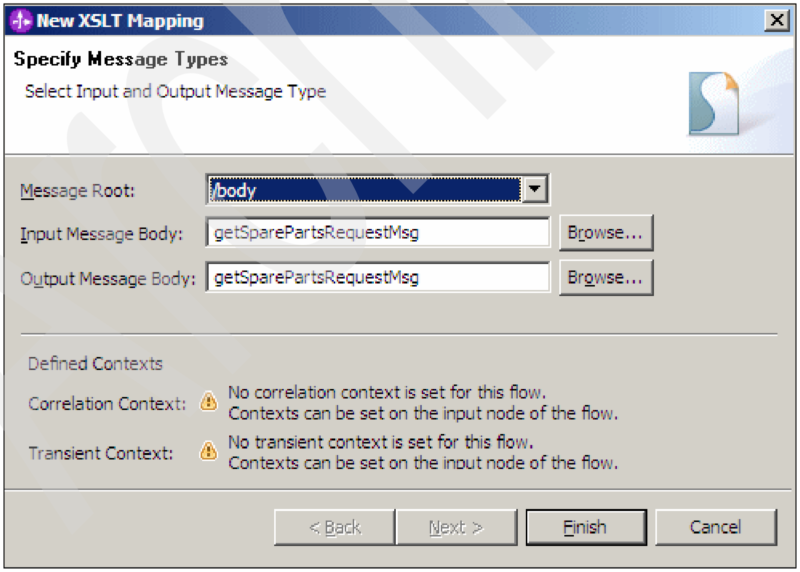


Figure 13-18 Specifying message types for getSparePartsRequestTransformer primitive

This opens the mapping file editor.

2. Select the **body** element on the source and the target. Right-click and select **Match Mapping** (Figure 13-19).

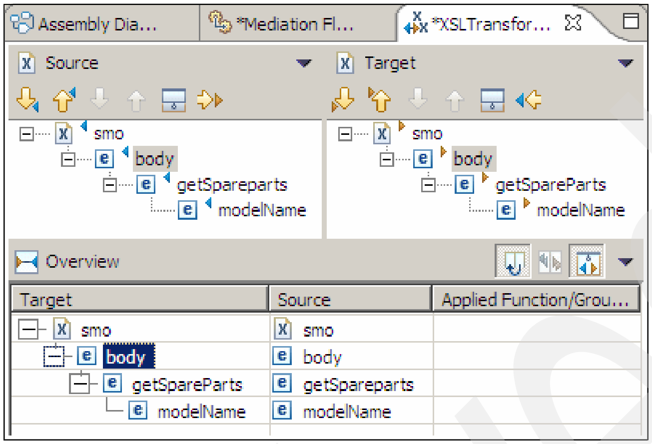


Figure 13-19 Creating a mapping for the request flow for getSpareParts operation

Create the response flow for getSpareParts operation

To do this:

1. Click the **Response** tab in the lower panel in the mediation flow editor.
2. Drag the **MessageLogger** primitive from the palette and drop it into the Mediation flow. Rename it `getSparePartsResponseLogger`. Accept its default properties.
3. Drag the **XSLTransformation** primitive from the palette and drop it into the mediation flow. Rename it `getSparePartsResponseTransformer`.

4. Wire the *out* of Callout Response Node to the *in* of getSparePartsResponseLogger, the out of getSparePartsResponseLogger to the in of getSparePartsResponseTransformer, and the out of getSparePartsResponseTransformer to the in of the Response node (Figure 13-20).

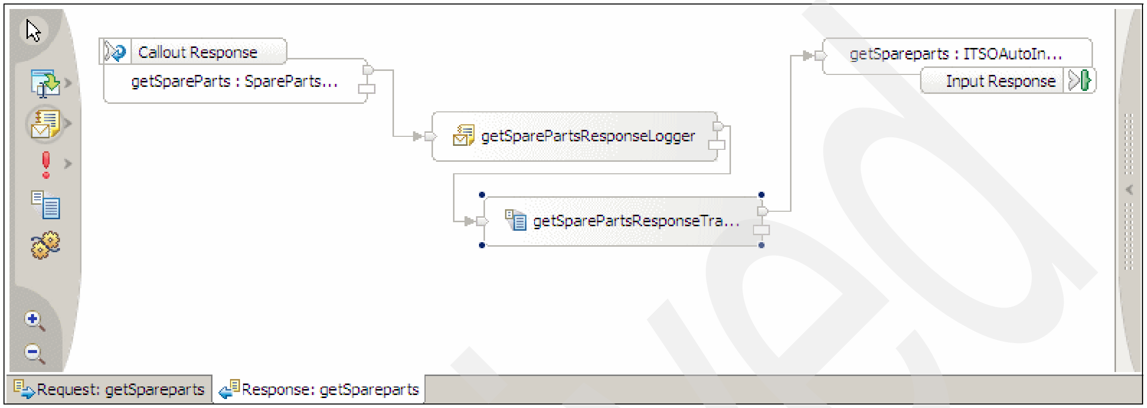


Figure 13-20 Response flow for getSpareParts operation

Next we create an XSL transformation mapping file.

1. In the getSparePartsResponseTransformer Properties view click **Details** → **New**. Accept the defaults for the Specify Message Types dialog box (Figure 13-21) and click **Finish**.

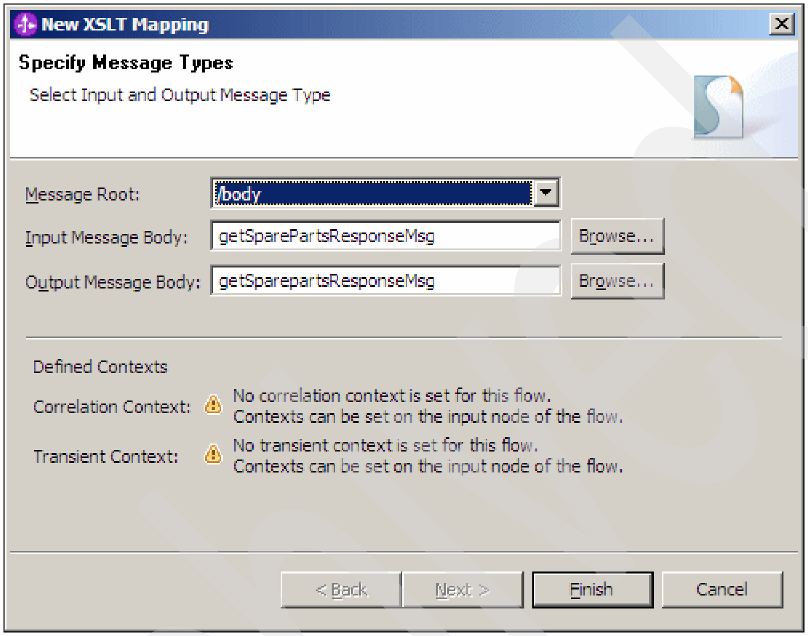


Figure 13-21 Specifying Message Types for getSparePartsResponseTransformer

This opens up the mapping file editor.

2. Select the **body** element in the source and the target. Right-click and select **Match Mapping** (Figure 13-22).

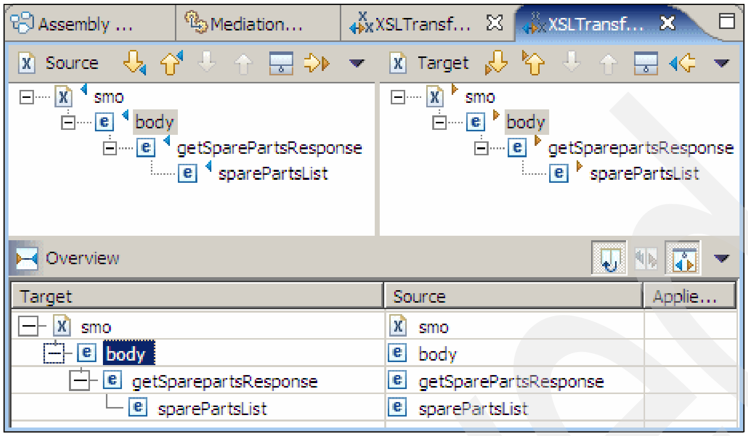


Figure 13-22 Map sparePartsList

13.6 Add getSpareParts to the InBoundAdapter class

We need to add another method to the InBoundAdapter class in the Web project to call the getSpareParts mediation. The code (Figure 13-23) is similar to the code for the logIn operation described in 12.7, “Create the InBoundAdapter class” on page 450, except that it invokes the getSpareParts operation on the ITSOMediationModule.

```
public String getSpareParts(String modelName) {
   JSONObject jo = null;
   String sparePartsResponse = null;
   try {
      System.out.println
         ("ITSOAuto::getSpareParts ************ creating service manager");
      ServiceManager serviceManager = new ServiceManager();
      BOFactory bof = (BOFactory) serviceManager
         .locateService("com/ibm/websphere/bo/BOFactory");
      System.out.println("ITSOAuto::getSpareParts ********* BOFactory created");
      DataObject sparePartsRequest = bof.createByElement(
         "http://ITSOAutoMediationModule/ITSOAutoInterface","getSpareParts");
      System.out.println
         ("ITSOAuto::getSpareParts ********* Spare Parts Request object created");
      sparePartsRequest.setString("modelName", modelName);
      System.out.println
         ("ITSOAuto::getSpareParts ********* locating ITSO Auto Service");
      ITSOAutoInterface autoService = (ITSOAutoInterface)
         serviceManager.locateService("ITSOAutoInterfacePartner");
      System.out.println
         ("ITSOAuto::getSpareParts ********* getting list of spare parts from ITSO
Auto service");
      sparePartsResponse = (String) autoService.getSpareParts(modelName);
      if (null == sparePartsResponse)
         System.out.println
         ("ITSOAuto::getSpareParts ********* sparePartsResponse is null");
      else
      System.out.println("ITSOAuto::getSpareParts ********* sparePartsResponse = "
          + sparePartsResponse);
   } catch (Exception e) {
      System.out.println(e);
   }
   return sparePartsResponse;
}
```

Figure 13-23 Add getSpareParts method to InBound Adapter class

To do this:

1. Switch to the Web perspective and select **Dynamic Web Projects AutoCompanyLoginWeb** → **Java Resources** → **Java Source** → **com.ibm.itso.autocompany**. Open the **ITSOInBoundAdapter.java** file.
2. Add the code in Figure 13-23 on page 486 for the getSpareParts method and click **Save All**.
3. Click **Project** → **Clean** → **Clean all projects** → **Build Automatically**.

There should be no errors.

13.7 Deployment and testing

To deploy and test the entire scenario, we make use of the integrated test environment on WebSphere Integration Developer (WID) 6.0.2, as we did in the inbound scenario.

The two Web services in your workspace were imported in for the inbound scenario. The instructions to do this again are in 12.8.1, “Import the Web services” on page 454.

In the section we first import the third-party spare parts service project from the project interchange zip file ThirdPartySparePartsService.zip (see Appendix A, “Additional material” on page 499), then add the required projects to the test server (created in the inbound scenario in 12.8, “Deploy and test” on page 453). Next we test the application through a browser and finally make sure that everything works by checking the message log database.

13.7.1 Import the third-party Spare Parts Service

To do this:

1. Switch to the Web perspective and click **File** → **Import** → **Project Interchange**.

2. Select **ThirdPartySparePartsService.zip** and specify your current workspace directory as the project location root. Click **Select All** (Figure 13-24) → **Finish**.

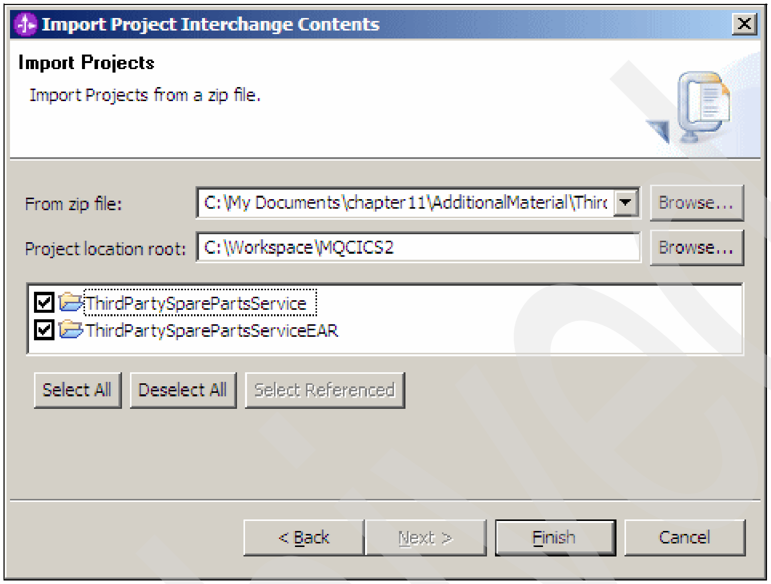


Figure 13-24 Importing the ThirdPartySparePartsService project interchange file

13.7.2 Deploy third-party spare parts service on the test server

Verify that the test server created in 12.8.2, “Create a test server and deploy the applications” on page 455, for the inbound scenario is up and running.

Right-click **WebSphere ESB Server v6.0** and select **Add and Remove Projects**. Make sure that the following four projects are in the Configured projects pane by clicking **Add>** (Figure 13-25) → **Finish**. Click the Publish icon.

- ThirdPartySparePartsServiceEAR
- ITSOAutoMediationModuleApp
- UserAuthenticationModuleApp
- InventoryModuleApp

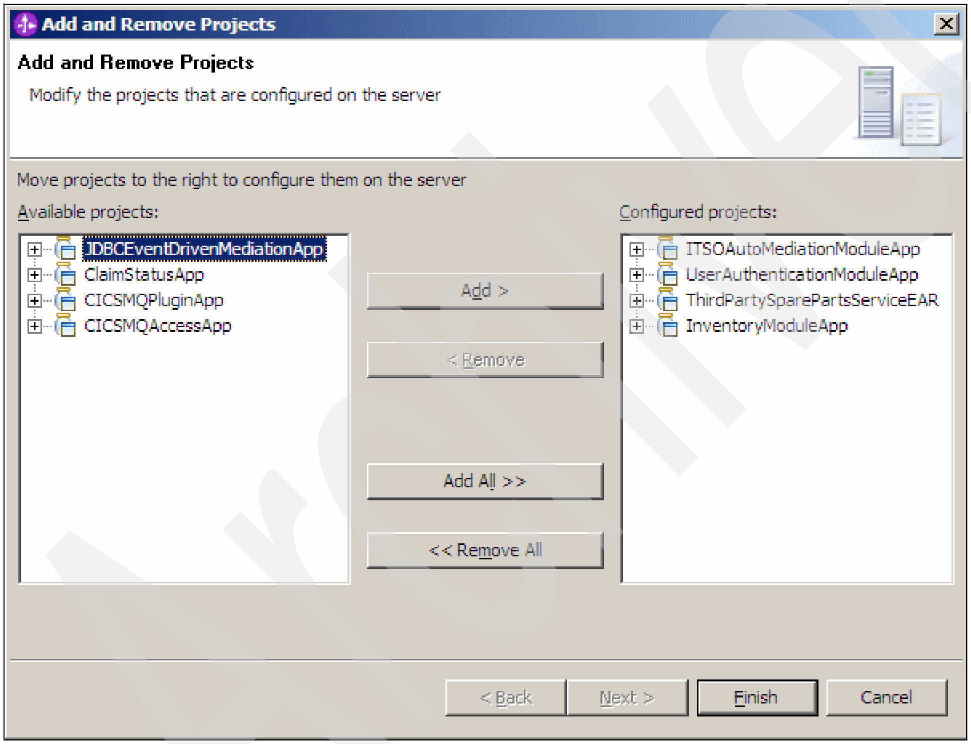


Figure 13-25 Adding ThirdPartySparePartsServiceEAR project to the test server

13.8 Test third-party spare parts service

To do this:

1. To test the application, do either of the following:
   - Open a browser window and point it to:

     `http://localhost:9080/AutoCompanyLoginWeb/login.jsp`
   - Find login.jsp in the AutoCompanyLoginWeb project and click **Run** → **Run on server**.
2. Follow the same steps as in “Test the adapter” on page 456. Enter `customerId_3` both as the customer ID and password, and click **SignIn**.

   This sends a JSON RPC request to the InBoundAdapter object that invokes the logIn operation on the mediation module.

We get back a list of car models, as shown in Figure 12-46 on page 458.

3. Next we select a particular car model (Figure 13-26 on page 492). The processing is as follows:

   a. Invoke the getSpareParts operation on the InBound Adapter object using the JSON RPC client-side library.

   b. The InBoundAdapter invokes the getSpareparts operation on the mediation module

   c. The request is routed to the Outbound Adapter object.

   d. The Outbound Adapter invokes the third-party spare parts service over JSON RPC.

   e. The OutBound Adapter object gets the response in JSON format and returns it back to the InBound Adapter via the mediation module.

   f. The InBoundAdapter returns the response back to the client, where it is parsed by the client-side JavaScript code and used to populate the list of spare parts.

Figure 13-26 Testing the application

Example 13-1 continues the trace from Example 12-4 on page 458.

Example 13-1 Trace continued

```
[23/06/07 22:04:39:625 BST] 000000ad SystemOut     O ITSOAuto::getSpareParts ************ creating service manager
[23/06/07 22:04:39:625 BST] 000000ad SystemOut     O ITSOAuto::getSpareParts ********* BOFactory created
[23/06/07 22:04:39:625 BST] 000000ad SystemOut     O ITSOAuto::getSpareParts ********* Spare Parts Request object created
[23/06/07 22:04:39:625 BST] 000000ad SystemOut     O ITSOAuto::getSpareParts ********* locating ITSO Auto service
[23/06/07 22:04:39:625 BST] 000000ad SystemOut     O ITSOAuto::getSpareParts ********* getting list of spare parts from
ITSO Auto service
...
[23/06/07 22:04:40:078 BST] 000000ac JSONRPCServle I   auto_session_bridge=true, keepalive=true
[23/06/07 22:04:40:094 BST] 000000ac ServletWrappe A   SRVE0242I: [ThirdPartySparePartsServiceEAR]
[/ThirdPartySparePartsService] [com.metaparadigm.jsonrpc.JSONRPCServlet]: Initialization successful.
[23/06/07 22:04:41:125 BST] 000000ac JSONRPCBridge I   analyzing SpareParts
```

```
[23/06/07 22:04:41:156 BST] 000000ac SystemOut     O SpareParts::getSpareParts() called
[23/06/07 22:04:41:188 BST] 000000ad SystemOut     O SparePartsAdapterComponentImpl:: JSON response is
{"result":{"list":["SparePart_20","SparePart_10","SparePart_22","SparePart_19","SparePart_26","SparePart_13","SparePart_16
","SparePart_7","SparePart_7","SparePart_7","SparePart_11","SparePart_7","SparePart_24","SparePart_27","SparePart_18","Spa
rePart_2","SparePart_19","SparePart_18","SparePart_18","SparePart_22"],"javaClass":"java.util.ArrayList"},"id":0}
[23/06/07 22:04:41:188 BST] 000000ad SystemOut     O SparePartsAdapterComponentImpl:: JSON result string is
{"list":["SparePart_20","SparePart_10","SparePart_22","SparePart_19","SparePart_26","SparePart_13","SparePart_16","SparePa
rt_7","SparePart_7","SparePart_7","SparePart_11","SparePart_7","SparePart_24","SparePart_27","SparePart_18","SparePart_2",
"SparePart_19","SparePart_18","SparePart_18","SparePart_22"],"javaClass":"java.util.ArrayList"}
...
[23/06/07 22:04:41:297 BST] 000000ad SystemOut     O ITSOAuto::getSpareParts ********* sparePartsResponse =
{"list":["SparePart_20","SparePart_10","SparePart_22","SparePart_19","SparePart_26","SparePart_13","SparePart_16","SparePa
rt_7","SparePart_7","SparePart_7","SparePart_11","SparePart_7","SparePart_24","SparePart_27","SparePart_18","SparePart_2",
"SparePart_19","SparePart_18","SparePart_18","SparePart_22"],"javaClass":"java.util.ArrayList"}
```

13.8.1 Check the message log

To do this:

1. Follow the steps in “Checking the message log” on page 460 to open the Cloudscape control center.
2. Select one of the messages (for example, the one logged by getSparePartsRequestLoggger), and click the Text Editor icon to display the message in another editor (shown in Figure 12-27 on page 440). Since we chose the default message logger settings, only the message body is logged in the table.

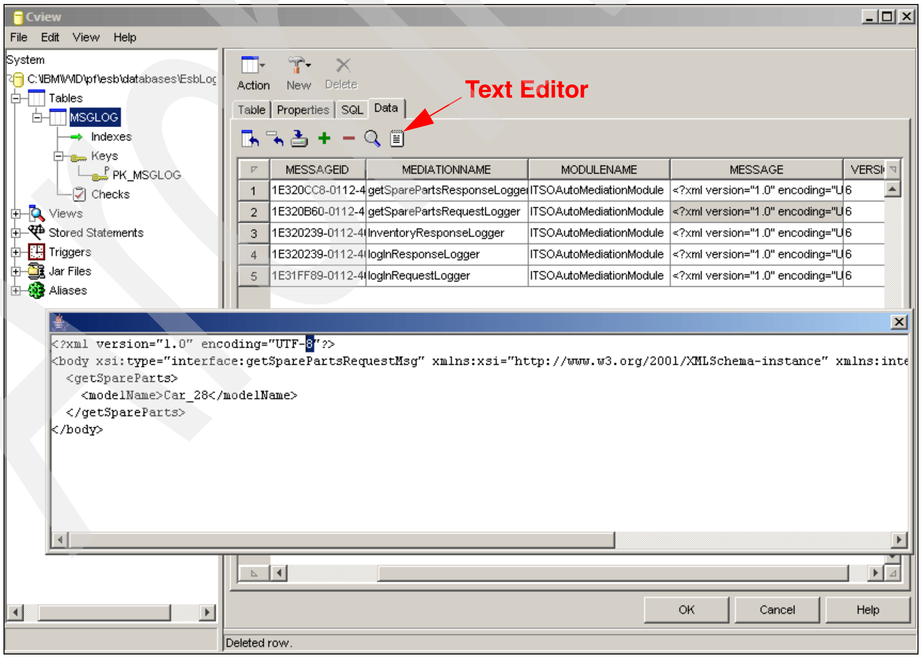


Figure 13-27 Checking the message log table in Cloudscape

13.9 Troubleshooting

If you see errors in your workspace that arise out of the generated projects (App, EJB, or WEB — ITSOAutoMediationModuleApp, ITSOAutoMediationModuleEJB, or ITSOAutoMediationModuleWeb projects, for example), you should go to the Resources view, delete these generated projects, do a clean on all projects, and then build them all again. You will also have to add these projects to the test server again. Also, if you delete the ITSOAutoMediationModuleEJB project and regenerate it, the AutoCompanyLoginWeb might still show errors. You should right-click the AutoCompanyLoginWeb project and select **Properties**. Select **Java JAR Dependencies**, deselect ITSOAutoMediationModuleEJB.jar, and select it again, and rebuild the AutoCompanyLoginWeb project.

You might experience some problems with the JavaScript functions in login.jsp in Internet Explorer®. We recommend that you use FireFox.

13.10 Summary

The outbound scenario provides a pattern for invoking a lightweight service from WebSphere Enterprise Service Bus using a Java component. Our example showed how to use the pattern in the context of a JSON RPC servlet. The same pattern could be used to invoke an servlet implementing any kind of protocol based on http.

In this, and in Chapter 12 on page 415, we have shown how to connect to and from the WebSphere Enterprise Service Bus using http. This is useful both for integrating WebSphere Enterprise Service Bus with traditional Web sites and Web clients, and also for integrating with the rapidly growing number of services that are loosely termed Web 2.0, which do not adhere to SOAP. The pattern is also useful to handle SOAP requests that do not conform to the WS-I basic profile 1.1, or for some reason the SOAP engine in WebSphere Enterprise Service Bus is unable to handle. We use the term *dirty SOAP* to characterize these cases.

The pattern is not limited to http. The examples show how to invoke mediation flows from a servlet or EJB, and how to build a Java component that can be used to connect to other environments outside the J2EE Web and EJB containers.

14

Summary

In this book we have looked at using WebSphere Enterprise Service Bus from the perspective of connecting Enterprise Applications to a service-oriented architecture. Figure 14-1 on page 496 summarizes the relationships between the main concepts explored in this book.

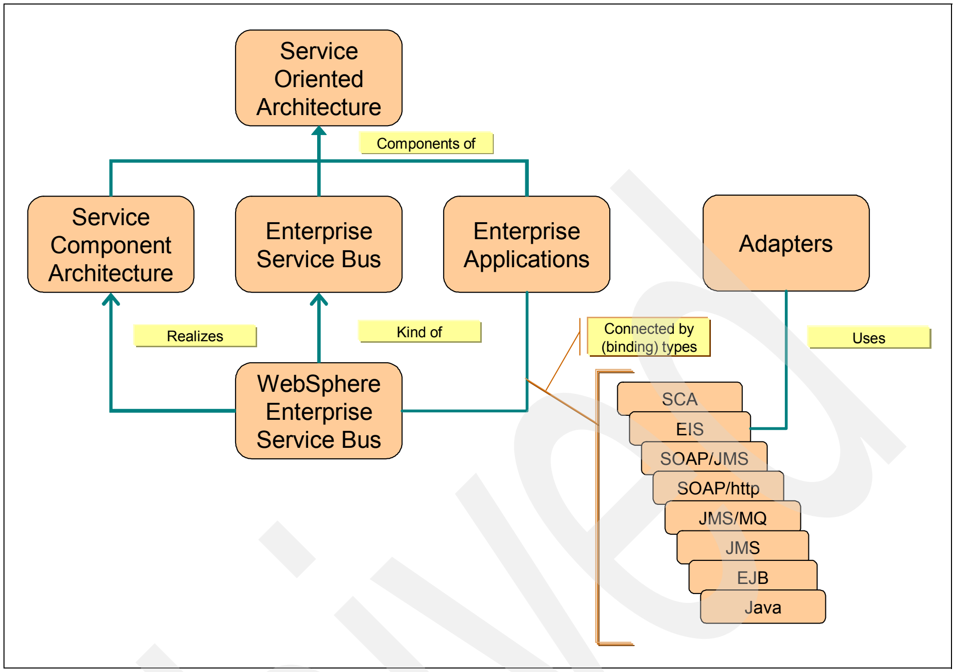


Figure 14-1 Connecting Enterprise Applications to an SOA

Part 4

Appendixes

A

Additional material

This book refers to additional material that can be downloaded from the Internet, as described below.

Locating the Web material

The Web material associated with this book is available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser to:

`ftp://www.redbooks.ibm.com/redbooks/SG24-7406.zip`

Alternatively, you can go to the IBM Redbooks Web site at:

`ibm.com/redbooks`

Select **Additional materials** and open the directory that corresponds with the book form number, SG24-7406.

Using the Web material

The additional Web material that accompanies this book is a project interchange file for WebSphere Integration Developer 6.0.2 fp 17 containing all of the projects in our examples.

There is a readme.txt file in the materials that contains further instructions.

System requirements for downloading the Web material

The following system configuration is recommended to use VMware to run the examples:

Hard disk space: 15 GB
Operating system: Windows
Processor: 2 GHZ
Memory: 2 GB

How to use the Web material

Load the SG24-7406.zip file into WebSphere Integration Developer using the Project Interchange importer and clean and rebuild all of the projects.

B

Source listings

This appendix provides source listings that were too lengthy to include in the text. A complete set of source material is available by downloading the WebSphere Integration Developer v6.0.2 Project Interchange file from the Web (Appendix A, “Additional material” on page 499).

Chapter 12, “Lightweight Web client integration using http” on page 415

WSDL and XSD definitions

The WSDL and XSD files are in two directories:

- WSDLs, which contains:
  - Figure B-1 AuthenticationRequest.xsd
  - Figure B-2 AuthenticationResponse.xsd
  - Figure B-3 InventoryResponse.xsd
  - Figure B-5 InventoryService.wsdl
  - Figure B-6 InventoryServiceExport1_InventoryServiceHttp_Service.wsdl
  - Figure B-7 UserAuthenticationService.wsdl
  - Figure B-8 UserAuthenticationServiceExport1_UserAuthenticationServiceHttp_Service.wsdl
- xsd-includes (sub-directory) containing:

  Figure B-4 http.UserAuthenticationModule.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://UserAuthenticationModule">
  <xsd:complexType name="AuthenticationRequest">
    <xsd:sequence>
      <xsd:element minOccurs="0" name="customerId" type="xsd:string"/>
      <xsd:element minOccurs="0" name="password" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

Figure B-1 AuthenticationRequest.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://UserAuthenticationModule">
  <xsd:complexType name="AuthenticationResponse">
    <xsd:sequence>
      <xsd:element minOccurs="0" name="response" type="xsd:string"/>
      <xsd:element minOccurs="0" name="customerId" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

Figure B-2 AuthenticationResponse.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://InventoryModule">
  <xsd:complexType name="InventoryResponse">
    <xsd:sequence>
      <xsd:element minOccurs="0" name="modelList" type="xsd:string"/>
      <xsd:element minOccurs="0" name="response" type="xsd:string"/>
      <xsd:element minOccurs="0" name="customerId" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

Figure B-3 InventoryResponse.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:bo="http://UserAuthenticationModule"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://UserAuthenticationModule">
  <xsd:annotation>
    <xsd:appinfo source="wid.federated.schema"/>
  </xsd:annotation>
  <xsd:include schemaLocation="../AuthenticationRequest.xsd"/>
  <xsd:include schemaLocation="../AuthenticationResponse.xsd"/>
</xsd:schema>
```

Figure B-4 http.UserAuthenticationModule.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions xmlns:bons1="http://UserAuthenticationModule"
xmlns:bons2="http://InventoryModule" xmlns:tns="http://InventoryModule/InventoryService"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
name="InventoryService" targetNamespace="http://InventoryModule/InventoryService">
  <wsdl:types>
    <xsd:schema targetNamespace="http://InventoryModule/InventoryService"
xmlns:bons1="http://UserAuthenticationModule" xmlns:bons2="http://InventoryModule"
xmlns:tns="http://InventoryModule/InventoryService"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
      <xsd:import namespace="http://UserAuthenticationModule"
schemaLocation="xsd-includes/http.UserAuthenticationModule.xsd"/>
      <xsd:import namespace="http://InventoryModule"
schemaLocation="InventoryResponse.xsd"/>
      <xsd:element name="getAutoList">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="authResponse" nillable="true"
type="bons1:AuthenticationResponse"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="getAutoListResponse">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="inventoryResponse" nillable="true"
type="bons2:InventoryResponse"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:schema>
  </wsdl:types>
    <wsdl:message name="getAutoListRequestMsg">
    <wsdl:part element="tns:getAutoList" name="getAutoListParameters"/>
  </wsdl:message>
    <wsdl:message name="getAutoListResponseMsg">
    <wsdl:part element="tns:getAutoListResponse" name="getAutoListResult"/>
  </wsdl:message>
    <wsdl:portType name="InventoryService">
    <wsdl:operation name="getAutoList">
      <wsdl:input message="tns:getAutoListRequestMsg" name="getAutoListRequest"/>
      <wsdl:output message="tns:getAutoListResponseMsg" name="getAutoListResponse"/>
    </wsdl:operation>
  </wsdl:portType>
</wsdl:definitions>
```

Figure B-5 InventoryService.wsdl

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions name="InventoryServiceExport1_InventoryServiceHttp_Service"
targetNamespace="http://InventoryModule/InventoryService/Binding"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:Port_0="http://InventoryModule/InventoryService"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:this="http://InventoryModule/InventoryService/Binding"
xmlns="http://schemas.xmlsoap.org/wsdl/">
  <wsdl:import namespace="http://InventoryModule/InventoryService"
location="InventoryService.wsdl"/>
  <wsdl:binding name="InventoryServiceExport1_InventoryServiceHttpBinding"
type="Port_0:InventoryService">
    <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="getAutoList">
      <soap:operation/>
      <wsdl:input name="getAutoListRequest">
        <soap:body use="literal"/>
      </wsdl:input>
      <wsdl:output name="getAutoListResponse">
        <soap:body use="literal"/>
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="InventoryServiceExport1_InventoryServiceHttpService">
    <wsdl:port name="InventoryServiceExport1_InventoryServiceHttpPort"
binding="this:InventoryServiceExport1_InventoryServiceHttpBinding">
      <soap:address
location="http://localhost:9080/InventoryModuleWeb/sca/InventoryServiceExport1"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

Figure B-6   InventoryServiceExport1_InventoryServiceHttp_Service.wsdl

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions xmlns:bons1="http://UserAuthenticationModule"
xmlns:tns="http://UserAuthenticationModule/UserAuthenticationService"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
name="UserAuthenticationService"
targetNamespace="http://UserAuthenticationModule/UserAuthenticationService">
  <wsdl:types>
    <xsd:schema targetNamespace="http://UserAuthenticationModule/UserAuthenticationService"
xmlns:bons1="http://UserAuthenticationModule"
xmlns:tns="http://UserAuthenticationModule/UserAuthenticationService"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
      <xsd:import namespace="http://UserAuthenticationModule"
schemaLocation="xsd-includes/http.UserAuthenticationModule.xsd"/>
      <xsd:element name="authenticateUser">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="systemAuthenticationRequest" nillable="true"
type="bons1:AuthenticationRequest"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="authenticateUserResponse">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="systemAuthenticationResponse" nillable="true"
type="bons1:AuthenticationResponse"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:schema>
  </wsdl:types>
    <wsdl:message name="authenticateUserRequestMsg">
    <wsdl:part element="tns:authenticateUser" name="authenticateUserParameters"/>
  </wsdl:message>
    <wsdl:message name="authenticateUserResponseMsg">
    <wsdl:part element="tns:authenticateUserResponse" name="authenticateUserResult"/>
  </wsdl:message>
    <wsdl:portType name="UserAuthenticationService">
    <wsdl:operation name="authenticateUser">
      <wsdl:input message="tns:authenticateUserRequestMsg" name="authenticateUserRequest"/>
      <wsdl:output message="tns:authenticateUserResponseMsg"
name="authenticateUserResponse"/>
    </wsdl:operation>
  </wsdl:portType>
</wsdl:definitions>
```

Figure B-7 UserAuthenticationService.wsdl

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
name="UserAuthenticationServiceExport1_UserAuthenticationServiceHttp_Service"
targetNamespace="http://UserAuthenticationModule/UserAuthenticationService/Binding"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:Port_0="http://UserAuthenticationModule/UserAuthenticationService"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:this="http://UserAuthenticationModule/UserAuthenticationService/Binding"
xmlns="http://schemas.xmlsoap.org/wsdl/">
  <wsdl:import namespace="http://UserAuthenticationModule/UserAuthenticationService"
location="UserAuthenticationService.wsdl"/>
  <wsdl:binding name="UserAuthenticationServiceExport1_UserAuthenticationServiceHttpBinding"
type="Port_0:UserAuthenticationService">
    <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="authenticateUser">
      <soap:operation/>
      <wsdl:input name="authenticateUserRequest">
        <soap:body use="literal"/>
      </wsdl:input>
      <wsdl:output name="authenticateUserResponse">
        <soap:body use="literal"/>
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service
name="UserAuthenticationServiceExport1_UserAuthenticationServiceHttpService">
    <wsdl:port name="UserAuthenticationServiceExport1_UserAuthenticationServiceHttpPort"
binding="this:UserAuthenticationServiceExport1_UserAuthenticationServiceHttpBinding">
      <soap:address
location="http://localhost:9080/UserAuthenticationModuleWeb/sca/UserAuthenticationServiceExp
ort1"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

Figure B-8 UserAuthenticationServiceExport1_UserAuthenticationServiceHttp_Service.wsdl

InBoundAdapter.java

Example: B-1 ITSOAutoInBoundAdapter class

```
ackage com.ibm.itso.autocompany;

import java.io.ByteArrayOutputStream;
import org.json.JSONException;
import org.json.JSONObject;
import org.json.XML;
```

```
import com.ibm.websphere.sca.ServiceManager;
import commonj.sdo.DataObject;
import com.ibm.websphere.bo.BOFactory;
import module.mediation.auto.itso.itso.auto.interface_.ITSOAutoInterface;
import com.ibm.websphere.bo.BOXMLSerializer;

public class ITSOAutoInBoundAdapter implements java.io.Serializable {

   public String logIn(String customerId, String passwd) {
      JSONObject jo = null;
      try {
         System.out
               .println("ITSOAuto::logIn ************ creating service manager");
         ServiceManager serviceManager = new ServiceManager();
         BOFactory bof = (BOFactory) serviceManager
               .locateService("com/ibm/websphere/bo/BOFactory");
         System.out
               .println("ITSOAuto::logIn ************ BOFactory created");
         DataObject authenticationRequest = bof.create(
               "http://UserAuthenticationModule", "AuthenticationRequest");
         System.out
               .println("ITSOAuto::logIn ************ Authentication Request object created");
         authenticationRequest.setString("customerId", customerId);
         authenticationRequest.setString("password", passwd);
         System.out
               .println("ITSOAuto::logIn ************ locating ITSO Auto service");
         ITSOAutoInterface autoService = (ITSOAutoInterface) serviceManager
               .locateService("ITSOAutoInterfacePartner");
         System.out
               .println("ITSOAuto::logIn ************ logging in to ITSO Auto service");
         DataObject inventoryResponse = (DataObject) autoService
               .logIn(authenticationRequest);
         if (null == inventoryResponse) {
            System.out
                  .println("ITSOAuto::logIn ************ inventoryResponse is null");
            throw new Exception("Inventory Response is null!");
         }
         ByteArrayOutputStream bos = new ByteArrayOutputStream();
         try {
            BOXMLSerializer xmlSerializerService = (BOXMLSerializer) new ServiceManager()
                  .locateService("com/ibm/websphere/bo/BOXMLSerializer");
            xmlSerializerService.writeDataObject(inventoryResponse, null,
                  "InventoryResponse", System.out);
            xmlSerializerService.writeDataObject(inventoryResponse, null,
                  "InventoryResponse", bos);
         } catch (Exception ex) {
            ex.printStackTrace();
         }
         jo = XML.toJSONObject(bos.toString());
```

```
         System.out
               .println("ITSOAuto::logIn ************ ITSOAuto:: JSON String "
                     + jo.toString());
      } catch (Exception e) {
         System.out.println(e);
         try {
            jo = new JSONObject(new String("Error: \"Error occurred: "
                  + e.getMessage() + "\""));
         } catch (JSONException ex) {
            System.out.println(ex.toString());
         }
      }
      return jo.toString();
   }
}
```

Chapter 10, “Custom application integration using JMS” on page 333

Example: B-2 ITSO-AutoFindPartsPrice.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
   pageEncoding="ISO-8859-1"%>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<meta name="GENERATOR" content="IBM Software Development Platform">
<title>ITSO-AutoFindPartsPrice.jsp</title>
</head>
<body>
<div style="text-align: center">
<h1><IMG border="0" src="redbooklogo.gif" width="288" height="66" alt=""><BR>
ITSO-Auto Purchase Order Price Search</h1>
<h2>Enter a part and quantity</h2>
<form method="get" action="ITSO-AutoFindPartsPrice.jsp">
<table align="center">
   <tr>
      <td>Spare Part ID:</td>
      <td><input type="text" name="productid" /></td>
   </tr>
   <tr>
      <td>Quantity:</td>
      <td><select name="quantity">
         <option>1</option>
         <option>2</option>
         <option>3</option>
```

```
            <option>4</option>
            <option>5</option>
        </select></td>
    </tr>
</table>
<br />
<input name="findPrice" type="submit" value="findPrice" /></form>
<%if (request.getParameter("productid") != null) {
            //The Initial Context Factory
            String icf = "com.ibm.websphere.naming.WsnInitialContextFactory";
            //the Provider URL (This port number may be different depending on your install)
            // The RMI Connector is available at port 2809
            String url = "iiop://localhost:2809/";
            //The Queue Connection Factory used to connect to the bus
            String sampleQCF = "jms/purchaseOrderQCF";
            //The Queue used to send requests to the mediation module
            String sampleSendQueue = "jms/purchaseOrderInputQueue";
            //The Queue used to receive responses from the mediation module
            String sampleReceiveQueue = "jms/purchaseOrderOutputQueue";
            //The XML representation of an Order Business Object required by the
            //order operation on the PartsPrice Interface
            String message = "<?xml version=\"1.0\" encoding=\"UTF-8\"?>";
            message += "<tns:requestParts xmlns:p=\"http://PurchaseOrderMediation\" ";
            message += "xmlns:tns=\"http://PurchaseOrderMediation/RequestForParts\">";
            message += "<partsRequest><partName>";
            message += request.getParameter("productid");
            message += "</partName><number>";
            message += request.getParameter("quantity");
            message += "</number></partsRequest></tns:requestParts>";
            System.out.println("message is " + message);
            try {
               //Create the Initial Context
               java.util.Hashtable env = new java.util.Hashtable();
               env.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY, icf);
               env.put(javax.naming.Context.PROVIDER_URL, url);
               javax.naming.Context ctx = new javax.naming.directory.InitialDirContext(
                     env);
               //Lookup the ConnectionFactory
               javax.jms.ConnectionFactory factory = (javax.jms.ConnectionFactory) ctx
                     .lookup(sampleQCF);
               //Create a Connection
               javax.jms.Connection connection = factory
                     .createConnection();
               //Start the Connection
               connection.start();
               //Create a Session
               javax.jms.Session jmsSession = connection.createSession(
                     false, javax.jms.Session.AUTO_ACKNOWLEDGE);
               //Lookup the send Destination
```

```
            javax.jms.Destination sendQueue = (javax.jms.Destination) ctx
                  .lookup(sampleSendQueue);
            //Create a MessageProducer
            javax.jms.MessageProducer producer = jmsSession
                  .createProducer(sendQueue);
            //Create the TextMessage that will hold our Order as text
            javax.jms.TextMessage sendMessage = jmsSession
                  .createTextMessage();
            //Set the content of the message to be the XML defined Order
            sendMessage.setText(message);
            //Set the operation to call on the OrderService interface to be placeOrder
            sendMessage.setStringProperty("TargetFunctionName",
                  "requestParts");
            //Send the message
            producer.send(sendMessage);
            //Lookup the receive Destination
            javax.jms.Destination receiveQueue = (javax.jms.Destination) ctx
                  .lookup(sampleReceiveQueue);
            //Create a MessageConsumer
            javax.jms.MessageConsumer consumer = jmsSession
                  .createConsumer(receiveQueue);
            //Wait 15 seconds to receive the response
            javax.jms.TextMessage receiveMessage = (javax.jms.TextMessage) consumer
                  .receive(8000);
            //If we receive a response print the contents of the message to the screen
            String confirmation = "Price Search failed.";
            if (receiveMessage != null) {
               //Print the contents of the message.
               confirmation = "Price Search Successful<br/>Price per Unit: $"
                     + receiveMessage.getText();
            }
            out.println("<p>" + confirmation + "</p>");
            //Close the Connection
            connection.close();
         } catch (Exception e) {
            out.println(e);
         }
      }

   %></div>
</body>
</html>
```

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this book.

IBM Redbooks

For information about ordering these publications, see “How to get IBM Redbooks” on page 514. Note that some of the documents referenced here may be available in softcopy only.

- *WebSphere Adapter Development*, SG24-6387
- *Connect WebSphere Service-Oriented Middleware to SAP*, SG24-7220
- *Getting Started with WebSphere Enterprise Service Bus V6*, SG24-7212
- *Patterns: Implementing an SOA using an Enterprise Service Bus*, SG24-6346
- *Patterns: Integrating Enterprise Service Buses in a Service-Oriented Architecture*, SG24-6773
- *A Simple Example: Using the WebSphere Adapter for Flat File*, REDP-4235
- *WebSphere Business Integration Adapters: An Adapter Development and WebSphere Business Integration Solution*, SG24-6345

Online resources

These Web sites are also relevant as further information sources:

- Specifications: Service Component Architecture (SCA) and Service Data Objects (SDO)

  http://www.ibm.com/developerworks/webservices/library/specification/ws-scasdosumm/
- Open Service Oriented Architecture collaboration

  http://www.osoa.org/display/Main/Home
- SOA programming model for implementing Web services, Part 4: An introduction to the IBM Enterprise Service Bus

  http://www.ibm.com/developerworks/webservices/library/ws-soa-progmodel4/index.html

- IBM WebSphere Developer Technical Journal: Building an Enterprise Service Bus using WebSphere ESB
  - Part 1: An introduction to using WebSphere ESB, or WebSphere ESB versus SIBus
  - Part 2: A JMS messaging example
  - Part 3: Adding Web services and promoting properties
  - Part 4: Connecting WebSphere ESB and WebSphere MQ
  - Part 5: Switching protocols

  http://www.ibm.com/developerworks/websphere/techjournal/archive.html
- WebSphere Enterprise Service Bus Product Page

  http://www-306.ibm.com/software/integration/wsesb/
- IBM developerWorks: Exploring the Enterprise Service Bus, Part 1: Discover how an ESB can help you meet the requirements for your SOA solution

  http://www-128.ibm.com/developerworks/architecture/library/ar-esbpat1/

How to get IBM Redbooks

You can search for, view, or download Redbooks, Redpapers, Hints and Tips, draft publications and Additional materials, as well as order hardcopy Redbooks or CD-ROMs, at this Web site:

ibm.com/redbooks

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services

Index

Symbols

Numerics

A

B

C

D

I

J

X

Z

IBM

Redbooks

Connecting Enterprise Applications to WebSphere Enterprise Service

IBM®

Connecting Enterprise Applications to WebSphere Enterprise Service Bus

Redbooks®

Patterns with worked examples for many integration scenarios

Code-free connection to CICS using WebSphere MQ

Integrate Web 2.0 into the service bus

Service Oriented Architecture (SOA) promises a great leap forward in the re-use of applications by simplifying application composition. The technology that simplifies application composition is Service Component Architecture (SCA). SCA is an open component architecture for wiring services together to build composite applications. WebSphere Enterprise Service Bus provides the on-ramp and off-ramp to incorporate many different applications and services into an SOA solution.

In this IBM Redbooks publication we introduce SCA and how it has evolved from earlier application integration architectures. We explain how WebSphere ESB connects applications and components to the service bus by using adapters and other types of SCA imports and exports.

Our main focus is helping you sort through the many choices that need to be made when deciding how to connect applications together to meet the requirements of a business scenario. We propose six different solution patterns, each with alternative implementations to choose from, to take on most integration scenarios. We also provide seven worked examples of some of the alternatives, which are fully described in the text, and are also available as working samples from the ITSO Redbooks Web site.

INTERNATIONAL TECHNICAL SUPPORT ORGANIZATION

BUILDING TECHNICAL INFORMATION BASED ON PRACTICAL EXPERIENCE

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

For more information:
ibm.com/redbooks

SG24-7406-00 ISBN 0738488828