WebSphere. software

IBM

Patterns: Implementing Self-Service in an SOA Environment

Integrate Web applications with the enterprise tier

Explore Web services, J2EE Connectors, and JMS solutions

Use SOA and ESB technology

Carla Sadtler
Anup Aggarwal
Diego Cotignola
Sandy Grewal
Peter Hood
Shashi Shrimali
Fernando Teixeira

Redbooks

ibm.com/redbooks

International Technical Support Organization

Patterns: Implementing Self-Service in an SOA Environment

January 2006

SG24-6680-01

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

Second Edition (January 2006)

This edition applies to WebSphere Application Server V6.

Contents

Notices

Trademarks

Preface

The Patterns for e-business are a group of proven, reusable assets that can be used to increase the speed of developing and deploying Web applications. This IBM® Redbook focuses on the use of service-oriented architecture and the enterprise service bus to build solutions that help organizations achieve rapid, flexible integration of IT systems.

It includes the Self-Service::Directly Integrated Single Channel pattern for implementing point-to-point connections with back-end applications, the Self-Service::Router pattern for implementing intelligent routing among multiple back-end applications, and the Self-Service::Decomposition pattern for decomposing a request into multiple requests and recomposing the results into a single response.

This IBM Redbook teaches you by example how to design and build sample solutions using WebSphere® Application Server V6 with Web services, J2EE™ Connectors and IBM CICS®, and JMS using the WebSphere Application Server default messaging provider. WebSphere Application Server service integration technology is used to implement enterprise service bus functionality.

The team that wrote this redbook

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

Carla Sadtler is a certified IT Specialist at the ITSO, Raleigh Center. She writes extensively about the WebSphere and Patterns for e-business areas. Before joining the ITSO in 1985, Carla worked in the Raleigh branch office as a Program Support Representative. She holds a degree in mathematics from the University of North Carolina at Greensboro.

Anup Aggarwal is a Staff Software Engineer with the WebSphere Application Server System Verification Test team in Research Triangle Park, Durham, North Carolina. He has five years of experience in designing, developing and simulating customer-like end-to-end J2EE applications to test WebSphere Application Server. His areas of expertise include WebSphere Application Server, J2EE technologies, tooling such as WebSphere Studio and Rational® Developer, and databases such as DB2®, Oracle and Sybase. He is currently the Transaction Management Strategist for SVT and has responsibility for coordinating test efforts on JCA, JTA and adapters across the WebSphere

Quality Center of Competence Organization. He holds a Bachelor of Science degree in Electronics and Communication Engineering from the CR State University, India, and a Master of Science degree in Computer Engineering from Wright State University of Dayton, Ohio.

Diego Cotignola Diego Cotignola is an IT Architect in Uruguay. He joined IBM in 1999, focusing on e-business application development, in particular Java™ and J2EE, WebSphere Application Server, and WebSphere MQ. He became a specialist in WebSphere and in its related technologies. Currently, he is also working as an IT Architect in EAI solutions using WebSphere Business Integration software.

Sandy Grewal is a Staff Software Developer at the IBM Toronto Lab. He has extensive experience with J2EE and Web technologies. His area of expertise includes J2EE object-oriented design and architecture, and data modeling. His interests include software development methodologies, application design and architecture, data modelling and design patterns. He has a masters degree in economics from the University of Economics in Wroclaw, Poland.

Peter Hood is a IT Specialist in Australia. He has seven years of experience in IBM A/NZ Global Services working as an IT Specialist. He has experience in a number of technologies, ranging from IBM WebSphere and Microsoft® .NET to general Internet based technology. He has utilized a number of different architectures ranging from high-performance n-tier web applications to more traditional client-server models. As a result of his experience, Peter has been involved in consulting and assisting a number of troubled projects in both short and long term secondments. He works in teams ranging from several people of similar skills to large multi-disciplinary teams with a range of backgrounds and skill-sets. Peter has been involved in the design of many Internet-based applications and been lead developer in many of these. He holds a bachelor degree in computer science from Melbourne University and also a masters degree in Internet and Web computing from the Royal Institute of Technology.

Shashi Shrimali is an Associate IT Architect in the AMS Organization, IBM Global Services India. He is a member of the Centre of Competence, Architecture & AMS Research within AMS. He is also a core team member for IGS India, SOA and Web Services community. He has over six years of IT experience mainly in the telecommunications sector. His area of expertise includes J2EE technologies, OOAD & Design Patterns, Web services, application architecture, and Service Oriented Architecture. He has conducted training in the areas of SOA, Web services and enterprise service bus technology.

Fernando Teixeira is a Certified IT Architect in the Application Services Organization, IBM Global Services, US. He has 16 years of IT experience, including lead technical roles in the telecommunications and financial industries.

His areas of expertise include application architecture, object-oriented design, J2EE technologies, and Web application design. He holds a Bachelor of Science degree in Computer Science and Math from the University of Wisconsin, and an Master of Science in Computer Science from the University of Pennsylvania.

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

Martin Keen
International Technical Support Organization, Raleigh Center

Jonathan Adams
Paul Verschueren
Patterns for e-business leadership and architecture, IBM UK

Become a published author

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

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

Find out more about the residency program, browse the residency index, and apply online at:

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

We want our Redbooks™ to be as helpful as possible. Send us your comments about this or other Redbooks in one of the following ways:

- Use the online **Contact us** review redbook form found at:

  ibm.com/redbooks

- Send your comments in an email to:

  redbook@us.ibm.com

- Mail your comments to:

IBM Corporation, International Technical Support Organization
Dept. HZ8 Building 662
P.O. Box 12195
Research Triangle Park, NC 27709-2195

Summary of changes

This section describes the technical changes made in this edition of the book and in previous editions. This edition may also include minor corrections and editorial changes that are not identified.

Summary of Changes
for SG24-6680-01
for Patterns: Implementing Self-Service in an SOA Environment
as created or updated on February 20, 2006.

January 2006, Second Edition

This revision reflects the addition, deletion, or modification of new and changed information described below.

New information

- The Self-Service Router and Decomposition application patterns have been added to the text and to the ITSOMart sample. The new patterns are illustrated using Web services.

1 Patterns for e-business

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

To improve the process over time, we need to capture and reuse the experience of the IT architects in such a way that future engagements can be made simpler and faster. We do this by capturing knowledge gained from each engagement and using it to build a repository of assets. IT architects can then build future solutions based on these proven assets. Reusing proven assets saves time, money, and effort and helps ensure delivery of a solid, properly architected solution.

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

1.1 The Patterns for e-business layered asset model

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

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

Figure 1-1 on page 3 shows these assets and their relationships to each other.

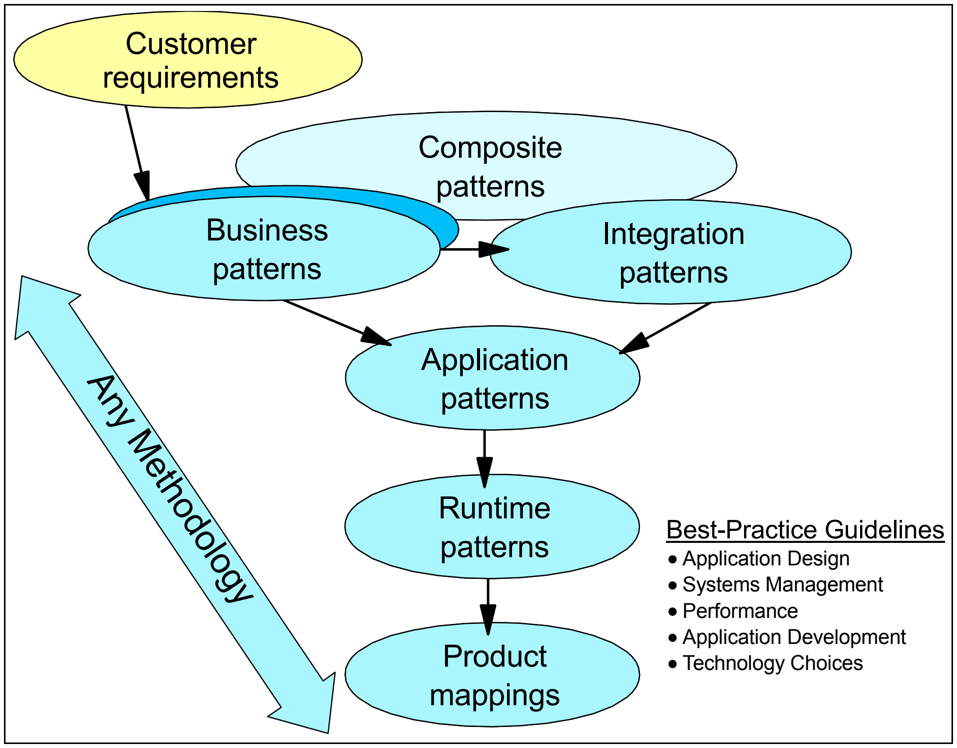


Figure 1-1 The Patterns for e-business layered asset model

Patterns for e-business Web site

The layers of patterns, along with their associated links and guidelines, allow the architect to start with a problem and a vision for the solution and then find a pattern that fits that vision. Then, by drilling down using the patterns process, the architect can further define the additional functional pieces that the application need to succeed. Finally, the architect can build the application using coding techniques that are outlined in the associated guidelines.

The Patterns Web site provides an easy way of navigating through the layered Patterns assets to determine the most appropriate assets for a particular engagement.

For easy reference, see the Patterns for e-business Web site:

`http://www.ibm.com/developerWorks/patterns/`

1.2 How to use the Patterns for e-business

As described in the previous section, the Patterns for e-business have a layered structure where each layer builds detail on the last. At the highest layer are Business patterns. These describe the entities involved in the e-business solution.

Composite patterns appear in the hierarchy shown in Figure 1-1 on page 3 above the Business patterns. However, Composite patterns are made up of a number of individual Business patterns and at least one Integration pattern. This section discusses how to use the layered structure of Patterns for e-business assets.

1.2.1 Selecting a Business, Integration, Composite pattern, or a Custom design

When faced with the challenge of designing a solution for a business problem, the first step is to get a high-level view of the goals that you want to achieve. You need to describe a proposed business scenario and match each element to an appropriate IBM Pattern for e-business. You might find, for example, that the total solution requires multiple Business and Integration patterns or that it fits into a Composite pattern or Custom design.

For example, suppose an insurance company wants to reduce the amount of time and money spent on call centers that handle customer inquiries. By allowing customers to view their policy information and request changes online, the company can cut back significantly on the resources that are spent handling this type of request by phone. The objective allows policy holders to view policy information that is stored in existing databases.

The Self-Service business pattern fits this scenario perfectly. You can use it in situations where users need direct access to business applications and data. The following sections discuss the available Business patterns.

Business patterns

A *Business pattern* describes the relationship between the users, the business organizations or applications, and the data to be accessed.

Four primary Business patterns are explained in Table 1-1.

Table 1-1 The four primary Business patterns

Business Patterns	Description	Examples
Self-Service (user-to-business)	Applications where users interact with a business with the Internet or intranet.	Simple Web applications
Information Aggregation (user-to-data)	Applications where users can extract useful information from large volumes of data, text, images, and so forth.	Business intelligence, knowledge management, and Web crawlers
Collaboration (user-to-user)	Applications where the Internet supports collaborative work between users	Community, chat, videoconferencing, e-mail, and so forth
Extended Enterprise (business-to-business)	Applications that link two or more business processes across separate enterprises.	EDI, supply chain management, and so forth

It would be very convenient if all problems fit nicely into these four slots, but the reality is that things can often be more complicated. The patterns assume that most problems, when broken down into their basic components, will fit more than one of these patterns. When a problem requires multiple Business patterns, you can use Integration patterns.

Integration patterns

Integration patterns allow you to tie together multiple Business patterns to solve a business problem. Table 1-2 describes the Integration patterns.

Table 1-2 Integration patterns

Integration Patterns	Description	Examples
Access Integration	Integration of a number of services through a common entry point	Portals
Application Integration	Integration of multiple applications and data sources without the user directly invoking them	Message brokers, workflow managers, data propagators, and data federation engines

The Access Integration pattern maps to User Integration. The Application Integration pattern is divided into two essentially different approaches:

- *Process Integration* is the integration of the functional flow of processing between the applications.
- *Data Integration* is the integration of the information that is used by applications.

You can combine the Business and Integration patterns to implement installation-specific business solutions called a *Custom design*.

Custom design

Figure 1-2 illustrates the use of a Custom design to address a business problem.

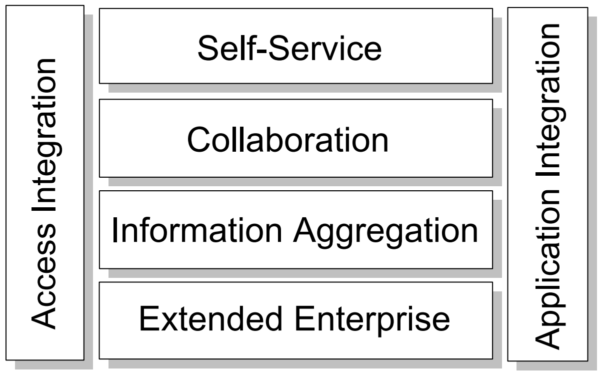


Figure 1-2 Patterns representing a Custom design

If you do not use any of the Business or Integration patterns in a Custom design, you can show the unused patterns as lighter blocks than those patterns that you do use. For example, Figure 1-3 shows a Custom design that does not have a Collaboration or an Extended Enterprise business pattern for a business problem.

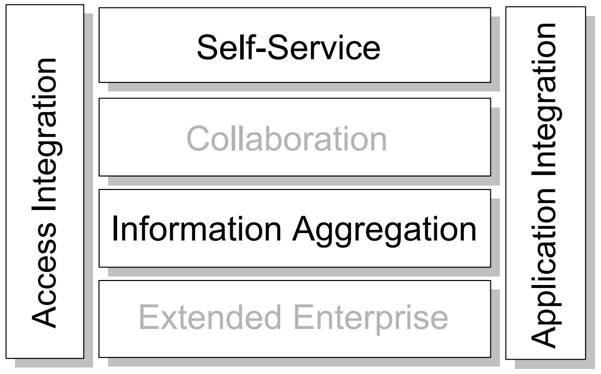


Figure 1-3   Custom design showing unused patterns

If a Custom design recurs many times across domains that have similar business problems, then it can also be a *Composite pattern*. For example, the Custom design in Figure 1-3 can also describe a Sell-Side Hub Composite pattern.

Composite patterns

Several common uses of Business and Integration patterns have been identified and formalized into Composite patterns. Table 1-3 on page 8 shows the identified Composite patterns.

Table 1-3 Composite patterns

Composite Patterns	Description	Examples
Electronic Commerce	User-to-online-buying	• http://www.macys.com • http://www.amazon.com
Portal	Typically designed to aggregate multiple information sources and applications to provide uniform, seamless, and personalized access for its users.	• Enterprise intranet portal providing self-service functions such as payroll, benefits, and travel expenses. • Collaboration providers who provide services such as e-mail or instant messaging.
Account Access	Provide customers with around-the-clock account access to their account information.	• Online brokerage trading applications • Telephone company account manager functions • Bank, credit card and insurance company online applications
Trading Exchange	Allows buyers and sellers to trade goods and services on a public site.	• Buyer's side: interaction between buyer's procurement system and commerce functions of e-Marketplace. • Seller's side:- interaction between the procurement functions of the e-Marketplace and its suppliers.
Sell-Side Hub (supplier)	The seller owns the e-Marketplace and uses it as a vehicle to sell goods and services on the Web.	http://www.carmax.com (car purchase)
Buy-Side Hub (purchaser)	The buyer of the goods owns the e-Marketplace and uses it as a vehicle to leverage the buying or procurement budget in soliciting the best deals for goods and services from prospective sellers across the Web.	http://www.wwre.org (WorldWide Retail Exchange)

The makeup of these patterns is variable in that there will be basic patterns present for each type. However, you can extend the Composite to meet additional criteria. For more information about Composite patterns, refer to *Patterns for e-business: A Strategy for Reuse* by Jonathan Adams, Srinivas Koushik, Guru Vasudeva, and George Galambos (ISBN 1-931182-02-7).

1.2.2 Selecting Application patterns

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

Application patterns break down the application into the most basic conceptual components that identify the goal of the application. In our example, the application falls into the Self-Service business pattern, and the goal is to build a simple application that allows users to access back-end information. Figure 1-4 shows the Self-Service::Directly Integrated Single Channel application pattern, which fulfills this requirement.

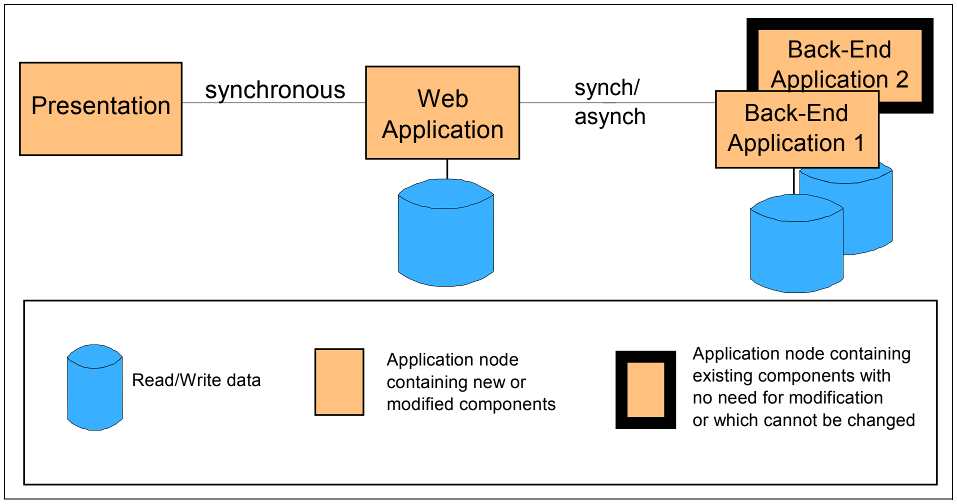

Presentation
synchronous
Web Application
synch/ asynch
Back-End Application 2
Back-End Application 1
Read/Write data
Application node containing new or modified components
Application node containing existing components with no need for modification or which cannot be changed

Figure 1-4 Self-Service::Directly Integrated Single Channel pattern

This Application pattern consists of a presentation tier that handles the request and response to the user. The application tier represents the component that handles access to the back-end applications and data. The multiple application boxes on the right represent the back-end applications that contain the business data. The type of communication is specified as synchronous (one request/one response, then next request/response) or asynchronous (multiple requests and responses intermixed).

Suppose that the situation is a little more complicated. Suppose that the automobile policies and the homeowner policies are kept in two separate and dissimilar databases. The user request actually needs data from multiple, disparate back-end systems. In this case, there is a need to break the request down into multiple requests (decompose the request) to be sent to the two different back-end databases, then to gather the information that is sent back from the requests, and put this information into the form of a response (recompose). In this case, the Self-Service::Decomposition application pattern (as shown in Figure 1-5) would be more appropriate.

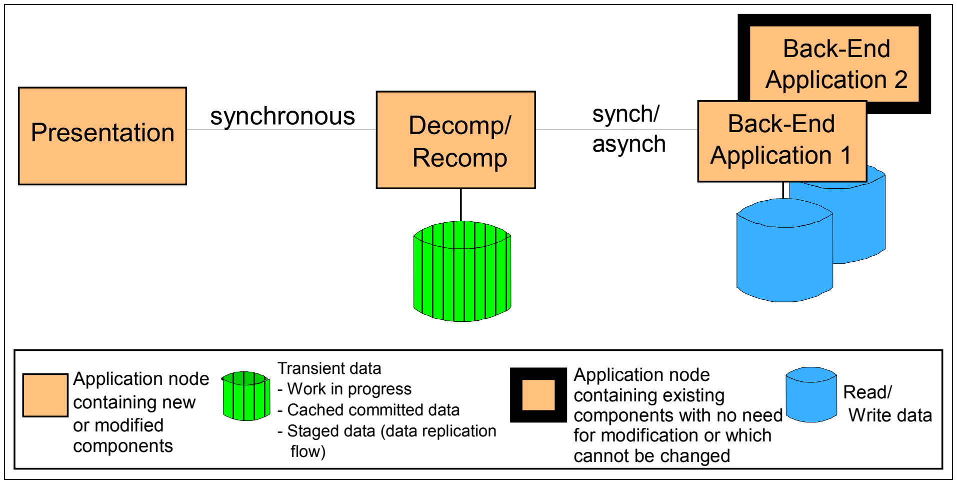

Presentation
synchronous
Decomp/ Recomp
synch/ asynch
Back-End Application 2
Back-End Application 1
Application node containing new or modified components
Transient data
- Work in progress
- Cached committed data
- Staged data (data replication flow)
Application node containing existing components with no need for modification or which cannot be changed
Read/ Write data

Figure 1-5 Self-Service::Decomposition pattern

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

1.2.3 Review Runtime patterns

You can refine the Application pattern further with more explicit functions. Each function is associated with a runtime node. In reality, these functions, or nodes, can exist on separate physical machines or can coexist on the same machine. In the Runtime pattern the physical location of the function is not relevant. The focus is on the logical nodes that are required and their placement in the overall network structure.

As an example, assume that our client has determined that their solution fits into the Self-Service business pattern and that the Directly Integrated Single Channel pattern is the most descriptive of the situation. The next step is to determine the Runtime pattern that is most appropriate for the situation.

They know that they will have users on the Internet what are accessing their business data, Therefore, they require a measure of security. You can implement security at various layers of the application, but the first line of defense is almost always one or more firewalls that define who and what can cross the physical network boundaries into the company network.

The client also needs to determine the functional nodes that are required to implement the application and security measures. Figure 1-6 on page 12 shows the Runtime pattern that is one option.

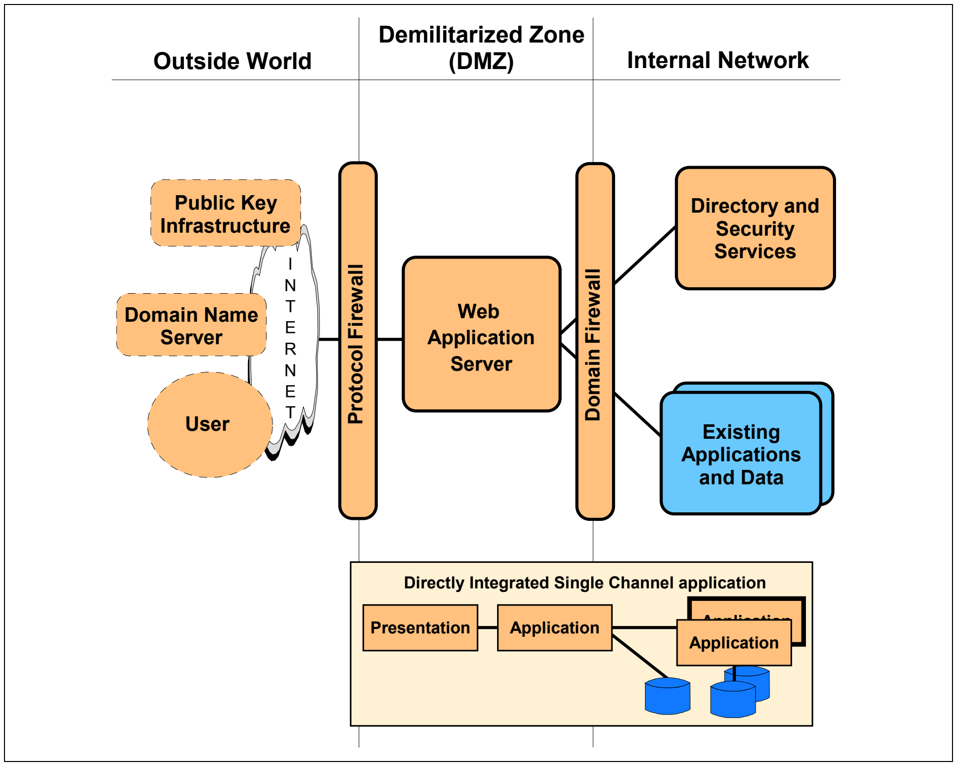


Figure 1-6 Directly Integrated Single Channel application pattern::Runtime pattern

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

Application security is handled by the Web application server through the use of a common central directory and security services node.

A characteristic that makes this Runtime pattern different from others is the placement of the Web application server between the two firewalls. Figure 1-7 on page 13 shows variation on this pattern. It splits the Web application server into two functional nodes by separating the HTTP server function from the application server. The HTTP server (Web server redirector) provides static Web pages and redirects other requests to the application server. This pattern moves the application server function behind the second firewall, adding further security.

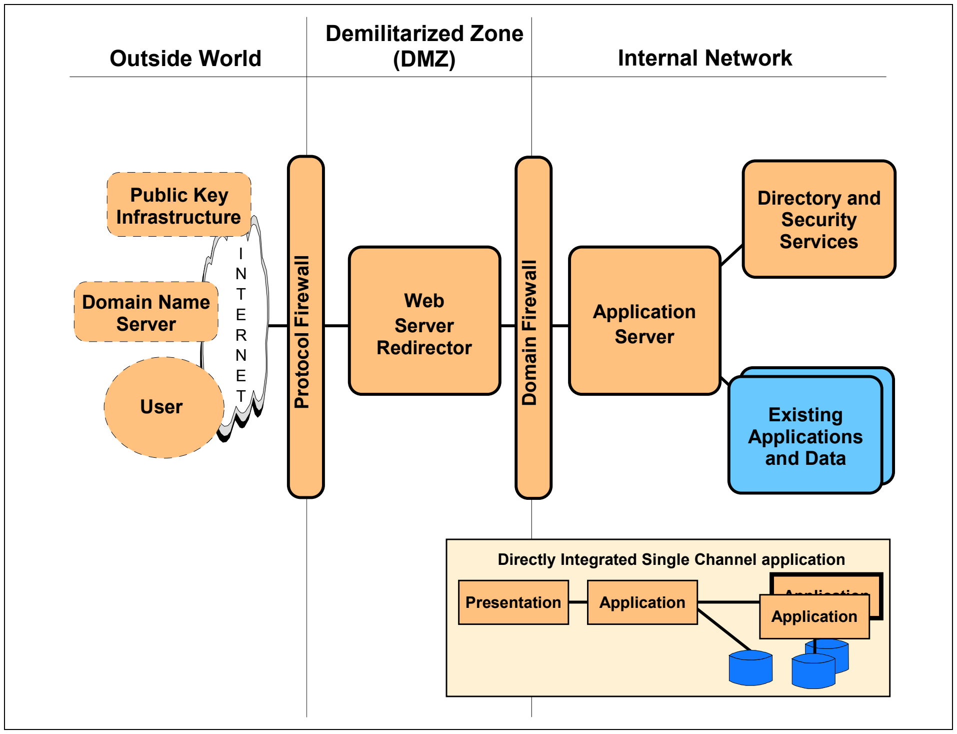


Figure 1-7 Directly Integrated Single Channel application pattern::Runtime pattern

These are just two examples of the possible Runtime patterns that are available. Each Application pattern will have one or more Runtime patterns defined. You can modify these Runtime patterns to suit the client's needs. For example, the client might want to add a load-balancing function and multiple application servers.

1.2.4 Reviewing Product mappings

The last step in defining the network structure for the application is to correlate real products with one or more runtime nodes. The Patterns Web site shows each Runtime pattern with products that have been tested in that capacity. The Product mappings are oriented toward a particular platform. However, it is more likely that the client will have a variety of platforms involved in the network. In this case, you can mix and match product mappings.

For example, you could implement the runtime variation in Figure 1-7 on page 13 using the product set that is depicted in Figure 1-8.

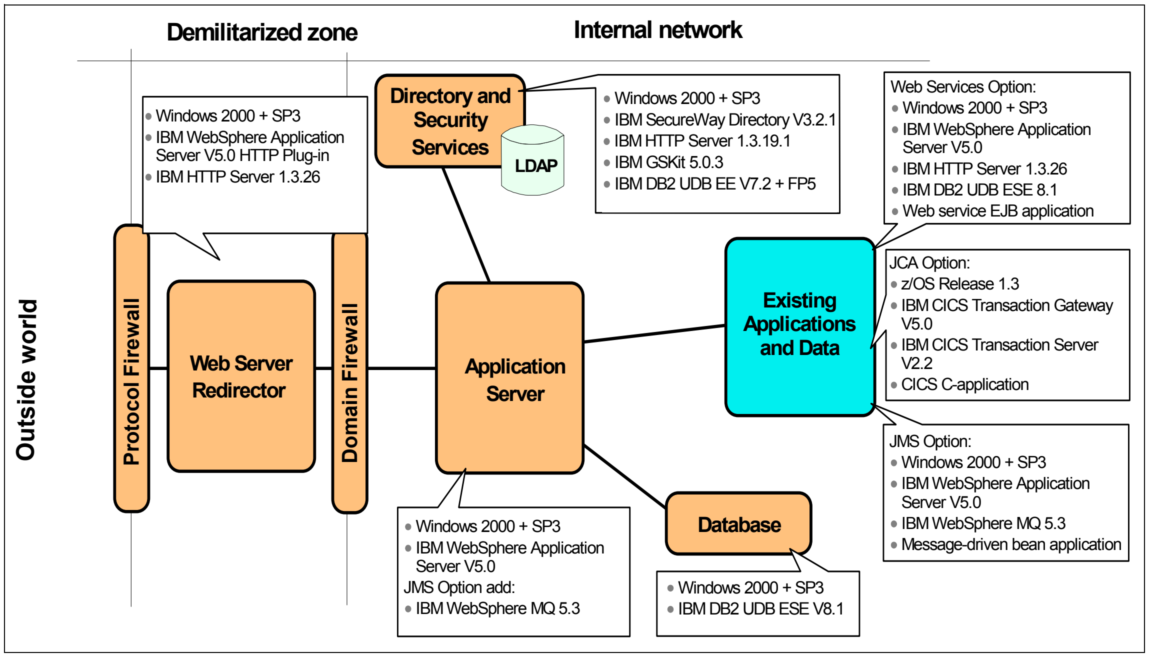


Figure 1-8 Directly Integrated Single Channel application pattern: Windows® 2000 Product mapping

1.2.5 Reviewing guidelines and related links

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

- Design guidelines provide tips and techniques for designing the applications.
- Development guidelines take you through the process of building the application, from the requirements phase all the way through the testing and rollout phases.
- System management guidelines address the day-to-day operational concerns, including security, backup and recovery, application management, and so forth.
- Performance guidelines give information about how to improve the application and system performance.

1.3 Summary

The IBM Patterns for e-business are a collected set of proven architectures. You can use this repository of assets to facilitate the development of Web-based applications. Patterns for e-business help you understand and analyze complex business problems and break them down into smaller, more manageable functions that you can then implement.

2

Self-Service business pattern

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

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

2.1 Self-service applications

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

The following situations are examples of self-service applications:

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

2.2 Self-Service application patterns

As you can see in Figure 2-1 on page 19, the Self-Service business pattern covers a wide range of uses. Applications of this pattern can range from the very simple function of allowing users to view data built explicitly for one purpose, to taking requests from users, decomposing them into multiple requests to be sent to multiple, disparate data sources, personalizing the information, and recomposing it into a response for the user.

For this reason, there are currently seven defined Application patterns that fit this range of functions. We summarize these for you here. More detailed information can be found in *Patterns for e-business: A Strategy for Reuse*, by Jonathan Adams, Srinivas Koushik, Guru Vasudeva, and George Galambos.

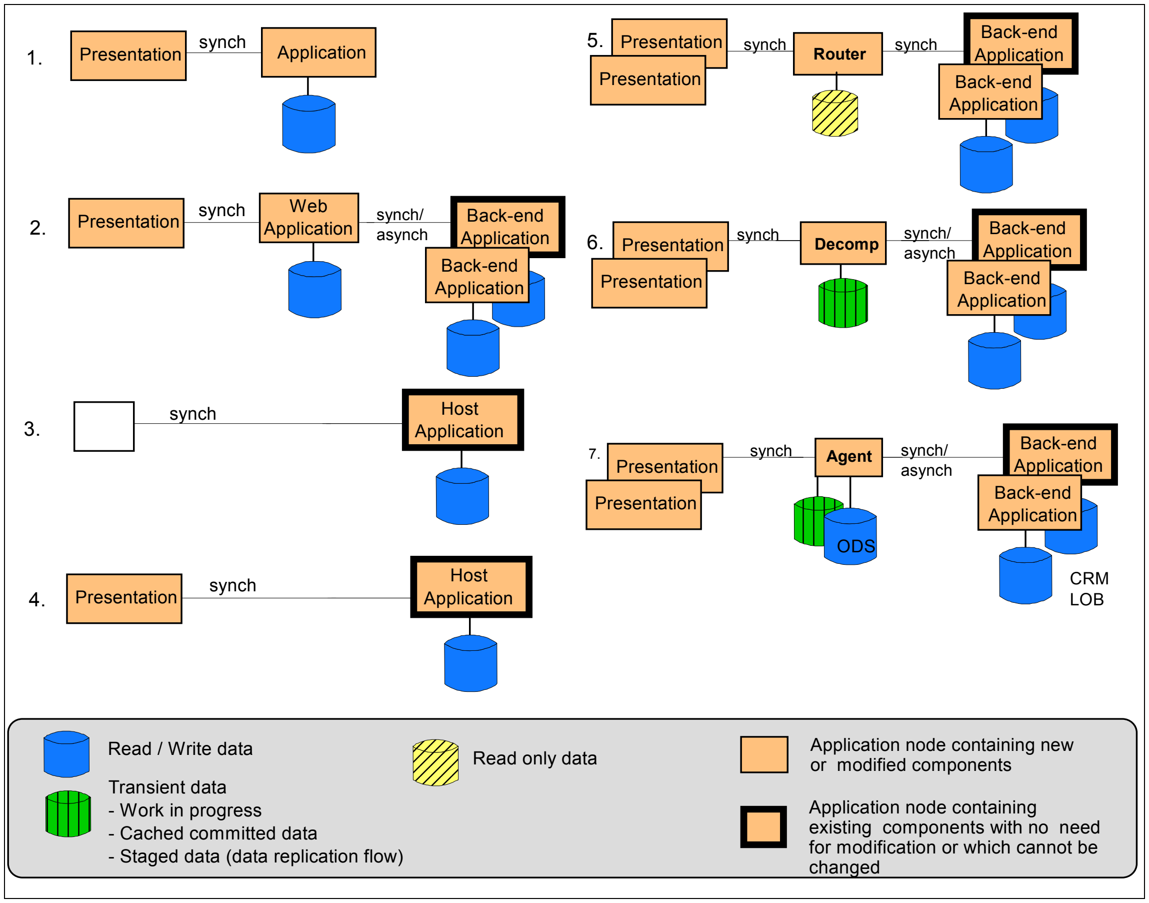


Figure 2-1 Self-Service application patterns

The Self-Service application patterns are defined as follows:

1. Stand-alone Single Channel application pattern

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

2. Directly Integrated Single Channel application pattern

   This pattern provides point-to-point connectivity between the user and the existing back-end applications. As with the Stand-alone Single Channel application pattern, it assumes one delivery channel, and the user interface is handled by the presentation tier. The business logic can reside in the Web application tier and in the back-end application.

   The Web application tier has access to local data that exists primarily as a result of this application, for example, customer profile information or cached data. It is also responsible for accessing one or more back-end applications. The back-end applications contain business logic and are responsible for accessing the existing back-end data. The communication between the presentation tier and Web application tier is synchronous. The communication between the Web application tier and the back-end can be either synchronous or asynchronous, depending on the characteristics and capabilities of the back-end application.

3. As-is Host application pattern

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

4. Customized Presentation to Host application pattern:

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

5. Router application pattern

   The Router application pattern provides intelligent routing from multiple channels to multiple back-end applications using a hub-and-spoke architecture. The interaction between the user and the back-end application is a one-to-one relation, meaning the user interacts with applications one at a time. The router maintains the connections to the back-end applications and pools connections when appropriate, but there is no true integration of the applications themselves. The router can use a read-only database, most probably to look up routing information. The primary business logic still resides in the back-end application tier.

This pattern assumes that the users are accessing the applications from a variety of client types such as Web browsers, voice response units (VRUs), or kiosks. The Router application pattern provides a common interface for accessing multiple back-end applications and acts as an intermediary between them and the delivery channels. In doing this, the Router application pattern can use elements of the Integration patterns.

6. Decomposition application pattern

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

   From a service-oriented architecture (SOA) perspective, the decomposition tier of this application pattern facilitates the invocation of business services hosted by a number of back-end applications. In doing so, the Decomposition application pattern fully leverages the integration capabilities described by the Application Integration::Broker pattern.

   If an interaction initiated by a user requires the execution of an end-to-end business process or workflow where process and workflow rules are better externalized, the Decomposition application pattern would leverage the integration capabilities of more advanced process integration alternatives such as the Application Integration::Serial Process or Serial Workflow and the Application Integration::Parallel Process or Parallel Workflow. Since the end result is the decomposition/recomposition capability discussed above, these variations are not documented as Decomposition application pattern variations, but rather may be captured as different runtime patterns where applicable.

7. Agent application pattern:

   The Agent application pattern structures an application design to provide a unified customer-centric view that can be exploited for mass customization of services, and for cross-selling purposes. The unified customer-centric view across Lines of Businesses (LOB) in this case is either dynamically developed or supported by an Operational Data Store (ODS) that collects near real time data about the user from multiple systems.

2.3 Application pattern used in this book

The rest of this redbook discusses e-business solutions using the following Application patterns:

- Directly Integrated Single Channel application pattern
- Router application pattern
- Decomposition application pattern

3

SOA and the Enterprise Service Bus

This chapter provides an introduction to service-oriented architecture (SOA). It also defines the Enterprise Service Bus (ESB) and describes the ESB in terms of the role that it plays in the implementation of an SOA.

3.1 Overview of SOA

SOA defines integration architectures based on the concept of a *service*. Applications collaborate by invoking each others services, and services are composed into larger sequences to implement business processes.

The drive for SOA

The main driver for SOA is to define an architectural approach that assists in the flexible integration of IT systems. Organizations spend a considerable amount of time and money trying to achieve rapid, flexible integration of IT systems across all elements of the business cycle. The drivers behind this objective include:

- Increasing the speed at which businesses can implement new products and processes, can change existing ones, or can recombine them in new ways
- Reducing implementation and ownership costs of IT systems and the integration between them
- Enabling flexible pricing models by outsourcing more fine-grained elements of the business than were previously possible or by moving from fixed to variable pricing, based on transaction volumes
- Simplifying the integration work that is required by mergers and acquisitions
- Achieving better IT use and return on investment
- Achieving implementation of business processes at a level that is independent from the applications and platforms that are used to support the processes

SOA prescribes a set of design principles and an architectural approach to achieve this rapid, flexible integration.

Definition of SOA

SOA is an integration architecture approach that is 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 user applications or other services.

SOA specifies that within any given architecture, there should be a consistent mechanism by which services communicate. That mechanism should be loosely coupled and should support the use of explicit interfaces.

SOA brings the benefits of loose coupling and encapsulation to integration at an enterprise level. It applies successful concepts that are proven 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 to SOA. They provide the function out of which you can build distributed systems. Services can be invoked independently by either external or internal service consumers to process simple functions or can be chained together to form more complex functionality and to quickly devise new functionality.

By adopting an SOA approach and implementing it using supporting technologies, you can build flexible systems that implement changing business processes quickly and make extensive use of reusable components.

Figure 3-1 illustrates a company that wants to implement a new business process to support customers who place orders from a Web site.

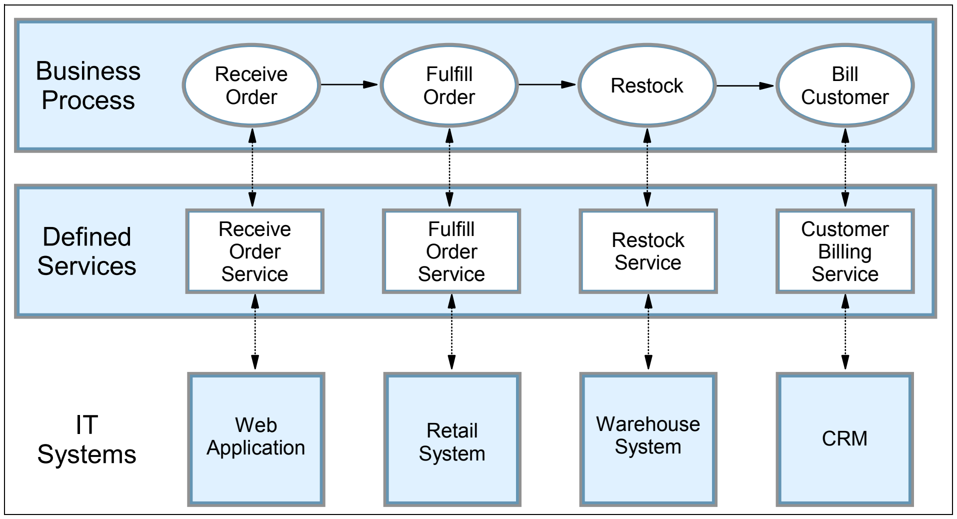


Figure 3-1 A service-oriented approach to building systems

The company already has existing retail, warehouse, and billing systems. It would like to build the new process by reusing the functionality that is provided by those systems, rather than having to write new applications or new interfaces to the existing systems.

If the company has already adopted an SOA approach, it will have defined the interfaces to its existing systems in terms of the functions or services that they can offer to support the building of business processes. The defined interfaces makes building the new Web front end to the system very simple. All the company needs to do is to develop an application that makes calls to the services to complete the new business process.

The SOA approach means companies are able to build horizontal business processes that integrate systems, people, and processes from across the enterprise quickly and easily in response to changing business needs.

As shown in Figure 3-1 on page 25, the company can use existing systems to implement new business processes that extend the use of the system beyond the processes that they were originally written to support. The company can maximize the previous IT investment by reusing existing IT systems without having to invest extensively to build new interfaces to the systems.

On Demand Business and SOA

SOA plays a crucial role for companies who are trying to implement the IBM vision of On Demand Business. The IBM vision of On Demand Business is to enable customers to succeed in an environment with an unprecedented rate of change.

In an on demand world, companies need to respond quickly and easily to any customer requirement, opportunity, or threat. To succeed in this environment, a company must be able to implement new processes quickly while leveraging existing investment. From a business perspective, On Demand Business provides a way for companies to realign their business and technology environment to match the need for reusable business functionality. For a fuller discussion about the On Demand Business vision from IBM and how it relates to SOA refer to the second chapter of *Patterns: Implementing an SOA Using an Enterprise Service Bus,* SG24-6346**.**

SOA can be an architectural enabler for On Demand Business. SOA touches on the four key elements of On Demand Business. These elements are:

- Open standards

  SOA provides a standard method of invoking services (business logic and functionality) for disparate organizations to share across network boundaries.

- Integration
  - SOA provides interfaces to wrap service endpoints for a system-independent architecture that promotes cross-industry communication and integrates end-to-end solutions both in and out of the enterprise.
  - SOA provides dynamic service discovery and binding, which means that service integration can occur on demand.
  - SOA provides an approach to integrate heterogeneous technologies inside an enterprise.

- Virtualization

  A key principle of SOA is that consumers who invoke the services are oblivious to implementation details, including location, platform, and if appropriate to the business scenario, even the identity of the service provider.
- Automation

  Technologies, such as grid technologies, can apply SOA principles to implementing infrastructure services that provide an evolutionary approach to increased automation.

3.1.1 Definition of a service

SOA is an architectural approach to defining integration architectures that are based on services. Now, it is important to define what is meant by a *service* in this context in order to fully describe SOA and to understand what you can achieve by using it.

A service can be defined as any discrete function that can be offered to an external consumer. The function can be an individual 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 an SOA. The most commonly agreed-on aspects of a service are that:

- Services encapsulate a reusable business function.
- Services are defined by explicit, implementation-independent interfaces.
- Services are invoked through communication protocols that stress location transparency and interoperability.

This book uses these commonly agreed upon aspects to define SOA.

Reusable function

Any business function can be a service. SOA often focusses on business functions. However, many technical functions can also be exposed as services. When defining function, there are several levels of granularity that you can consider. Many descriptions of SOA refer to the use of *large-grained* services; however, some powerful counter-examples of successful, reusable, fine-grained services exist. For example, getBalance is a very useful service, but is not large-grained.

More realistically, there are many useful levels of service granularity in most SOAs. For example, all of the following are services that each have a different granularity:

- Technical Function Services (for example auditEvent, checkUserPassword, and checkUserAuthorization)
- Business Function Services (for example calculateDollarValueFromYen and getStockPrice)
- Business Transaction Services (for example checkOrderAvailability and createBillingRecord)
- Business Process Services (for example openAccount, createStockOrder, reconcileAccount, and renewPolicy)

Some degree of choreography or aggregation is required between each granularity level for them to be integrated in an SOA.

A service can be any business function. In an SOA, however, it is preferable that the function is genuinely reusable. In an SOA, the service can be used and reused by one or more systems that participate in the architecture. For example, while the reuse of a Java logging API could be described as *design time* (when a decision is made to reuse an available package and bind it into application code), the intention of SOA is to achieve the reuse of services at:

- Runtime

  Each service is deployed in one place and one place only and is invoked remotely by anything that must use it. The advantage of this approach is that changes to the service (for example, to the calculation algorithm or the reference data it depends on) need only be applied in a single place.

- Deployment time

  Each service is built once but redeployed locally to each system or set of systems that must use it. The advantage of this approach is increased flexibility to achieve performance targets or to customize the service (perhaps according to geography).

The service definition should encapsulate the function well enough to make the reuse possible. The encapsulation of functions as services and their definition using interfaces enables the substitution of one service implementation for another. For example, the same service might be provided by multiple providers (such as a car insurance quote service, which might be provided by multiple insurance companies) and individual service consumers might be routed to individual service providers through some intermediary agent.

Granularity in SOA

The concept of *granularity* is used to mean several things in SOA, each of which is actually quite separate:

- Level of abstraction of services

  Is the service a high-level business process, a lower-level business sub-process or activity, or a very low-level technical function?

- Granularity of service operations

  How many operations are in the service? One, a few, or many? What factors determine which operations are collected together in a service?

- Granularity of service parameters

  How are the input and output data of service operations expressed? SOA prefers a small number of large, structured parameters rather than a small number of primitive types.

Explicit implementation of independent interfaces

The use of explicit interfaces to define and encapsulate service function is of particular importance in making services genuinely reusable. The interface should encapsulate only those aspects of process and behavior that are used in the interaction between the service consumer and the service provider. An explicit interface definition, or contract, is used to bind a service consumer and a service provider. It should specify only the mutual behavior that is required for the interaction and nothing about the implementation of the consumer or the provider.

By explicitly defining the interaction in this way, those aspects of either system (for example, the platform on which they are based) that are not part of the interaction are free to change without affecting the other system. This flexibility allows either system to change implementation or identity freely.

Figure 3-2 illustrates the use of explicit interfaces to define and encapsulate services function.

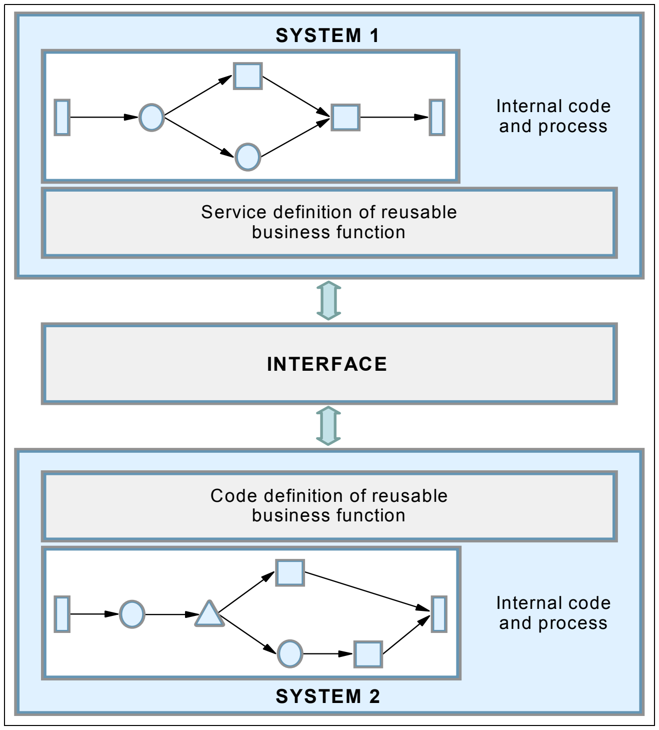

SYSTEM 1
Internal code and process
Service definition of reusable business function
INTERFACE
Code definition of reusable business function
Internal code and process
SYSTEM 2

Figure 3-2 Service implementation in SOA

Communication protocols that stress location transparency

Companies have a variety of choices when deciding how to connect applications. HTTP, HTTPS, JMS, CORBA, and SMTP are all examples of protocols that can be used to connect applications. There are also many middleware products, for example WebSphere MQ, that provide application-to-application connectivity. Typically, even within a single company, a variety of techniques, products, and protocols are used to address different integration requirements. This variety of techniques can create problems when trying to extend the integration to connect to applications that do not use the same protocols.

SOA does not specify that any specific protocol should be used to provide access to a service. A key principle in SOA is that a service is not defined by the communication protocol that it uses, but instead is protocol-independent so that different protocols can be used to access the same service.

Ideally, a service should be defined only once, through a service interface, and should have many implementations with different access protocols. This definition increases the reusability of any service definition. Also, services should be invoked, published, and discovered in a way that is abstracted away from the actual implementation using a single, standards-based form of interface. Thus, there is a complimentary nature between SOA and Web services.

3.1.2 Web services and SOA

An appropriate combination of both Web services technology and the SOA approach addresses many of the issues of building an SOA-enabled environment. That is not to say that Web services and SOA are intrinsically linked, because they can be implemented separately. In fact, many significant SOAs are proprietary or customized implementations that are based on messaging and Enterprise Application Integration middleware and do not use Web services technologies. Also, most existing Web services implementations consist of point-to-point integrations that address a limited set of business functions between a defined set of cooperating partners.

However, existing SOA implementations have demonstrated the benefits of SOA, usually within a single enterprise, and the existing uses of Web services have demonstrated the benefits of the Web services technologies in integrating heterogeneous systems both within and among organizations. A custom approach gives an organization the problem of supporting heterogenity; a proprietary approach gives it to one IT vendor. Adopting a standards-based approach, such as Web services, offers a solution to these issues.

There are logical links between Web services and SOA that suggest that they are complimentary:

- Web services provide an open-standard and machine-readable model for creating explicit, implementation-independent descriptions of service interfaces.
- Web services provide communication mechanisms that are location-transparent and interoperable.
- Web services are evolving, through Business Process Execution Language for Web Services (BPEL4WS), document-style SOAP, Web services Definition Language (WSDL), and emerging technologies (such as WS-ResourceFramework), to support the technical implementation of well-designed services that encapsulate and model reusable function in a flexible manner.

Working together, Web services and SOA have the potential to address many of the technical issues that enterprises face when trying to build an on demand environment. For example:

- A multitude of technologies and platforms are used to support business systems, all which need to be integrated into an SOA.

  Web services are a set of open-standard technologies that are supported by most of the IT industry and by the Web Services Interoperability (WS-I) organization. Their basis is in simple, text-based, and open-standard technologies such as XML and HTTP, and the fact that they can leverage more sophisticated interoperable technologies, such as asynchronous messaging, means that they can be supported in the vast majority of IT environments. Increasing ubiquity and maturity of product support means that implementing and integrating Web services will become increasingly efficient.

- Business process models are a mixture of people practices, application code, and interactions among people and systems or systems and systems.

  Although SOA is an approach to architecture that must be applied to systems and integrations, it specifies a set of principles and techniques that encourage the encapsulation and modeling of reusable business functions and processes. Recent and emerging trends in Web services, such as BPEL4WS and WS-ResourceFramework, will increasingly support the modeling concepts of SOA. In this way, process management can be centralized rather than being part of multiple applications.

- Changes to one system tend to imply ripples of change at many levels to many other systems.

  SOA specifies several principles and techniques for achieving the encapsulation of service function and the loose coupling of service interactions. These techniques minimize the cases where change to one part of a system implies changes to other parts.

- In a true SOA, the integration solution should be able to invoke services offered outside the enterprise by partners and should be extendable to support future partners.

The Web services technologies have proven effective in many business-to-business integrations, where their open standards basis and use of simple, existing infrastructure and protocols makes them particularly effective. Recent and emerging standards, such as WS-Security, add to the sophistication of interaction that is possible when using Web services in this model.

- There is no single data, business, or process model across, or beyond, the enterprise.

  Although they are not a magic solution to this issue, the SOA principles define an approach that enables organizations to progressively expose functions across their business as services and to combine those services into processes. SOA encourages processes to be centrally managed and explicitly defined and modelled. Over time, businesses that take this approach will improve the consistency of their business and process models and will leverage the use of business process modeling and automation technology to more explicitly control and monitor their execution of processes.

- Not all integration technologies work as well across a wide area network or the Internet as they do across a local area network.

  The Web services technologies support multiple protocols, so they can use the simplest protocols available, such as HTTP when that offers an advantage, or leverage other infrastructures such as WebSphere MQ when that is more appropriate.

For these reasons, SOA and Web services are often seen together as the future direction for system integration. However, note that in Web services that WSDL does not specify all that is implied by what SOA means by an interface. It does not specify service levels, and it does not specify pre- and post- conditions. BPEL4WS can do something equivalent but only for a subset of requirements, for example in process models. All SOA projects have to make up this shortfall in standard project documentation, custom service descriptions, or some other means.

3.1.3 Messaging and SOA

SOAs that are based on reliable messaging and Enterprise Application Integration middleware (for example WebSphere MQ and WebSphere Business Integration Message Broker) support the principles of an SOA implementation by:

- Decoupling the consumer's view of a service from the actual implementation of the service
- Decoupling technical aspects of service interactions
- Integrating and managing services in the enterprise

Decoupling the consumer's view of a service from the actual implementation greatly increases the flexibility of the architecture. It allows the substitution of one service provider for another (for example because another provider offers the same services for lower cost or with higher standards) without the consumer being aware of the change, or needing to be altered to support it.

This decoupling is better achieved by having the consumers and providers interact via an intermediary. Intermediaries publish services to consumers. The consumer binds to the intermediary to access the service, with no direct coupling to the actual provider of the service. The intermediary maps the request to the location of the real service implementation.

Figure 3-3 shows requestor and provider connected using a messaging infrastructure as intermediary.

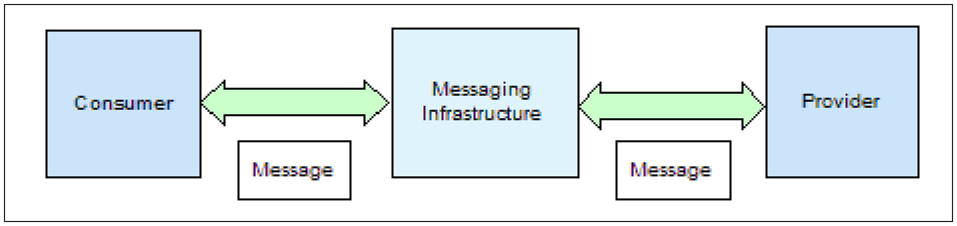


Figure 3-3 Decoupling requestor and provider using messaging as intermediary

The SOA principles of granularity and modularity are primarily solved through the proper structuring of the application. In contrast, the aspect of loose coupling is greatly addressed by using messaging middleware.

Table 3-1 on page 34 shows an overview about the coupling aspects related to the use of messaging middleware.

Table 3-1 Coupling aspects of messaging middleware

Coupling aspect	**Justification**
Language independence	The payload of a message can be passed in a language independent manner. An appropriate interface to the messaging middleware needs to exist for requestor and provider.
Transport protocol transparency	The transport protocol used is encapsulated by the interface of the messaging infrastructure. A requestor does not need to know if a provider is connected using the same transport protocol.
Location transparency	For a service requestor or provider the messaging infrastructure is just a communication medium. A requestor does not need to know the route a message takes as long as it gets the result it expects.

Coupling aspect	Justification
Data format independence	The payload of a message is passed in a data format independent manner.
Platform independence	Messaging infrastructure supports the communication between different platforms and even provides mapping functionality between different data and encoding formats.
Communication model transparency	Messaging not only supports the synchronous communication model but also the asynchronous thus providing enhanced flexibility in binding and orchestrating services.

3.1.4 The advantages of SOA

Use of SOA has the following advantages to achieving loosely coupled, flexible integration of IT systems:

- Heterogeneous systems can be integrated because of implementation-independent interfaces that describe services.
- The description of service interfaces in terms of a common business process and data model minimizes any interdependencies to only what matters to the business.
- The encapsulation of services with standard interfaces enables reuse and flexibility. Each service is defined and implemented in only one place, so changing it is straightforward.

There are benefits in development and maintenance costs, but flexibility is the primary goal in SOA.

With clearly defined interfaces between all business systems, it is possible to model and change the business process that are captured by them at a level above individual systems. Thus, SOA is an enabler for process modelling and automation at an enterprise scale.

Currently, and for some time to come, many of the technologies that are used to implement SOAs are evolving rather than maturing and stablizing. Therefore, individual SOA solutions must make carefully balanced decisions among customized, proprietary, and open-standard technologies, which characteristics and components of SOA to implement, and which areas of business function and process to which to apply them. Of course, you should balance these decisions between business benefits, technology maturity, and implementation or maintenance efforts.

3.1.5 SOA summary

SOA and Web services enable new opportunities for more flexible, rapid, and widespread integration in a model that is consistent with the exposure of business function as services. SOA and Web services offer the choreography of those services into processes that can be modeled, executed, and monitored with features such as:

- SOA defines concepts and general techniques for designing, encapsulating, and invoking reusable business functions through loosely bound service interactions. Most of the techniques have been proven individually in previous technologies or design styles. SOA unites them in an approach that is intended to bring encapsulation and reuse to the enterprise level.
- Web services provide an emerging set of open-standard technologies that can be combined with proven existing technologies to implement the concepts and techniques of SOA.
- Industry support for Web services standards, interoperability among different implementations of Web services, and the infrastructure technology that is required to support an SOA give technology customers increasingly mature and sophisticated technologies that are suitable for SOA implementation.

These techniques and technologies give you the tools that are required to implement flexible SOAs and to evolve toward an on demand business model. However, SOA is an architectural approach, not a technology or a product. In order to implement an SOA, you must have the infrastructure to support the architecture, such as an Enterprise Service Bus.

3.2 Overview of the Enterprise Service Bus

Successfully implementing an SOA requires applications and infrastructure that can support the SOA principles. Applications can be enabled by creating service interfaces to existing or new functions that are hosted by the applications. The service interfaces should be accessed using an infrastructure that can route and transport service requests to the correct service provider. As organizations expose more and more functions as services, it is vitally important that this infrastructure should support the management of SOA on an enterprise scale.

3.2.1 SOA infrastructure requirements

The Enterprise Service Bus (ESB) is emerging as a middleware infrastructure component that supports the implementation of SOA within an enterprise. The need for an ESB can be seen by considering how it supports the concepts of SOA implementation by:

- Decoupling the consumer's view of a service from the implementation of a service
- Decoupling technical aspects of service interactions
- Integrating and managing services in the enterprise

Decoupling the consumer's view of a service from the actual implementation greatly increases the flexibility of the architecture. It allows the substitution of one service provider for another (for example, because another provider offers the same services for lower cost or with higher standards) without the consumer being aware of the change or without the need to alter the architecture to support the substitution.

This decoupling is better achieved by having the consumers and providers interact through an intermediary. Intermediaries publish services to consumers. The consumer binds to the intermediary to access the service, with no direct coupling to the actual provider of the service. The intermediary maps the request to the location of the real service implementation.

In an SOA, services are described as being loosely coupled. However, at implementation time, there is no way to loosely couple a service or any other interaction between systems. The systems must have some common understanding to conduct an interaction. Instead, to achieve the benefits of loose coupling, consideration should be given to how to couple or decouple various aspects of service interactions, such as the platform and language in which services are implemented, the communication protocols used to invoke services, the data formats used to exchange input and output data between service consumers and providers.

Further decoupling can be achieved by handling some of the technical aspects of transactions outside of applications. This could apply aspects of interactions such as:

- How service interactions are secured
- How the integrity of business transactions and data are maintained, for example through reliable messaging, the use of transaction monitors, or compensation techniques
- How the invocation of alternative service providers is handled in the event that the default provider is unavailable

These aspects imply a need for middleware to support an SOA implementation. Some of the functions that might be provided by the middleware are:

- 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 might 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

This middleware support is the role of an ESB.

3.2.2 Definition of an ESB

An *ESB* provides an infrastructure that removes any direct connection between service consumers and providers. Consumers connect to the bus and not the provider that actually implements the service. This type of connection further decouples the consumer from the provider. A bus also implements further value add capabilities. For example, security and delivery assurance can be implemented centrally within the bus instead of having this buried within the applications.

Integrating and managing services in the enterprise outside of the actual implementation of the services in this way helps to increase the flexibility and manageability of SOA.

The primary driver for an ESB, however, is that it increases decoupling between service consumers and providers. Protocols such as Web services define a standard way of describing the interface to a service provider that allow some level of decoupling because the actual implementation details are hidden. However, the protocols imply a direct connection between the consumer and provider.

Although it is relatively straight forward to build a direct link between a consumer and provider, these links can lead to an interaction pattern that consists of building multiple point-to-point links that perform specific interactions. With a large number of interfaces, this quickly leads to the build up of a complex spaghetti of links with multiple security and transaction models. Routing control is distributed throughout the infrastructure, and probably no consistent approach to logging, monitoring, or systems management is implemented. This environment is difficult to manage or maintain and inhibits change.

A common approach to reducing this complexity is to introduce a centralized point through which interactions are routed, as shown in Figure 3-4.

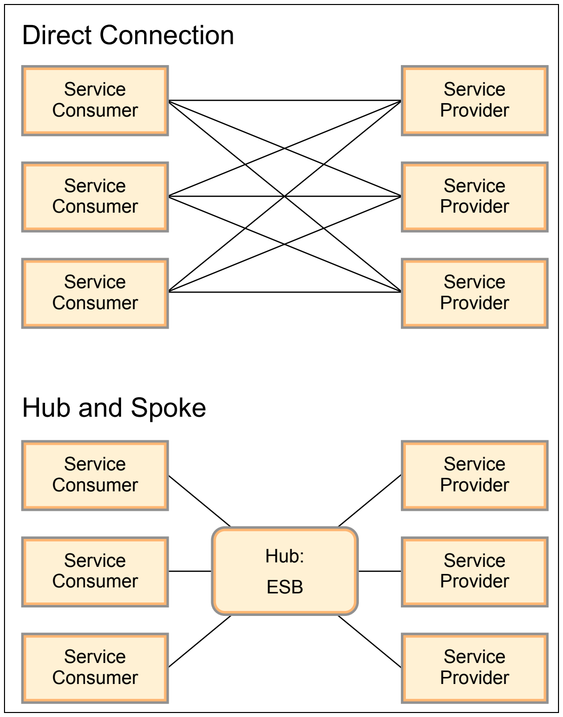


Figure 3-4 Direct connection and central hub integration styles

A hub and spoke architecture is a common approach that is used in application integration architectures. In a hub, the distribution rules are separated from applications. The applications connect to the hub and not directly to any application. This type of connection allows a single interaction from an application to be distributed to multiple target applications without the consumer being aware that multiple providers are involved in servicing the request. This connection can reduce the proliferation of point-to-point connections.

Note that the benefit of reducing the number of connections only truly emerges if the application interfaces and connections are genuinely reusable. For example, consider the case where one application needs to send data to three other applications. If this is implemented in a hub, the sending application must define a link to the hub, and the hub must have links that are defined to the three receiving applications, giving a total of four interfaces that need to be defined. If the same scenario was implemented using multiple point-to-point links, the sending application would need to define links to each of the three receiving applications, giving a total of just three links. A hub only offers the benefit of reduced links if another application also needs to send data to the receiving applications and can make use of the same links as those that are already defined for the first application. In this scenario, the new application only needs to define a connection between itself and the hub, which can then send the data correctly formatted to the receiving applications.

Hubs can be federated together to form what is logically a single entity that provides a single point of control but is, in fact, a collection of physically distributed components. This arrangement is commonly termed a *bus.* A bus provides a consistent management and administration approach to a distributed integration infrastructure.

3.2.3 Enterprise requirements for an ESB

Using a bus to implement an SOA has a number of advantages. In an SOA services should, by definition, be reusable by a number of different consumers, so that the benefits of reduced connections are achieved. In addition, the ESB:

- Supports high volumes of individual interactions.
- Supports 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.
- Supports centralization of enterprise-level qualities of service and manageability requirements into the hub.

Figure 3-5 shows a high-level view of the ESB.

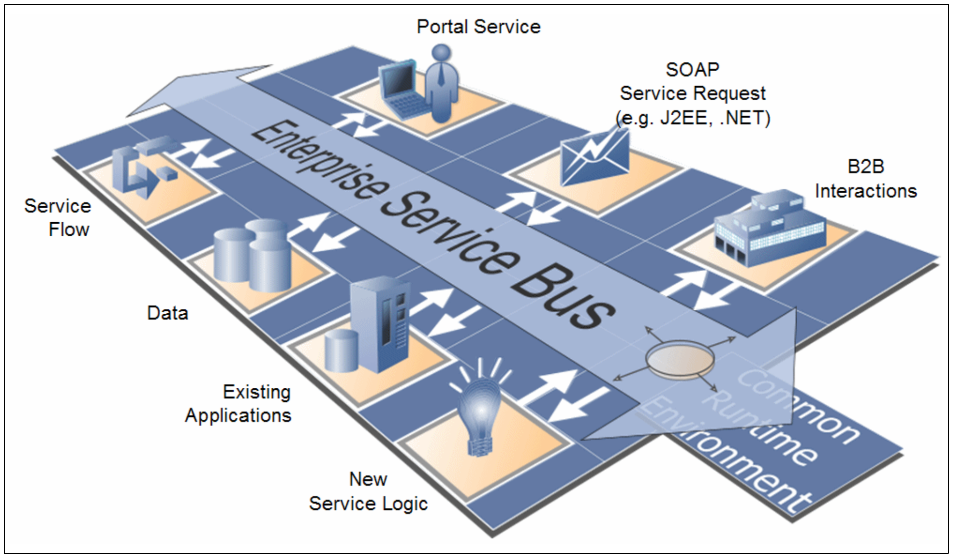


Figure 3-5 The Enterprise Service Bus

As discussed in 3.1, “Overview of SOA” on page 24, SOA applications are built from services. Typically, a business service relies on many other services in its implementation. The ESB is the component that provides access to the services and so enables the building of SOA applications.

Mediation support

The ESB is more than just a transport layer. It must provide mediation support to facilitate service interactions. An example of medation support would be to find services that provide capabilities for which a consumer is asking or to take care of interface mismatches between consumers and providers that are compatible in terms of their capabilities. It must support a variety of ways to get on and off the bus, such as adapter support for existing applications or business connections, that enable external partners in business-to-business interaction scenarios. To support these different ways to get on and off the bus, it must support service interaction with a wide variety of service endpoints. It is likely that each endpoint will have its own integration techniques, protocols, security models and so on. This level of complexity should be hidden from service consumers. They need to be offered a simpler model. In order to hide the complexity from the consumers, the ESB is required to mediate between the multiple interaction models that are understood by service providers and the simplified view that is provided to consumers.

Protocol independence

As shown in Figure 3-5 on page 41, services can be offered by a variety of sources. Without an ESB infrastructure, any service consumer that needs to invoke a service needs to connect directly to a service provider using the protocol, transport, and interaction pattern that is used by the provider. With an ESB, the infrastructure shields the consumer from the details of how to connect to the provider.

In an ESB, there is no direct connection between the consumer and provider. Consumers access the ESB to invoke services, and the ESB acts as an intermediary by passing the request to the provider using the appropriate protocol, transport, and interaction pattern for the provider. This intermediary connection enables the ESB to shield the consumer from the infrastructure details of how to connect to the provider. The ESB should support several integration mechanisms. These mechanisms can be described as invoking services through specific addresses and protocols, even if, in some cases, the address is the name of a CICS transaction and the protocol is a J2EE resource adapter integrating with the CICS Transaction Gateway. By using the ESB, the consumers are unaware of how the service is invoked on the provider.

Because the ESB removes the direct connection between service consumer and providers, an ESB enables the substitution of one service implementation by another with no effect to the consumers of that service. Thus, an ESB allows the reach of an SOA to extend to non-SOA enabled service providers. It can also be used to support migration of the non-SOA providers to using an SOA approach without impacting the consumers of the service.

Support for multiple interaction patterns

To support fully the variety of interaction patterns (request/response, publish/subscribe, and events) that are required in a comprehensive SOA, the ESB must support in one infrastructure the following major styles of enterprise integration:

- SOAs in which applications that communicate through reusable services with well-defined, explicit interfaces

  Service-oriented interactions leverage underlying messaging and event-communication models.

- Message-driven architectures in which applications send messages through the ESB to receiving applicationsEvent-driven architectures in which applications generate and consume messages independently of one another

The ESB support the enterprise integration while providing additional capabilities to mediate or transform service messages and interactions, enabling a wide variety of behaviors and supporting the various models of coupling interaction.

3.2.4 Minimum ESB capabilities

This section discusses the minimum capabilities an ESB must have to support the requirements of an SOA enabling infrastructure component. Understanding the minimum capabilities allows you to assess the suitability of individual technologies or products for implementing an ESB by analyzing the functionality that they offer to support the minimum ESB capabilities.

In discussions on ESB, the most commonly agreed upon elements for defining an ESB are:

- The ESB is a logical architectural component that provides an integration infrastructure consistent with the principles of SOA.
- The ESB can be implemented as a distributed, heterogeneous infrastructure.
- The ESB provides the means to manage the service infrastructure and the capability to operate in a distributed, heterogeneous environment.

Table 3-2 on page 44 summarizes the minimum capabilities that an ESB should have in order to provide an infrastructure consistent with these elements, and thus consistent with the benefits of SOA. The sections that follow discuss these capabilities in more detail.

Table 3-2 Minimum capabilities of an ESB

Category	Capabilities	Reasons
Communications	► Routing ► Addressing ► At least one messaging style (request/response, publish/subscribe) ► At least one transport protocol that is or can be made widely available	Provides location transparency and supports service substitution
Integration	► Several integration styles or adapters ► Protocol transformation	Supports integration in heterogeneous environments and supports service substitution
Service interaction	► Service interface definition ► Service messaging model ► Substitution of service implementation	Supports SOA principles, separating application code from specific service protocols and implementations
Management	Administration capability	A point of control over service addressing and naming

Communication

The ESB needs to supply a communication layer to support service interactions. It should support communication through a variety of protocols. It should provide underlying support for message and event-oriented middleware and integrate with existing HTTP infrastructure and other enterprise application integration (EAI) technologies. As a minimum capability, the ESB should support at least the protocols that fit the requirements of a specific situation. The ESB should be able to route between all these communication technologies through a consistent naming and administration model.

Integration

The ESB should support linking to a variety of systems that do not directly support service-style interactions so that a variety of services can be offered in a heterogeneous environment. This includes existing systems, packaged applications and other EAI technologies. Integration technologies might be protocols (for example JDBC™, FTP, or EDI) or adapters such as the J2EE Connector Architecture resource adapters or WebSphere Business Integration Adapters. It also includes service client invocation through client APIs for various languages (Java, C+, or C#) and platforms (J2EE or .Net), CORBA, and RMI.

Service interaction

The ESB needs to support SOA concepts for the use of interfaces and support declaration service operations and quality of service requirements. The ESB should also support service messaging models consistent with those interfaces, and be capable of transmitting the required interaction context, such as security, transaction or message correlation information.

Management

As with any other infrastructure component, the ESB needs to have administration capabilities so that it can be managed and monitored to provide a point of control over service addressing and naming. In addition, it should be capable of integration into systems management software.

3.2.5 ESB and Web services technologies

Given the prominence of Web services technologies in current discussions of SOA and the fact that many successful implementations of Web services technologies exist, it is interesting to analyze what the use of basic Web services technologies (WSDL and SOAP/HTTP) achieves against the minimum ESB capabilities that are described in 3.2.4, “Minimum ESB capabilities” on page 43. Using basic Web services technologies achieves:

- URL addressing and the existing HTTP and DNS infrastructure provide a bus with routing services and location transparency.
- SOAP/HTTP supports the request/response messaging paradigm.
- The HTTP transport protocol is widely available.
- SOAP and WSDL are an open, implementation-independent messaging and interfacing model.

Although the use of SOAP/HTTP and WSDL in this way has many advantages, this scenario falls short of the minimum capabilities of the ESB in the following ways:

- The scenario relies on the provision of interoperable SOAP/HTTP enablement of each participating system. Because the Web services standards are continuing to mature, there are many systems for which this will not be feasible. An ESB should provide some form of support for alternative integration techniques.
- Control over service addressing and routing is dispersed between client invocation code, adapter configurations, and the DNS infrastructure. There is no single point of infrastructure control. In other words, this is a point-to-point integration style.

Vitally, there is no capability to substitute one implementation of a service provider for another without changing the service consumers. Clients and provider code tend to be bound to service invocations over specific protocols and to specific addresses.

In conclusion, the use of basic Web services technologies on their own is not sufficient to build an ESB. This technology only supports a subset of the minimum capabilities that an ESB needs to provide. Support of the Web services technologies is highly desirable within the ESB, as is support for other technologies that are required in combination to fully implement an ESB infrastructure.

3.2.6 Extended ESB capabilities

The minimum capabilities described in 3.2.4, “Minimum ESB capabilities” on page 43 can help assess the suitability of individual technologies or products for implementing an ESB. However, checking your avialable technologies against these minimum capabilities will establish only those technologies that are candidates. The detailed requirements of any particular scenario drive additional ESB capabilities that can then be used to select specific, appropriate products.

In particular, the following types of requirements are likely to lead to the use of more sophisticated technologies, either now or over time:

- Non-functional requirements such quality of service demands and service-level capabilities
- Higher-level SOA concepts, such as a service directory, and transformations
- Advanced management capabilities, such as system management, and autonomic and intelligent capabilities
- Truly heterogeneous operation across multiple networks, multiple protocols, and multiple domains of disparate ownership

Figure 3-6 on page 47 shows the vision of the IBM On Demand Operating Environment based on SOA.

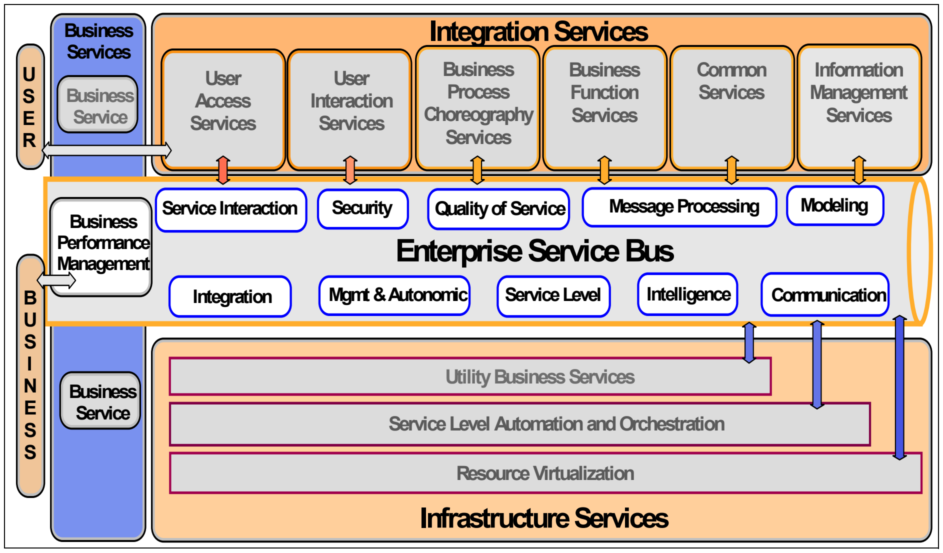


Figure 3-6 On Demand Operating Environment architecture

Figure 3-6 shows the capabilities that the ESB requires to facilitate the interactions between the levels in the On Demand Operating Environment. These capabilities include service level, service interface, quality of service, intelligence, communication, security, message management, modeling, management/automation, and integration capabilities.

If we consider the requirements for an ESB in light of both the minimum requirements described in 3.2.4, “Minimum ESB capabilities” on page 43 and the IBM On Demand Operating Environment requirements, then several additional capability requirements can be identified, as shown in Table 3-3 on page 48. Note that many situations not in the On Demand Operating Environment can also require some of these capabilities in addition to the minimum requirements.

Table 3-3 Categorized ESB capabilities

Communication	Service interaction
► Routing ► Addressing ► Protocols and standards (HTTP, HTTPS) ► Publish/subscribe ► Response/request ► Fire and forget, events ► Synchronous and asynchronous messaging	► Service interface definition (WSDL) ► Substitution of service implementation ► Service messaging models required for communication and integration (SOAP, XML, or proprietary Enterprise Application Integration models) ► Service directory and discovery
Integration	**Quality of service**
► Database ► Legacy and application adapters ► Connectivity to enterprise application integration middleware ► Service mapping ► Protocol transformation ► Data enrichment ► Application server environments (J2EE and .Net) ► Language interfaces for service invocation (Java, C/C++, or C#)	► Transactions (atomic transactions, compensation, WS-Transaction) ► Various assured delivery paradigms (WS-ReliableMessaging or support for Enterprise Application Integration middleware)
Security	**Service level**
► Authentication ► Authorization ► Non-repudiation ► Confidentiality ► Security standards (Kerberos, WS-Security)	► Performance (response time, throughput and capacity) ► Availability ► Other continuous measures that might form the basis of contracts or agreements

Message processing	Management and autonomic
► Encoded logic ► Content-based logic ► Message and data transformations ► Message and service aggregation and correlation ► Validation ► Intermediaries ► Object identity mapping ► Service / message aggregation ► Store and forward	► Administration capability ► Service provisioning and registration ► Logging ► Metering ► Monitoring ► Integration to systems management and administration tooling ► Self-monitoring and self-management
Modeling	**Infrastructure Intelligence**
► Object modeling ► Common business object models ► Data format libraries ► Public versus private models for business-to-business integration ► Development and deployment tooling	► Business rules ► Policy-driven behavior, particularly for service level, security and quality of service capabilities (WS-Policy) ► Pattern recognition

Integration

Because additional integration capabilities could be supported, the ESB should be capable of connectivity to a wide range of different service providers, using adapters and EAI middleware. It should be capable of data enrichment to alter the service request content and destination on route, and map an incoming service request to a one or more service providers.

Quality of service

The ESB might be required to support service interactions that require different qualities of service to protect the integrity of data mediated through those interactions. This support can involve transactional support, compensation, and levels of delivery assurance. These features should be variable and driven by service interface definitions. Other ESB quality of service considerations include:

- Support qualities of service on top of communication protocols that are fundamentally more brittle
- Business transactions spanning several systems that need to be monitored as a whole
- Support for exception and error handling

Security

The ESB should ensure the integrity and confidentiality of the services that it carries is maintained. It should integrate with the existing security infrastructures to address the essential security functions, such as:

- Identification and authentication
- Access controls
- Confidentiality
- Data integrity
- Security management and administration
- Disaster recovery and contingency planning
- Incident reporting

Additionally, the ESB should integrate with the overall management and monitoring of the security infrastructure. The ESB can either provide security directly or can integrate with other components, such as authentication, authorization, and directory components.

Service level

The ESB should mediate interactions between systems supporting specific performance, availability and other requirements. It should offer a variety of techniques and capabilities to meet these requirements. The ESB should provide support that allows technical and business service level agreements to be monitored and enforced.

Message processing

The ESB needs to be capable of integrating message, object, and data models between the application components of an SOA. It should also be able to make decisions, such as routing, based on content of service messages. The ESB needs a mediation model that allows message processing to be customized. The model should also allow sequencing of infrastructure services (for example, security logging and monitoring) around business services invocations. Mediations can be located close to consumers, providers, or anywhere in the ESB infrastructure that is transparent to consumers and providers. Mediations can also be chained. The ESB should be able to validate content and format.

Management and autonomic capabilities

In addition to basic management capabilities, the ESB should also support the migration to autonomic and on demand infrastructure by supporting metering and billing, self-healing and dynamic routing, and it should be able to react to events to self-configure, heal, and optimize.

Modeling

The ESB should support the increasing array of cross-industry and vertical standards in both the XML and Web services spaces. It should support custom message and data models. The ESB should also support the use of development tooling and be capable of identifying different models for internal and external services and processes.

Infrastructure intelligence

The ESB should be capable of evolving towards a more autonomic, infrastructure. It should allow business rules and policies to affect ESB function, and it should support pattern recognition.

3.2.7 The ESB and other SOA components

The ESB is not the only infrastructure component in an SOA. Although individual scenarios vary, other commonly occurring components are:

- Business Service Directory, which provides details of available services to systems that participate in the SOA.
- Business Service Choreography, which is used to orchestrate sequences of service interactions into short or long-lived business processes.
- ESB Gateway, which is used to provide a controlled point of external access to services where the ESB does not provide this natively. Larger organizations are likely to keep the ESB Gateway as a separate component. An ESB Gateway can also be used to federate ESBs within an enterprise.

4

Runtime patterns

The next step is to choose Runtime patterns that most closely match the requirements of the application.

Runtime patterns are used to define the logical middleware structure supporting the Application patterns. In other words, Runtime patterns describe the logical architecture required to implement an Application pattern. Runtime patterns depict the major middleware nodes, their roles, and the interfaces between these nodes.

The Runtime patterns illustrated in this chapter give some typical examples of possible solutions, but these examples should not be considered exhaustive.

4.1 An introduction to the node types

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

- User node

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

- Domain Name System (DNS) node

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

- Public Key Infrastructure (PKI)

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

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

- Web application server node

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

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

The node can contain these data types:

- HTML text pages, images, multimedia content for the client browser
- Servlets, JavaServer™ Pages™
- Enterprise beans
- Application program libraries, such as Java applets for dynamic download to client workstations

- ► Web server redirector node

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

- ► Application server node

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

- ► Integration server node

  The purpose of the *integration server node* is to interface between any front end access channel, such as the Web, a call center, or a client/server (*fat client*) PC, and any needed back-end application system. This can include applications from other companies. The integration server node performs the following kinds of services:

  - Convert protocols from the front end to match what the back-end systems understand.
  - Decompose a single message from the front end (such as a Web server) into several back-end messages (or transactions), and then recompose the replies.
  - Navigate from the front end to any back-end system that needs to be accessed.
  - In more complex cases, control the process or unit of work for a number of back-end interactions based on a request from the front end.

The intent is to relieve each front end from handling the complexity of interfacing with potentially multiple back-end systems which might be in different companies. The front end, such as the Web server, should just need to send a message to the integration server and have it look after the interface.

A second purpose for locating these interface services on the Integration server concerns security. There is a firewall between the Web server and the integration server. The Web server does not need to know about all the back-end addresses. Many locations do not want a server located in the DMZ to have access directly to sensitive data and systems. In this case, the Web server can only send messages to the integration server, nowhere else.

- Directory and security services node

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

- Protocol firewall node

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

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

  – Screening routers (the *protocol* firewall)
  – Application gateways (the *domain* firewall)

  A pair of firewall nodes provides increasing levels of protection at the expense of increasing computing resource requirements.

  The protocol firewall is typically implemented as an IP router.

- Domain firewall node

  The domain firewall is typically implemented as a dedicated server node. See "Protocol firewall node" on page 56 for a description of firewalls.

- Existing applications and data node

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

- Business service directory

  The role of the *business service directory* is to provide details of services that are available to perform business functions identified within a taxonomy. The business service directory can be implemented as an open-standard UDDI registry. Catalogs, such as a UDDI registry, can achieve one of the primary goals of a business service directory: to publish the availability of services and encourage their reuse across the development activity of an enterprise.

  The vision of Web services defines an open-standard UDDI registry that enables the dynamic discovery and invocation of business services. However although technologies mature toward that vision, more basic solutions are likely to be implemented in the near term.

- Enterprise service bus

  The *ESB* is a key enabler for a SOA as it provides the capability to route and transport service requests from the service requester to the correct service provider. The true value of the ESB concept, however, is to enable the infrastructure for SOA in a way that reflects the needs of today's enterprise: to provide suitable service levels and manageability, and to operate and integrate in a heterogeneous environment.

  Furthermore the ESB needs to be centrally managed and administered and have the ability to be physically distributed.

4.1.1 Why use an enterprise service bus?

Using an ESB node is key to designing an SOA solution. For that reason, we will go into the ESB node in more detail to help you understand the advantages.

Selecting a Runtime pattern that uses an ESB helps you achieve the following:

- Minimizes the number of adapters required for each point-to-point connection to link service consumers to service providers.
- Improves reuse in multiple point-to-point scenarios.

- Addresses any technical and information model discrepancies between services.
- Provides a single configuration point for distributed deployment.
- Decouples service requesters from providers
- Provides a common access point for service requesters
- Provides centralized security for services

The service bus can span across multiple system and application tiers, and can extend beyond the enterprise boundary.

Figure 4-1 shows a first level decomposition of the major components that make up an ESB node.

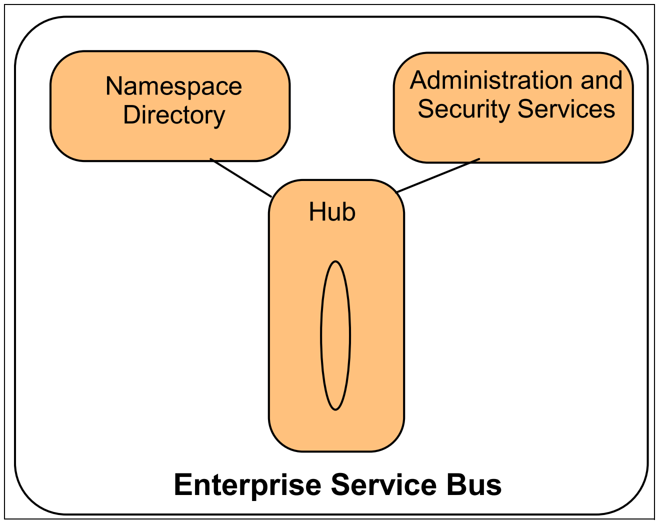


Figure 4-1 ESB runtime pattern: Level one

The ESB is a key enabler for an SOA because it provides the capability to route and transport service requests from the service consumer to the correct service provider. The ESB controls routing within the scope of a service namespace, indicated symbolically in Figure 4-1 by the oval on the ESB Hub.

The true value of the ESB concept, however, is to enable the infrastructure for SOA in a way that reflects the needs of today's enterprise: to provide suitable service levels and manageability and to operate and integrate a heterogeneous environment. Furthermore, the ESB needs to be centrally managed and administered and have the ability to be physically distributed.

The Hub, as shown in Figure 4-1 on page 58, supports the key ESB functions and, therefore, fulfills a large part of the ESB capabilities. The Hub has a fundamental service integration role and should be able to support various styles of interaction.

The following are the minimum set of functions that this node should support:

- Routing

  This function removes the need for applications to know anything about the bus topology or its traversal. The interaction that a requester initiates is sent to one provider.

- Addressing

  Addressing complements routing to provide location transparency and support service substitution. Service addresses are transparent to the service consumer and can be transformed by the hub. The hub obtains the service address from the namespace directory.

- Messaging styles

  The hub should support at least one or more messaging styles. The most common are request/response, fire and forget, events, publish/subscribe, and synchronous and asynchronous messaging.

- Transport protocols

  The hub should support at least one transport that is or can be made widely available, such as HTTP/S. The hub can provide protocol transformation. If a protocol transformation is required that is not supported by the hub, then a specific connector can be used to perform the transformation.

- Service interface definition

  Services should have a formal definition, ideally in an industry-standard format, such as WSDL.

- Service messaging model

  The hub should support at least one model such as SOAP, XML, or a proprietary EAI model.

A Namespace Directory component may be present to provide addressing information in order for the hub to properly forward requests.

An Administration and Security Services component provides a single point of administration that, at a minimum, should support service addressing and naming. The key services that need to be provided by this node are:

- ESB configuration
- Service provisioning and registration
- Logging

- Metering
- Monitoring
- Integration with systems management and administration tooling

More advanced administration features that can be provided by this node include self-monitoring and self-management.

The hub should support security capabilities such as authentication, authorization, non-repudiation, confidentiality, and security standards, such as Kerberos and WS-Security. The Administration and Security Services component provides the interface to the security services.

4.2 Runtime patterns for Directly Integrated Single Channel

The Directly Integrated Single Channel application pattern provides point-to-point connectivity between the user and the existing back-end applications.

The Runtime pattern for the Directly Integrated Single Channel application pattern has been broken down into a generic Runtime pattern and then further refined using an SOA profile.

4.2.1 Generic Runtime pattern for Directly Integrated Single Channel

The Runtime pattern shown in Figure 4-2 represents one solution for the Directly Integrated Single Channel application pattern. Based on the Enterprise Solution Structure (ESS) Thin Client Transactional pattern, this runtime is a starting point for extending business to the Web.

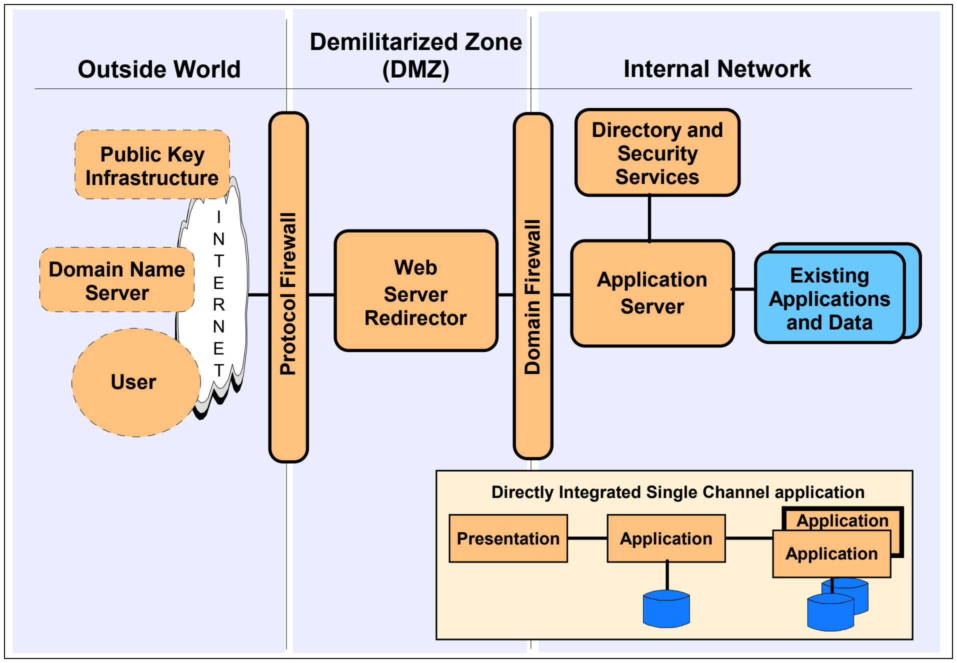


Figure 4-2 Directly Integrated Single Channel application pattern::Generic profile

The generic profile Runtime pattern uses a Web server redirector containing the Web server and an application server, effectively splitting the function of a Web application server across two machines. Placing the application server in the internal network provides a higher level of security than you might find in installations with one Web application server protected by a single layer of firewall security. The application server node will run both presentation and business logic. The Web server remains in the DMZ and serves static pages. The Web server redirector forwards requests from the Web server to the application server.

4.2.2 SOA profile for Directly Integrated Single Channel

The Runtime pattern for the Directly Integrated Single Channel application pattern can be refined as shown in Figure 4-3 to take advantage of service oriented architecture technology.

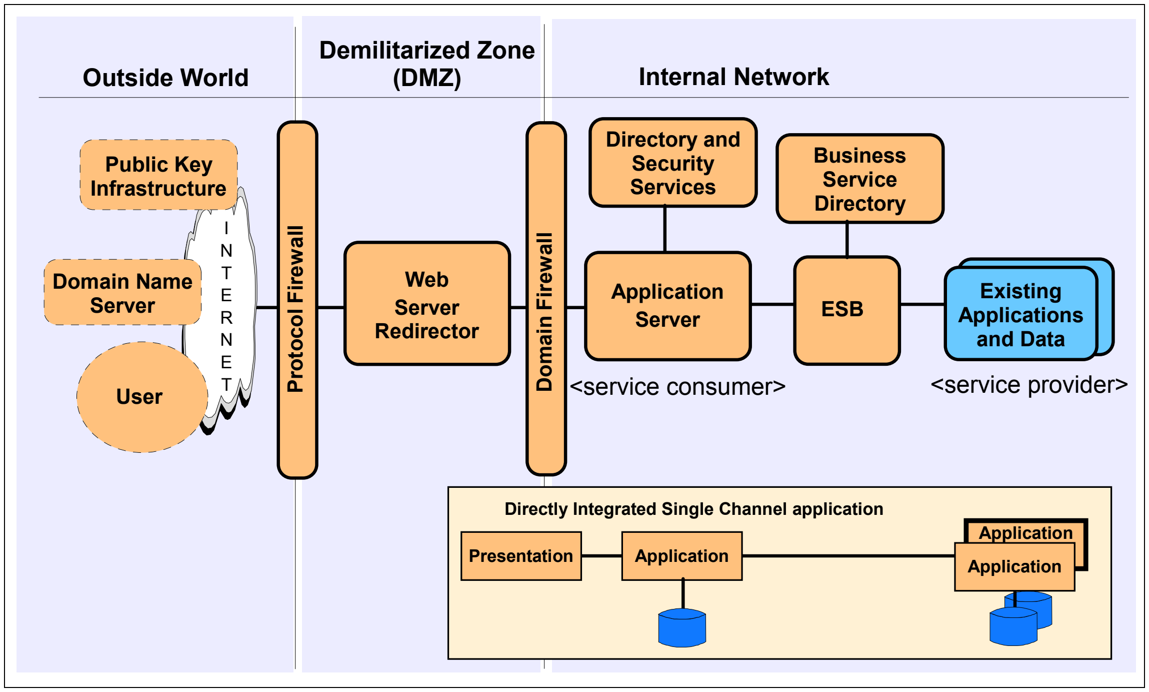


Figure 4-3 Directly Integrated Single Channel application pattern::Runtime pattern: SOA profile

In this SOA profile, the application server node becomes the service consumer with the back-end applications acting as service providers. The service consumer is connected to the service providers with a simple enterprise service bus. Due to the nature of the SOA approach, the consumer and provider could be reversed.

Implementing the SOA profile with an ESB adds extra capabilities to the runtime pattern, for example routing and decomposition capability. Because of this, the SOA profile for the Directly Integrated Single Channel runtime pattern can be applicable to multiple Self-Service application patterns. This highlights the fact that using SOA facilitates the future expansion of solution functionality without requiring major changes to the middleware structure.

4.3 Runtime patterns for Router

The Router application pattern provides intelligent routing from multiple channels to multiple back-end applications using a hub-and-spoke architecture. The interaction between the user and the back-end application is a one-to-one relation, meaning the user interacts with applications one at a time.

The Runtime pattern for the Router application pattern has been broken down into a generic Runtime pattern and then further refined using an SOA profile.

4.3.1 Generic Runtime pattern for Router

The Runtime pattern shown in Figure 4-4 represents one solution for the Router application pattern.

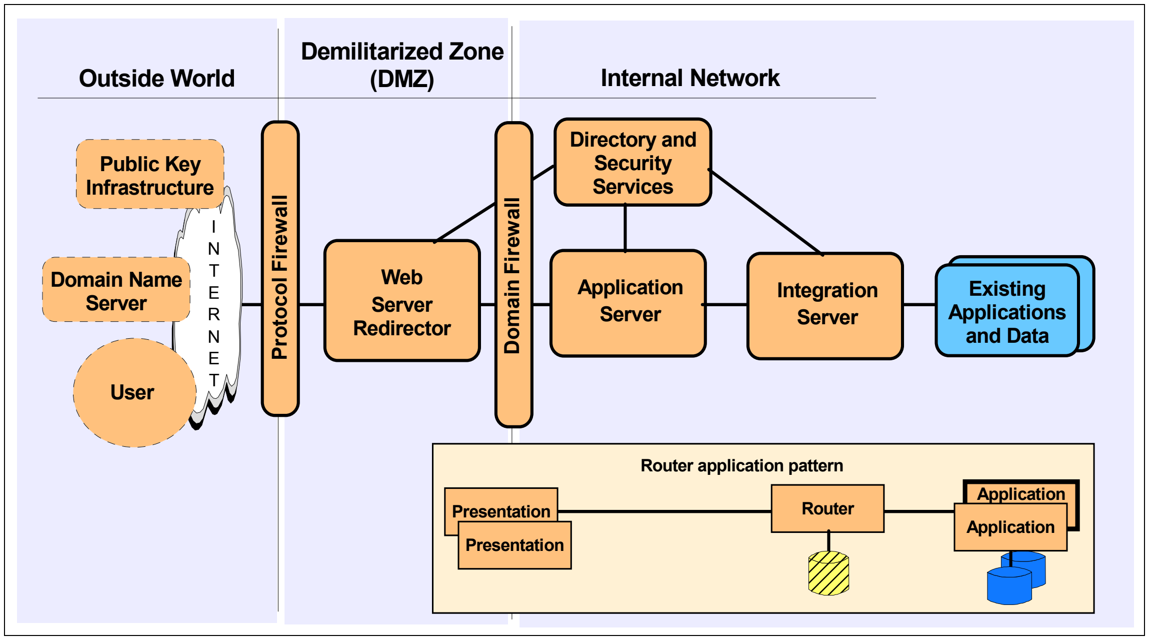


Figure 4-4 Router application pattern::Generic runtime pattern

This Runtime pattern uses a Web server redirector node in the DMZ to serve static HTML pages to the client. Requests for dynamic data are forwarded to the application server in the internal network. Together, these two nodes provide the presentation tier, capable of handling multiple, diverse, presentation styles. Using a redirector allows you to place the bulk of the business logic behind the protection of both the protocol and domain firewalls.

In addition to presentation logic, primarily in the form of JavaServer Pages (JSPs), the application server contains some business logic. This is primarily in the form of the controlling servlets required to access the back-end applications. The application server builds a request based on user input and passes it to the integration server.

The integration server examines the request, determines the appropriate destination, and forwards it to the chosen back-end application, where the primary business logic resides. This may involve activities such as message transformation, protocol conversion, security management, and session concentration. The integration server may use a database to look up routing information, as a caching device, or for holding intermediary data.

4.3.2 SOA profile for Router

The Runtime pattern for the Router application pattern can be refined as shown in Figure 4-5 to take advantage of service oriented architecture technology. In this SOA profile, the application server node becomes the service consumer with the back-end applications acting as service providers. The service consumer is connected to the service providers via a simple enterprise service bus.

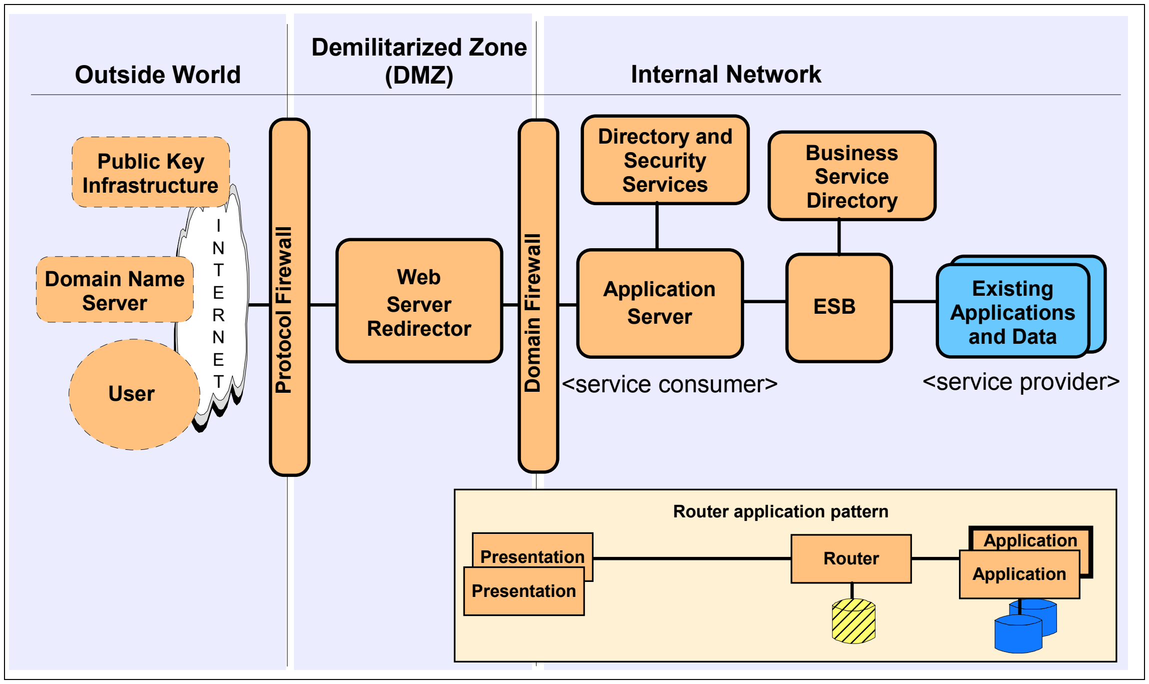


Figure 4-5 Router application pattern::Runtime pattern: SOA profile

In this pattern the ESB node has replaced the integration server node. It provides the message routing function, as well as protocol conversion, logging, transformation, or other tasks related to the proper routing of messages.

The ESB approach:

- Minimizes the number of adapters required to link service consumers to service providers.

- Improves reuse.
- Addresses any technical and information model discrepancies amongst services.
- Provides a single configuration point for distributed deployment.
- Decouples service requesters from providers
- Provides a common access point for service requesters
- Provides centralized security for services

The ESB can span across multiple system/application tiers, and may extend beyond the enterprise boundary.

Note that this Runtime pattern appears to be identical to the SOA profile of the Runtime pattern for the Directly Integrated Single Channel application pattern. The use of SOA and an ESB introduces the flexibility to accommodate a wide range of functionality, including point-to-point, routing, and as we will see later, decomposition capability.

4.4 Runtime patterns for Decomposition

The Decomposition application pattern expands on the Router application pattern, providing all the features and functions of that pattern and adding recomposition/decomposition. This capability allows an incoming request to be split into multiple requests directed to separate back-end applications. The results of these requests is recombined into a single response to the user.

The Runtime pattern for the Decomposition application pattern has been broken down into a generic Runtime pattern and then further refined using an SOA profile.

4.4.1 Generic Runtime pattern for Decomposition

The Runtime pattern shown in Figure 4-6 on page 66 represents one solution for the Decomposition application pattern.

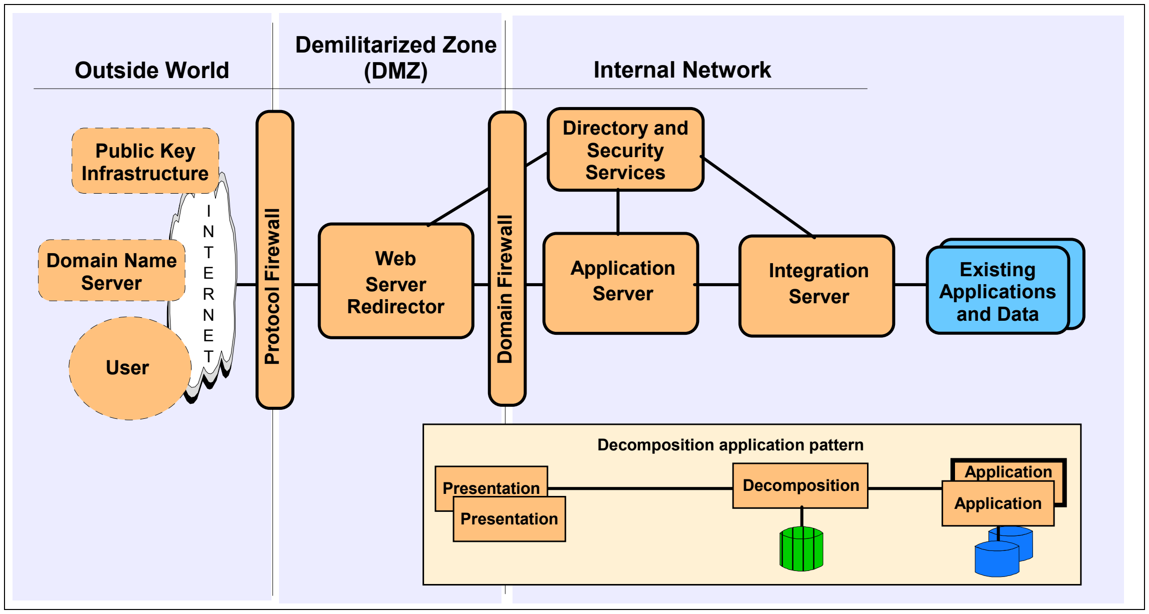


Figure 4-6 Decomposition application pattern::Generic runtime pattern

In the Decomposition application pattern, the decomposition tier serves as an integration point for delivery channels in the presentation tier, allowing access to individual back-end applications. In the Generic runtime pattern (Figure 4-6 on page 66), the functions of the decomposition tier are performed by an integration server node. The functions of the presentation tier are performed jointly by a Web server redirector node and the application server node. Placing a Web server redirector in the DMZ provides an extra layer of security by putting all application logic behind the firewall. Only a portion of the presentation function is left in the DMZ.

The Web server redirector serves static HTTP pages, while forwarding dynamic servlet and JSP requests to the application server. The presentation logic, therefore, spans both nodes. Together, these two provide the presentation tier, capable of handling multiple, diverse presentation styles. Using a redirector allows you to place the bulk of the business logic behind the protection of both the protocol and domain firewalls.

In addition to presentation logic (for example, JSPs), the application server contains some business logic. This is primarily in the form of the controlling servlets required to access the back-end applications. The application server builds a request based on user input and passes it to the integration server node. The primary business logic resides in the back-end applications.

The integration server examines messages and routes them to the appropriate back-end applications. It can go a step further by taking a single complex message, decomposing it into multiple messages, and routing those messages to the appropriate back-end applications. It is also capable of managing these messages such that it can wait for responses and recompose them into a single response to be sent back to the user. This effectively takes multiple, diverse back-end applications and unifies them into one interface for the user.

The integration server can use a local database as a work-in-progress database to store information required for message decomposition and recomposition.

4.4.2 SOA profile for Decomposition

The Runtime pattern for the Decomposition application pattern can be refined as shown in Figure 4-7 to take advantage of service oriented architecture technology. In this SOA profile, the application server node becomes the service consumer with the back-end applications acting as service providers. The service consumer is connected to the service providers via a simple enterprise service bus.

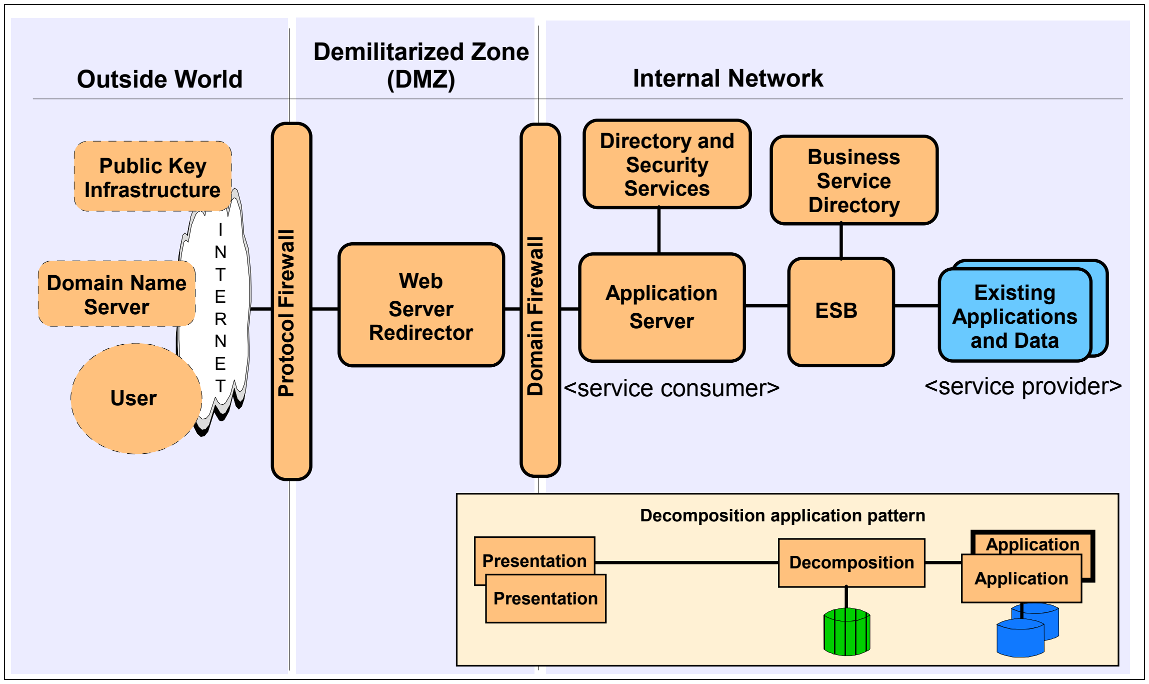


Figure 4-7 Decomposition application pattern::Runtime pattern:SOA profile

The functionality related to handling messages including decomposition/recomposition, routing, transformation, logging, and protocol conversion moves to the ESB.

The ESB approach:

- Minimizes the number of adapters required to link service consumers to service providers.
- Improves reuse.
- Addresses any technical and information model discrepancies amongst services.
- Provides a single configuration point for distributed deployment.
- Decouples service requesters from providers
- Provides a common access point for service requesters
- Provides centralized security for services

Note that this Runtime pattern appears to be identical to the SOA profile of the Runtime patterns for the Directly Integrated Single Channel and Router application patterns. The use of SOA and an ESB introduces the flexibility to accommodate a wide range of functionality, including point-to-point, routing, and decomposition capability.

5

Product mappings and product overview

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

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

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

Our sample application has been implemented using IBM WebSphere Application Server V6 in the Microsoft Windows 2000 environment.

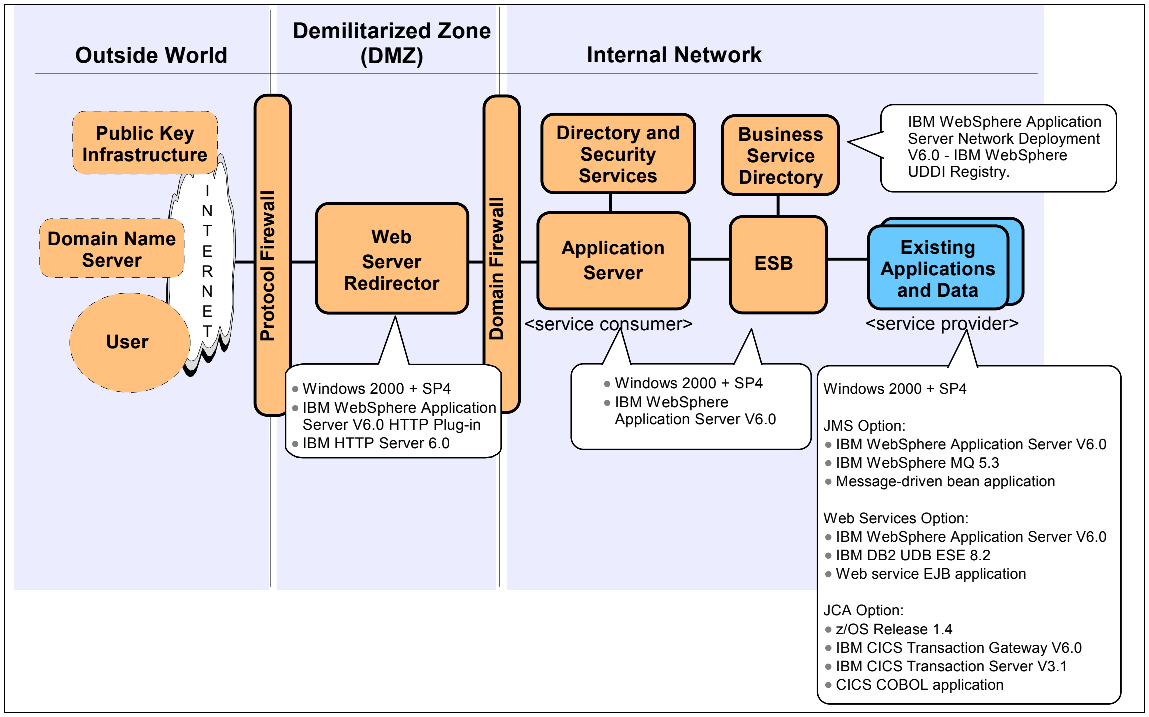
This chapter introduces the major products used in the application and provides an overview of the products as they apply to the selected SOA profile Runtime patterns.

5.1 Product mapping

Figure 5-1 shows a Product mapping based on the Windows 2000 operating system platform and the SOA profile of the Runtime patterns described in:

- "Runtime patterns for Directly Integrated Single Channel" on page 60.
- "Runtime patterns for Router" on page 62
- "Runtime patterns for Decomposition" on page 65

The SOA profile of each of these Runtime patterns uses the same nodes. As we pointed out earlier, the use of SOA and an ESB provides a wide range of functionality. The differentiator between the Runtime patterns is the functionality implemented within the ESB. Our selection of products (see Figure 5-1) was such that the same product set could be used for the implementation of each of the Runtime patterns.



Outside World
Demilitarized Zone (DMZ)
Internal Network
Public Key Infrastructure
Domain Name Server
User
INTERNET
Protocol Firewall
Web Server Redirector
Domain Firewall
Directory and Security Services
Business Service Directory
Application Server
ESB
Existing Applications and Data
<service consumer>
<service provider>
IBM WebSphere Application Server Network Deployment V6.0 - IBM WebSphere UDDI Registry.
• Windows 2000 + SP4
• IBM WebSphere Application Server V6.0 HTTP Plug-in
• IBM HTTP Server 6.0
• Windows 2000 + SP4
• IBM WebSphere Application Server V6.0
Windows 2000 + SP4
JMS Option:
• IBM WebSphere Application Server V6.0
• IBM WebSphere MQ 5.3
• Message-driven bean application
Web Services Option:
• IBM WebSphere Application Server V6.0
• IBM DB2 UDB ESE 8.2
• Web service EJB application
JCA Option:
• z/OS Release 1.4
• IBM CICS Transaction Gateway V6.0
• IBM CICS Transaction Server V3.1
• CICS COBOL application

Figure 5-1 Windows 2000 Product mapping

This product mapping implements the ESB using the service integration technology in WebSphere Application Server. The service integration bus provides the service endpoints and queue destinations used by the consumers to access the providers. The service integration bus has the ability to mediate messages as they pass through the bus. Mediation can be used to affect the processing of the message, to translate the format, change the destination, log information, or other actions before the message reaches the service provider. Use of the bus provides a central management location that insulates changes in the service providers from the service consumers.

The following alternatives are shown for the implementation of the services provided by the back-end enterprise applications:

- Web services using IBM WebSphere Application Server V6

  The SOA approach in this case is implemented using Web services. The application server acts as the service consumer and the back-end application as the service provider. The service integration bus provides the Web service endpoint for the service consumer.

- J2EE Connectors using IBM CICS

  The SOA approach in this case is also implemented using Web services. The service provider uses a CICS ECI J2EE Connector adapter to access the existing CICS enterprise application using the CICS Transaction Gateway. The functionality that invokes the CICS application is wrapped as a Web service.

- Java Message Service (JMS) using the WebSphere Application Server V6 default messaging provider

  The SOA approach is implemented using JMS messaging. The implementation shown uses the default messaging provider integrated in WebSphere Application Server V6. This is acceptable because the enterprise application receiving the messages is a WebSphere application on an application server connected to the same bus. For more advance messaging environments, the use of WebSphere MQ, or the default messaging provider in conjunction with WebSphere MQ are better options. In either case, the Java application uses JMS to place messages on a local queue. The messaging provider is then responsible for ensured delivery of this message to the proper destination.

  The service integration bus provides the communication infrastructure for the default messaging provider.

By using a Web server redirector node, we can place the majority of the business logic in the internal network, placing it behind two firewalls. The redirector is implemented using the IBM HTTP Server and WebSphere Application Server Web server plug-in. The redirector serves static HTML pages and forwards requests for dynamic content to a WebSphere application server using the HTTP protocol.

The network protocols used are:

- HTTP/HTTPS

  Hypertext Transfer Protocol (HTTP V1.1), or Hypertext Transfer Protocol Secure (HTTPS - HTTP V1.1/SSL V3), is used from the user's Web browser to the HTTP server in the Web server redirector node.

  HTTP (or HTTPS) is also used from the WebSphere Web server plug-in in the Web server redirector node to the Web container in the Application server node.

- LDAP

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

- JDBC

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

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

  – SOAP/HTTP

    Simple Object Access Protocol (SOAP 1.1) and HTTP V1.1 are used between the Web services client in the application server node and the Web service provider in the existing enterprise tier.

    Message data is passed using XML V1.0 with UTF-8 encoded character strings. Messages are validated using the message's XML schema definition.

  – CICS Transaction Gateway TCP

    The proprietary CICS Transaction Gateway V6.0 TCP protocol is used from the J2C resource adapter in the application server to the CICS Transaction Gateway in the existing enterprise tier.

    Message data is passed using a byte array representing the CICS COMMAREA. Character data can flow in ASCII or EBCDIC, or it could be binary.

  – WebSphere MQ and the WebSphere Application Server default messaging provider:

    The proprietary WebSphere MQ protocol is used from the messaging provider on the application server node to the messaging provider in the existing enterprise tier.

    Message data is passed using XML V1.0 with UTF-8 encoded character strings. Messages are validated using the message's XML schema definition.

5.2 IBM WebSphere Application Server

The IBM WebSphere Application Servers are a suite of servers that implement the J2EE specification. This simply means that any Web applications that are written to the J2EE specification can be installed and deployed on any of the servers in the WebSphere Application Server family.

The primary component of the WebSphere Application Server products is the application server, which provides the environment to run your Web-enabled e-business applications. You can think of an application server as *Web middleware*, the middle tier in a three-tier, e-business environment. The first tier is the Web server that handles requests from the browser client. The third tier is the business database, for example DB2 UDB, and the business logic, for example, traditional business applications such as order processing. The middle tier is IBM WebSphere Application Server, which provides a framework for consistent, architected linkage between the HTTP requests and the business data and logic.

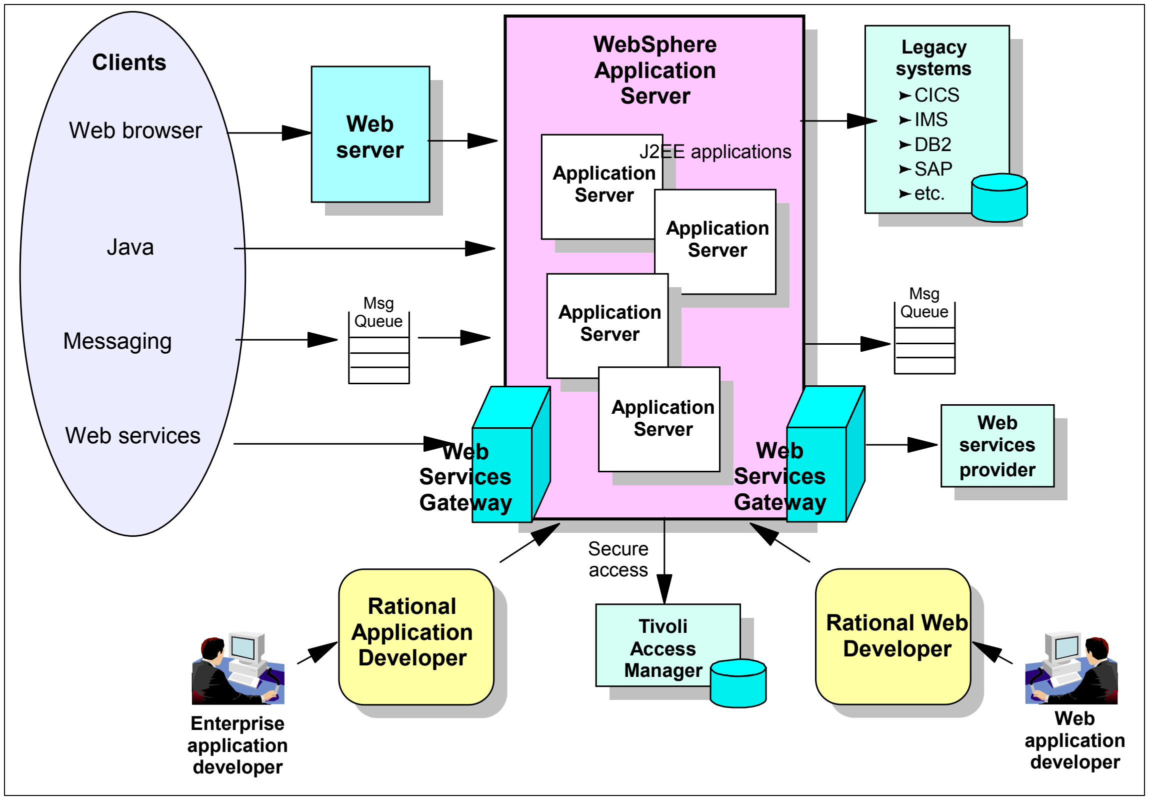


Figure 5-2 WebSphere Application Server product overview

WebSphere Application Servers are available in multiple packages to meet specific business needs. They also serve as the base for other WebSphere products, such as WebSphere Commerce, by providing the application server required for running these specialized applications.

WebSphere Application Servers are available on a wide range of platforms, including UNIX®-based platforms, Microsoft operating systems, IBM z/OS®, and IBM @server® iSeries.

5.2.1 WebSphere Application Server V6 for distributed platforms

The latest product to be announced in the WebSphere Application Server family is IBM WebSphere Application Server V6. It features:

- Full J2EE 1.4 support
- High-performance connectors to many common back-end systems, reducing the coding effort required to link dynamic Web pages to real line-of-business data.
- Application services for session and state management
- Web services

  Web services enable businesses to connect applications to other business applications, to deliver business functions to a broader set of clients and partners, to interact with marketplaces more efficiently, and to create new business models dynamically.
- A fully integrated JMS 1.1 messaging provider

  This messaging provider complements and extends WebSphere MQ and application server. It is suitable for messaging among application servers and for providing messaging capability between WebSphere Application Server and an existing WebSphere MQ backbone.
- Many of the programming model extensions previously found in WebSphere Business Integration Server Foundation

Because varying e-business application scenarios require different levels of application server capabilities, WebSphere Application Server is available in multiple packaging options. Although they share a common foundation, each option provides unique benefits to meet the needs of applications and the infrastructure that supports them. At least one WebSphere Application Server product package fulfills the requirements of any particular project and the prerequisites of the infrastructure that supports it. As your business grows, the WebSphere Application Server family provides a migration path to higher configurations.

WebSphere Application Server - Express V6

The Express package is geared to those who need to get started quickly with e-business. It is specifically targeted at medium-sized businesses or departments of a large corporation, and is focused on providing ease of use and ease of application development. It contains full J2EE 1.4 support but is limited to a single-server environment.

WebSphere Application Server - Express is unique from the other packages in that it is bundled with an application development tool. Although there are WebSphere Studio and Rational Developer products designed to support each WebSphere Application Server package, normally they are ordered independent of the server. WebSphere Application Server - Express includes the Rational Web Developer application development tool. It provides a development environment geared toward Web developers and includes support for most J2EE 1.4 features with the exception of Enterprise JavaBeans™ (EJB™) and J2EE Connector Architecture (JCA) development environments. However, keep in mind that WebSphere Application Server - Express V6 does contain full support for EJB and JCA, so you can deploy applications that use these technologies.

WebSphere Application Server V6

The WebSphere Application Server package is the next level of server infrastructure in the WebSphere Application Server family. Though the WebSphere Application Server is functionally equivalent to that shipped with Express, this package differs slightly in packaging and licensing. The development tool included is a trial version of Rational Application Developer, the full J2EE 1.4 compliant development tool.

WebSphere Application Server Network Deployment V6

WebSphere Application Server Network Deployment is an even higher level of server infrastructure in the WebSphere Application Server family. It extends the WebSphere Application Server base package to include clustering capabilities, Edge components, and high availability for distributed configurations. These features become more important at larger enterprises, where applications tend to service a larger customer base, and more elaborate performance and availability requirements are in place.

Application servers in a cluster can reside on the same or multiple machines. A Web server plug-in installed in the Web server can distribute work among clustered application servers. In turn, Web containers running servlets and Java ServerPages (JSPs) can distribute requests for EJBs among EJB containers in a cluster.

The addition of Edge components provides high performance and high availability features. For example:

- The Caching Proxy intercepts data requests from a client, retrieves the requested information from the application servers, and delivers that content back to the client. It stores cacheable content in a local cache before delivering it to the client. Subsequent requests for the same content are served from the local cache, which is much faster and reduces the network and application server load.
- The Load Balancer provides horizontal scalability by dispatching HTTP requests among several, identically configured Web server or application server nodes.

5.2.2 Service integration

The service integration functionality within WebSphere Application Server V6 provides the infrastructure to support both message-oriented and service-oriented applications. This new functionality is based on the concept of the service integration bus, or simply, the bus.

The bus provides advanced support for application integration. It combines support for applications connecting through native JMS, WebSphere MQ JMS, WebSphere MQ, and Web services. It supports the message-oriented middleware and request-response interaction models. As a part of this, the service integration bus supports multiple message distribution models, reliability options, and transactional messaging.

Figure 5-3 on page 77 gives you a high-level view of the bus functionality.

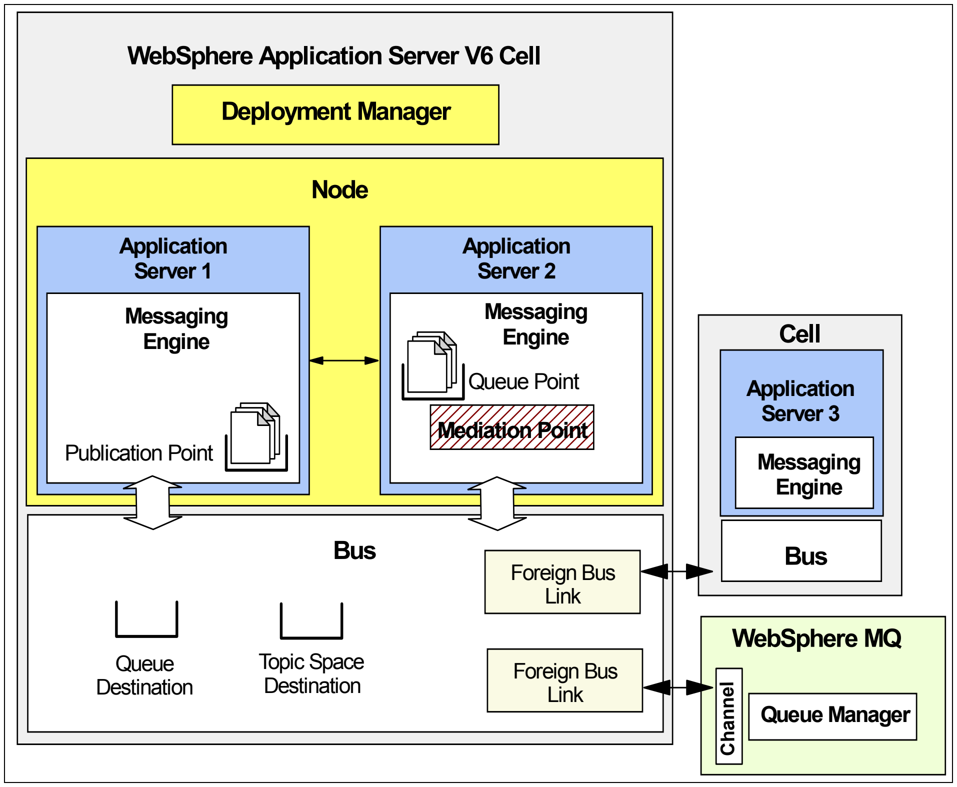


Figure 5-3 Service integration bus

- Bus

  A service integration bus, or bus, provides a conceptual connection point for destinations and services. The application integration capabilities of the service integration bus are provided by a number of connected messaging engines.

- Messaging engine

  While the conceptual entity clients connect to is the bus, the physical connection is to a messaging engine. A *messaging engine* manages bus resources and provides the connection point for applications. Each messaging engine is associated with a server or cluster that is a member of the bus.

  A messaging engine manages messages by routing them to the appropriate endpoint, through additional messaging engines if required. These messages can be persisted to a database and managed within a transactional scope.

Clients can connect into any messaging engine in the bus and send messages to it. If the destination is assigned to a different messaging engine, the messaging engine will route it to the correct messaging engine.

- Destination

  A *destination* is an addressing point within a bus. A destination is assigned to one bus member and, therefore, one or more messaging engines. Clients send messages to a destination and the bus ensures that it is routed to the correct localization on the bus. The following destination types are supported by the service integration bus:

  – Web service destinations

    Web service destinations are a representation of an outbound Web service in the bus. They are used as a placeholder for a port selection mediation.

  – Port destinations

    Port destinations are a representation of an outbound Web service port. Sending a Web service request to a port destination will result in the target Web service being invoked.

  – Queue destinations

    Queue destinations are destinations that are configured for point-to-point messaging.

  – Topic space destinations

    Topic space destinations are destinations that are configured for publish/subscribe messaging.

  – Alias destinations

    Alias destinations are destinations that are configured to refer to another destination. They provide an extra level of indirection for messaging applications. An alias destination can also be used to override some of the values specified on the target destination, such as default reliability and maximum reliability. An alias destination can also refer to a destination on a foreign bus. Foreign buses are discussed in “Foreign bus link” on page 79.

  – Foreign destinations

    Foreign destinations are not actual destinations within a service integration bus, but they can be used override the default reliability and maximum reliability properties of a destination that exists on a foreign bus. Foreign buses are discussed in “Foreign bus link” on page 79.

  Destinations can be mediated to provide advanced message formatting and routing function.

- Message point

  When a destination is assigned to a bus member, a *message point* is created. The messages are stored on the message point. The following are the types of message point that can be contained with a messaging engine:

  - A *Queue point* is the message point for a queue destination,
  - A *Publication point* is the message point for a topic space.

    Creating a topic space destination automatically defines a publication point for each messaging engine within the bus.
  - *Mediation points*, are where messages are stored while they wait to be mediated. A mediated destination also has mediation points.
- Mediation

  A *mediation* processes in-flight messages between the production of a message by one application and the consumption of a message by another application. Mediations enable the messaging behavior of a bus to be customized. Examples of the processing that can be performed by a mediation are:

  - Transforming a message from one format into another
  - Dynamically routing messages to one or more target destinations that were not specified by the sending application
  - Augmenting messages by adding data from a data source
  - Disaggregation of a request into several requests and then aggregation of the responses

  When you configure a mediation for use at a particular destination, the physical location is called a *mediation point*.
- Foreign bus link

  A bus can be configured to connect to and exchange messages with other messaging networks. A *foreign bus* is how the service integration bus refers to one of these networks.

  A foreign bus encapsulates information related to the remote messaging network, such as the type of the foreign bus and whether messaging applications are allowed to send messages to the foreign bus. When buses are interconnected, applications can send messages to destinations defined on other buses.

  When a foreign bus is configured on a bus, it simply names a foreign bus. It does not define a link between the two. In order for the two buses to be able to communicate with each other at runtime, links must be configured between a specific messaging engine within the local bus and a specific messaging engine, or queue manager, within the foreign bus. When configuring a direct service integration bus link, these links must be configured in both directions in order for the two buses to be able to communicate. At runtime, messages that are routed to a foreign bus will flow across the corresponding link.

5.2.3 ESB capabilities

WebSphere Application Server V6 provides several runtime features that support ESB capabilities. It has support for Web services standards and for programming models that enable data and message manipulation. Development tools for WebSphere Application Server, such as Rational Application Developer, include tools and wizards that simplify the development of application, framework, or infrastructure code to leverage those runtime features.

WebSphere Application Server provides, through the service integration bus component, communication infrastructure for messaging and Web services applications that enables it to support the communication and message processing requirements of an ESB. WebSphere Application Server and tools also provide support for a wide variety of integration methods, either directly (databases, J2EE connectors, and so forth) or through support for Enterprise Application Integration middleware (such as WebSphere MQ).

WebSphere Application Server V6 meets the ESB capabilities that the following sections describe.

Communication

WebSphere Application Server supports all of the minimum and extended capabilites that we have defined for communication, including support for:

- SOAP-based Web service hosting and invocation
- Asynchronous messaging

  The support is provided by the service integration bus component that provides a JMS V1.1 compliant JMS provider for reliable message transport.
- Synchronous messaging with HTTP and HTTPS transports
- Point-to-point, request/response, fire and forget, events, and publish/subscribe styles of messaging
- Routing support that allows:
  - Dynamic service and port selection
  - Web service requests converted from one WSDL definition to another
  - Internet routing with proxy
- WSDL as the service interface definition and the service implementation definition

  WSDL can publish services to a UDDI directory.

Integration

WebSphere Application Server supports all of the minimum and extended capabilites that we have defined for integration, including support for:

- JDBC used for connectivity to external data sources, for example, a relational database.
- Protocol transformation capability comes in the form of the service integration bus supporting transformation from SOAP/HTTP to SOAP/JMS and vice versa.
- Existing software and application adapters can be incorporated into the system by implementing the J2EE Connector Architecture support for connecting the J2EE platform to heterogeneous Enterprise Information Systems (EIS). Examples include ERP, mainframe transaction processing, database systems, and existing applications not written in the Java programming language.
- Connectivity to enterprise application middleware

  The service integration bus is tightly integrated with WebSphere MQ. Connections can be defined so that WebSphere MQ queue managers view a service integration bus as a queue manager, and so the service integration bus views queue managers as another bus. The JMS support means messages can be exchanged with any other JMS V1.1 compliant provider.
- Data enrichment of services messages within the ESB

  The service integration bus provides mediation support that allows processing of in-flight messages. Examples of the processing that can be performed by a mediation are transforming a message from one format into another, routing messages to one or more target destinations that were not specified by the sending application, augmenting messages by adding data from a data source, and distributing messages to multiple target destinations.
- WebSphere Application Server is a fully J2EE V1.4 compliant application server.
- Language interfaces for service invocation

  WebSphere Application Server supports Java interfaces. It provides Web service support so that it can act as both a Web service provider and as a consumer. As a provider, it hosts Web services that are published for use by clients. As a consumer, it hosts applications that invoke Web services from other locations

Security

WebSphere Application Server supports some of the extended capabilites that we have defined for security, including support for:

- Tokens, keys, signatures, and encryption according to the WS-Security specification can be applied to every deployed Web service.
- Authentication and authorization as part of J2EE.
- HTTPS
- Enabled proxy authentication
- Message-level security, as part of the WS-Security specification, and implemented using JAX-RPC

Message processing

WebSphere Application Server supports some of the extended capabilites that we have defined for message processing. It provides:

- Content based logic support

  The mediation support in the service integration bus allows messages to be routed and altered based on content.
- Message and data transformation support using mediations in the service integration bu.
- Message aggregation and correlation support

  The mediation framework requires custom Java coding to perform aggregation and correlation.
- Validation supported only through mediation support

  This means validation must be coded instead of configured.
- Intermediary support

  The service integration bus allows WebSphere Application server to act as an intermediary.
- Store and forward support

Modeling

WebSphere Application Server has limited support for the extended capabilites that we have defined for modeling.

Service interaction

WebSphere Application Server supports all of the minimum and some of the extended capabilites that we have defined for service interaction. It provides:

- WSDL support for the service interface definition and the service implementation definition
- Service directory and discovery support

- Substitution of service implementation

  Using WebSphere Application Server as an ESB means service implementations can be substituted without the service consumer needing to be aware of the change.

Quality of service

WebSphere Application Server supports some of the extended capabilites that we have defined for quality of service. It provides:

- Assured delivery support

  The service integration bus supports five levels of message reliability and persistence. The integration with WebSphere MQ means that it can also use the assured delivery features of WebSphere MQ.

- Transaction support

  WebSphere Application Server can act as an XA compliant transaction manager or as a participant in transactions controlled by another transaction controller.

Service level

WebSphere Application Server supports some of the extended capabilites that we have defined for service level. It provides:

- Performance tuning and monitoring tools

  In particular, Web service performance can be monitored through the Performance Monitoring Infrastructure (PMI), including the number of asynchronous and synchronous requests and responses.

- High availability

  WebSphere Application Server Network Deployment provides a number of facilities for provide high availability across all components of the WebSphere Application Server environment.

Management and autonomic systems

WebSphere Application Server supports all of the minimum and some of the extended capabilites that we have defined for management and autonomic systems. It provides:

- Administration tools and support
- Provision for service provision and registration
- Integration to system management and administration tooling, in particular IBM Tivoli® products

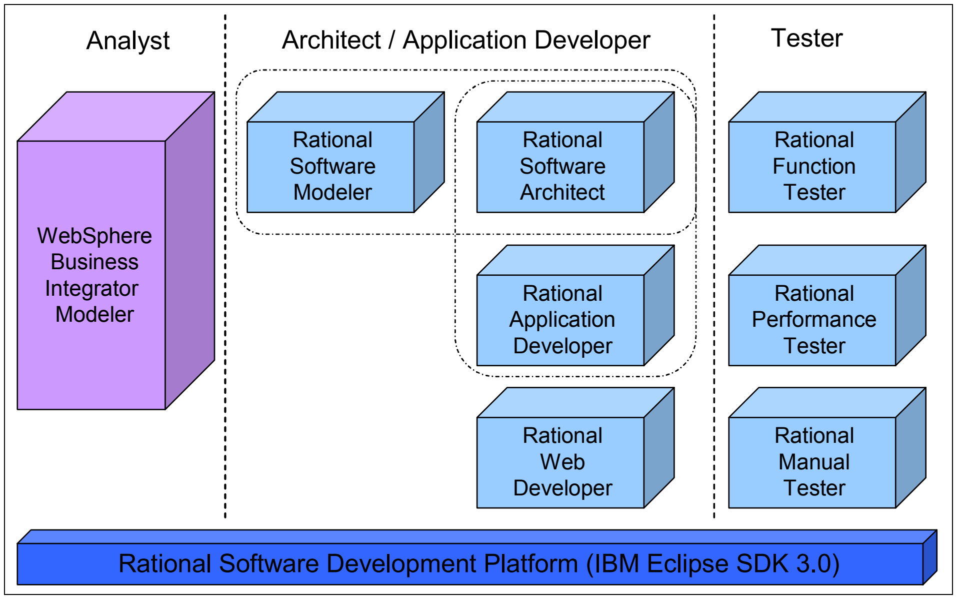
Infrastructure intelligence

WebSphere Application Server has limited support for the extended capabilites that we have defined for infrastructure intelligence.

5.3 IBM Rational Software Development Platform

The IBM Rational Software Development Platform is not a single product, but rather an integrated set of products which share a common technology platform built on the Eclipse 3.0 framework in support of each phase of the development life cycle.

The IBM Rational Software Development Platform provides a team-based environment with capabilities that are optimized for the key development team roles, including: business analyst, architect, developer, tester, and deployment manager. It enables a high degree of team cohesion through shared access to common requirements, test results, software assets, and workflow and process guidance. Combined, these capabilities improve both individual and team productivity.



Analyst
Architect / Application Developer
Tester
WebSphere Business Integrator Modeler
Rational Software Modeler
Rational Software Architect
Rational Application Developer
Rational Web Developer
Rational Function Tester
Rational Performance Tester
Rational Manual Tester
Rational Software Development Platform (IBM Eclipse SDK 3.0)

Figure 5-4 Rational Software Development Platform products

We have included a brief description of each of the products included in the IBM Rational Software Development Platform (see Figure 5-4) that share common tooling based on the IBM Eclipse SDK V3.0 (IBM supported Eclipse 3.0):

- Rational Software Modeler

  The Software Modeler is a UML-based visual modeling and design tool for system analysts, software architects, and designers who need to clearly define and communicate their architectural specifications to stakeholders.

  This product was known in previous releases as Rational XDE™ Modeler and is targeted at development shops where the business analyst has the distinct role of architecture and design but no development.

- Rational Software Architect

  The Software Architect is a design and construction tool that leverages model-driven development with UML for creating well-architected applications, including those based on a Service-Oriented Architecture (SOA). It unifies modeling, Java structural review, Web Services, J2SE™, J2EE, database, XML, Web development, and process guidance for architects and senior developers creating applications in Java or C++.

  This product was known in previous releases as Rational Rose® and Rational XDE for Java. Software Architect includes architecture and design capability as well as full J2EE development functionality provided by Rational Application Developer. This product is targeted at development shops where the architect has a strong architecture and design role as well as application development. If architects only need the modeling functionality, they should use the Rational Software Modeler product.

  Note: IBM Rational Software Architect V6.0 is used to model and develop the ITSOMart application.

- Rational Web Developer

  The IBM Rational Web Developer is targeted to the Web application developer and provides visual tools for Web, Java, and rich client applications, and full support for XML, Web services, and Enterprise Generation Language.

  In previous releases this product was known as WebSphere Studio Site Developer. Rational Web Developer is packaged with IBM WebSphere Application Server Express V6.0.

- Rational Application Developer

  The IBM Rational Application Developer is a full suite of development, analysis, and deployment tools for rapidly implementing J2EE applications, Enterprise JavaBeans, portlets, and Web applications.

  In previous releases this product was known as WebSphere Studio Application Developer and is targeted at J2EE developers.

- Rational Functional Tester

  The Rational Function Tester is an automated testing tool for Java, HTML, VB.NET, and Windows applications. It provides the capability to record robust scripts that can be played back to validate new builds of an application.

- Rational Performance Tester

  The Rational Performance Tester is a multi-user, system-performance tester designed to test Web applications, focusing on ease-of-use and scalability.

- WebSphere Business Integrator Modeler

  The WebSphere Business Integrator Modeler does not carry the Rational brand name, but is an important product of the Rational Software Development Platform. WebSphere Business Integrator Modeler targets the business analyst who models business processes. WebSphere Business Integrator Modeler can be used to generate Business Process Execution Language (BPEL) definitions to be deployed to WebSphere Business Integrator production environments. The WebSphere Business Integrator Modeler BPEL provides a more seamless move to production and eliminates the need to create Visio® diagrams and then move to production.

5.3.1 Workbench

An *integrated development environment* (IDE) is a set of software development tools such as source editors, compilers, and debuggers, that are accessible from a single user interface. In Rational Software Development Platform, the IDE is called the *Workbench*. Rational Application Developer's Workbench provides customizable perspectives that support role-based development and a common way for all members of a project team to create, manage, and navigate resources easily. It consists of a number of interrelated views and editors.

Views provide different ways of looking at the resource on which you are working, while *editors* allow you to create and modify the resource. Perspectives are a combination of views and editors that show various aspects of the project resource, and are organized by developer role or task. For example, a Java developer would work most often in the Java perspective, while a Web designer would work in the Web perspective.

Several perspectives are provided in Rational Software Development Platform. Team members can customize them, according to their current role or preference. You can open more than one perspective at a time, and switch perspectives while you are working with Rational Software Development Platform. If you find that a particular perspective does not contain the views or editors you require, you can add them to the perspective and position them to suit your preference.

Perspectives

Perspectives provide a convenient grouping of views and editors which match the way you use Rational Software Development Platform. Depending on the role you are in and the task you preform, you open a different perspective. A perspective defines an initial set and layout of views and editors for performing a particular set of development activities (for example, EJB development or profiling). You can change the layout and the preferences, then save the customized perspective so that you can open it again later.

Views

Views provide different presentations of resources or ways of navigating through the information in your workspace. For example, the Navigator view displays projects and other resources that you are working as a folder hierarchy, like a file system view. Rational Application Developer provides synchronization between different views and editors. Some views display information obtained from other software products, such as database systems or software configuration management (SCM) systems.

A view might appear by itself, or stacked with other views in a tabbed notebook arrangement. A perspective determines the initial set of views that you are likely to need. For example, the Java perspective includes the Package Explorer and the Hierarchy views to help you work with Java packages and hierarchies.

Editors

When you open a file, Rational Software Development Platform automatically opens the editor that is associated with that file type. For example, the Page Designer editor is opened for .html, .htm and .jsp files, while the Java editor is opened for .java and .jpage files.

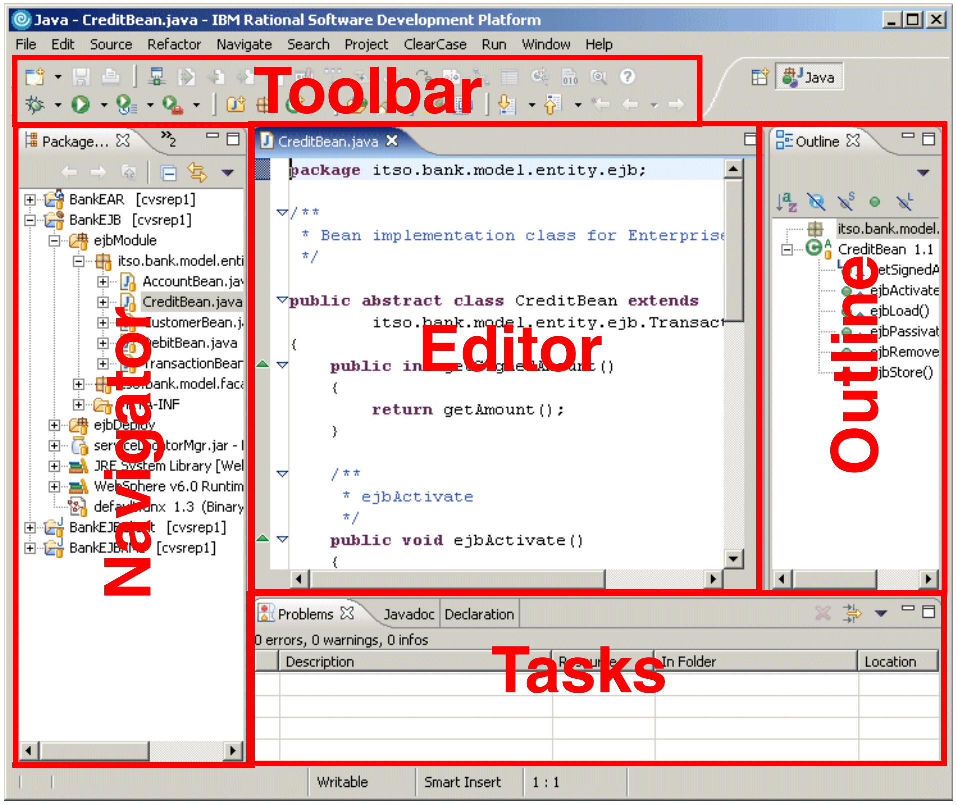
Editors that have been associated with specific file types open in the editor area of the Workbench. By default, editors are stacked in a notebook arrangement inside the editor area. If there is no associated editor for a resource, Rational Application Developer will open the file in the default editor, which is a text editor.

Wizards

There are a significant number of wizards available that lead you through the process of creating almost any type of file, project, or artifact that you will need. The wizards take you through a series of input panels where you receive guidance in selecting and entering values to create the new item. The wizard process provides many defaults, making the process easy. The results include the new files and automatic updates to affected files. For example, the wizard that creates a new dynamic Web project creates the Web project, its file structure and deployment descriptors. The deployment descriptor for the EAR file it belongs to is also updated to reflect the new module.

Perspective layout

Many of Rational Software Development Platform's perspectives use a similar layout. Figure 5-5 shows a layout of a perspective which is quite common.



```
package itso.bank.model.entity.ejb;

/**
 * Bean implementation class for Enterprise
 */
public abstract class CreditBean extends
        itso.bank.model.entity.ejb.Transact
{
    public int getSignedAmount()
    {
        return getAmount();
    }

    /**
     * ejbActivate
     */
    public void ejbActivate()
    {
```

0 errors, 0 warnings, 0 infos

Figure 5-5 Perspective layout

On the left side are views for navigating through the workspace. In the middle of the Workbench is larger pane, usually the source editor or the design pane, which allows you to change the code and design of files in your project. The right pane usually contains outline or properties views. In some perspectives, the editor pane is larger and the outline view is at the bottom left corner of the perspective.

The content of the views is synchronized. For example, if you change a value in the properties view, the editor view is automatically updated to reflect the change.

5.4 Rational Application Developer

IBM Rational Application Developer V6.0 is the full function development platform for developing Java 2 Platform Standard Edition (J2SE), and Java 2 Platform Enterprise Edition (J2EE) applications with a focus on applications to be deployed to IBM WebSphere Application Server and IBM WebSphere Portal.

Rational Application Developer contains a great deal of functionality for Web, Java, and J2EE application developers. This section gives a high-level overview of just a few of the features found in Rational Application Developer. These are the features that you will see used in the development scenarios later in this book.

For more information about Rational Application Developer and a more complete listing of its capabilities, we would suggest the following IBM Redbooks:

- *Rational Application Developer V6 Programming Guide,* SG24-6449
- *WebSphere Version 6 Web Services Handbook Development and Deployment,* SG24-6461

5.4.1 Web development

Web developers can use the Web perspective and supporting views within Rational Application Developer to build and edit Web resources such as servlets, JSPs, HTML pages, style sheets and images, as well as the deployment descriptor files.

You create and maintain Web resources in Web projects. They provide an environment that enables you to perform activities such as link-checking, building, testing, and publishing. Within a Web project, Web resources can be treated as a portable, cohesive unit.

Web projects can be static or dynamic. Static Web projects are comprised solely of static resources, which can be served by a traditional HTTP server (HTML files, images, and so on), and are useful for when you do not have to program any business logic. J2EE Web projects, on the other hand, can deliver dynamic content as well, which gives them the ability to define Web applications.

Wizards provide the means to create Web projects and the artifacts that go in them. These wizards include:

- HTML File wizard
- JSP File wizard
- Faces JSP File wizard
- Page Template File wizard
- JavaScript™ File wizard

- CSS File wizard
- Image File wizard

Web perspective

The Web perspective is designed for Web application development. It contains a default set of views, wizards, and toolbar icons that are often used by Web developers.

Page Designer

The Page Designer is the primary editor for developing HTML, XHTML, JSPs, and Faces JSP source code. It has three representations of the page, including Design, Source, and Preview. The Design tab provides a WYSIWYG environment to visual design the contents of the page. As its name implies, the Source tab provides access to the page source code. The Preview tab shows what the page would like if displayed in a Web browser.

Web Site Designer

The Web Site Designer is provided to simplify and speed up the creation of the Web site navigation and creation of HTML and JSP pages. You can view the Web site in a Navigation view to add new pages, delete pages and move pages in the site. The Web Site Designer is especially suited for building pages that use a page template.

The Web Site Designer is used to create the structure for your application in much the same way you would create a book outline to serve as the basis for writing a book. You use the Web Site Designer to visually lay out the flow of the application, rearranging the elements (JSPs, HTML pages) until it fits your needs. Then you continue by creating pages based on this design.

As you build your Web site design, the information is stored so that navigation links and site maps can be generated automatically. This means that when the structure of a site changes, for example, when a new page is added, the navigation links are automatically regenerated to reflect the new Web site structure.

Struts

Struts development can take advantage of the Web development features available. In addition, Rational Application Developer provides the following support for Struts-based Web applications:

- A Web project can be configured for Struts. This adds the Struts runtime and dependent JARs, tag libraries, and action servlet to the project, and creates skeleton Struts configuration and application resources files. Rational Application Developer provides support for Struts 1.1, selectable when setting up the project. This field is selectable as at the time of this writing support for Struts 1.2.x is being added to Rational Application Developer.

- A set of Struts Component Wizards define action form classes, action classes with action forwarding information, and JSP skeletons with the tag libraries included.
- The Struts Configuration Editor maintains the control information for the action servlet.
- A graphical design tool edits a graphical view of the Web application from which components (forms, actions, JSPs) can be created using the wizards. This graphical view is called a *Web diagram*. The Web diagram editor provides top-down development (developing a Struts application from scratch), bottom-up development (diagramming an existing Struts application that you might have imported) and meet-in-the-middle development (enhancing or modifying an existing diagrammed Struts application).
- Validators validate the Struts XML configuration file and the JSP tags used in the JSP pages.

JavaServer Faces (JSF)

As with Struts development, JSF developers can also take advantage of the many Web development features available in Rational Application Developer. In addition, Rational Application Developer provides a set of views and wizards to make working with JSF components simple. The New Faces JSP File wizard creates a JSP for use with JavaServer Faces components and automatically creates the corresponding backing bean. Built-in tools exist to simplify and automate event handling and to simplify page navigation. The Palette view contains a wide array of Faces components that you can drag and drop to the Page Designer.

Development of a Web application using JSF can be seen in Chapter 9, “JSF front-end scenario” on page 239.

5.4.2 EJB development

EJB development is done in the J2EE perspective. Rational Application Developer supports the Enterprise JavaBeans 1.1, 2.0, and 2.1 specification levels. Features for EJB development include:

- The Enterprise Bean wizard

  This wizard creates stateless and stateful enterprise beans. You can choose to create a session bean, message-driven bean, bean-managed persistent entity bean, or container-managed persistent entity bean.

- The Deployment Descriptor editor

  This editor provides an intuitive interface for modifying the EJB deployment descriptor.
- EJBs

  The EJBs automatically validated and updated as they are modified.
- Tools creating access beans
- Tools creating session bean facades
- EJB mapping tools

  These tools help you map entity enterprise beans to back-end data stores, such as relational databases. There is support for top-down, bottom-up, and meet-in-the-middle mapping development. You can also create schemas and maps from existing EJB JAR files.
- Deployment code generation

5.4.3 Web services support

The Rational Software Development Platform provides tools that are geared toward the development and deployment of both Web service providers and requesters. Web services development is normally done in the Web or J2EE perspective.

Web services providers

The Web Service wizard assists you with the creation of Web services, either from scratch, or from an existing application.

The bottom-up method of creating Web services takes existing artifacts, including Java beans, enterprise beans, DADX files, and URLs, then creates a Web service from them. It wraps the existing artifacts as SOAP-accessible services and describes them in WSDL.

The alternate method, the top-down method, creates Web services from WSDL that exists or is created using the WSDL Editor. This approach lets you start by designing the Web service implementation by building a WSDL file that describes it. You can then use the Web Services wizard to create the Web service and skeleton Java classes to which you can add the required code.

In addition to creating the Web service, the wizard can be used to test the Web service in the Web Service Explorer and to generate a Java bean client proxy to the Web service.

The Web Service wizards and deployment descriptor editors assist you with configuring Web services security (WS-Security) for the WebSphere Application Server environment.

Web services clients

The Web Service Client wizard assists you in generating and a Java bean proxy and sample application to access a Web service. The input to the wizard is the WSDL file for the Web service. The wizard allows you to select the client type from the following:

- Web: Servlets, JSPs, or JavaBeans invoked by a servlet or JSP
- EJB: Session EJBs or JavaBeans invoked by a session EJB
- Managed Java client: Java program running in an application client container
- Stand-alone Java client: Java program running outside a container

The wizard also assists you in testing the new client.

Web Services Explorer

The IBM Web Services Explorer assists you in discovering and publishing your Web service descriptions. The Web Services Explorer provides an interface allowing you to publish your Web service to a UDDI v2 or v3 Business Registry. Publishing the Web service allows you to advertise and make your Web services available so that other businesses and clients can access them.

The Web Services Explorer can also be used to test HTTP bound Web services.

Web services sample test JSPs

The Web Service wizard can create test JavaServer Pages (JSPs) for a Web service. This function is part of generating client-side code, such as proxy classes, into a client Web project. The test client can be generated by the Web Service wizard when generating server and client code and by the Web Service Client wizard when generating a client from a WSDL file.

Web service client JSPs have an advantage over the Web Services Explorer, because they can also be used to test SOAP/JMS Web services.

WSDL support

Wizards and editors are available to work with WSDL. You can use a graphical editor to create a WSDL file from a template and to add WSDL elements (service, port, port types, messages). WSDL files are validated for WS-I compliance.

5.4.4 Connector support

Starting with Rational Application Developer version 6.0.0.1 support for connectors is provided by the J2EE Connector (J2C) Architecture tools. These tools allow developers to create J2EE applications that integrate and extend operations and data on their existing Enterprise Information Systems (EIS).

Wizards

The following wizards assist in developing applications for connectors.

- J2C Java Bean Wizard: For the generation of EIS specific Java beans.
- CICS/IMS™ Data Binding Wizard: For the creation of reusable data types for input or output into CICS/IMS transactions.
- Deployable Code Creation Wizard: For the creation of Web pages, Web Services, or EJBs from your J2C Java bean.

Java bean editing

The following features are available when editing Java beans:

- Add method snippets. Add a new method to access an EIS operation.
- J2C doclet tag code assist, change J2C Java beans to expose or modify generated J2C tags.

Resource adapters

The following resource adapters are supported in the J2C tools:

- CICS ECI 5.1 (J2C 1.0)
- CICS ECI 6.0 (J2C 1.5) - Outbound only
- IMS 9.1.0.1.1 (J2C 1.0)
- IMS 9.1.0.2 (J2C 1.5) - Outbound only

5.4.5 Test environment

Rational Application Developer provides support for testing, debugging, profiling and deploying enterprise applications to local and remote test environments. To run an enterprise application or Web application, the application must be published, or deployed, to the server. Publishing the application is achieved by installing the EAR project for the application into an application server. The server can then be started and the application can be tested in a Web browser, or by using the Universal Test Client (UTC) for EJBs and Web services.

Rational Application Developer includes integrated test environments for IBM WebSphere Application Server V5.0/V5.1/V6.0 and IBM WebSphere Portal V5.0.2/V5.1. You can simultaneously run multiple server configurations and test environments in the same development environment.

5.4.6 Team development

Rational Application Developer provides clients for ClearCase® and CVS. Using a version control mechanism allows multiple members of a team to work on an application by checking out files from a repository, updating them and checking them back in.

Rational ClearCase is a software configuration management (SCM) product that helps to automate the tasks required to write, release, and maintain software code. Rational ClearCase offers the essential functions of version control, workspace management, process configuration, and build management. By automating many of the necessary and error-prone tasks associated with software development, Rational ClearCase helps teams of all sizes build high quality software.

Concurrent Version System (CVS) is a simple open-source software configuration management (SCM) system. CVS only implements version control. CVS can be used by individual developers as well as by large, distributed teams.

5.5 Rational Software Architect

Rational Software Architect contains the features found in Rational Application Developer, plus features that appeal to software architects. Rational Software Architect is a design and development tool that leverages model-driven development with the Unified Modeling Language (UML) for creating well-architected applications and services.

5.5.1 Rational Unified Process guidance

You can access process guidance content and features directly in the Rational Software Development Platform to guide you and other team members in your software development project. A configuration of the Rational Unified Process® (RUP®) platform is provided with topics on software development best practices, tool mentors, and other process-related information.

5.5.2 Model-driven development

Model-driven development (MDD) is the concept of using models as the basis of application development. One approach to model-driven development, called Model Driven Architecture (MDA), is in the process of being defined by the Object Management Group (OMG). MDA defines development using UML models at different levels of abstraction. Transformations are used to take a model at one level and transform it to a model at a different level. You can see the Model Driven Architecture (MDA) standards at:

http://www.omg.org/mda

Rational Software Architect provides full support for MDD with UML 2 modeling, transformations, code generation from models, and patterns. As a part of the model-driven development support, Rational Software Architect also supports the current principles of Model Driven Architecture (MDA).

Transformations

A transformation converts elements of a source model to elements of a target model. For example, the source and target model can be text files, code models or UML models. When the source and target models are both UML models, the transformation usually converts the elements from one level of abstraction to another. You can apply a transformation to an entire model or a subset of model elements in a model to generate output such as code.

Rational Software Architect comes with the following set of transformations

- Business Tier Transformations
  - UML → EJB Business Tier
- Integration Tier Transformations
  - UML → EJB Integration Tier
  - UML → EJB UML
- Presentation Tier Transformations
  - UML → IBM Portlet (JSF)
- RSA Transformations
  - EMF Deployment → UML2
  - UML → EMF Deployment
  - UML → C++
  - UML → EJB
  - UML → Java

Additional transformations may be available on the Web.

Patterns

Rational design patterns capture frequently used or complex structures and processes for reuse. They are used to integrate repeatable software design solutions into UML 2.0 models. Rational patterns are a type of transform.

Rational Software Architect provides the tools needed to create design patterns. When developers recognize repeatable structures or processes they can create patterns from them, allowing others to use these designs.

Patterns are stored in a RAS repository as a unique type of reusable asset in the form of a plug-in. Users can browse the repository for useful patterns as they model systems. The pattern user relies on the pattern documentation for information about selecting and applying a pattern. Depending on the pattern design, the pattern applier has the flexibility to apply all or only part of a pattern as needed.

A set of sample patterns is supplied with Rational Software Architect and can be seen in the RAS perspective. Another set of patterns is included with the product and can be installed as an example.

5.5.3 Modeling

UML modeling provides a way of architecting systems in such a way they can be communicated to the stakeholders. UML models show a specific perspective of a system. Models are visual representations and as such are easily verified and communicated. Models start at the conceptual levels and can be refined down to detailed levels. Rational Software Architect supports UML Version 2 (UML 2).

Rational Software Architect supports modeling through all phases of software development.

1. Capturing system requirements

   The first step in any system design is to determine the requirements for the solution. The IBM Rational RequisitePro® solution is a requirements and use case management tool. This tool can be integrated with Rational Software Architect, allowing you to map existing requirements to existing UML model elements. You can also create requirements from existing model elements, or create model elements from existing requirements definitions. The result of this development phase is one or more use case diagrams that describe how the system will be used.

2. Domain analysis

   The next step is to build on the use case model by describing the high-level structure of the system based on the system domain requirements. An analysis model is used to capture this information. The analysis model consists of class diagrams that model the static structure of the system and of sequence diagrams that model the interactions between participants. The analysis model describes the logical structure of the system but does not define how it will be implemented.

3. Architectural design

   The next step is to create a design model to define the architecture and implementation choices for the application. The design model builds on the analysis model by adding details about the system structure and how implementation will occur. Classes that were identified in the analysis model are refined to include the implementation constructs.

   You can use a variety of diagrams for this purpose, including sequence, state machine, component, and deployment diagrams. It is during this stage that you can apply proven design patterns and automated model-to-model transformations.

- Implementation

  Developers transition from design to implementation by using automated transformations to convert the model to code (such as Java, EJB, or C++) and by continuing to develop and deploy the application by using software and Web development, debugging, testing and deployment capabilities.

The Modeling perspective is the primary workbench interface for working with models.

5.5.4 Asset-based development

As business needs are increasingly solved using more and more complex software solutions, it has become apparent that many of these solutions are created using the same integral key components structured in a similar manner. The idea that the same actions can be performed over and over in a variety of ways to create a solution has given rise to many of the fundamental concepts used in software development today, namely the use of patterns to structure solutions, and the reuse of assets within a context to build the key components of a solution.

Asset-based development embodies the idea of developing solutions by reusing defined and documented assets. These assets are made up of software artifacts that detail the requirements, design elements, development and testing process, and deployment requirements. Reusing these assets streamlines the development process and leverages previous investments.

The success of asset-based development within a department, organization, or on a more widespread basis, the community, lies in the ability to identify potential assets for reuse. Once a potential asset is identified, often through repeated experiences during development and deployment processes, it must be defined and made available for reuse by storing it in a central repository.

Potential consumers browse the repository for assets they can use. Documentation, an integral part of each asset, is key to the successful use effectiveness of an asset. The documentation details not only how the asset is to be used, but should give enough information that a potential consumer knows if the asset is appropriate for their use.

As a final step, feedback to the managers of the asset will help in tracking the effectiveness and value of the asset.

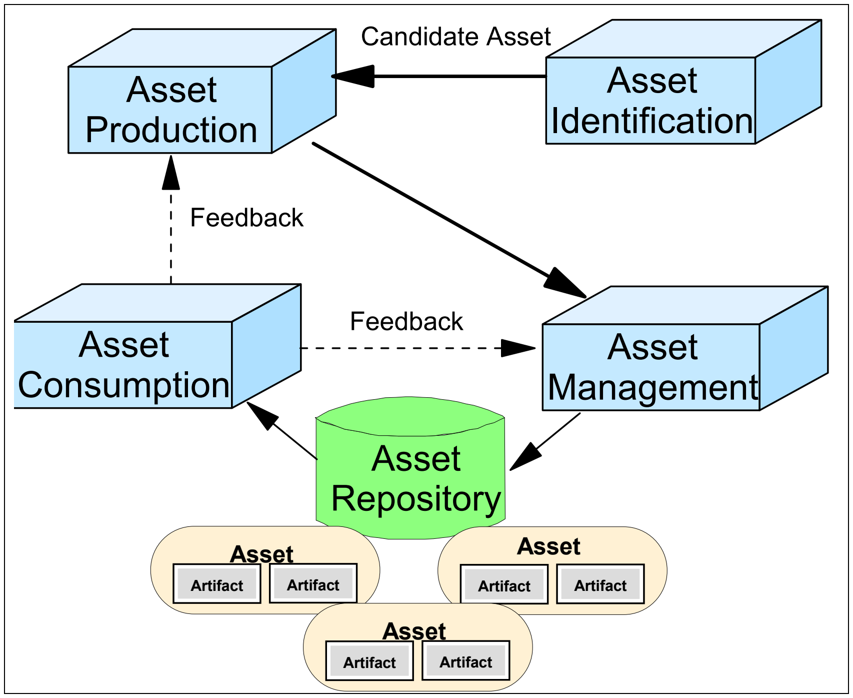


Figure 5-6 Asset-based development cycle

The following platforms deliver asset-based development:

- Rational Unified Process (RUP) has defined processes for producing and consuming assets.
- The IBM Software Development platform has incorporated asset-based development capabilities into its products. Rational Software Architect and Rational Software Modeler both contain tools for producing, consuming, and managing assets.

The Reusable Asset Specification (RAS) defines a standard way to package assets and describe their contents. Rational Software Architect provides a *reusable asset* (RAS) perspective for working with reusable assets.

5.6 For more information

For further information about these topics, see the following publications:

- *An introduction to Model Driven Architecture Part 1: MDA and today's systems*

  http://www-128.ibm.com/developerworks/rational/library/3100.html

- *An introduction to Model-Driven Architecture (MDA): Part II: Lessons from the design and use of an MDA toolkit*

  http://www-128.ibm.com/developerworks/rational/library/content/RationalEdge/apr05/brown/

- Model Driven Architecture (MDA) standards at:

  http://www.omg.org/mda

- Reusable Asset Specification

  http://www.flashline.com/ras/RAS_060604.pdf

6

Technology options

This chapter provides an overview of the Web application technology options you should consider when building a solution. Given the vast number of products in the market today, a complete analysis of all options would be out of the scope of this book. Therefore, our focus will be on IBM technology, and IBM supported industry standards. The recommendations are guided by the demands of reuse, flexibility, and interoperability, and subsequently are based on the open industry standards outlined by Java 2 Platform, Enterprise Edition (J2EE). Many of the choices continue to evolve and expand as the J2EE specification matures to include a broader view of the enterprise architecture. These recommendations are based on the J2EE1.3 specification and parts of the J2EE1.4 specification.

6.1 The big picture

Our discussion of technology options is organized along the enterprise application tiers shown in Figure 6-1:

- Client technologies for interfacing with end users
- Web application server technologies for providing server-side presentation and business logic
- Integration technologies for providing access to the enterprise tier

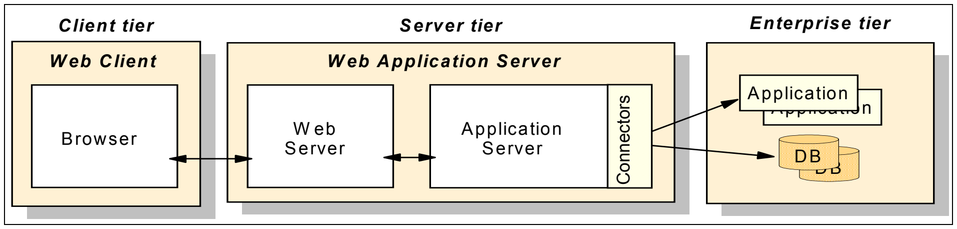


Figure 6-1 Self-Service application tiers

6.2 Client technologies

Figure 6-2 on page 103 shows the recommended technologies for clients.

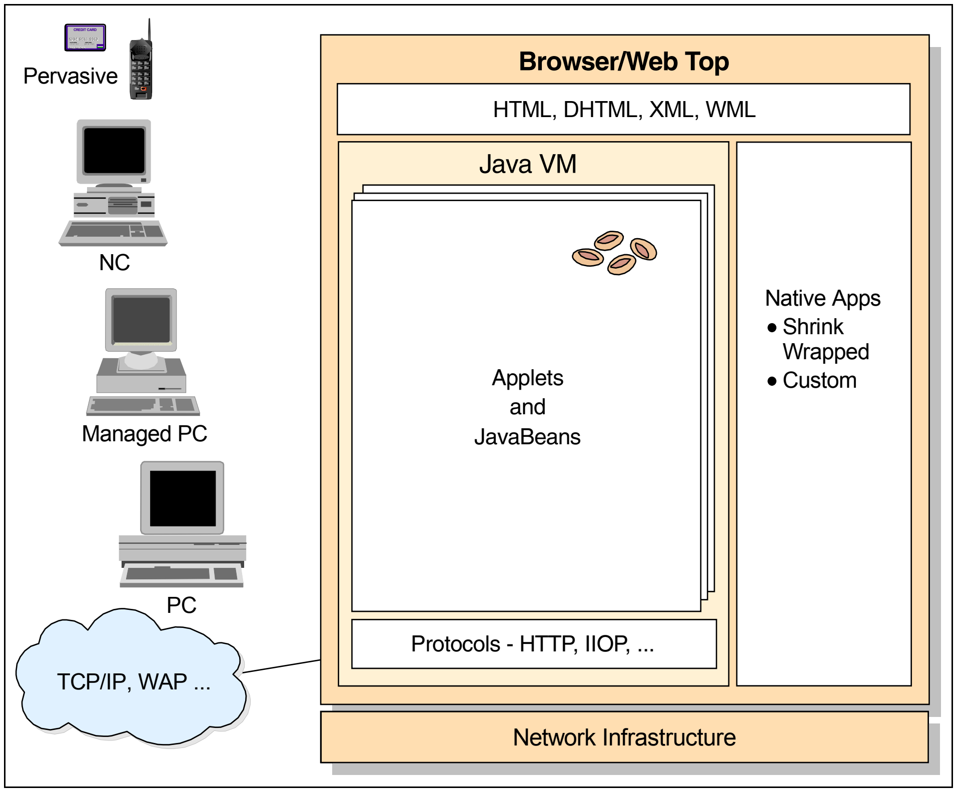


Figure 6-2 Web client technology model

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

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

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

We also touch on some of the current wireless technology choices.

6.2.1 Web-based clients

In this section we discuss the key technologies involved with Web-based clients.

Web browser

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

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

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

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

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

HTML

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

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

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

`http://validator.w3.org/`

Dynamic HTML

DHTML allows a high degree of flexibility in designing and displaying a user interface. In particular, DHTML includes Cascading Style Sheets (CSS), which enable different fonts, margins, and line spacing for various parts of the display to be created. These elements can be accurately positioned using absolute coordinates. See “Cascading Style Sheets” on page 106, for further details.

Another advantage of DHTML is that it increases the level of functionality of an HTML page through a document object model and event model. The document object enables scripting languages such as JavaScript to control parts of the HTML page. For example, text and images can be moved about the window, and hidden or shown, under the command of a script. Also, scripting can be used to change the color or image of a link when the mouse is moved over it, or to validate a text input field of a form without having to send it to the server.

Unfortunately, there are several disadvantages to using DHTML. The greatest of these is that two different implementations (Netscape and Microsoft) exist and are found only on the more recent browser versions. A small, basic set of functionality is common to both, but differences appear in most areas. The significant difference is that Microsoft allows the content of the HTML page to be modified by using either JScript® or VBScript, while Netscape allows the content to be manipulated (moved, hidden, shown) using JavaScript only.

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

Cascading Style Sheets

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

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

CSS is defined as level1 and level 2 specifications. Level 1 was written with HTML in mind, while level 2 was expanded to include general markup styles for XML documents. Developers using CSS should unit test against a validator tool, such as the W3C CSS Validation Service at:

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

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

JavaScript

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

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

`http://ecma-international.org`

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

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

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

Java applets

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

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

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

```
<APPLET CODEBASE="/mydir" CODE="myapplet.class" width=400 height=100>
  <PARAM NAME="myParameter" VALUE="myValue">
</APPLET>
```

For this example, a Java applet called `myapplet` will run. An effective way to send data to an applet is with the use of the `PARAM` tag. The applet has access to this parameter data and can easily use it as input to the display logic.

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

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

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

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

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

- Conventional APPLET tag
- OBJECT tag instead of APPLET tag for Internet Explorer or the EMBED tag with the APPLET tag for Netscape

A second disadvantage of Java applets is that any classes that are not included as part of the Java support in the browser, such as widgets and business logic, must be loaded from the Web server as they are needed. If these additional classes are large, the initialization of the applet may take from seconds to minutes, depending upon the speed of the connection to the Internet.

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

Java applets can also require substantially more memory than JavaScript or DHTML. As previously mentioned, applets are generally composed of several distinct Java classes. All class objects will require some memory, particularly those classes holding application data. Great care must be taken with the manipulation of Java data classes, or an applet could severely tax the client's memory resources.

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

XML (client side)

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

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

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

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

For most Web application designs, you should focus your attention on the use of XML on the server side. See 6.3.8, “XML” on page 120, for additional discussion of the server-side use of XML.

XHTML 1.1 (HTML 4.01)

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

The advantages of XHTML are:

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

Some issues with XHTML are:

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

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

Some of the common features found in Wireless Markup Language (WML) and other subsets of HTML have been used as the basis for developing XHTML Basic:

- Basic text
- Basic forms and tables
- Hyperlinks

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

XForms

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

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

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

6.2.2 Mobile clients

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

Devices

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

Voice

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

Voice technology fall into two categories: Those that recognize speech and those that generate speech. The ability to recognize human voice by computers is called *Automatic Speech Recognition* (ASR). The ability to generate speech from written text is called *speech synthesis* or *Text-to-Speech* (TTS).

Architecture

Support for mobile clients impacts the runtime topology and therefore must be designed and implemented using best practices for system architecture. The good news is that any past investment in Web architecture to support Internet-based applications can be extended to support mobile clients.

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

WAP

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

Microbrowser

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

WML

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

WMLScript

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

Bluetooth

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

J2ME

The *Java 2 Platform Micro Edition* (J2ME™) provides a standard environment for applications running on small embedded devices, such as mobile phones, pagers, PDAs, and TV set-top boxes. Given the wide variance in the available resources of such devices, the J2ME specification is divided in multiple configurations and profiles.

J2ME configurations include a Java virtual machine (JVM™) and Java class libraries. The two existing configurations differ primarily on the computing resources available to the device, such as memory, power, and storage. Within a configuration, a J2ME profile will provide additional Java APIs geared to a specific class of device, such as pagers or PDAs.

For further details about J2ME, see:

http://java.sun.com/j2me

6.3 Web application server

Figure 6-3 on page 114 shows the recommended technology model for a Web application server.

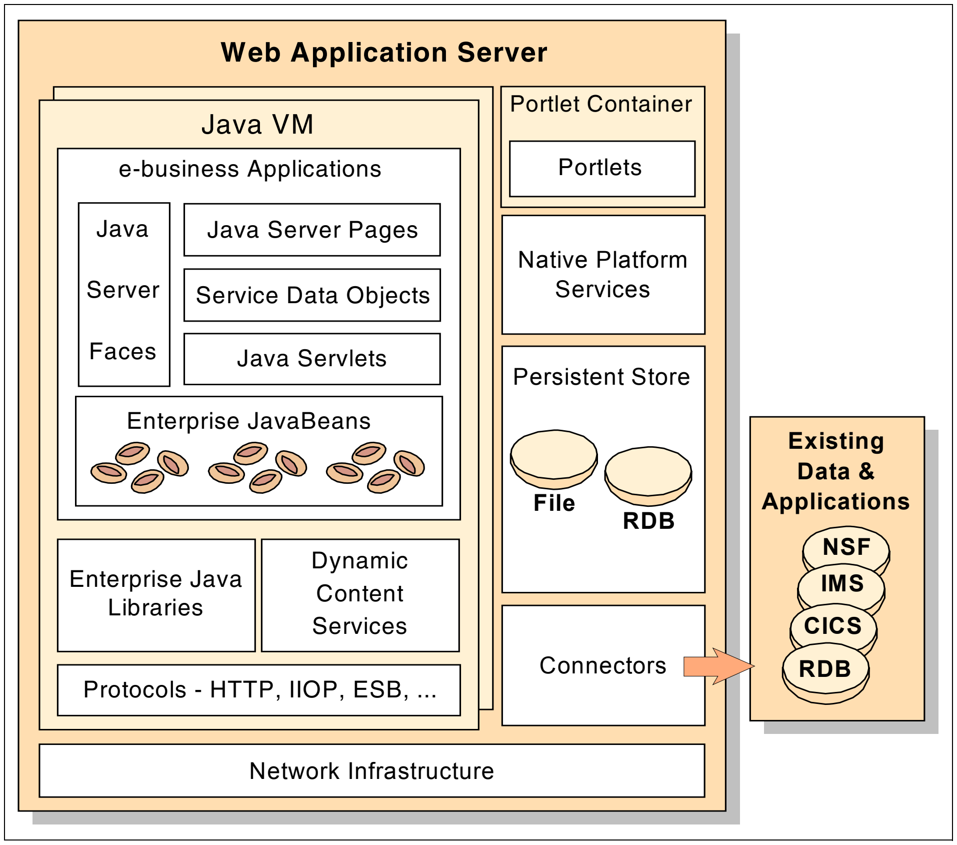


Figure 6-3 Web application server technology model

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

Before looking at the technologies and APIs available in the Web application programming environment, we need to have a word about two fundamental operational components on this node: the HTTP server and the application server. For production applications, they should be chosen for their operational characteristics in areas such as robustness, performance, and availability.

We follow the well-known Model-View-Controller (MVC) architectural pattern so often used in user interfaces. For the Web application programming model:

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

- The View is a visual representation of the model. Multiple views can exist simultaneously for the same model, and each view is responsible for making sure that it is presenting the most current data by either subscribing to state change events or by making periodic queries to the model. With J2EE, the view is generally implemented using JavaServer Pages (JSP).
- The Controller decouples the visual presentation from the underlying business data and logic by handling user interactions and controlling access to the model. It processes the incoming HTTP requests and invokes the appropriate business or UI logic. Using J2EE, the controller is often implemented as a servlet.

Given its wide spread acceptance, there are several design frameworks supporting the implementation of the Model-View-Controller pattern. These include the Java Server Faces (JSF) Framework, Apache Jakarta Project's Struts, and IBM Global Services's EAD4J.

6.3.1 Java servlets

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

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

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

One of the attractions of using servlets is that the API is a very accessible one for a Java programmer to master. The specification of the J2EE 1.4 platform requires Servlet API 2.4 for support of packaging and installation of Web applications.

Servlets are a core technology in the Web application programming model. They are the recommended choice for implementing the Controller classes that handle HTTP requests received from the Web client.

6.3.2 JavaServer Pages (JSPs)

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

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

JSP technology uses XML-like tags and scriptlets written in Java programming language to encapsulate the conditional logic that generates dynamic content for an HTML page. In the runtime environment, JSPs are compiled into servlets before being executed on the Web application. Output is not limited to HTML but also includes JavaScript, WML, XML, cHTML, DHTML, and VoiceXML. The JSP API for J2EE 1.4 is JSP 2.0.

6.3.3 JavaServer Faces

JavaServer Faces (JSF) is a framework for developing Java Web applications The JSF framework aims to unify techniques for solving a number of common problems in Web application design and development, such as:

- User interface development

  JSF allows direct binding of user interface (UI) components to model data. It abstracts request processing into an event-driven model. Developers can use extensive libraries of prebuilt UI components that provide both basic and advanced Web functionality.

- Navigation

  JSF introduces a layer of separation between business logic and the resulting UI pages; stand-alone, flexible rules drive the flow of pages.

- Session and object management

JSF manages designated model data objects by handling their initialization, persistence over the request cycle, and cleanup.

- Validation and error feedback

  JSF allows direct binding of reusable validators to UI components. The framework also provides a queue mechanism to simplify error and message feedback to the application user. These messages can be associated with specific UI components.

- Internationalization

  JSF provides tools for internationalizing Web applications, supporting number, currency, time, and date formatting, and externalizing of UI strings.

JSF is particularly well-suited to implement the MVC architectural pattern. Specifically, the JSF framework maps to the pattern as follows:

- Model

  Managed beans make up the model of a JSF application. These Java beans typically interface with reusable business logic components or external systems, such as a mainframe or database.

- View

  JSPs make up the view of a JSF Web application. These JSPs are created by combining model data with predefined and custom-made UI components.

- Controller

  The FacesServlet, which drives navigation and object management, makes up most of a JSF application's controller. Event listeners also contribute to the controller logic.

Note that unlike other frameworks supporting the MVC pattern, JavaServer Faces are a standard component of the J2EE 1.4 specification. Therefore, it has received wide support from IBM and other leading vendors.

6.3.4 Struts

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

Struts addresses a number of common design and implementation issues associated with most servlet projects:

- Mapping HTTP parameters to JavaBeans
- Standard input validation
- Standard error display

- Message internationalization
- Hard coded JSP URIs

Struts is widely used, and supported by most J2EE IDE tools (including Rational Software Architect and Rational Application Developer). However, JavaServer Faces are likely to become the industry standard, because it has been adopted as a J2EE standard. There are many similarities between the two frameworks, and, in fact, many of the concepts originated in Struts were adopted in the JSF specification.

6.3.5 Service Data Objects

Service Data Objects (SDO) is a data programming architecture and API for the Java platform that unifies data programming across data source types, provides support for common application patterns, and enables applications, tools, and frameworks to more easily query, view, bind, update, and introspect data. SDO is currently the subject of a Java specification request (JSR-235), but has not yet been standardized under this process.

SDOs are designed to simplify and unify the way in which applications handle data. Using SDO, application programmers can uniformly access and manipulate data from heterogeneous data sources, including relational databases, XML data sources, Web services, and enterprise information systems. The SDO architecture consists of three major components:

- Data object

  The data object is designed to be an easy way for a Java programmer to access, traverse, and update structured data. Data objects have a rich variety of strongly and loosely-typed interfaces for querying and updating properties. This enables a simple programming model without sacrificing the dynamic model required by tools and frameworks. A data object may also be a composite of other data objects.

- Data graph

  SDO is based on the concept of disconnected data graphs. A data graph is a collection of tree-structured or graph-structured data objects. Under the disconnected data graphs architecture, a client retrieves a data graph from a data source, mutates the data graph, and can then apply the data graph changes to the data source. The data graph also contains some metadata about the data object including change summary and metadata information. The metadata API allows applications, tools, and frameworks to introspect the data model for a data graph, enabling applications to handle data from heterogeneous data sources in a uniform way.

- Data mediator

The task of connecting applications to data sources is performed by a data mediator. Client applications query a data mediator and get a data graph in response. Client applications send an updated data graph to a data mediator to have the updates applied to the original data source. This architecture allows applications to deal principally with data graphs and data objects, providing a layer of abstraction between the business data and the data source.

6.3.6 Portal applications

Portal applications have several important features:

- They can collect content from a variety of sources and present them to the user in a single unified format.
- The presentation can be personalized so that each user sees a view based on their own characteristics or role.
- The presentation can be customized by the user to fulfill their specific needs.
- They can provide collaboration tools, which allow teams to work in a virtual office.
- They can provide content to a range of devices, formatting and selecting the content appropriately according to the capabilities of the device.

Portlets

Portlets are Java based components much like servlets, but are designed to be aggregated to produce a portal page. A single portal page request will generally trigger the invocation of multiple portlets. The Java Portlet V1.0 specification has been developed to provide a standard for the development of Java portlets for portal applications. For further details, see:

`http://jcp.org/en/jsr/detail?id=168`

Portlets will run within a portlet container, which provides them the required runtime environment. The container manages the portlet life cycle and storage needs.

One such container, is the one provided by the IBM WebSphere Portal. It runs within WebSphere Application Server, using the J2EE standard services and management capabilities of the server as the basis for portal services.

6.3.7 JavaBeans

JavaBeans is an architecture developed by Sun™ Microsystems™, Inc. describing an API and a set of conventions for reusable, Java-based components. Code written to Sun's JavaBeans architecture is called *JavaBeans* or just beans. One of the design criteria for the JavaBeans API was support for builder tools that can compose solutions incorporating beans. Beans may be visual or non-visual.

Beans are recommended for use in conjunction with servlets and JSPs in the following ways:

- As the client interface to the Model layer

  A Controller servlet will use this bean interface.
- As the client interface to other resources

  In some cases, this can be generated for you by a tool.
- As a component that incorporates a number of property-value pairs for use by other components or classes

  For example, the JavaServer Pages specification includes a set of tags for accessing JavaBeans properties.

6.3.8 XML

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

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

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

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

http://xml.apache.org/

Defining XML documents

XML documents are defined using DTDs or XML Schemas.

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

DTDs have two major problems:

- Poor data typing

  In DTDsm elements can only be specified as EMPTY, ANY, element content, or mixed element-and-text content, and there is no standard way to specify null values for elements.

  Data typing such as date formats, numbers, or other common data types cannot be specified in the DTD. As a result, an XML document might comply with the DTD but still have data type errors that can only be detected by the application.

- Not defined in XML

  DTD uses its own language to define XML syntax that is not compliant to the XML specification. This makes it difficult to manipulate a DTD.

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

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

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

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

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

XSLT

Extensible Stylesheet Language Transformations (XSLT) is a W3C specification for transforming XML documents into other XML documents. The XSLT is built on top of the Extensible Stylesheet Language (XSL), which, like CSS2 seen in“Cascading Style Sheets” on page 106, is a stylesheet language for XML. Unlike CSS2, XSL is also a transformation language.

A transformation expressed in the XSLT language defines a set of rules for transforming a source tree to a result tree, and it is expressed in the form of a stylesheet.

An XSLT processor is used for transforming a source document to a result document. There are currently a number of XSLT processors available on the market. DataPower has introduced an XSL just-in-time (JIT) compiler, which speeds up the time taken for the XSL transformation.

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

XML security

XML security is an important issue, particularly where XML is being used to by organizations to interchange data across the Internet. Several new XML security specifications are working their way through three standards bodies:

- World Wide Web Consortium (W3C)
- Internet Engineering Task Force (IETF)
- Organization for the Advancement of Structured Information Standards (OASIS)

We highlight a few specifications here:

- XML Signature Syntax and Processing is a specification for digitally signing electronic documents using XML syntax.

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

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

- XML encryption will allow encryption of digital content, such as Graphical Interchange Format (GIF) images or XML fragments. XML Encryption allows parts of an XML document to be encrypted while leaving other parts open, encryption of the XML itself, or the super-encryption of data (that is, encrypting an XML document when some elements have already been encrypted).
- XML Key Management Specification (XKMS) establishes a standard for XML-based applications to use Public Key Infrastructure (PKI) when handling digitally signed or encrypted XML documents. XML signature addresses message and user integrity, but not issues of trust that key cryptography ensures.
- Security Assertion Markup Language (SAML) is the first industry standard for secure e-commerce transactions using XML. It aims to standardize the exchange of user identities and authorizations by defining how this information is to be presented in XML documents, regardless of the underlying security systems in place.

Advantages of XML

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

- Acceptability of use for data transfer

  XML is a standard way of putting information in a format that can be processed and exchanged across different hardware devices, operating systems, software applications, and the Web.
- Uniformity and conformity

  XML gives you an common format that could be developed upon and is accepted industry-wide.
- Simplicity and openness

  Information coded in XML is human readable.
- Separation of data and display

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

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

Disadvantages of XML

Some XML issues to consider are:

- Complexity

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

  When multiple applications use XML to communicate with each other they need to agree on the tag names they are using. While industry-specific standard tag definitions often do exist, you can still declare your own nonstandard tags.
- Large size

  XML documents tend to be larger in size than other forms of data representation.

6.3.9 Enterprise JavaBeans

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

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

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

There are three types of Enterprise JavaBeans:

- Session beans
- Entity beans
- Message-driven beans

A typical session bean has the following characteristics:

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

A typical entity bean has the following characteristics:

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

A typical message-driven bean has the following characteristics:

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

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

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

The business logic of a Web application often accesses data in a database. EJB entity beans are a convenient way to wrap the relational database layer in an object layer, hiding the complexity of database access. Because a single business task can involve accessing several tables in a database, modeling rows in those tables with entity beans makes it easier for your application logic to manipulate the data.

An important change to the specification in EJB 2.0 is the addition of a new enterprise bean type, the message-driven bean (MDB). The message-driven bean is designed specifically to handle incoming JMS messages. The EJB container uses message properties and a bean deployment descriptor to select the bean to invoke when a message arrives, so your application logic only needs to process the message contents.

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

6.3.10 Additional enterprise Java APIs

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

- JNDI: Java Naming and Directory Interface™

  This package provides a common API to a directory service independent of any directory access protocol. This allows for easy migration to new directory services. Through this interface, component providers can store and retrieve Java object instances by name. Service provider implementations include those for JDBC data sources, LDAP directories, RMI, and CORBA object registries. Sample uses of JNDI include:

  – Accessing a user profile from an LDAP directory
  – Locating and accessing an EJB home
  – Locating a driver-specific data source

- RMI-IIOP: Remote Method Invocation (RMI) and RMI over IIOP

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

- JTA: Java Transaction API

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

- JAF: JavaBeans Activation Framework

  This API is not intended for typical application use, but it is required by the JavaMail™ API.

- JavaMail

  This is a set of classes for supporting e-mail. Functionally it provides APIs for reading, sending, and composing Internet mail. This API models a mail delivery system and requires the SMTP for sending mail and POP3 or IMAP for receiving mail. Special data wrapper classes are provided to view and edit data in the mail content. Support for MIME data is delegated to the JAF-aware beans.

- JAXP is an API for parsing and transforming XML documents.
- JAAS is the Java Authentication and Authorization Service.

6.4 Integration technologies

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

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

- The current infrastructure

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

- Time to market

  Web service enabling an application is relatively fast with the Web services development tools available.

- Future expansion plans

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

- Reliability

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

- Transaction support

  Web services currently do not offer support for transactions. If your application needs transactional management, it might be worthwhile to consider either JMS or J2EE Connectors.

6.4.1 Web services

Web services is a recent reinvention of concepts that have been around for sometime. They introduce many new advantages and capabilities. In a sense, none of the function provided by Web services is new; CORBA has provided much of this function for many years. What Web services does do that is new is to build upon existing open Web technologies such as XML, URL and HTTP. Web services are defined in several different standards such as SOAP and

WSDL which build upon general Web and other Web services standards. These standards are defined by the World Wide Web Consortium, the Organization for the Advancement of Structured Information Standards (OASIS) and Web Services Interoperability Organization (WS-I).

The basic Web services support provides for three simple usage models. These are:

- One-way usage scenario

  A Web services message is sent from a consumer to a provider and no response message is expected.

- Synchronous request/response usage scenario

  A Web services message is sent from a consumer to a provider and a response message is expected.

- Basic callback usage scenario

  A Web service message is sent from a consumer to a provider using the two-way invocation model, but the response is just treated as an acknowledgement that the request has been received. The provider then responds by calling making use of a Web service callback to the consumer.

Other Web service standards are built upon these basic standards and invocation models to provide higher level functions and qualities of service. Examples of these standards are WS-Transaction, WS-Security and WS-ResourceFramework.

One of the main aims of Web services is to provide a loose coupling between service consumer and service providers. While this is limited to a certain extent by a requirement for the consumers and providers to agree on a WSDL interface definition, Web services have been created with significant flexibility with regard to their location. Figure 6-4 on page 130 shows how the Web services interaction model has been designed with this form of loose coupling.

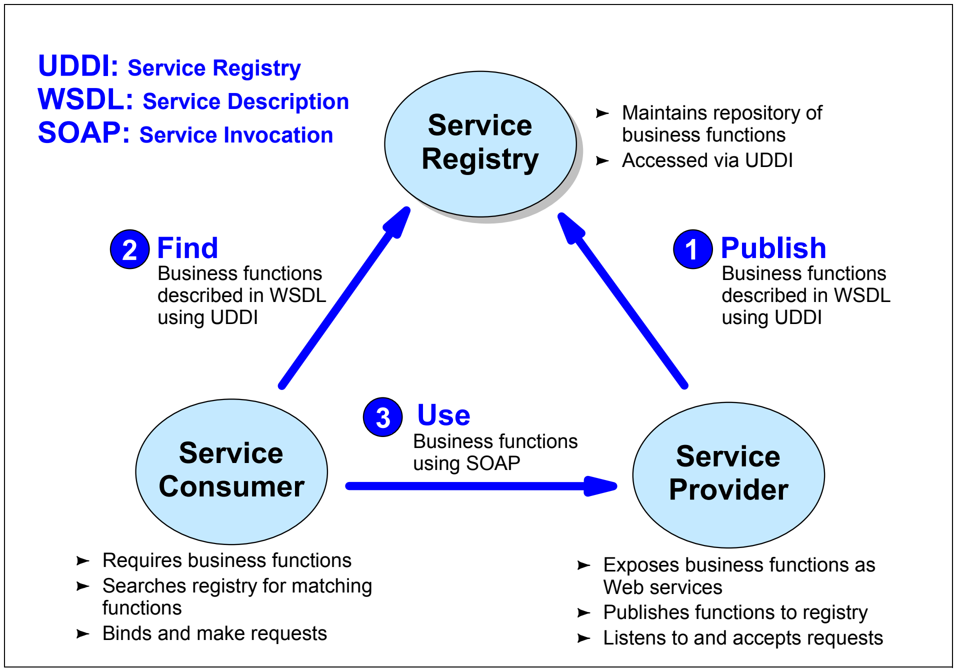

UDDI: Service Registry
WSDL: Service Description
SOAP: Service Invocation
Service Registry
Maintains repository of business functions
Accessed via UDDI
2 Find
Business functions described in WSDL using UDDI
1 Publish
Business functions described in WSDL using UDDI
3 Use
Business functions using SOAP
Service Consumer
Service Provider
Requires business functions
Searches registry for matching functions
Binds and make requests
Exposes business functions as Web services
Publishes functions to registry
Listens to and accepts requests

Figure 6-4 Basic Web service interaction model.

The interactions work as follows:

1. The service provider publishes some WSDL defining its interface and location to a service registry.
2. The service consumer contacts the service registry to obtain a reference to a service provider.
3. The service consumer, having obtained the location of the service provider, makes calls on the service provider.

Note: Although this model is regularly discussed, the service registry is often removed from the cycle in real implementations in the interests of simplicity and lack of trust of the services in the service registry. This has the drawback that if the service provider is relocated, the service consumer needs to be changed to refer to the new location of the service provider.

SOAP

SOAP is an-XML based format for constructing messages in a transport independent way and a standard on how the message should be handled. SOAP messages consist of an envelope containing a header and a body. It also defines a mechanism for indicating and communicating problems that occurred while processing the message. These are known as SOAP *faults*.

The headers section of a SOAP message is extensible and can contain many different headers defined by different schemas. The extra headers can be used to modify the behavior of the middleware infrastructure. For example, the headers can include information about transactions that can be used to ensure that actions performed by the service consumer and service provider are coordinated.

The body section contains the content of the SOAP message. When used by Web services, the SOAP body contains XML-formatted data. This data is specified in the WSDL describing the Web service.

When talking about SOAP, it is common to talk about SOAP in combination with the transport protocol used to communicate the SOAP message. For example, SOAP transported using HTTP is referred to as SOAP over HTTP or SOAP/HTTP.

The most common transport used to communicate SOAP messages is HTTP. This is to be expected because Web services are designed to make use of Web technologies. However, SOAP can also be communicated using JMS as a transport. When using JMS the address of the Web service is expressed in terms of a JMS connection factory and a JMS destination. Although using JMS provides a more reliable transport mechanism it is not an open standard, requires extra and potential expensive investment, and does not interoperate as easily as SOAP over HTTP.

The SOAP version 1.1 and 1.2 specifications are available from the World Wide Web Consortium.

Web Services Description Language (WSDL)

Web Services Description Language (WSDL) is an XML-based interface definition language that separates function from implementation, and enables design by contract as recommended by SOA. WSDL descriptions contain a *port type* (the functional and data description of the operations that are available in a Web service), a *binding* (providing instructions for interacting with the Web service through specific protocols, such as SOAP over HTTP), and a *port* (providing a specific address through which a Web service can be invoked using a specific protocol binding).

It is common for these three aspects to be defined in three separate WSDL files, each importing the others.

The value of WSDL is that it enables development tooling and middleware for any platform and language to understand service operations and invocation mechanisms. For example, given the WSDL interface to a service that is implemented in Java, running in a WebSphere environment, and offering invocation through HTTP, a developer working in the Microsoft .Net platform can import the WSDL and easily generate application code to invoke the service.

As with SOAP, the WSDL specification is extensible and provides for additional aspects of service interactions to be specified, such as security and transactionality.

Universal Description, Discovery, Integration

Universal Description, Disccover, Integration (UDDI) servers act as a directory of available services and service providers. SOAP can be used to query UDDI to find the locations of WSDL definitions of services, or the search can be performed through a user interface at design or development time. The original UDDI classification was based on a U.S. government taxonomy of businesses, and recent versions of the UDDI specification have added support for custom taxonomies.

A public UDDI directory is provided by IBM, Microsoft, and SAP, each of whom runs a mirror of the same directory of public services. However, there are many patterns of use that involve private registries. For more information, see the following articles:

- *The role of private UDDI nodes in Web services, Part 1: Six species of UDDI*

  `http://www.ibm.com/developerworks/webservices/library/ws-rpu1.html`
- *The role of private UDDI nodes, Part 2: Private nodes and operator nodes*

  `http://www.ibm.com/developerworks/webservices/library/ws-rpu2.html`

6.4.2 J2EE Connector Architecture

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

J2EE Connector Architecture V1.5 support is a requirement of the J2EE V1.4 specification. Resource adapters allow J2EE applications to connect to a particular EIS. The J2EE Connector Architecture specification defines two different types of resource adapters:

- Outbound adapters

  Outbound adapters are used by application initiated requests to an EIS.
- Inbound adapters

  Inbound adapters are used by the EIS making calls to a message-driven bean.

The J2EE Connector Architecture provides a Common Client Interface API (CCI) with both common, and resource adapter specific, interfaces. Application programmers code to this single API rather than needing to use different interfaces for each proprietary system. However it is common for a resource adapter to make use of its own, or an existing API, such as JDBC or JMS.

The J2EE Connector Architecture specification provides support for transactions, security and sharing of connections between different clients.

Advantages of the J2EE Connector Architecture

Some reasons to use J2EE Connector Architecture resource adapters are:

- The CCI simplifies application integration with diverse EISs. This common interface makes it easy to plug third-party or home-grown resource adapters into your applications.
- Inbound adapters provide a way to get a message-driven bean invoked when an event occurs in the EIS. For example, a message arrives at a JMS destination.
- Outbound adapters that are XA capable automatically participate in any transactions in effect without requiring action by an application.
- Outbound adapters can pick up security credentials from the container in which they execute.
- Connections to the EIS can be shared to reduce resource overhead.

Disadvantages of the J2EE Connector Architecture

Although the CCI provides a common interface definition, some resource adapter specific interfaces still need to be used and the usage of these interfaces varies depending on the EIS the resource adapter used.

6.4.3 Java Message Service

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

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

In IBM WebSphere Application Server V6.0, the J2EE 1.4 specification is implemented, which includes JMS 1.1 and EJB 2.1.

According to the JMS 1.1 specification, a message provider is integrated in an application server. As shown in Figure 6-5, the integrated message provider makes it possible to communicate asynchronously with other WebSphere applications, without installing separate messaging software such as IBM WebSphere MQ. WebSphere's integrated JMS server is based on IBM WebSphere MQ.

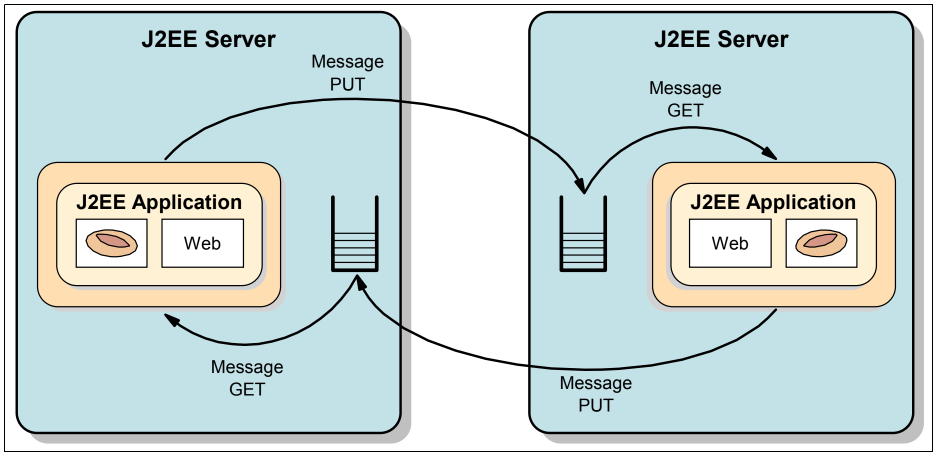


Figure 6-5 Integrated JMS provider

An important feature of EJB 2.1 is message-driven beans (MDB). As described in 6.3.9, "Enterprise JavaBeans" on page 124, message-driven beans are designed specifically to handle incoming JMS messages.

What is messaging?

Messaging is a form of communication between two or more software applications or components. One strength of messaging is application integration. Messaging communication is *loosely* coupled, as compared to *tightly* coupled technologies such as Remote Method Invocation (RMI) or Remote Procedure Calls (RPC). The sender does not need to know anything about the receiver for communication. The message to be delivered is sent to a destination (*queue*) by a sender component. The recipient picks it up from there. Moreover, the sender and receiver do not both have to be available at the same time to communicate.

JMS has two messaging styles, or two domains:

- One-to-one, or point-to-point model
- Publish/subscribe model

Advantages of JMS

The JMS standard is important because:

- It is the first enterprise messaging API that has achieved wide cross-industry support.
- It simplifies the development of enterprise applications by providing standard messaging concepts and conventions that apply across a wide range of enterprise messaging systems.
- It leverages existing, enterprise-proven messaging systems.
- It allows you to extend existing message-based applications by adding new JMS clients that are integrated fully with their existing non-JMS clients.
- Developers have to learn only one common interface for accessing diverse messaging systems.

Disadvantages of JMS

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

6.4.4 Enterprise Service Bus

The Enterprise Service Bus (ESB) is emerging as a middleware infrastructure component that supports the implementation of SOA within an enterprise. An ESB provides an infrastructure that removes any direct connection between service consumers and providers. Consumers connect to the bus and not the provider that actually implements the service. A bus also implements further value added capabilities. For example security and delivery assurance can be implemented centrally within the bus instead of having this buried within the applications.

Note that an Enterprise Service Bus is a concept, not a specific product. It describes a coherent architectural approach for effectively implementing a service-oriented architecture. The actual implementation of an ESB, can be accomplished using a variety of different technologies.

For more detail on Enterprise Service Bus, see 3.2, “Overview of the Enterprise Service Bus” on page 36.

WebSphere service integration bus

The service integration bus is the key component in the WebSphere Application Server V6 that supports the implementation of an Enterprise Service Bus. The flexibility of the service integration bus allows the ESB to connect consumers and providers utilizing different transport mechanism. The service integration bus combines support for applications connecting with native JMS, WebSphere MQ, and Web services. It supports the message-oriented middleware and request-response interaction models. As a part of this, the service integration bus supports multiple message distribution models, reliability options, and transactional messaging.

For more information about the service integration bus, see 5.2.2, “Service integration” on page 76.

6.4.5 Others

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

- RMI/IIOP
- CORBA

RMI/IIOP

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

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

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

CORBA

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

CORBA can be used for integration with existing applications. This is done by creating a CORBA wrapper for the existing application, which can then be invoked by other applications.

CORBA is just a specification, and there are a number of vendors (such as IONA or Borland) that implement it. Each vendor will provide additional value-added services such as persistence, security, and so on, which can be leveraged by CORBA developers.

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

6.5 Where to find more information

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

- *WebSphere Version 6 Web Services Handbook Development and Deployment,* SG24-6461
- *WebSphere Studio 5.1.2, JavaServer Faces and Service Data Objects,* SG24-6361
- *Patterns: Implementing an SOA using an Enterprise Service Bus in WebSphere Application Server V6,* SG24-6494
- *Java Connectors for CICS: Featuring the J2EE Connector Architecture*, SG24-6401
- *Revealed! Architecting Web Access to CICS*, SG24-5466
- *MQSeries Programming Patterns*, SG24-6506
- Flanagan, David, *JavaScript: The Definitive Guide*, Third Edition, O'Reilly & Associates, Inc., 1998
- Maruyama, Hiroshi, Kent Tamura and Naohiko Uramoto, *XML and Java: Developing Web Applications*, Addison-Wesley 1999
- Flanagan, David, Jim Farley, William Crawford and Kris Magnusson, *Java Enterprise in a Nutshell*, O'Reilly & Associates, Inc., 1999
- IBM CICS

  `http://www.ibm.com/software/ts/cics`
- ECMAScript language specification

  `http://www.ecma-international.org/publications/standards/ECMA-262.HTM`
- Java APIs and technology

  `http://java.sun.com/products`

- Validator tools

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

- World Wide Web Consortium (W3C) site

  http://www.w3.org/

- Open source XML frameworks

  http://xml.apache.org/

- Sun ONE™ article, *Riddle Me This: Is Your XML Data Safe?* by Brett Mendel

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

- Service-oriented architecture and Web services

  http://www.ibm.com/software/solutions/webservices/resources.html

7

Application and system design guidelines

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

7.1 e-business application design considerations

The following list provides key issues to consider when designing e-business applications:

- The user experience and the look and feel of the site need to be constantly enhanced to leverage emerging technologies and to attract and retain site users.
- New features have to be constantly added to the site to meet customer demands.
- Such changes and enhancements will have to be delivered at record speed to avoid losing customers to the competition.
- e-business applications in essence represent the corporate brand online. Developers have to work closely with the marketing department to ensure that the digital brand effectively represents the company image. Such intra-group interactions usually present content management challenges.
- It is hard to predict the runtime load of e-business applications. Based on the marketing of the site, the load can increase dramatically over time. If the load increases, the design must allow such applications to be deployed in various high-volume configurations. It is important to be able to move Web applications between these runtime configurations without making significant changes to the code.
- Security requirements are significantly higher for e-business applications compared to traditional applications. To execute traditional applications from the Web, a special set of security-related software might be needed to access private networks.
- The emergence of the personal digital assistant (PDA) and broadband Internet markets will require the same information to be presented in various UI formats. PDAs will require a lightweight presentation style to accommodate the low network bandwidth. Broadband users, on the other hand, will demand a highly interactive, rich, GUI.

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

7.2 Application structure

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

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

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

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

7.2.1 Model-View-Controller design pattern

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

This section will describe the MVC pattern in detail. Note that in practice, we expect that an existing MVC based design framework, such as JSF or Struts will be used. However, in order to provide a broader view of the key design issues, this section will not focus on any one framework. The frameworks described in a later section.

As shown in Figure 7-1 on page 142, *Model* represents the application object that implements the application data and business logic. The *View* is responsible for formatting the application results and dynamic page construction. The *Controller* is responsible for receiving the client request, invoking the appropriate business logic, and based on the results, selecting the appropriate view to be presented to the user.

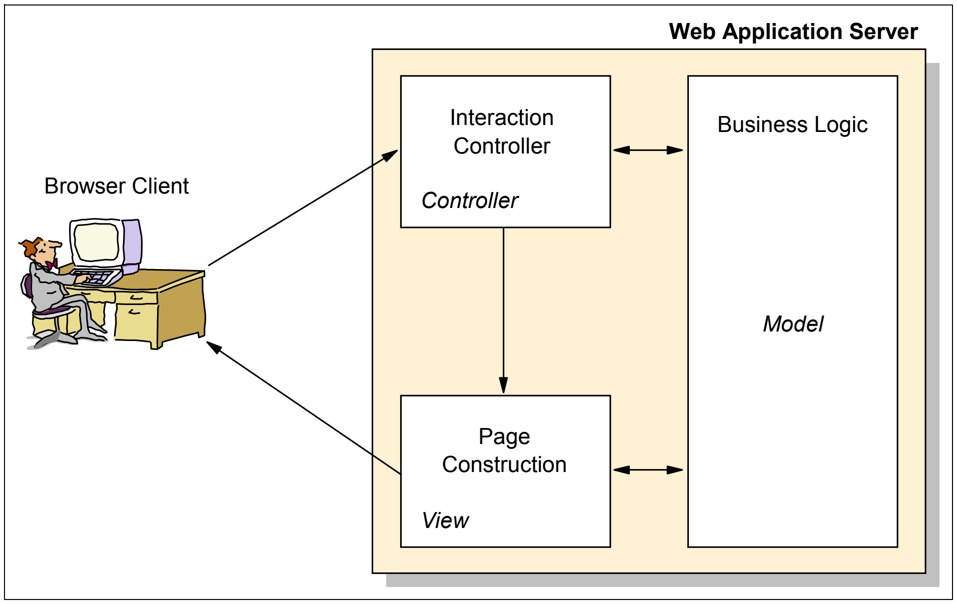


Figure 7-1 The Model-View-Controller design pattern

A number of different types of skills and tools are required to implement various parts of a Web application. For example, the skills and tools required to design an HTML page are vastly different from the skills and tools required to design and develop the business logic part of the application. In order to leverage these scarce resources effectively and to promote reuse, we recommend structuring Web applications to follow the Model-View-Controller design pattern.

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

Interaction controller (Controller)

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

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

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

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

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

- Display pages need to be reusable because they can be called by multiple interaction controllers. For example, Figure 7-2 on page 144 shows that an error page can be called by more than one interaction controller. In such a scenario, if we were to combine the error page construction logic and the interaction controller logic, then the error page logic would need to be duplicated in several places throughout the application.

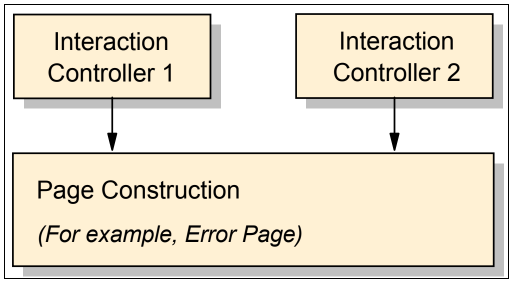


Figure 7-2 Single display page used by multiple interaction controllers

- The interaction controller is required to do page selection. There are several reasons for this, such as the need to include different display pages depending on runtime results, national language support, client browser types, customer types, etc. For example, Figure 7-3 shows an interaction controller calling a page constructor for either an administrator page or a page for normal users, based on the user type.

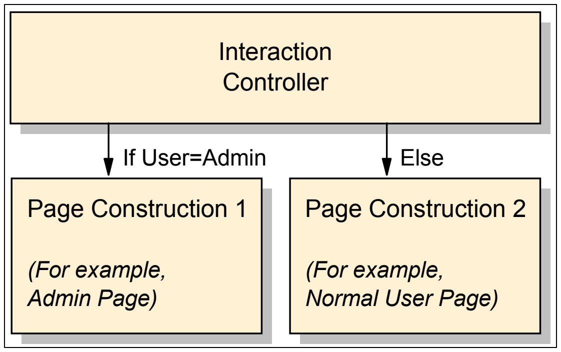


Figure 7-3 One interaction controller calling multiple page constructors

- If there is a need to use different tools and skills for coding interaction controllers and display pages then it is good to separate the two to simplify the development process. Failing to do so could result in multiple people having to write different parts of the same file, thus complicating the version control and code management process.

Note: It is recommended that servlets be used in most cases to implement interaction controllers. However, for simple applications where there is no conditional or transactional logic involved, it is possible to combine the interaction controller and page construction logic into one component. Under such conditions, a JSP would be the best choice.

Another common design issue to consider is the relationship between interactions and interaction controllers. The following is an overview of the options:

- One interaction to one interaction controller

  For every unique interaction, there is a unique interaction controller. For example, login is handled by loginServlet, and logoff is handled by logoffServlet.

- One interaction group to one interaction controller

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

- All interactions to one interaction controller

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

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

Page constructor (View)

The page constructor is responsible for generating the HTML page that will be returned to the client. It is the view component of the application. Similar to interaction controllers, WebSphere allows display pages to be implemented as either servlets or JSPs. JSPs allow template pages to be developed directly in HTML, with scripting logic inserted for dynamic elements of the page and jsp include actions for multipart pages. Hence, a JSP is the best choice for implementing page construction components.

In many cases, the interaction controller passes the dynamic data as JavaBeans to the display page for formatting. In other cases, the display page invokes business logic directly to obtain dynamic data. It makes sense to have the interaction controller pass the data when it has already obtained it and when the data is an essential component of the contract between the interaction controller and the display page. In other cases, the data needed for display is not an essential part of the interaction and can be obtained independently by inserting calls to business logic directly in the display page. However, such direct access to business logic from the page construction component increases the complexity of the display page, since the page designer must know the details of the business logic methods. For this reason, you must be careful to minimize such direct access to business logic from the display pages. Ideally, the source data the page constructor requires can be pre-packaged by the Controller, which eliminates direct manipulation of the Model by the View.

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

Business logic (Model)

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

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

Advantages and disadvantages of the MVC design pattern

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

- Leverages different skill sets

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

- Increases reusability

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

- Can support multiple user interfaces

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

- Improves maintainability of the site

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

- Reduces complexity

  Any non-trivial application implemented without clear separation of concerns could result in large and complex code. For such applications, the MVC separation reduces the complexity.

On the other hand, it is important to recognize the following disadvantages of the MVC design pattern:

- Possible overkill for small applications

  The MVC design pattern can introduce extra artifacts that might not be necessary for very simple cases and might, in fact, increase the complexity of the application. However, if the application is likely to evolve over time, then it can be beneficial to purchase a more complex system than you need in the present to gain the flexibility for the future provided by the MVC design pattern.

- High level of communication requirements between various groups

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

7.2.2 Result bean design pattern

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

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

Figure 7-4 on page 149 demonstrates the relationship between MVC components, result beans, and view beans. See 7.2.3, “View bean design pattern” on page 149, for our discussion on view beans.

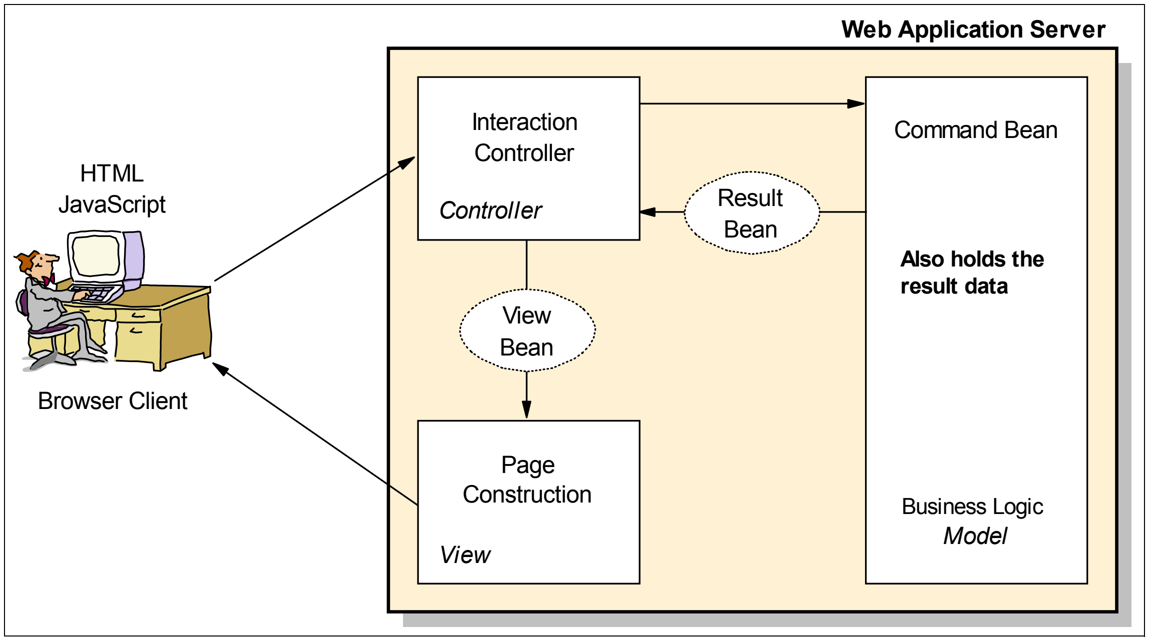


Figure 7-4 Result bean and view bean design pattern

Advantages of result beans

Some reasons to use result beans are:

- Result beans clearly define what the controller expects back from the business logic.
- Once a result bean is defined, the developers of the Controller and the Model can develop their components independently. This simplifies and optimizes the development process and allows for parallel development.
- Since result beans can be serialized, they can be sent to remote servers or received from remote servers, such as EJB-based distributed applications. In addition, they can be stored in a file for persistence purposes.
- The result bean data structure can be reused by multiple business logic objects and interaction controller objects.

7.2.3 View bean design pattern

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

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

Figure 7-4 on page 149 demonstrates the relationship between MVC components, view beans, and result beans. See 7.2.2, “Result bean design pattern” on page 148, for our discussion on result beans.

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

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

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

Advantages and disadvantages of view beans

Some advantages of view beans are:

- Clearly defines all the fields a View (JSP) can display.
- Once a view bean is clearly defined, the developers of the Controller and the View can develop their components independently. This simplifies and optimizes the development process and allows for parallel development.
- The View (JSP) designer can get all the dynamic data from one view bean and use <jsp:useBean> and <jsp:getProperty> tags to insert these values. This allows the JSP designer to concentrate on the look and feel of the page rather than worry about gathering data from various sources (for example, sessions, cookies, result beans, and so on) and coding complex View-specific Java code. View beans effectively hide these complexities from the display page designer.

- Complete separation of the View-specific logic from the business logic, for example, currency conversion based on the user preference.
- Using inheritance can promote view bean reuse and ensure that similar information is received by all users, for example, CSRs and customers.
- View beans can be used with tools such as Rational Developer or IBM WebSphere Studio Application Developer, which allows a developer to insert JavaBean properties into JSPs.
- JSPs and associated view beans can be exhaustively unit tested by the View developer prior to integration with the rest of the application.

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

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

7.2.4 Formatter beans design pattern

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

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

Advantages and disadvantages of formatter beans

Some advantages of formatter beans are:

- Eliminates the need for inserting complex scripting logic inside a JSP.
- Promotes reusability of the formatting logic.
- Hides the complexity of scripting logic from view designers.

- Promotes the ability to drop common information on multiple pages, such as displaying the current weather information and current stock price on all the pages of the Web application.

A disadvantage of formatter beans is that some of the display page functionality that is best expressed in a JSP is now moved into Java code.

7.2.5 Command bean design pattern

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

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

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

Figure 7-5 on page 153 shows how such a command bean interacts with the other components of the Web application.

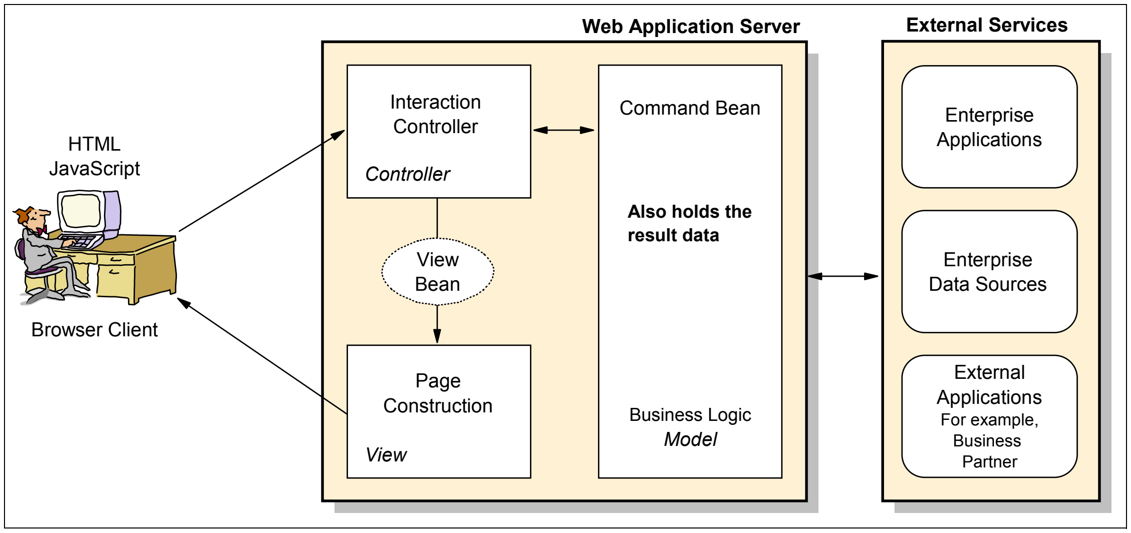


Figure 7-5 Command beans

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

7.2.6 Frameworks

The goal of frameworks is to make it easier to build and maintain Web applications with reusable components. In addition to providing software components, a framework will also defined a design or approach.

For instance, when we discussed the MVC pattern, we detailed a wide variety of possible design alternatives. Without the use of a framework, the project team would have to carefully chose between the different design approaches, and them implement all the components using servlets and JSPs.

Using a standard framework, many of the design decisions are already done, generally following industry best practices. Each framework will define a design approach in implementing the MVC pattern. The framework will also provide reusable components which will accelerate and standardize this implementation.

In this section, we examine the following Web application frameworks:

- JavaServer Faces (JSF) framework
- The Apache Jakarta Project's Struts framework
- IBM's Enterprise Application Development Frameworks for Java (EAD4J)

Although we recommend the use of a framework when implementing the MVC pattern, there are some issues to consider:

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

JavaServer Faces

The JavaServer Faces is a standard J2EE technology, created by Java Specification Request (JSR) 127. Although it is still a relatively new concept, JSFs have wide acceptance by the tool vendors, including the Rational and WebSphere product line.

The main pieces of a JSF application are:

- JSF pages

  JSPs are built from JSF components, and each component is represented by a server-side class.
- Faces servlet

  One servlet (*FacesServlet*) controls the execution flow.
- Configuration file

  An XML file (*faces-config.xml*) that contains the navigation rules between the JSPs, validators, and managed beans.
- Tag libraries

  The JSF components are implemented in tag libraries.
- Validators

  The Java classes validate the content of JSF components, for example, to validate user input.
- Managed beans

  Java beans defined in the configuration file hold the data from JSF components. Managed beans represent the data model and are passed between business logic and user interface. JSF moves the data between managed beans and user interface components.

- Events

  Events are Java code executed in the server for events (for example, a push button). Event handling is used to pass managed beans to business logic.

Now that we have introduced the JSP components, we examine how they fit in to the MVC architectural pattern:

- Model

  Managed beans make up the model of a JSF application.

- View

  JSPs make up the view of a JSF Web application. These JSPs are created by combining model data with predefined (tag libraries) and custom-made UI components.

- Controller -

  The FacesServlet, which drives navigation and object management, makes up most of a JSF application’s controller. Event listeners also contribute to the controller logic.

See Chapter 9, “JSF front-end scenario” on page 239 for a sample implementation of the MVC pattern using JavaServer Faces. In addition, the following can provide more information:

- The official JSF specification

  `http://java.sun.com/j2ee/javaserverfaces/`

- *WebSphere Studio 5.1.2, JavaServer Faces and Service Data Objects,* SG24-6361

Struts

Struts is a Model II servlet-JSP framework offered by the Apache Software Foundation. Struts supports application architectures based on the Model II approach, which is an implementation of the traditional MVC paradigm discussed earlier. Before the emergence of JSF, Struts was the industry standard choice for implementing the MVC pattern. In fact the JSF specification is heavily influenced by Struts.

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

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

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

Figure 7-6 shows an example Struts form bean, TransferFundsForm, that has been defined in the struts-config.xml file and linked to an action mapping. When a request calls for the TransferFundsAction Struts action, the ActionServlet retrieves the form bean (or creates it if it does not exist), and passes it to the action.

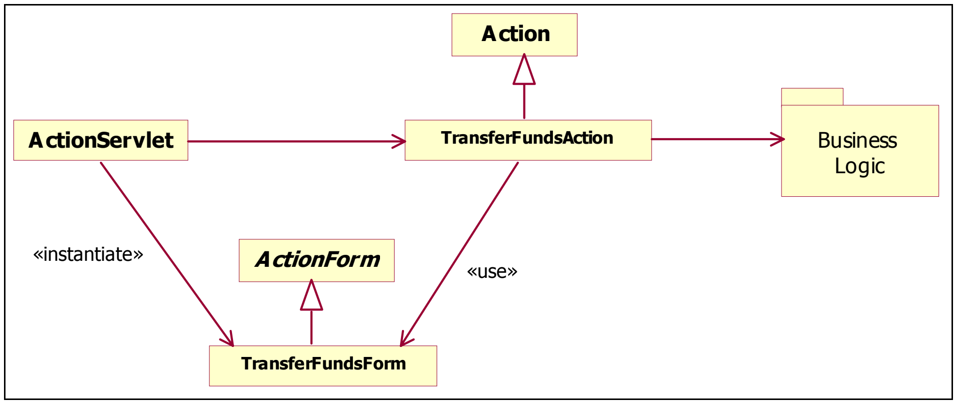


Figure 7-6 Struts action class diagram

The action can then check the contents of the form bean before its input form is displayed, and also queue messages to be handled by the form. When ready, the action can return control with a forward to its output form, usually a JSP. The ActionServlet can then respond to the HTTP request and direct the client to the JSP. Figure 7-7 on page 157 summarizes this sequence.

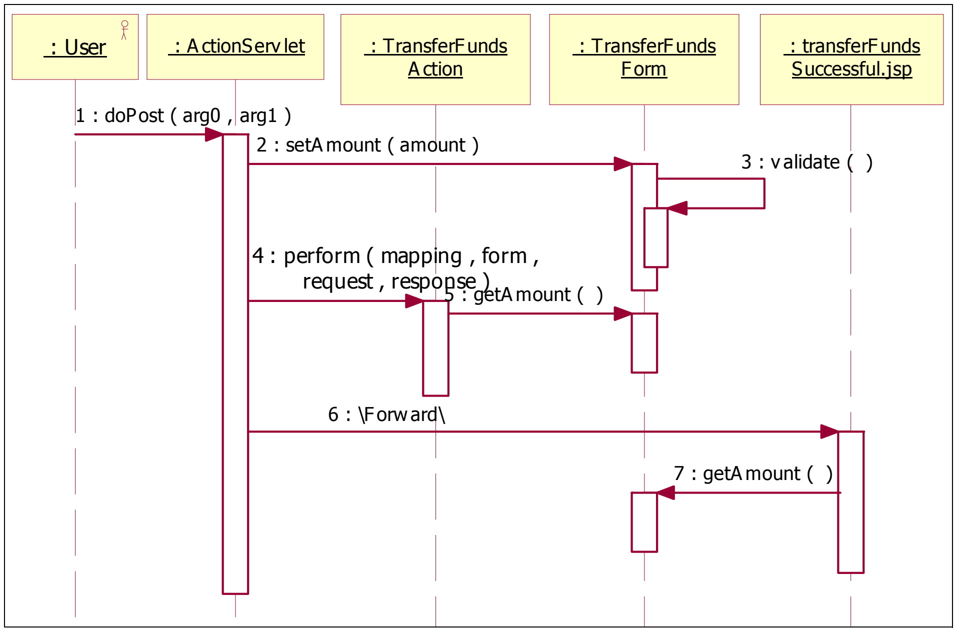


Figure 7-7 Struts action sequence diagram

For more information about Struts see:

- *Legacy Modernization with WebSphere Studio Enterprise Developer*, SG24-6806
- WebSphere Developer Domain for a number of articles on Struts:

  http://www7b.software.ibm.com/wsdd/
- The Apache Jakarta Project Struts Web site:

  http://jakarta.apache.org/struts/

EAD4J

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

EAD4J is comprised of a set of components, each responsible for a different portion of the J2EE space, for example:

- EAD4J.Jade is based on the Model II Servlet-JSP pattern.
- EAD4J.Topaz provide support in the model/persistence areas.

- EAD4J.Opal handles logging.
- EAD4J.Ruby handles the XML/XSLT transcoding issues.

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

7.2.7 WebSphere command framework with EJBs

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

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

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

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

Command model

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

To use commands you must perform the following steps:

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

Optionally, you can perform the next step:

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

The command package

WebSphere V6.0 includes support for the command model as formalized in the command package (com.ibm.websphere.command) and extended to accommodate command shipping (called TargetableCommands). The concept behind command shipping is to intercept the execute method, ship the command to a better execution point (say on a remote server), execute it there, and then ship it back to the caller.

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

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

Advantages and disadvantages of command shipping

The command shipping model has several compelling advantages:

- It is a direct extension of the base command model and therefore maintains the same programming style and tooling advantages.
- It isolates application logic from communication protocols and routing policies. This allows the best protocol to be selected without requiring extensive application changes. Indeed, using techniques such as dynamic class loading, new protocols can be supported *on-the-fly* without the need to change or recompile existing code.
- It supports an agent-oriented service definition model in which the service provider provides a functional interface without consideration for distribution overhead. The service client then defines commands based on the service

interface. The commands are shipped to the service and run there. This allows the service client to control the granularity of remote communication and avoids many of the performance and complexity issues associated with remote interfaces.

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

There are also some disadvantages.

- The simplest implementation of command shipping uses Java serialization to generate the messages that flow between servers. This might hamper the use of a messaging infrastructure such as MQSeries® Integrator, because it is difficult to interpret a serialized bean at an intermediate point. However, command shipping does not dictate an encoding for requests. It is perfectly reasonable to provide a command target that encodes the command in XML, or even SOAP, for transport. Currently this will require per-command encoding logic, but this logic would be required on a per-request basis with any approach.
- The simplest implementation of command shipping uses the same class for both the server-side and the client-side implementation of the command. Thus, if the server-side implementation of the execute method needed to be changed, it would be necessary to redeploy the command class to all clients as well, for example, by including the command classes in a deployed EJB JAR file. Given the goal of agent-oriented service definitions, this does not seem like a serious issue. However, if this is a concern, then a simple dynamic delegation pattern can be followed where the execute method of a command is implemented by delegation to a dynamically linked helper class with classForName. In this way the helper class can be changed at any time with no impact on the client code.

Command caching

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

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

Advantages and disadvantages of command caching

The advantages of command caching are:

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

The disadvantages of command caching are:

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

Command classes

The complete command hierarchy is shown in Figure 7-8 on page 162. This shows the command interface as the base for all commands. Each command has to implement at least the Command interface. When using the base command interface, the command is executed locally in the same JVM as the calling servlet. An application that requires a command to be executed remotely (a shippable command) needs to implement the TargetableCommand and TargetableCommandImpl interfaces. Finally, if a command is to undo the work done by another command, then it must implement the CompensableCommand interface.

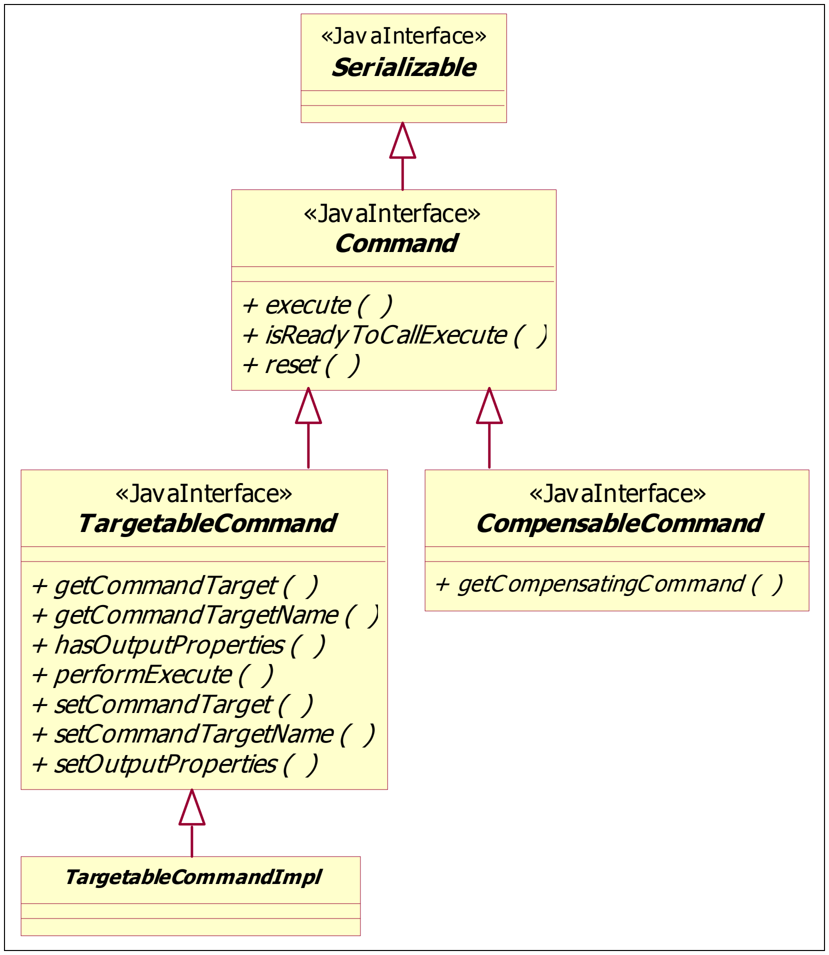


Figure 7-8 Command hierarchy

7.2.8 Best practices for EJBs

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

- Use session beans to represent large-grained tasks in the business process.

  Session beans are used by one client and may or may not have associated properties.

- Use entity beans to represent fine-grained business domain elements.

  Entity beans are shared by many clients and typically have persistent data.

- Use JavaBeans as helper objects to get work done.

  Use JavaBeans as general-purpose utility elements, but avoid using helper objects in create() and finder methods in entity beans. This makes reuse of the bean challenging and deployment difficult.

- Wherever possible use stateless session beans.

  Stateless session beans can be pooled by the EJB container for efficiency, since they do not retain data between client invocations.

- Use session beans as facades to entity beans.

  This prevents a new transaction from being created on every method call, and eliminates the need to code business logic to back out a failed transaction.

- Use Container Managed Persistence (CMP) entity beans in most cases.

  Use Bean Managed Persistence (BMP) for situations requiring complex SQL or relational joins. When using BMP, always use WebSphere data source objects and connection pooling.

- Cache read-only objects in a Singleton JavaBean.

  Cache read-only objects or objects that do not change state on a per-JVM basis by storing them in a Singleton JavaBean. Access these objects using a stateless session bean.

- Use session beans for write-only objects.

  For write-only objects that do not need to be read into memory, such as a log entries, use session beans or consider using asynchronous messaging.

- Cache the EJB home interface.

  Looking up the context, the remote home, and the remote interface is expensive, so cache the home interface to improve performance. Beware of stale handles.

- EJB local versus remote interface

  Only provide a local interface for entity EJBs. This ensures that fine-grained entity access is only available within the JVM. Provide remote coarse-grained entity access using a session bean as a facade.

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

7.3 Design guidelines for Web services

In this section we discuss the following Web services topics:

- Web services architecture
- Web services design considerations
- The key challenges in Web services
- Best practices for Web services

7.3.1 Web services architecture

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

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

As we can see in Figure 7-9:

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

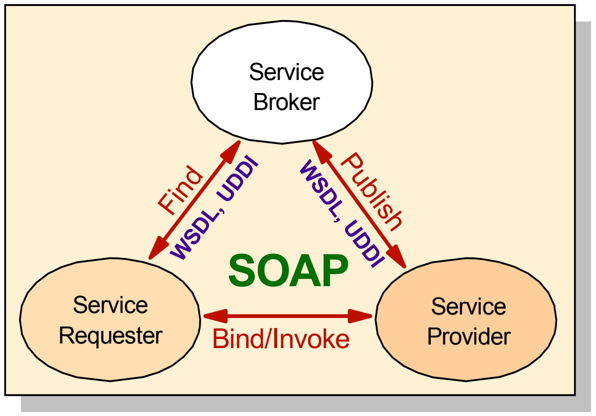


Figure 7-9 Web services roles and operations

We can now look at the building blocks of Web services:

- SOAP
- UDDI
- WSDL

SOAP

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

SOAP has the following characteristics:

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

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

WSDL

The Web Services Description Language (WSDL) is an XML-based interface and implementation description language. WSDL1.1 provides a notation to formally describe both the service invocation interface and the service location.

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

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

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

UDDI

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

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

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

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

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

- Service broker
- Service provider
- Service requester

Service broker

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

- Public registries, such as the IBM UDDI Business Registry and the IBM UDDI Business Test Registry:

  http://www.ibm.com/services/uddi/protect/registry.html
  http://www.ibm.com/services/uddi/testregistry/protect/registry.html

- Private registries such as the UDDI registry provided with IBM WebSphere Application Server

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

Service provider

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

Figure 7-10 on page 167 shows in more detail the application architecture of a Web service provider. Note that we have introduced the concept of a Web service provider bean. This bean can be a simple Java bean, an EJB (stateless session bean) or a simple Java class. This bean is a facade for the actual business logic that is present in the form of business objects. The Web service provider bean exposes methods that might mirror methods in the actual business objects, or the bean might expose methods that call a number of business objects. In any case, think of this as a facade bean for the real business objects. Using this architecture, we do not have to change the existing business logic or business objects in order to create a Web service from the existing enterprise business objects.

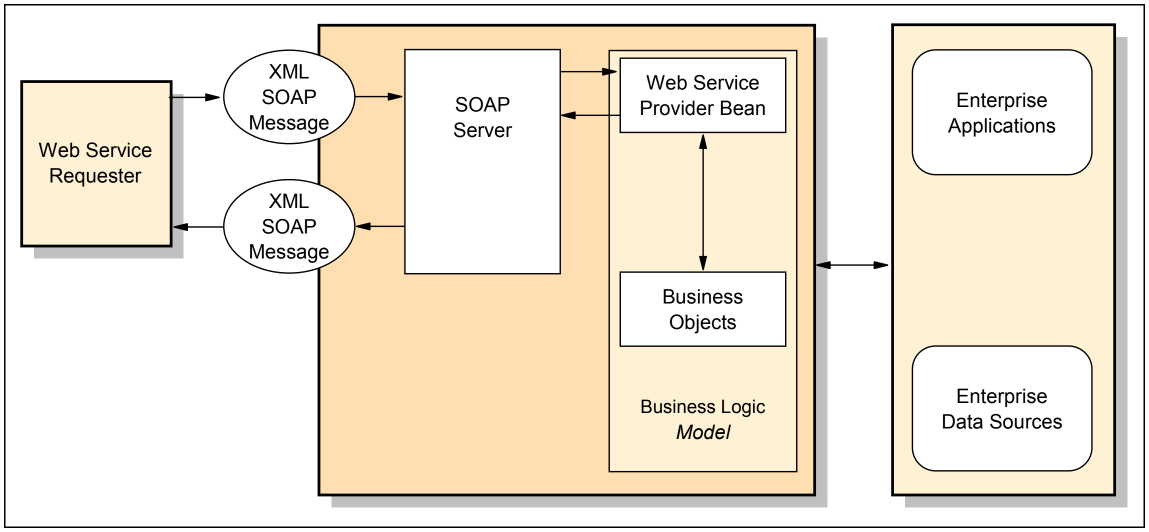


Figure 7-10 Web service provider architecture

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

Service requester

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

Figure 7-11 on page 168 shows the architecture of a Web service requester. Note that the architectural model follows the Model-View-Controller (MVC) pattern, with the servlet as the main component of the controller; the JSP the main component of the View; and the commands and the Web services residing the Model layer. Web services provide a link to another system within the Model layer.

This is considered a best practice for building Web-based applications. We can see that Web services fit very easily into this model.

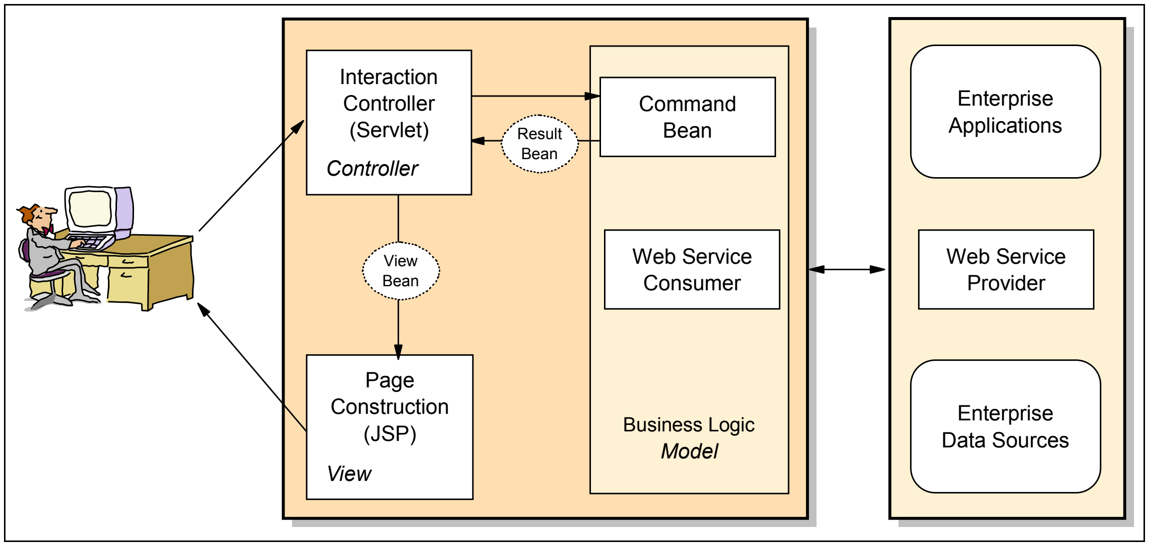


Figure 7-11 Web service requester architecture

7.3.2 Web services design considerations

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

Transmission patterns

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

- Request-response
- One-way
- Solicit-response
- Notification operation

Request-response

The request-response transmission primitive is shown in Figure 7-12 on page 169.

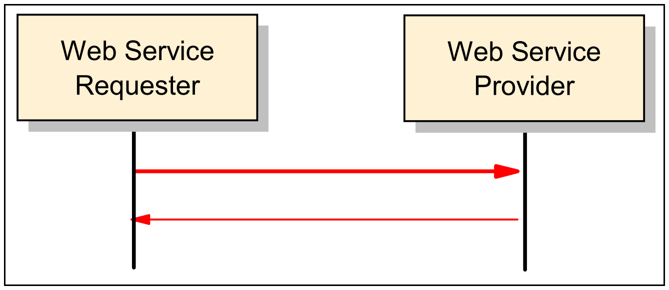


Figure 7-12 Request-response transmission

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

One-way

The one-way transmission primitive is shown in Figure 7-13.

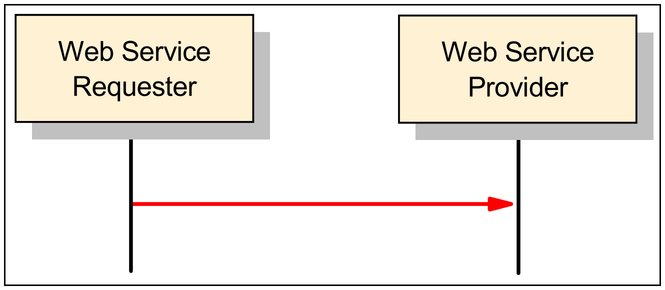


Figure 7-13 One-way transmission

Solicit-response

The solicit-response transmission primitive is shown in Figure 7-14 on page 170.

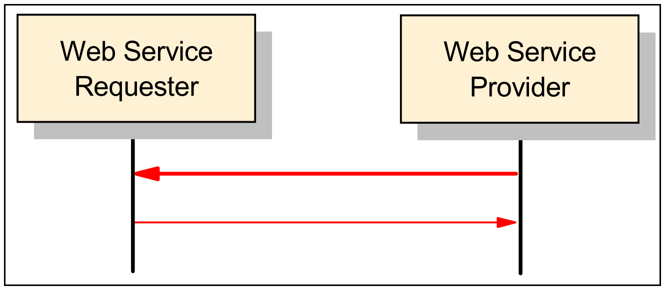


Figure 7-14 Solicit-response transmission

Notification operation

The notification operation transmission primitive is shown in Figure 7-15.

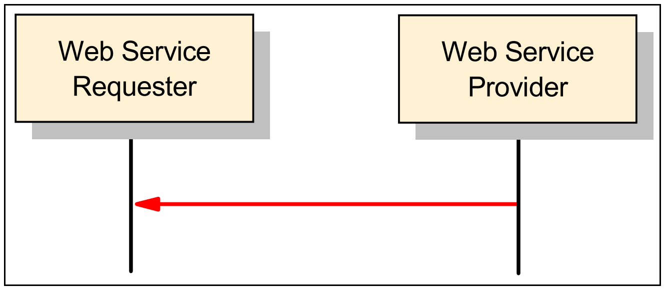


Figure 7-15 Notification operation transmission

SOAP messaging mechanisms

The next design point in architecting a Web service is to choose the SOAP messaging mechanism to use. Figure 7-16 on page 171 shows the two general categories of Web services SOAP messaging mechanisms:

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

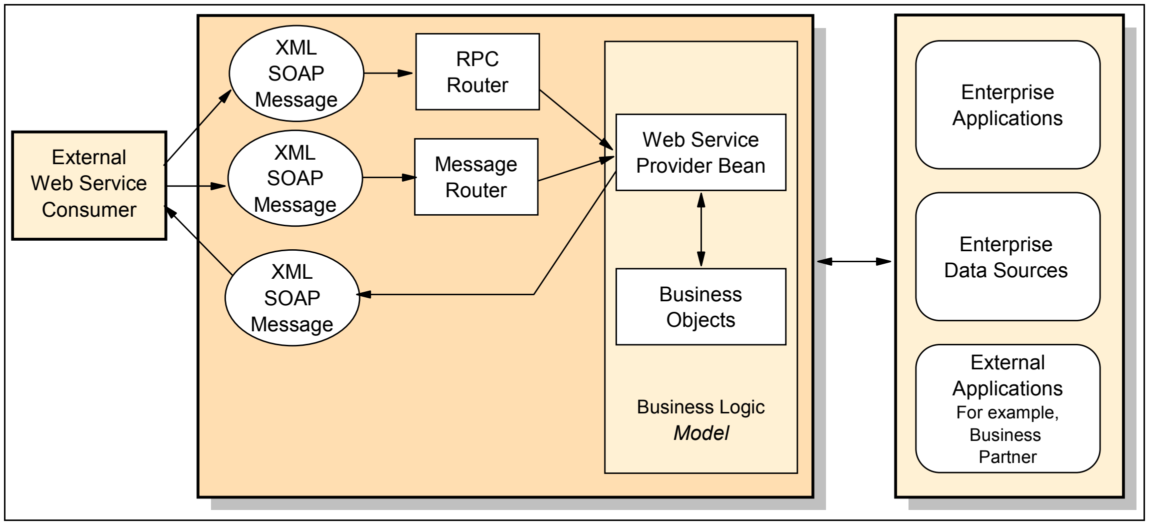


Figure 7-16 SOAP messaging operations

RPC versus message-oriented

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

- SOAP RPC advantage:

  Simpler development:

- SOAP RPC disadvantages

  Requester is too dependent on the availability of the Web service provider

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

- SOAP message-oriented disadvantage:

  Relatively more complex development because it uses assured delivery of asynchronous messages and can require compensating transactions.

Static versus dynamic Web services discovery

Our next design point is to decide if the Web service requester will use static or dynamic discovery of available Web services. The requester has to begin with the WSDL file that describes the interface and implementation specification of the Web service to be invoked. This WSDL file can be retrieved dynamically using a service registry, or statically, as shown in Figure 7-17.

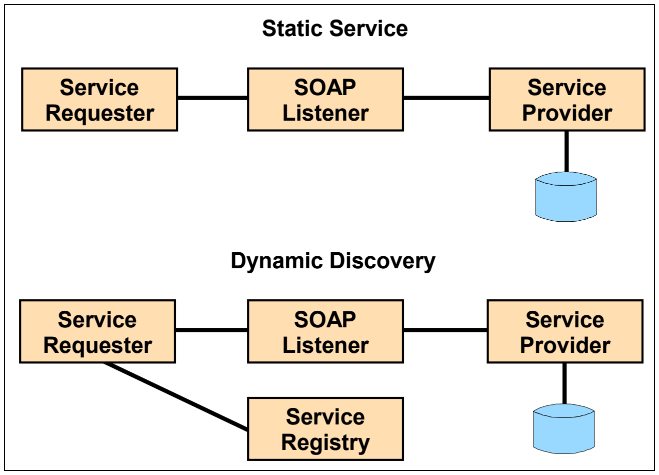


Figure 7-17   Web services discovery methods

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

- Static service

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

- Provider-dynamic

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

- Type-dynamic

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

Message structure

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

SOAP encoding versus literal encoding

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

An alternative is to use literal encoding where the payload of a SOAP message is defined completely by a specific schema, often an XML Schema. Instead of having the Web service and the client agree to follow a set of rules for serializing the data, they agree on the exact format of the data. If the format is described by using an XML Schema, a development tool can read it and provide automatic marshalling of the data from the native language structures into XML. In this case, all the toolkit has to understand is the entire XML Schema specification instead of the combination of the particular encoding rules as well as the chosen type system. The only issue left with using literal encoding is how the tool finds the particular service's XML Schema. This issue is solved by WSDL.

Currently, no machine-understandable standard exists for describing data models for use with SOAP Section 5 encoding, so developers are leaning towards literal encoding with XML Schema. Development tools often provide features such as syntax assistance and data model validation with XML Schema. This may change soon, however, as the Web Services Description working group is considering creating a language to describe SOAP Section 5 data models as part of the WSDL binding for SOAP in WSDL version 1.2.

Synchronous versus asynchronous Web services

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

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

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

Transports and local interfaces

The transports that can be used for Web services communications vary in their capabilities to facilitate the support of asynchronous operations. Thus, it is not only Web services behavior that can be described as either asynchronous or synchronous; the transport used for exchanging Web services messages also falls into one category or the other. Transports whose interfaces inherently support the correlation of response messages to request messages for application use and support a push and pull type of message exchange are often described as being asynchronous transports. Synchronous transports do not provide these facilities and, when used for asynchronous operations, require that the applications (the client and service provider, for the purposes of this discussion) manage the correlation of messages exchanged by not only defining how the correlator will be passed within each message, but by also matching responses with requests. Examples of transports that can be used in support of asynchronous operations are listed in Table 7-1.

Table 7-1 Web services transports

Asynchronous transports	**Synchronous transports**
HTTPR JMS IBM WebSphere MQ Messaging MS Messaging	HTTP HTTPS RMI/IIOP SMTP

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

Correlation ID

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

Return address

In addition there must be an agreed-upon mechanism to identify the return address to which to send the response. You could set up a return address in a profile database or its return address could be part of every request.

The asynchronous transports enable a client to continue processing on its thread of execution immediately after requesting a service invocation. They also provide mechanisms to enable a client to determine the status of its Web service requests, and to retrieve responses to those requests.

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

Asynchronous Web services approaches

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

A typically asynchronous scenario would include the following:

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

We examine the two approaches for asynchronous Web services:

- Decoupled publication-subscription
- Polling

Decoupled publication-subscription

This approach requires partners A and B to be both Web service provider and consumer. They alternate roles. This means both need the SOAP server footprint. Figure 7-18 on page 177 illustrates the steps to implement publication-subscription asynchronous Web services messaging.

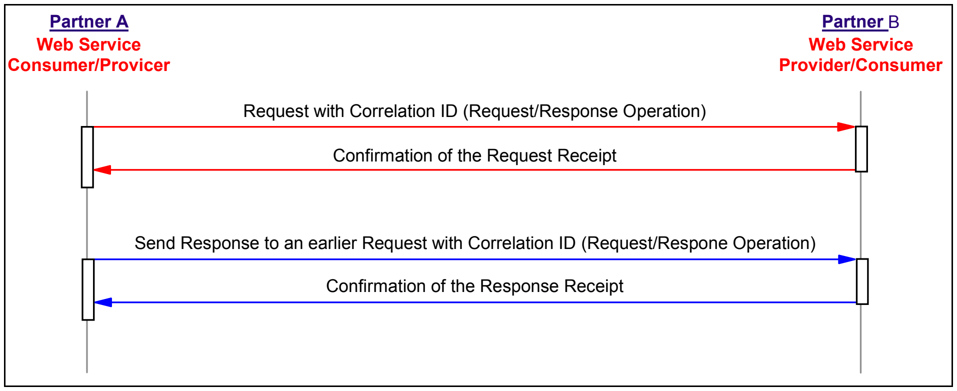

Partner A
Web Service
Consumer/Provicer
Partner B
Web Service
Provider/Consumer
Request with Correlation ID (Request/Response Operation)
Confirmation of the Request Receipt
Send Response to an earlier Request with Correlation ID (Request/Respone Operation)
Confirmation of the Response Receipt

Figure 7-18 Publication-subscription asynchronous Web service approach

Polling

Here the consumer polls on a periodic or event basis to retrieve the response for an earlier request. Partner A remains as a pure Web service consumer and does not need the SOAP server footprint. Figure 7-19 illustrates how to implement a polling approach to asynchronous Web services.

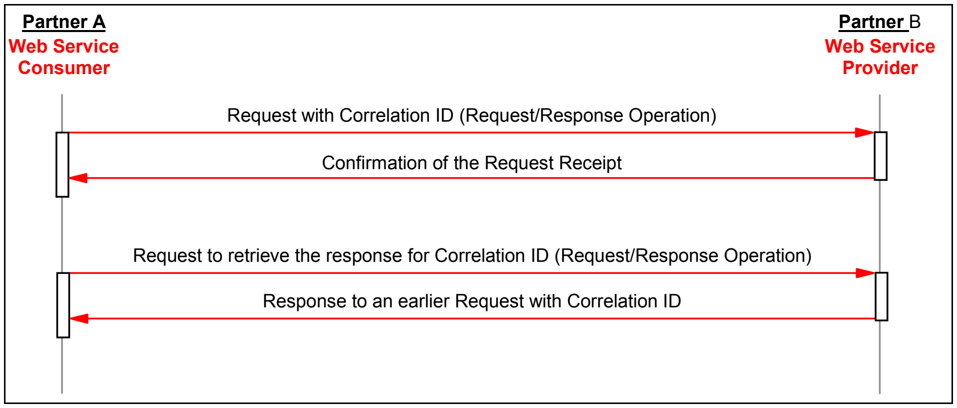

Partner A
Web Service
Consumer
Partner B
Web Service
Provider
Request with Correlation ID (Request/Response Operation)
Confirmation of the Request Receipt
Request to retrieve the response for Correlation ID (Request/Response Operation)
Response to an earlier Request with Correlation ID

Figure 7-19 Polling asynchronous Web service approach

Development strategies for Web service providers

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

- Top-down

  When following the top-down approach, both the server-side and client-side Java code are developed from an existing WSDL specification.

- Bottom-up

  If some server-side Java code already exists, the WSDL specification can be generated from it. The client-side Java proxy is still generated from this WSDL document.

- Meet-in-the-middle

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

  - MIM variant 1

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

  - MIM variant 2

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

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

Level of integration between requester and provider

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

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

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

7.3.3 The key challenges in Web services

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

Extended Web Services Architecture

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

The proposed architecture consists of a basic architecture that defines the interactions between service requesters and service providers, as discussed in 7.3.1, "Web services architecture" on page 164.

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

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

See the W3C Web Services Architecture specification for more detail:

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

Security

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

We further discuss the latest development in Web services security in 10.8.1, "Security considerations for Web services" on page 354.

Interoperability

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

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

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

See the WS-I Basic Profile Version 1.0 specification for more detail:

`http://www.ws-i.org/Profiles/Basic/2002-10/BasicProfile-1.0-WGD.htm`

Quality of Service (QoS)

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

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

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

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

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

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

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

`http://www-106.ibm.com/developerworks/library/ws-quality.html`

Distributed transactions

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

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

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

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

  See the following Web site for the published specification:

  `http://www-106.ibm.com/developerworks/library/ws-coor/`

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

  See the following Web site for the published specification:

  `http://www-106.ibm.com/developerworks/library/ws-transpec/`

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

7.3.4 Best practices for Web services

Web services constitute a distributed computer architecture made up of different computers communicating over a network to form one system. In this section, we focus on best practices for Web services development and deployment within a J2EE environment, that is Web services that are built using servlets, JSP pages, EJB architecture, and all the other standards that are part of the J2EE technology.

Apply distributed computing principles

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

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

- Design systems that are layered

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

- Design coarse-grained Web services

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

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

- Design for loosely coupled components.

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

- Limit dependency on other components.

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

  – Call flow dependency

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

  – Object association dependency

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

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

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

Utilize standard XML structures to pass data

One of the biggest challenges to using Web services effectively is the need to not only pass data, but also the meaning of data between services. The standard currency for Web services in this case is an XML document. Of course, simply using XML is not enough. In order for a Web service to understand the elements (or metadata) of a document, the structure and meaning must be standardized:

- Design components to store metadata as well as data. Understanding the meaning and value of the information passed is critical to implementing any distributed system.
- Manage data within the Web service component responsible for the data. By exposing a business process as a Web service that is available to users within and outside an organization, it is best to make that service as transparent as possible. Web service requesters should not have detailed knowledge of database structures or data formats.
- When designing XML schema to define business-related, more advanced data types, try not to use complex XML schema constructs (such as choice), and decompose them into simple and clean interfaces with primitive data types. Also avoid using specific techniques (for example, INOUT parameter passing) that are not widely supported.
- Whenever possible, provide complete XML Schema and WSDL definitions in one WSDL file rather than importing them from various locations.

Use existing Web services tools

This begins with using standards-based tools for service lookup. While UDDI promises to provide the capability for dynamic lookup of services to call, the reality might be closer to a dynamic binding capability to a known service.

Use an IDE that supports Web services, such as Rational Application Developer. This allows you to expose assets and services using WSDL and proxy-generation tools, shielding you from the underlying XML messages in Web services.

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

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

Use Web scalability principles

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

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

XML performance

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

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

Interoperability

Consider the following interoperability principles:

- Web services should be used when you need interoperability across heterogeneous platforms. In other words, use Web services when you need to expose all or part of your application to other applications on different platforms.
- There will be some issues with vendor compliance, albeit less than CORBA. Beware of vendor-specific extensions.
- Use Apache as your benchmark for SOAP compliance.

- Use low-level XML APIs to adjust SOAP requests to overcome glitches:
  - MS SOAP API
  - JDOM or other Java XML API
- Build sample stubs that use WSDL for all supported platforms to share with partners.

Maintainability

Consider the following maintainability principles:

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

Reuseability

Consider the following reuseability principles:

- Leverage reuse of services by establishing an enterprise-wide service-oriented architecture.
- Reduce the cost and complexity of maintaining multiple, distributed infrastructures:
  - Phase out CORBA and others with Web services where applicable.
  - Expose coarse-grained services for application integration. This approach helps to promote reuse of components.
  - Use CORBA for lightweight, tightly-coupled, fine-grained access to applications if necessary.
- Limit the use of complex types in distributed environments.

Reliability

There are no Quality of Service agreements in Web services or guaranteed delivery. With Web services. you will be accessing services that are not under your control. Therefore, plan for the cases where the service you are requesting fails.

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

7.4 Design guidelines for J2EE Connector Architecture

The J2EE Connector Architecture (JCA) defines a set of standards about how a resource adapter provides connectivity to an enterprise system application (EIS) and how the system contracts with an application server.

There are set of contracts between the resource adapter and the application server which provides services to the resource adapter and maintains a pluggable mechanism when running in the application server.

There are currently two versions of the JCA specification. The JCA 1.0 specification includes outbound communication and JCA 1.5 includes outbound as well as inbound communication.

You can find these specifications here:

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

7.4.1 Components of J2EE Connector Architecture

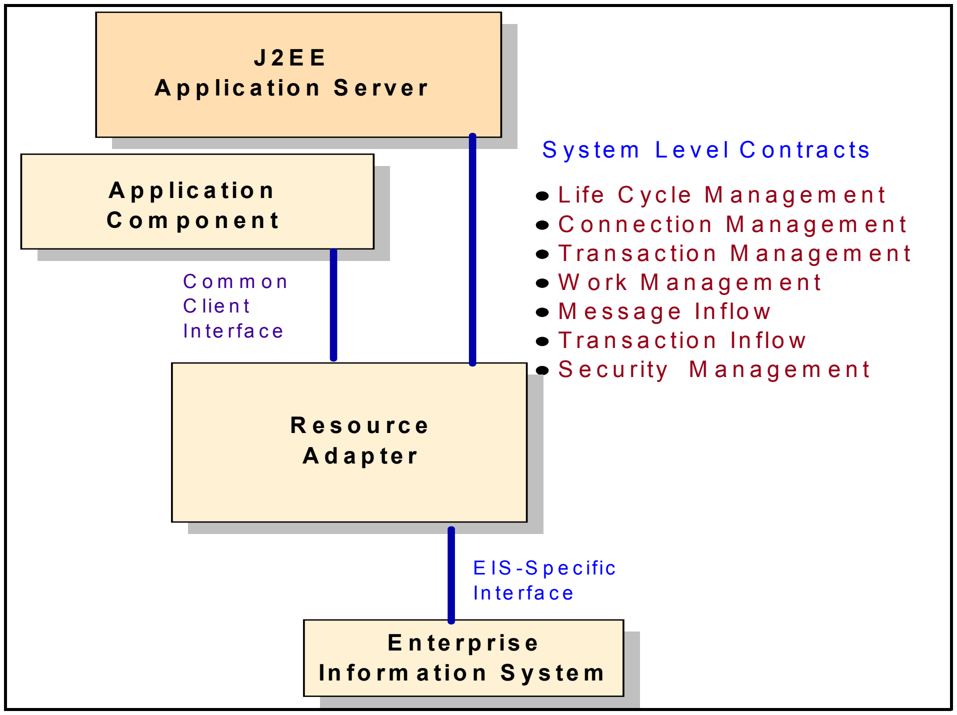


Figure 7-20 J2EE Connector Architecture components

As shown in Figure 7-20 on page 187, Version 1.5 of the J2EE Connector Architecture defines a number of components and interfaces that make up this architecture to connect to any EIS. JCA main components include resource adapter, system level contracts between application server and resource adapter, and the Common Client Interface (CCI).

JCA 1.5 defines the following:

- Common Client Interface (CCI)

  The CCI defines a common API for interacting with resource adapters. It is independent of a specific EIS. A Java developer communicates to the resource adapter using this API.

- System contracts

  These contracts are a set of system-level contracts between an application server and EIS. These extend the application server to provide:

  – *Connection management* enables an application server to pool connections to the underlying EIS and enables application components to connect to an EIS. This leads to a scalable application environment that can support a large number of clients requiring access to an EIS.

  – *Transaction management* enables an application server to use a transaction manager to manage transactions across multiple resource managers. This contract also supports transactions that are managed internal to an EIS resource manager without the necessity of involving an external transaction manager.

  – *Security management* provides support for a secure application environment that reduces security threats to the EIS and protects valuable information resources managed by the EIS

  – *Life cycle management* (new in JCA 1.5) enables an application server to manage the life cycle of a resource adapter. This contract provides a mechanism for the application server to bootstrap a resource adapter instance during its deployment or application server startup, and to notify the resource adapter instance during its undeployment or during an orderly shutdown of the application server.

  – *Work management* (new in JCA 1.5) enables a resource adapter to do work (monitor network endpoints, call application components, and so on) by submitting Work instances to an application server for execution. The application server dispatches threads to execute submitted Work instances. This allows a resource adapter to avoid creating or managing threads directly, and allows an application server to efficiently pool threads and have more control over its runtime environment. The resource adapter can control the security context and transaction context with which Work instances are executed.

  - *Transaction inflow management* (new in JCA 1.5) enables a resource adapter to propagate an imported transaction to an application server. This contract also allows a resource adapter to transmit transaction completion and crash recovery calls initiated by an EIS, and ensures that the ACID (Atomicity, Consistency, Isolation and Durability) properties of the imported transaction are preserved.
  - *Message inflow management* (new in JCA 1.5) enables a resource adapter to asynchronously deliver messages to message endpoints residing in the application server independent of the specific messaging style, messaging semantics, and messaging infrastructure used to deliver messages. This contract also serves as the standard message provider pluggability contract that allows a wide range of message providers (Java Message Service (JMS), Java API for XML Messaging (JAXM), etc.) to be plugged into any J2EE compatible application server via a resource adapter.

  These system contracts are transparent to the application developer. They do not implement these services themselves.

- Resource adapter deployment and packaging

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

For a full description of all of the system contracts listed above, please refer to the J2EE Connector Architecture Version 1.5 specification. All of the above contracts discussed are valid for both managed and non-managed environment.

7.4.2 Managed and non-managed environments

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

- Managed environment

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

- Non-managed environment

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

  `http://www7b.boulder.ibm.com/wsdd/library/techarticles/0109_kelle/0109_kelle.html`

7.4.3 Outbound and inbound communication

JCA specification defines two types of communication (Figure 7-21 on page 191) that a Java application using J2EE Connectors can use to connect and work with EIS are:

- *Outbound communication* allows an application in your J2EE application server environment to initiate a request to another outside system via resource adapter. System contracts defined in the JCA specification such as Connection management and Transaction management provides mechanism for outbound calls.
- *Inbound communication* (new in JCA 1.5) allows you to initiate calls to a J2EE application server environment with a resource adapter from any outside system. System contracts in the JCA specifications, such as transaction inflow management and message inflow, provide the mechanism for inbound calls. Inbound messages delivered through this communication may introduce transaction context into the system.

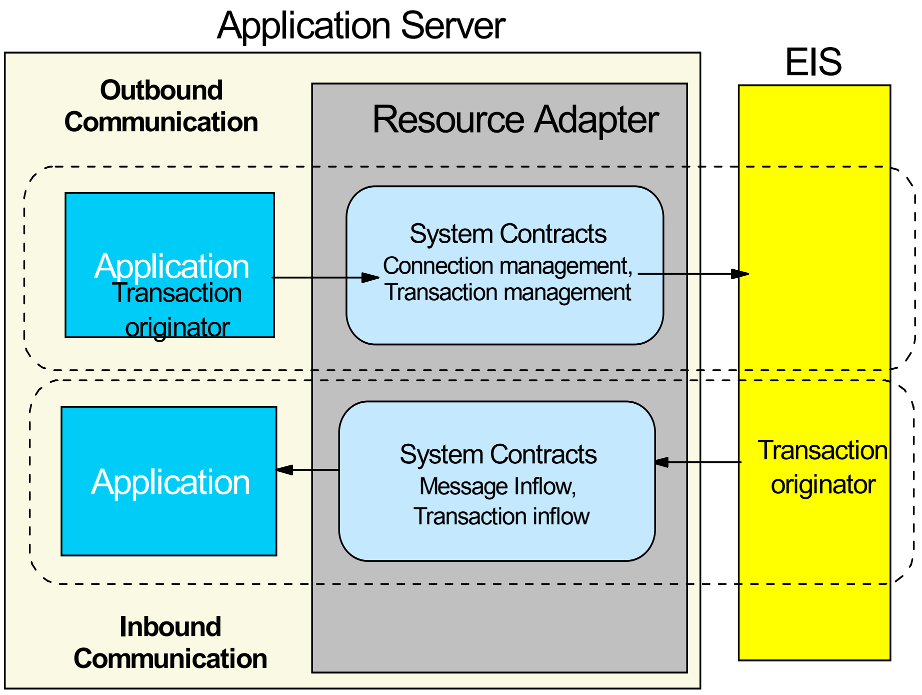


Figure 7-21 Inbound and outbound communication with a resource adapter

7.4.4 WebSphere Application Server and JCA

WebSphere Application Server Connection Management architecture is based on JCA specification for both procedural and relational access to EIS. WebSphere Application Server has two programming model for connection manager, that is, JDBC and J2C. WebSphere Application Server provides its own Relational Resource Adapter (WebSphere Relational Resource Adapter) to make JDBC data source connections managed by the connection manager, which also manages JCA connections for J2EE 1.3 and up specifications. Users still configure JDBC resources as it were a data source which uses this adapter to connect to database. They do not experience any difference in their applications because of underlying use of JCA architecture.

WebSphere Application Server V6 supports both JCA 1.0 and JCA 1.5 specifications.

7.4.5 Common Connector Interface

The Common Client Interface (CCI) defines a standard client API so that application components can access multiple resource adapters, as shown in Figure 7-22. This API can be used directly, or you can use enterprise application integration frameworks to generate EIS access code for the developer. The CCI is designed to be an EIS independent API, so that an enterprise application development tool can produce code for any J2EE-compliant resource adapter that implements the CCI interface. Such tools include the Enterprise Service toolkit of Rational Application Developer and IBM WebSphere Studio Application Developer Integration Edition V5.1.

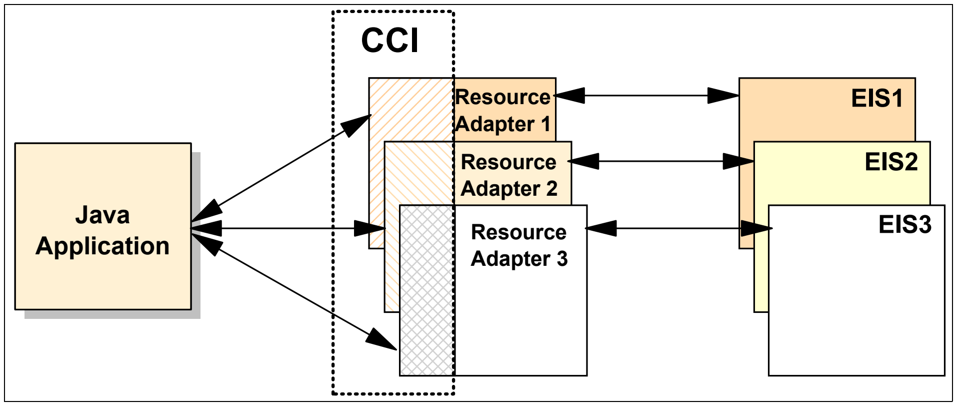


Figure 7-22 CCI with multiple resource adapters

The CCI has the following characteristics:

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

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

- The JDBC API is used to access relational databases.
- The CCI API is used to access EISs which are not relational databases.

The CCI classes can be used directly by the programmer, but are more likely to be used by Enterprise Application Integration (EAI) vendors and application development tools as the interface to resource adapters.

CCI classes

The CCI interface can be divided into the following parts:

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

A resource adapter provides an implementation of the CCI interfaces.

Connection related interfaces

The following information relates to Figure 7-23.

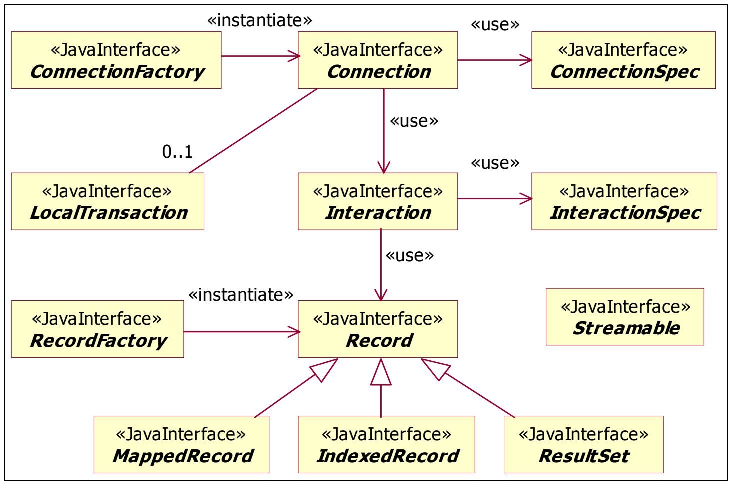

«instantiate»
«use»
«JavaInterface» ConnectionFactory
«JavaInterface» Connection
«JavaInterface» ConnectionSpec
«use»
0..1
«use»
«JavaInterface» LocalTransaction
«JavaInterface» Interaction
«JavaInterface» InteractionSpec
«use»
«instantiate»
«JavaInterface» RecordFactory
«JavaInterface» Record
«JavaInterface» Streamable
«JavaInterface» MappedRecord
«JavaInterface» IndexedRecord
«JavaInterface» ResultSet

Figure 7-23 CCI classes from the java.resource.cci package

The connection interface provides an interface for the connection with an EIS application. The following classes are available in the javax.resource.cci package:

- ConnectionFactory is the interface for getting a connection to the EIS.
- Connection is the application level connection handle used by a component to access the EIS.

- ConnectionSpec is for passing connection-specific properties for the Connection.
- LocalTransaction enables a component to demarcate resource manager local transactions.

Interaction related interfaces

An interaction instance supports the following interaction with an EIS system:

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

Data representation related interfaces

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

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

Metadata related interfaces

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

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

Additional classes

The additional classes consist of exception interfaces.

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

7.4.6 CICS resource adapters

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

The CICS Transaction Gateway (Figure 7-24) consists of the following components:

- The gateway daemon is a long-running process that acts as a server for network-attached Java applications.
- The client daemon provides client-server connectivity.
- The configuration tool provides a graphical interface for configuring the client and the gateway daemon.
- The terminal servlet that allows you to use a Web browser as an emulator for a 3270 CICS application.
- A Java class library containing the three basic interfaces: ECI, EPI, ESI.

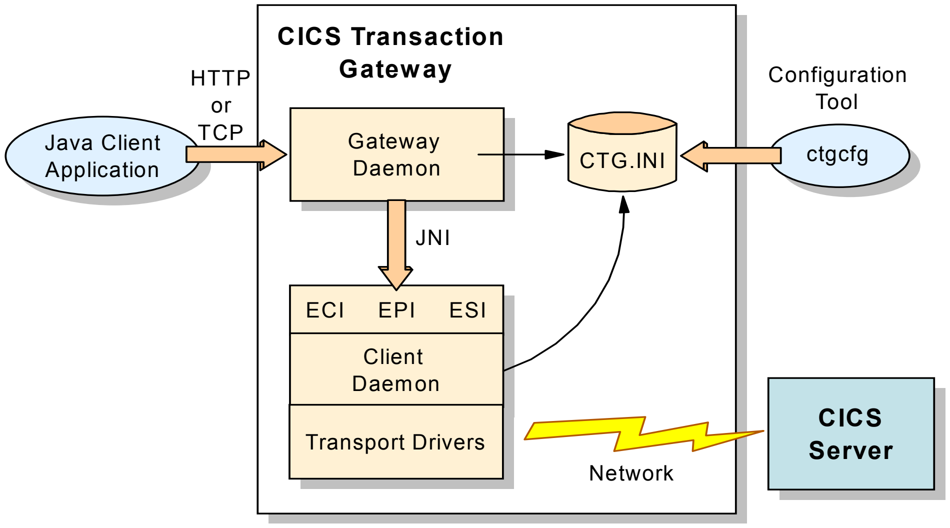


Figure 7-24 CICS Transaction Gateway

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

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

ECI resource adapter

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

EPI resource adapter

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

7.4.7 Selecting a CICS resource adapter

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

- External Call Interface (ECI)

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

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

Table 7-2 shows the characteristics of ECI and EPI.

Table 7-2 CICS ECI and EPI characteristics

Characteristic	ECI	EPI
Protocol type	Remote call	Conversational
Interface	COMMAREA	3270 data stream
Maximum data length	32 KB	24x80 plus control characters
CICS TG JCA support	Distributed or z/OS	Distributed only
Recommendation	Use with new applications or existing COMMAREA-based applications	Use with existing 3270 applications only

7.4.8 CICS ECI design considerations

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

- If your existing CICS application does not use a COMMAREA interface, it must be changed to use COMMAREA.
- The COMMAREA size and interaction complexity

  For performance reasons, the size of COMMAREA and the number of interactions between the Web application and enterprise application should be minimized. The maximum COMMAREA size is 32 KB.
- DPL considerations

  In the CICS world, ECI calls are treated as Distributed Program Link (DPL) calls. Refer to *CICS Transaction Server for OS/390: Application Programming Guide*, SC33-1687 for details on DPL considerations.

7.4.9 Best practices for J2EE Connector Architecture

Some best practices for J2C application developers are:

- Use J2C in a managed environment.

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

- Minimize the resource adapter-specific calls.

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

- Use container-managed transactions.

  WebSphere Application Server is a transaction manager that supports the coordination of resource managers and participates in distributed global transactions with transaction managers that support specific protocols. If there is only a single resource manager in a global transaction then the WebSphere transaction manager will optimize the XA flows using what is called only-agent optimization (such as a one phase commit), providing an automatic performance optimization on the prepare flows. It does not however, provide a mechanism to optimize down to a local transaction.

  The use of container-managed transactions allows you to exploit the QoS provided by the container and resource adapter, and to concentrate on providing the business logic in the enterprise application.

7.5 Design guidelines for JMS

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

- JMS clients are Java programs that send and receive messages.
- Messages are defined for each application and used for communication.
- A JMS *provider* is a message system that implements JMS in addition to the other administrative and control functionality required for a full-featured messaging product.
- Administrated objects are preconfigured JMS objects created by an administrator for JMS clients. There are two types of administrative objects:

– ConnectionFactory is used to create a connection with the provider.
– Destination is used to access a source or destination of messages.

Both are required for the JMS client to send a message.

7.5.1 Message models

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

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

JMS is based on some common messaging concepts which are defined in JMS parent classes. See Figure 7-25. Each messaging domain defines a customized set of these classes for its own domain. There are also classes defined that are transaction aware, such as XAQueueConnection Factory.

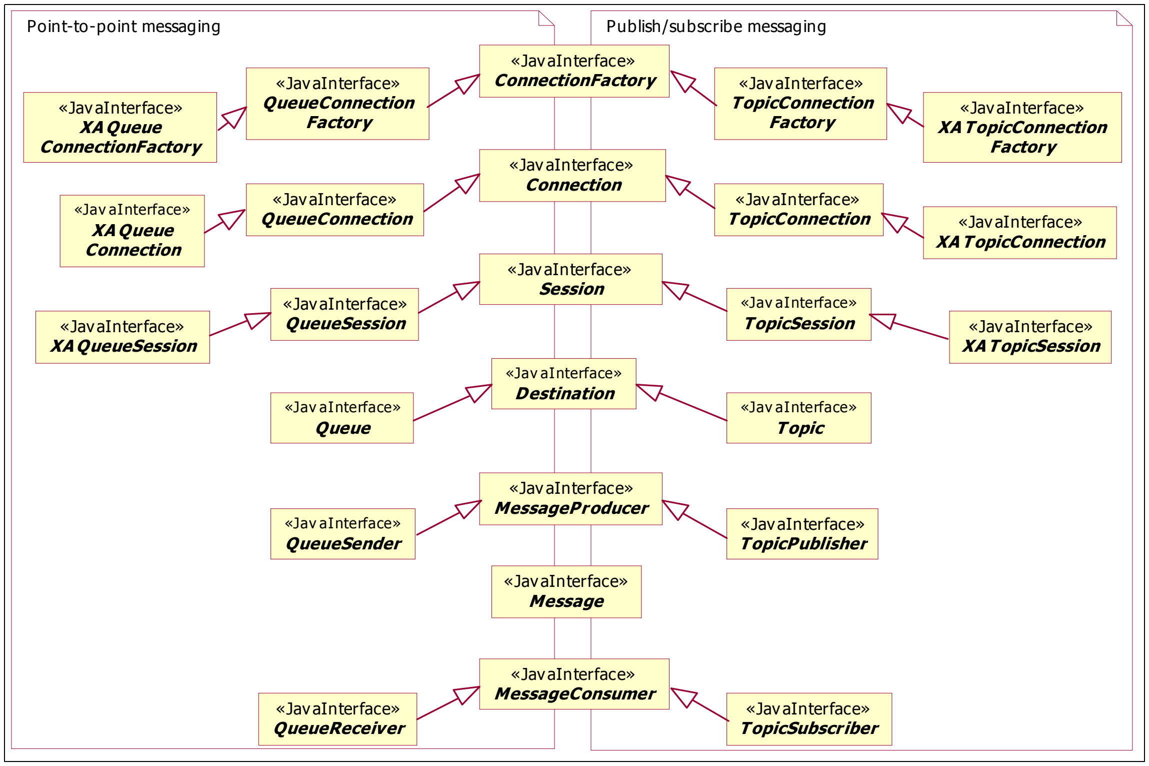

Point-to-point messaging
Publish/subscribe messaging
«JavaInterface» ConnectionFactory
«JavaInterface» QueueConnection Factory
«JavaInterface» XA Queue ConnectionFactory
«JavaInterface» TopicConnection Factory
«JavaInterface» XA TopicConnection Factory
«JavaInterface» Connection
«JavaInterface» QueueConnection
«JavaInterface» XA Queue Connection
«JavaInterface» TopicConnection
«JavaInterface» XA TopicConnection
«JavaInterface» Session
«JavaInterface» QueueSession
«JavaInterface» XA QueueSession
«JavaInterface» TopicSession
«JavaInterface» XA TopicSession
«JavaInterface» Destination
«JavaInterface» Queue
«JavaInterface» Topic
«JavaInterface» MessageProducer
«JavaInterface» QueueSender
«JavaInterface» TopicPublisher
«JavaInterface» Message
«JavaInterface» MessageConsumer
«JavaInterface» QueueReceiver
«JavaInterface» TopicSubscriber

Figure 7-25 JMS classes

The JMS parent classes define the following basic message concepts:

- ConnectionFactory is an administrative object used by a client to create a connection.
- Connection is an active connection to a JMS provider.
- Destination is an administrative object encapsulating the identity of a message destination.
- Session is a single-threaded context for sending and receiving messages.
- MessageProducer is an object created by a Session for sending messages to a Destination.
- MessageConsumer is an object created by a Session for receiving messages from a Destination.

Not all JMS objects can be used concurrently. Table 7-3 shows the objects that can be used concurrently and those that cannot.

Table 7-3 Concurrent JMS classes

Object	**Concurrent use**
ConnectionFactory	Yes
Connection	Yes
Destination	Yes
Session	No
MessageProducer	No
MessageConsumer	No

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

JMS point-to-point model

Point-to-point (PTP) messaging in Figure 7-26 on page 201 involves working with queues of messages. The sender sends messages to a specific queue to be consumed normally by a single receiver. In point-to-point communication, a message has at most one recipient. A sending client addresses the message to the queue that holds the messages for the intended or receiving client. You can think of the queue as a mailbox. Many clients might send messages to the queue, but a message is taken out by only one client. Like a mailbox, messages remain in the queue until they are removed. Thus the availability of the recipient client does not affect the ability to deliver a message. In a point-to-point system, a client can be a sender (message producer), a receiver (message consumer), or both. In JMS, PTP types are prefixed with *Queue*.

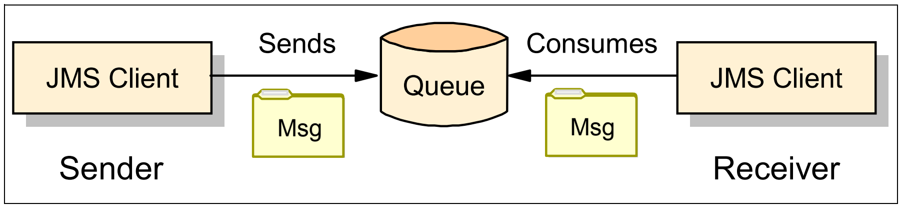


Figure 7-26 JMS point-to-point model

JMS publish/subscribe model

In contrast to the point-to-point model of communication, the publish/subscribe model, shown in Figure 7-27, enables the delivery of a message to multiple recipients. A sending client addresses, or publishes, the message to a topic to which multiple clients can be subscribed. There can be multiple publishers, as well as subscribers, to a topic. A durable (or persistent) subscription, or interest, exists across client shutdowns and restarts. While a client is down, all objects that are delivered to the topic are stored and then sent to the client when it renews the subscription. A non-durable subscription will deliver messages when the consumer is connected, but discard messages when the consumer is not connected. In a publish/subscribe system, a client can be a publisher (message producer), a subscriber (message consumer), or both. In JMS, pub/sub types are prefixed with *Topic*.

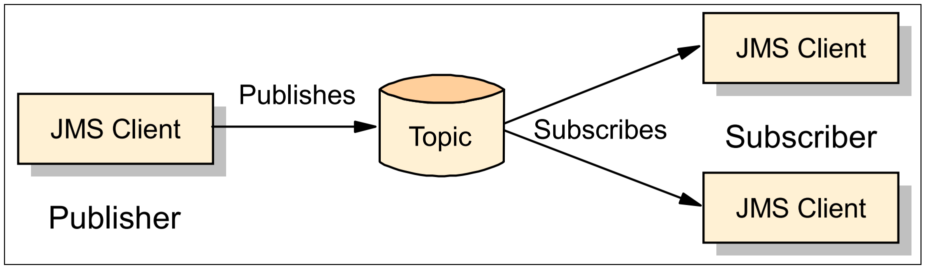


Figure 7-27 Publish/subscribe model

JMS also supports the optional durability of subscribers and remembers that they exist while they are inactive. All an application has to do is send information it wants to share to a standard destination managed by the JMS provider publish/subscribe, and let the JMS provider publish/subscribe distribute it.

Similarly, the target application does not have to know anything about the source of the information it receives.

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

7.5.2 JMS messages

Another design choice to use is JMS message type. JMS messages are composed of the following parts:

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

If you decide to use certain JMS providers, such as IBM WebSphere MQ, then you need to perform a mapping of the JMS message parts so that the message can be delivered by IBM WebSphere MQ, as in Figure 7-28.

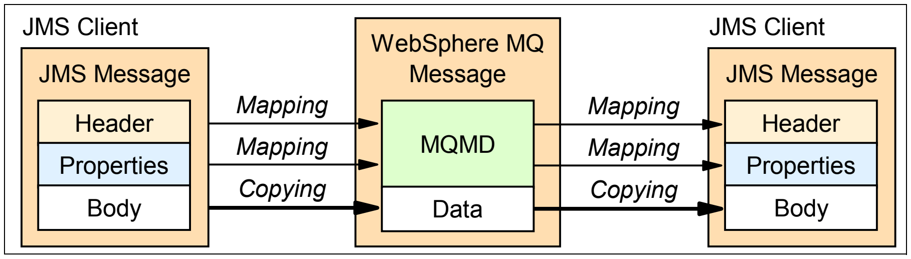


Figure 7-28 Message content mappings using IBM WebSphere MQ

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

The message types that can be used in JMS are:

- BytesMessage contains operations for storing and accessing a stream of bytes.

- StreamMessage contains operations for storing and accessing a stream of Java primitive values. It is filled and read sequentially.
- ObjectMessage contains operations for storing and accessing a serialized Java object. If the application design requires more than one object to be serialized, then use a Collection object.
- MapMessage contains operations for storing and accessing a set of key-value pairs from the message body. The keys must be strings and the values must be primitive types.
- TextMessage contains operations for storing and accessing the body of a message as a string. Text messages can be used to store XML-data. This type of message can be used for sending messages to non-Java applications.

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

- Delivery mode

  When delivery needs to be assured by the business requirements, persistent messages are needed. But when this is not needed, performance can be gained by the use of nonpersistent messages.
- Message expiration

  When using nonpersistent messages, message expiration can be used to discard messages that have remained on a queue or topic for longer than required. This prevents unprocessed messages from building up over time.

7.5.3 Message-driven beans

A *message-driven bean* (MBD) is an asynchronous message consumer. The onMessage method of the message-driven bean is invoked by the container on arrival of a JMS message on a queue. Rather than writing application code to poll for messages on a queue, you can use a message-driven bean instead. The main difference between message-driven beans and other enterprise beans is that message-driven beans have only a bean class. There is no home or remote interface for a message-driven bean. Message-driven beans can only be invoked by the container.

As with all programming models, certain best practices have emerged for using the message-driven bean programming model. These best practices are:

- Delegate business logic to another handler.

  Traditionally, the role of a stateless session bean is to provide a facade for business logic. Message-driven beans should delegate the business logic concerned with processing the contents of a message to a stateless session bean. Message-driven beans can then focus on what they were designed to do, which is processing messages. See Figure 7-29.

An additional benefit of this approach is that the business logic within the stateless session bean can be reused by other EJB clients.

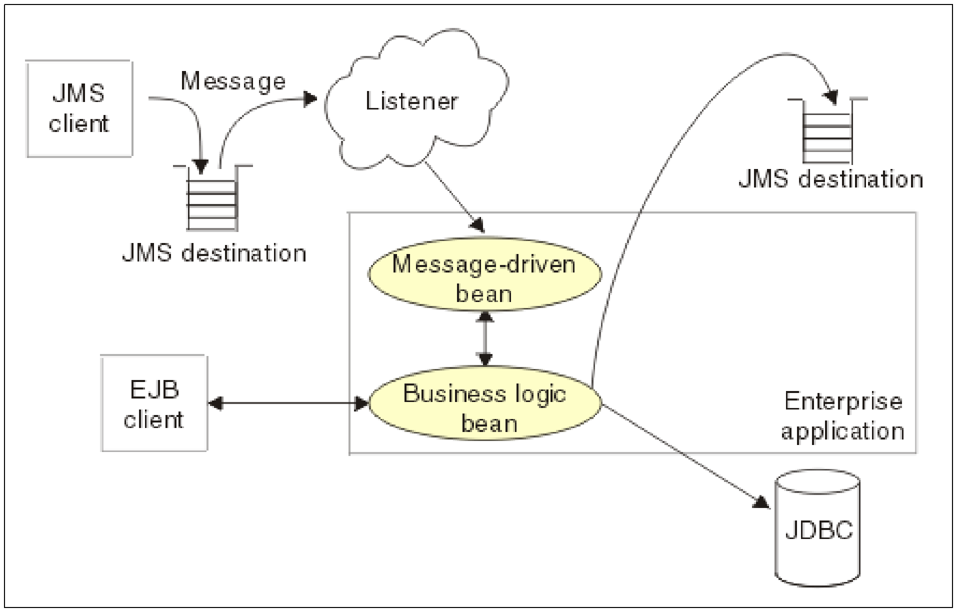


Figure 7-29 Delegating business logic to a session bean

- Do not maintain a client-specific state within an MDB.

  Message-driven bean instances should not maintain any conversational state on behalf of a client. This enables the EJB container to maintain a pool of message-driven bean instances and to select any instance from this pool to process an incoming message. However, this does not prevent a message-driven bean from maintaining a state that is not specific to a client, for instance, data source references or references to another EJB.

- Avoid large message bodies.

  A JMS message probably will travel over the network at some point in its life. It will definitely need to be handled by the JMS provider. All of these components contribute to the overall performance and reliability of the system. The amount of data contained in the body of a JMS message should be kept as small as possible to avoid impacting the performance of the network or the JMS provider.

- Minimize message processing time.

  Instances of a message-driven bean are allocated from a method-ready pool to process incoming messages. These instances are not returned to the method-ready pool until message processing is complete. Therefore, the longer it takes for a message-driven bean to process a message, the longer it is unavailable for reallocation.

  If an application is required to process a high volume of messages, the number of message-driven bean instances in the method-ready pool could be rapidly depleted if each message requires a significant processing. The EJB container would then need to spend valuable CPU time creating additional message-driven bean instances for the method-ready pool, further impacting the performance of the application.

  Additional care must be taken if other resources are enlisted into a global transaction during the processing of a message. The EJB container will not attempt to commit the global transaction until the MDB's onMessage method returns. Until the global transaction commits, these resources cannot be released on the resource managers in question.

  For these reasons, the amount of time required to process each message should be kept to a minimum.

- Avoid dependencies on message ordering.

  Try to avoid having an application making any assumptions with regard to the order in which JMS messages are processed. This is due to the fact that application servers enable the concurrent processing of JMS messages by MDB's and that some messages can take longer to process than others. Consequently, a message delivered later in a sequence of messages might finish message processing before a message delivered earlier in the sequence. It might be possible to configure the application server in such a way that messaging ordering is maintained within the application, but this is usually done at the expense of performance or architectural flexibility, such as the inability to deploy an application to a cluster.

- Be aware of poison messages.

  Sometimes, a badly-formatted JMS message arrives at a destination. Such a message might cause an exception to be thrown within the MDB during message processing. An MDB that is making use of container-managed transactions then marks the transaction for rollback. The EJB container rolls back the transaction, causing the message to be placed back on the queue for redelivery. However, the same problem occurs within the MDB the next time the message is delivered. In this situation, such a message might be received, and then returned to the queue, repeatedly. These messages are known as *poison* messages.

Fortunately, some messaging providers have implemented mechanisms that can detect poison messages and redirect them to a another destination. WebSphere MQ and the default messaging provider are two such providers.

Listener ports versus JCA activation specifications

Message-driven beans can be configured as listeners on a Java Connector Architecture (JCA) 1.5 resource adapter or against a listener port, as in WebSphere Application Server V5.

Using JCA activation specifications

With a JCA 1.5 resource adapter, message-driven beans can handle generic message types, not just JMS messages. This makes message-driven beans suitable for handling generic requests inbound to WebSphere Application Server from enterprise information systems through the resource adapter. In the JCA 1.5 specification, such message-driven beans are commonly called *message endpoints* or simply endpoints.

For JMS messaging, message-driven beans can use a JMS provider that has a JCA 1.5 resource adapter, such as the default messaging provider, part of WebSphere Application Server V6. With a JCA 1.5 resource adapter, you deploy EJB 2.1 message-driven beans as JCA 1.5-compliant resources, to use a J2C activation specification. See Figure 7-30.

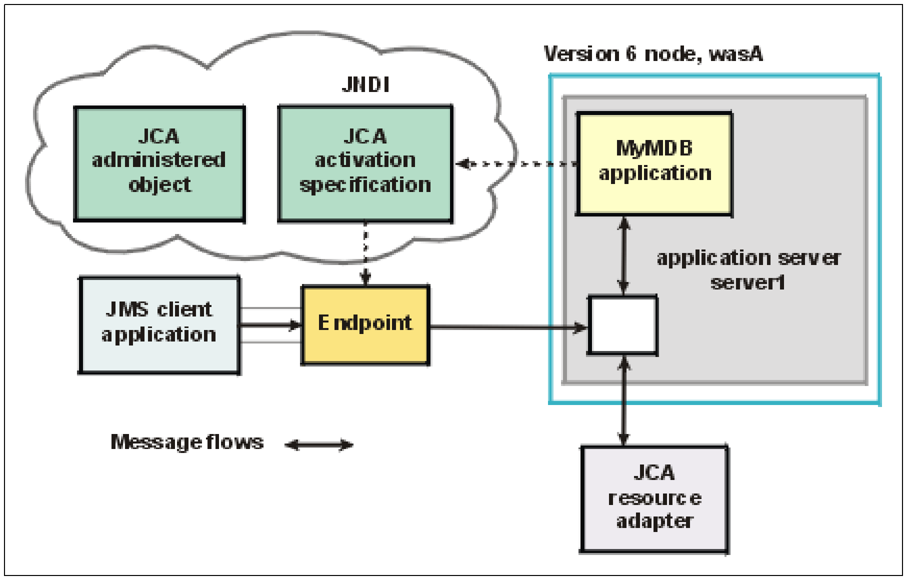


Figure 7-30 Message-driven bean components for an JCA resource adapter

With the SIB JMS Resource Adapter, the default messaging provider listed under JMS providers, a message-driven bean acts as a listener on a specific JMS destination.

Using listener ports

If the JMS provider does not have a JCA 1.5 resource adapter, such as the V5 default messaging and WebSphere MQ, you must configure JMS message-driven beans against a listener port, as in WebSphere Application Server V5. See Figure 7-31.

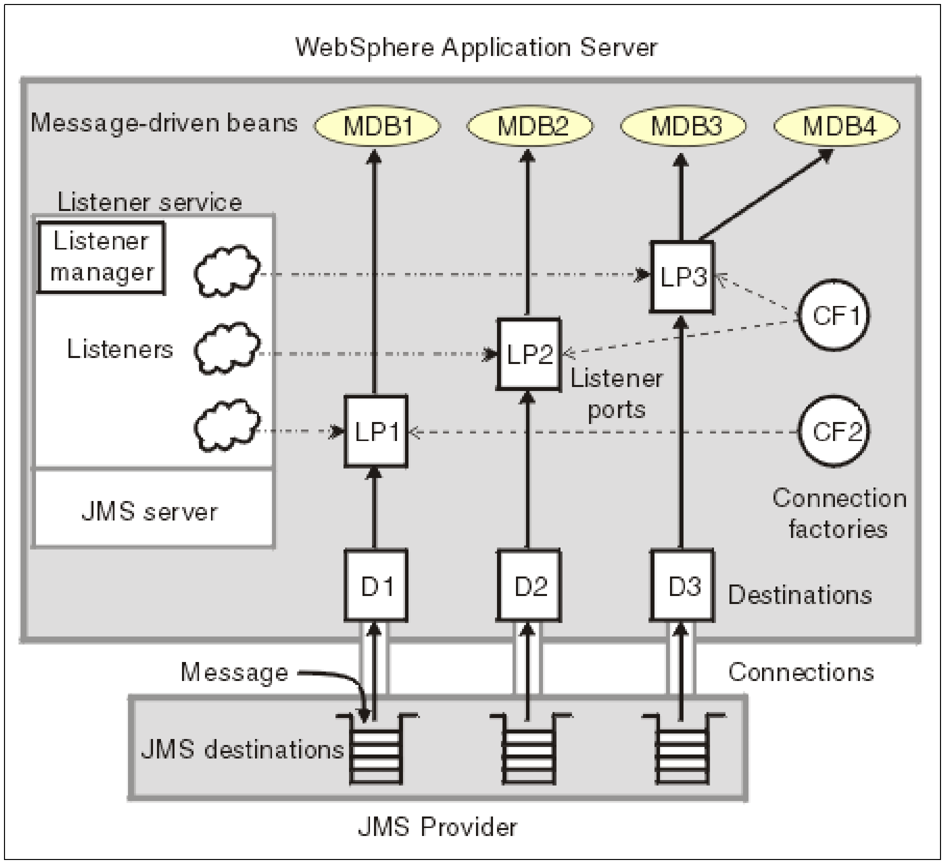


Figure 7-31 Message-driven bean components using listener ports

The message listener service is an extension to the JMS functions of the JMS provider and provides a listener manager, which controls and monitors one or more JMS listeners.

Each listener monitors either a JMS queue destination for point-to-point messaging, or a JMS topic destination for publish/subscribe messaging.

A connection factory is used to create connections with the JMS provider for a specific JMS queue or topic destination. Each connection factory encapsulates the configuration parameters needed to create a connection to a JMS destination.

A listener port defines the association between a connection factory, a destination, and a deployed message-driven bean. Listener ports are used to simplify the administration of the associations between these resources.

When a deployed message-driven bean is installed, it is associated with a listener port and the listener for a destination. When a message arrives on the destination, the listener passes the message to a new instance of a message-driven bean for processing.

When an application server is started, it initializes the listener manager based on the configuration data. The listener manager creates a dynamic session thread pool for use by listeners, creates and starts listeners, and during server termination controls the cleanup of listener message service resources. Each listener completes several steps for the JMS destination that it is to monitor, including:

- Creating a JMS server session pool, and allocating JMS server sessions and session threads for incoming messages.
- Interfacing with JMS ASF to create JMS connection consumers to listen for incoming messages.
- If specified, starting a transaction and requesting that it is committed (or rolled back) when the EJB method has completed.
- Processing incoming messages by invoking the onMessage() method of the specified enterprise bean.

7.5.4 Managing JMS objects

JMS Connection is the first point of access to JMS objects. JMS Connection is created from JMS ConnectionFactory. Once a connection is created, one or more sessions can be created in the context of the connection. JMS Sessions allow you to create message consumers and producers. When consumers or producers are created, the connection needs to be started to receive or send messages. JMS Connections can be cached, similar to the way EJB home objects are cached, and reused by many clients.

JMS Sessions are designed for synchronous access only. A session can only be used by a single client and not shared among other clients. Similarly, an instance of either MessageConsumer and MessageProducer can only be used by a single client. JMS Sessions are opened for the duration of message sending or receiving; after this the session can be closed.

When a session is opened, the correct session acknowledgment must be selected from a performance perspective. In our sample scenario, we selected AUTO_ACKNOWLEDGE. This policy specifies that the message be delivered once and only once. The server must send an acknowledgment back, so the server incurs an overhead to implement this policy. The DUPS_OK_ACKNOWLEDGE setting resends the message until an acknowledgment is sent from the server. The server will operate in a lazy acknowledge mode, thereby reducing the overhead on the server but resulting in an increase in network traffic. With the most overhead of the three settings, CLIENT_ACKNOWLEDGE will cause the server to wait until a request for acknowledgment is sent from the client. Usually the client calls the sent message's acknowledge method.

On completion of interaction with the message producer or consumer (sender or receiver), the session needs to be closed. If the connection is closed, the session belonging to this connection is automatically closed.

The message producers and consumers must also be closed when you finish sending and receiving messages. Again if the connection is closed, the producers and consumers are automatically closed.

Garbage collection of Java cannot be relied upon to clean out objects in a timely manner. It is always a good practice to call the close of any resource-bound object.

For further information, read the JMS specification at:

`http://java.sun.com/products/jms/docs.html`

7.5.5 JMS and JNDI

The Java Naming and Directory Interface (JNDI) API implementation provides directory and naming functionality to programs developed in Java. This allows Java programs to discover and retrieve objects of any type from the JNDI name space.

JMS has two types of administered objects:

- ConnectionFactory
- Destination

An administrator can place objects of these types in the JNDI name space to be accessed by messaging applications.

Figure 7-32 shows the role of JMS and JNDI relative to a Java application. These two APIs sit above any specific service providers and encapsulate any vendor-specific information.

As a result, a developer using these technologies in a messaging-enabled application need only be familiar with the APIs, not the specific messaging systems.

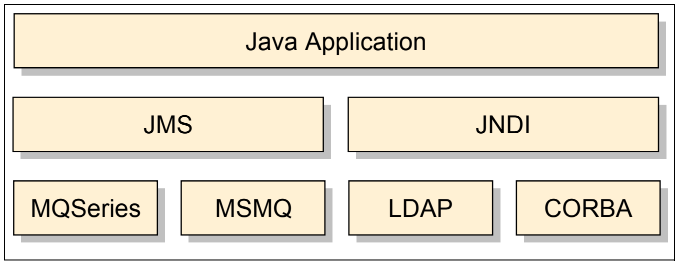


Figure 7-32 The role of JMS and JNDI relative to an application

So, how does an administrator put these objects in the JNDI name space? This step is vendor-specific. If you are using WebSphere MQ V5.3 with WebSphere Application Server, or just the WebSphere default messaging provider, you can administer these objects right from the WebSphere administrative console. If you are using another application server, WebSphere MQ V5.3 provides a tool called JMSAdmin for this purpose.

7.5.6 Choosing a JMS provider

WebSphere Application Server V6 comes with a JMS 1.1 compliant default messaging provider. This provider uses the service integration bus for transport. WebSphere Application Server V6 also provides support for the IBM WebSphere MQ product as the JMS provider, as well as the capability to define and use generic JMS providers.

When deciding which JMS provider suits your environment, consider the following about each provider:

- The following are some reasons why you might choose to use the default messaging provider:
  - You have a requirement for your message-driven beans to handle generic message types, not just JMS messages.
  - You would like to use mediations that act on inbound or outbound messages.
  - You are only connecting J2EE applications hosted by WebSphere Application Server to each other.

- The following are some reasons why you might choose to use IBM WebSphere MQ:
  - Your message-driven beans are written following the EJB 2.0 specification.
  - You want to use WebSphere MQ specific features.
  - Your message-driven beans have to handle JMS messages generated by non-J2EE applications.
  - You want to integrate your solution in a more complex scenario that requires the use of WebSphere MQ, like using WebSphere Business Integration Message Broker.

If you decide to use WebSphere MQ, you could decide to use both options, using the default messaging provider's MQ Links, to communicate both JMS providers. For details about the WebSphere MQ Links, refer to *WebSphere Application Server V6 System Management and Configuration Handbook,* SG24-6451.

7.5.7 WebSphere default messaging provider design considerations

The service integration technologies of IBM WebSphere Application Server can act as a messaging system when you have configured a service integration bus that is accessed through the default messaging provider. This support is installed as part of WebSphere Application Server, administered through the administrative console, and is fully integrated with the WebSphere Application Server runtime.

The JCA 1.5 message inflow management enables a resource adapter to deliver messages asynchronously to message endpoints residing in the application server independent of the specific messaging style, messaging semantics, and messaging infrastructure used to deliver messages. This contract also serves as the standard message provider pluggability contract that allows a wide range of message providers (Java Message Service (JMS), Java API for XML Messaging (JAXM), etc.) to be plugged into any J2EE compatible application server with a resource adapter.

JMS ActivationSpec bean

An ActivationSpec Java bean instance encapsulates the configuration information needed to setup asynchronous message delivery to a message endpoint.

Service integration bus and WebSphere MQ

The service integration bus is the underlying messaging provider for the default messaging provider, replacing the embedded messaging provider that was supported in WebSphere Application Server V5.

If you operate within a WebSphere Application Server environment, sending messages across a service integration bus, you can also exchange point-to-point and publish/subscribe messages with applications in a WebSphere MQ network. The method of exchange uses a component called WebSphere MQ link. The WebSphere MQ link makes exchanging messages very simple by automatically converting them so their characteristics are retained or mapped to similar settings. However, there are some circumstances where the two systems work differently and you can select from the conversion options available.

In most cases messages will flow either from a:

- Service integration bus in a WebSphere Application Server to a WebSphere MQ network
- WebSphere MQ network directly to a service integration bus in WebSphere Application Server

However, you can also send messages between two different:

- WebSphere Application Server service integration buses by way of an intermediate WebSphere MQ network
- WebSphere MQ networks through a service integration bus of an intermediate WebSphere Application Server.

Regardless of the route you implement, the requirements are the same: the requirements of the receiving WebSphere MQ destination must be recognized, and the message transformed to achieve a smooth transfer. This is why the WebSphere MQ link was designed. It handles both styles of messaging familiar to WebSphere MQ programmers: point-to-point and publish and subscribe.

For details about WebSphere MQ Links, refer to *WebSphere Application Server V6 System Management and Configuration Handbook,* SG24-6451.

Message reliability levels (Quality of Service)

Messaging queues and topics are defined as destinations on the bus. It is on a destination that an administrator specifies the default quality of service levels that will be applied when a message producer or message consumer interacts with the destination. An administrator is able to configure a default reliability and a maximum reliability for each service integration bus destination.

- Best effort nonpersistent

  Messages that are sent to this destination are discarded when the messaging engine with which it associated stops or fails. Messages can also be discarded if the connection used to send them becomes unavailable, or as a result of constrained system resources. Messages delivered asynchronously to non-transactional MessageListeners or message-driven beans will not be redelivered if an exception is thrown.

- Express nonpersistent

  Messages that are sent to this destination are discarded when the messaging engine with which it is associated is stopped or if it fails. Messages can also be discarded if the connection used to send them becomes unavailable.

- Reliable nonpersistent

  Messages that are sent to this destination are discarded when the messaging engine with which it is associated stops or fails.

- Reliable persistent

  Messages that are sent to this destination can be discarded when the messaging engine with which it is associated fails, but are persisted if the messaging engine stops normally.

- Assured persistent

  Messages that are sent to this destination are never discarded.

Note: Reliability should be chosen according to your messaging needs. More reliable qualities of service might not perform as well as less reliable qualities of service.

Administrators can also allow message producers to override the default reliability specified on a destination. The mechanism used to achieve this depends on the type of the message producer. For instance, a JMS message producer can use the quality of service properties on the default messaging provider connection factory to map the JMS PERSISTENT and NON_PERSISTENT delivery modes onto the required service integration bus reliabilities.

Note: The reliability specified by a message producer can never exceed the maximum reliability specified on a service integration bus destination. In the case of a JMS message producer, attempting to do this will cause a JMS exception to be thrown to the client application.

7.5.8 WebSphere MQ design considerations

If you have elected to use WebSphere MQ as the messaging provider, there are a few design issues you should consider.

Connection options

A message placed on an IBM WebSphere MQ queue from an application server may originate directly from a servlet, or may be sent from a command bean or EJB. We recommend the latter two methods and not so much from servlets. Regardless of the method, the messages are sent to a queue manager using one of the two available WebSphere MQ Java APIs by IBM WebSphere MQ. Each API has certain characteristics that make it appropriate for a situation, depending on your priorities. However, the API chosen can have an effect on you options for distributing the application components.

The two APIs that we discuss here are:

- The IBM WebSphere MQ for Java Message Service package, com.ibm.mq.jms.jar and com.ibm.jms

  IBM WebSphere MQ for JMS classes implements the J2EE Java Message Service (JMS) interface to enable JMS programs to access a subset of IBM WebSphere MQ features from a vendor-neutral point of view, as defined by the JMS specification. The JMS interface is implemented by a set of IBM WebSphere MQ classes for JMS.

- The IBM WebSphere MQ for Java package, com.ibm.mq.jar

  IBM WebSphere MQ for Java classes enable Java applets, applications, servlets, and EJBs to issue direct calls and queries to IBM WebSphere MQ using specific calls designed to take advantage of IBM WebSphere MQ features.

A JMS Java application uses the vendor-independent JMS interfaces to access the MQ-specific implementation of the JMS classes.

A key idea in JMS is that it is possible, and strongly recommended, to write application programs that use only references to the interfaces in javax.jms. All vendor-specific information is encapsulated in implementations of:

- QueueConnectionFactory
- TopicConnectionFactory
- Queue
- Topic

Coding outside the JMS interface to access WebSphere MQ-specific features will, of course, reduce the portability of the application, because it is now referencing WebSphere MQ-specific classes directly. If application portability, vendor independence, and location transparency are of importance, pure JMS is the obvious choice. JMS uses abstracted concepts of messaging to provide a vendor-independent API to messaging, while underneath lies the IBM WebSphere MQ implementation of the JMS interfaces. The real-world entities that are IBM WebSphere MQ queue managers and queues are accessed by JMS clients through the use of the Java Directory and Naming Service (JNDI). The IBM WebSphere MQ entities are published to JNDI from the WebSphere Administrative Console, or through a tool called JMSAdmin. MQ JMS supports both the point-to-point and publish/subscribe models of JMS.

MQ base JMS classes provide two connection options to IBM WebSphere MQ:

- Bindings mode to connect to a queue manager directly
- Client mode using TCP/IP to connect to a queue manager (not supported on z/OS or OS/390®)

All options support connection pooling.

Java bindings mode

In *bindings mode*, also known as *server connection*, the communication to the queue manager utilizes interprocess communications. One of the key factors that should be kept in mind is that binding mode is available only to programs running on the WebSphere MQ server that hosts the queue manager.

The key connection parameter in this case is the queue manager name.

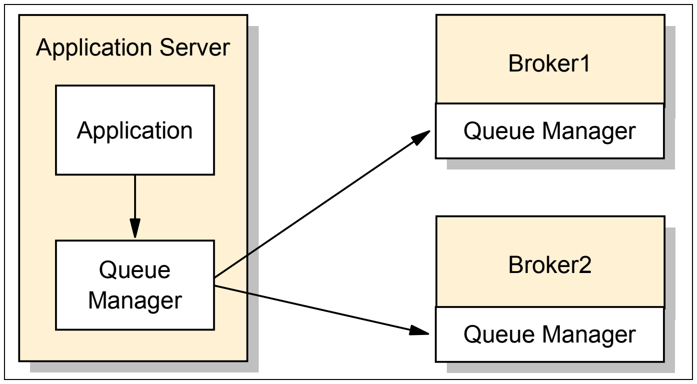


Figure 7-33 Java bindings mode

Connecting to the local queue manager has several major advantages:

- The probability of establishing a connection to a queue manager in your own host is high, as opposed to a connection with a remote queue manager.

- The time taken to establish a network connection to the queue manager is avoided.
- The local queue manager can distribute the work among multiple brokers. If connection performance is a high priority in your network, then using bindings mode is the clear choice.

Using bindings mode, you can also use WebSphere as an XA resource coordinator for units of work that involve WebSphere MQ updates and database updates, for databases and drivers that support the XOpen/XA standards.

Java client mode

Client connection uses a TCP/IP connection to the WebSphere MQ Server and enables communications with the queue manager. Programs using client connections can run on an WebSphere MQ client machine as well as on a WebSphere MQ server machine. Client connections use client channels on the queue manager to communicate with the queue manager. The client connection does not support XA transaction coordination by the queue manager.

Both the WebSphere MQ classes for Java and the WebSphere MQ classes for JMS are needed to use WebSphere MQ client mode. The key connection parameters are host name, TCP/IP port, and server connection channel name. If your code is using only the JMS interfaces (for maximum portability), then client mode cannot be achieved because there are no methods exposed in the JMS interface to select a host or port number.

The client mode is best used when you do not want IBM WebSphere MQ to reside on the same machine as the application server. It allows you to connect directly to a remote IBM WebSphere MQ queue manager.

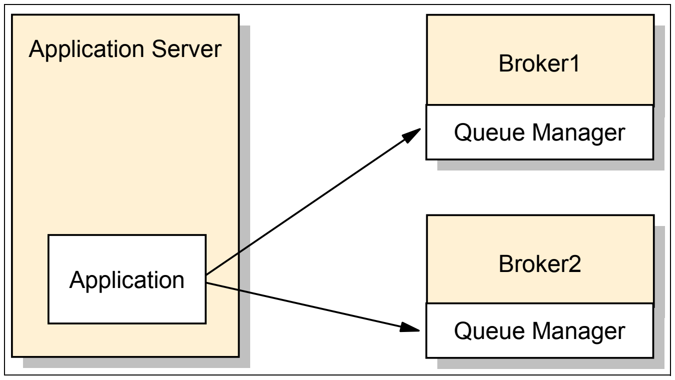


Figure 7-34 Client mode to remote brokers

When you connect directly to a queue manager on a broker, as in Figure 7-34 on page 216, you relinquish any workload distribution the queue manager offers. The application must decide which broker to send the work to and any workload distribution would have to be done in the application itself, which is not recommended. Even having the queue managers in a cluster does not help, because a queue manager will always send the work to the local instance of the broker. Another pitfall is that XOpen/XA facilities for coordinated commits to WebSphere MQ/JMS and databases are no longer available using WebSphere as the transaction coordinator.

One way around this issue is to connect to a remote queue manager that does not have a broker instance, but is there purely for workload distribution, as shown in Figure 7-35.

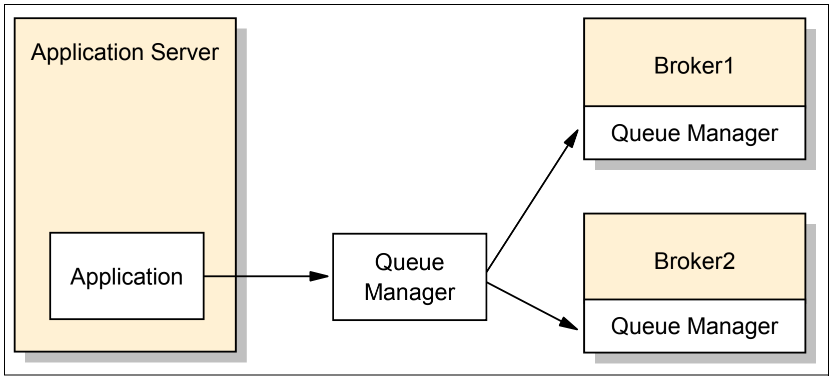


Figure 7-35 Client mode to a remote queue manager

You still have the network connectivity time. In fact, you have made it a little worse by introducing an intermediate system. But you do have the advantage of the queue manager workload distribution and the ability to connect to a remote queue manager. Yet another way would be to use TCP/IP load balancing, but this is not a function of WebSphere MQ.

The client mode can also be used to connect to the local queue manager by passing through the internal TCP/IP stack. This is obviously not as efficient as using the bindings mode, but it does allow your program to be used in a generic environment where you do not know if the queue manager will be local or not. You can also make the different connection options parameter-driven so that the application is ready no matter the connection type. You do need to ensure that the correct parameters are passed so you get the connection type you desire. A database table would be a good place to store these if you are interacting with a database.

Both MQ JMS classes (not the pure JMS interface) and MQ base Java allow you to put messages from the Java application in WebSphere directly into the remote broker's queue. If you are thinking of doing this, you should consider the performance implications. The cost of creating a network connection is added to the total cost of each request. For each request, an IBM WebSphere MQ-to-client session is created. There is no long-lasting network connection. This will impact the ability to run thousands of sessions in parallel. If you create a local IBM WebSphere MQ session for each request, the overhead will be much lower. The network connection is now maintained by a sender-receiver channel pair and is long running. Ideally, long-running IBM WebSphere MQ sessions are preferable.

WebSphere MQ clustering

WebSphere MQ offers the ability to create clusters. MQ clusters provide a number of benefits that JMS applications use silently. Clusters offer:

- Simpler administration of logically related queue managers

  Clustering allows communication between queue managers to promote information about the queues they offer. Once in a cluster, queues on remote queue managers are visible to all queue managers if the queues are defined as cluster queues. The number of explicit definitions within IBM WebSphere MQ administration is reduced with the use of clusters.

- Workload and failover management

  Adding queue managers to clusters allows access to WebSphere MQ workload and failover features.

  As shown in Figure 7-36, QM3 is able to load balance across the queue named ReplyQ, since it is available on both QM1 and QM2. Similarly, if QM1 is disabled, all messages for ReplyQ are routed to QM2.

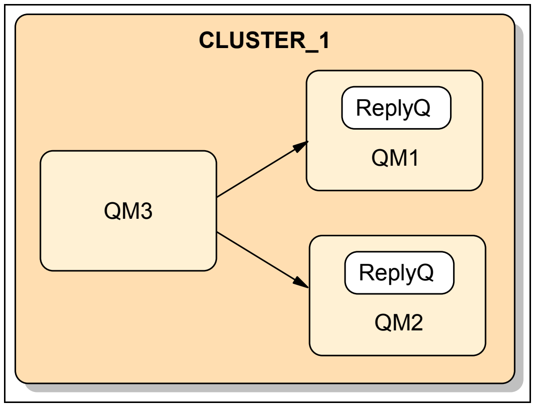


Figure 7-36 Cluster workload management

None of these features can be controlled through the JMS interfaces. However, MQ will automatically utilize the workload and failover under JMS.

These and other features of MQ offer significant benefits and demonstrate that IBM WebSphere MQ is a reliable, scalable, and mature JMS provider.

7.5.9 For more information

These documents and Web sites are further information sources:

- Java Message Service API documentation

  http://java.sun.com/products/jms

- *WebSphere Studio 5.1.2, JavaServer Faces and Service Data Objects,* SG24-6361
- WebSphere Developer Domain article *Integrating IBM WebSphere Application Server and the WebSphere MQ Family*

  http://www7b.boulder.ibm.com/wsdd/techjournal/0110_yusuf/yusuf.html

- WebSphere Developer Domain article *IBM WebSphere and MQSeries Integration Using Servlets and JavaServer Pages*

  http://www7b.boulder.ibm.com/wsdd/library/techtip/pwd/wsmq_integration.html

- IBM Redbook *MQSeries Programming Patterns*, SG24-6506
- IBM Redbook *EJB 2.0 Development with WebSphere Studio Application Developer*, SG24-6819
- *WebSphere MQ Application Programming Guide*, SC34-6064
- *WebSphere MQ Using Java*, SC34-6066

7.6 Design guidelines for the ESB

An enterprise service bus is a concept that can be implemented using a variety of software programs. In the implementation scenarios in this book, the ESB is implemented using the service integration bus in WebSphere Application Server.

7.6.1 Service integration bus

A service integration bus supports applications using message-based and service-oriented architectures. A *bus* is a group of one or more interconnected servers or server clusters that have been added as members of the bus. Applications connect to a bus at one of the messaging engines associated with its bus members.

The service integration bus provides the backbone support for the WebSphere Application Server default messaging provider. It supports sending messages asynchronously (possible whether the consuming application is running or not and whether or not the destination is reachable). Both point-to-point and publish/subscribe messaging are supported.

In addition, the service integration bus Web services enablement support provides the following support for Web services:

- You can take an internal service, define it as a service destination, and make it available as a Web service.
- You can take an external Web service, and make it available at a service destination.
- (Network Deployment only) You can use the Web services gateway to map an existing service - either an internal service, or an external Web service - to a new Web service that seems to be provided by the gateway.

The Web services enablement support is included with WebSphere Application Server but is not installed by default.

The bus appears to its applications as though though were a single logical entity, which means applications only need to connect to the bus and do not need to be aware of the bus topology. In many cases, the knowledge of how to connect to the bus and which bus resources are defined are handled by a suitable API abstraction, such as the administered JMS connection factory and JMS destination objects.

Many scenarios only require relatively simple bus topologies, perhaps even just a single server. You can add multiple servers to a single bus to increase the number of connection points for applications to use. You can also increase scalability, and achieve high availability, by adding server clusters as members of a bus. However, servers do not have to be bus members to connect to a bus. In more complicated scenarios, multiple buses are configured, which can be interconnected to form more complicated networks. An enterprise might deploy multiple interconnected buses for organizational reasons. For example, an enterprise with several autonomous departments might want to have separately administered buses in each location.

The service integration bus introduces a number of new concepts:

- Buses

  A service integration bus within WebSphere Application Server V6 is simply an architectural concept. It gives an administrator the ability to group a collection of resources together that provide the messaging capabilities of the bus. At runtime, the bus presents these cooperating messaging resources to applications as a single entity, hiding from those applications the details of how the bus is configured and where on the bus the different resources are located.

- Bus members

  The members of a service integration bus are the application servers and server clusters within which messaging engines for that bus can run.

- Messaging engines

  A messaging engine is a server component that provides the core messaging functionality of a service integration bus. A messaging engine manages bus resources and provides a connection point for applications.

- Data stores

  Every messaging engine defined within a bus has a data store associated with it. A messaging engine uses this data store to persist durable data, such as persistent messages and transaction states. Durable data written to the data store survives the orderly shutdown, or failure, of a messaging engine, regardless of the reason for the failure.

- Destinations

  A bus destination is a virtual location within a service integration bus, to which applications attach as producers, consumers, or both to exchange messages. To support the default messaging provider, you can define the following types of destinations:

  - Queue destinations
  - Topic space destinations
  - Alias destinations
  - Foreign destinations

  In addition, destinations are created for Web services defined to the bus.

- Mediations

  A *mediation* processes in-flight messages between the production of a message by one application, and the consumption of a message by another application. Mediations enable the messaging behavior of a service integration bus to be customized. Mediations are defined for a destination.

- Foreign buses

  A *foreign bus* is a property of a service integration bus, and represents other service integration buses with which this bus can exchange messages. See Figure 7-37.

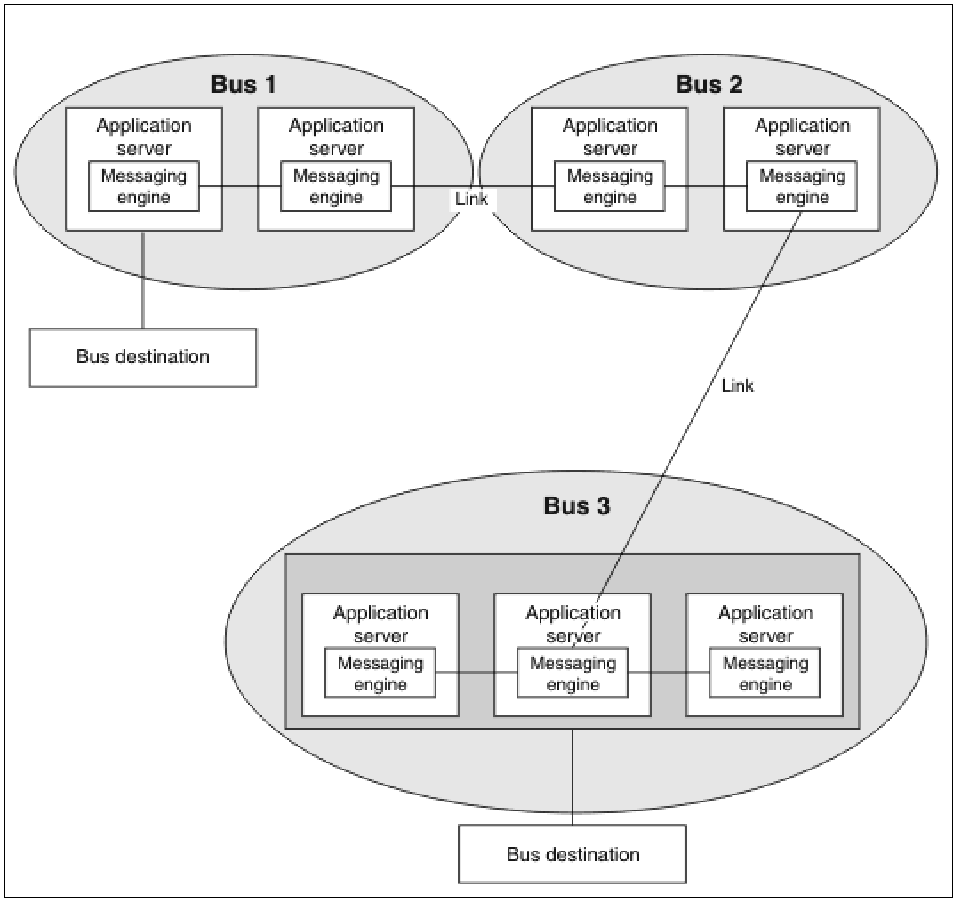


Figure 7-37 Service integration buses in a multiple-bus topology

For details about the bus, please refer to *WebSphere Application Server V6 System Management and Configuration Handbook,* SG24-6451.

7.6.2 Mediations

A *mediation* in the WebSphere Application Server service integration bus processes inflight messages between the production of a message by one application, and the consumption of a message by another application. Mediations provide you with functionality to customize the messaging behavior of the bus. This may include processing such as:

- ► Transforming a message from one format into another.
- ► Routing messages to one or more target destinations that were not specified by the sending application.

- Augmenting messages by adding data from a data source.
- Distributing messages to multiple target destinations.

A mediation is associated with a destination on the bus to create a mediated destination. A mediated destination has two parts: *pre-mediated* and *post-mediated*. Applications send messages to the pre-mediated part, and receive them from the post-mediated part. A mediation receives messages from the pre-mediated part, transforms the messages in some way, and places one or more messages on the post-mediated part. In this way, the mediation controls the progress of messages to their intended target destination.

A mediation can operate within a global unit of work to ensure transactional integrity. Several mediations can operate at the same time to improve the throughput of messages at a destination.

The behavior of a mediation is defined by a mediation handler list. Mediation handler lists contain one or more mediation handlers.

Mediation handlers

A *mediation handler* is a Java program that performs the function of a mediation, and can be deployed in a mediation handler list. The mediation handler class implements the following interface:

```
com.ibm.websphere.sib.mediation.handler.MediationHandler.
```

A mediation handler can have properties that control its behavior.

A mediation handler is packaged for deployment with a supplied EJB. At deployment, you can set properties to control the behavior of the mediation at run-time, and specify membership of one or more mediation handler lists. You deploy a mediation handler as an EAR file and install it into WebSphere Application Server.

Mediation handler lists

A *mediation handler list* is a collection of mediation handlers that are invoked in sequence.

A mediation handler list is a simple pipeline of mediation handlers. The same parameters are passed from one mediation handler to the next mediation handler in the list.

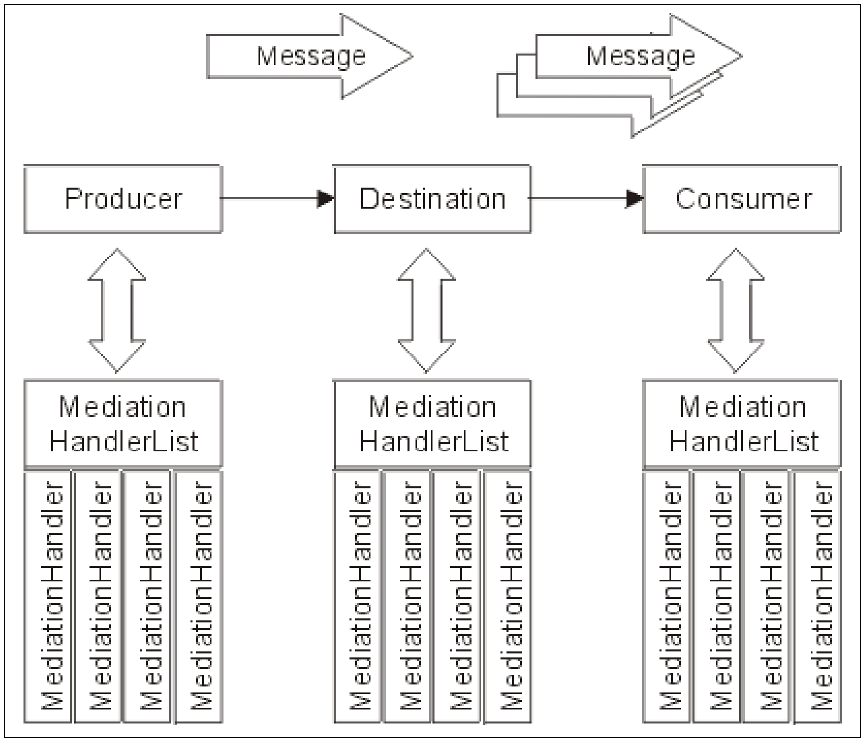


Figure 7-38 Mediation handler lists

A mediation handler list can contain one or more mediation handlers. Membership of a mediation handler list is specified when a mediation is deployed.

When you configure the messaging provider in WebSphere Application Server, you can associate a mediation handler list with one or more selected destinations. This makes the mediation handler list eligible to mediate messages that arrive at the destination.

7.6.3 Working with messages in mediations

This section describes some concepts you need to understand to work with messages in mediations.

Mediation APIs

Several application programming interfaces (APIs) are provided to allow you to work with the message context and code mediations.

- MediationHandler

  This interface defines the method that i the mediation runtime invokes. The method returns boolean `true` if the message passed into this method should continue along the handler list. Otherwise, it returns `false`. The API has just one method handle, `handle()`, that the runtime uses to invoke a mediation.

  In addition to the context information that is passed from one handler to another, it can return a reference to an SIMessage and an SIMediationSession. The SIMessage is the service integration bus representation of the message that is processed by the MediationHandler. The SIMediationSession is a handle to the runtime resources.

- MessageContext

  This interface abstracts the message context that is processed by a handler in the handle method. The MessageContext interface provides methods to manage a property set. The API has two methods:

  – `getSIMessage()` a method to get the service integration bus representation of the message being mediated

  – `getSession()` is a method to get an SIMediationSession object, which is a handle to the core runtime.

- SIMessage

  This interface is the public interface to a service integration bus message for use by mediations. The SIMessage interface has many methods which allow you to work with the message properties, header contents, routing path, metadata, and others.

  In particular, the method `getDataGraph()` returns the SDO data graph which contains the SIMessage content in a tree representation. This method allows you to work directly with the individual fields in the message payload.

  Forward and reverse routing paths define a sequential list of intermediary bus destinations that messages must pass through to reach a target bus destination. A routing path applies the mediations configured on several destinations to messages sent along the path. The methods `getForwardRoutingPath()`, `setForwardRoutingPath()`, `getReverseRoutingPath()`, and `getReverseRoutingPath()` allow you to get and set the contents of the forward routing path and reverse routing path for this SIMessage.

- SIMediationSession

  This interface defines the methods for querying and interacting with the service integration bus. and also includes methods that provide information about where the mediation is invoked from, and the criteria that are applied before the message is mediated.

The API has these methods:

- `getBusName()` returns the name of the bus upon which the mediation is associated.
- `getDestinationName()` returns the name of the destination with which the mediation is associated.
- `getDiscriminator()` returns the discriminator that is defined in the mediation definition.
- `getMediationName()` returns the name of the mediation that is being executed.
- `getMessageSelector()` returns the message selector that is defined in the mediation definition.
- `getMessagingEngineName()` returns the name of the messaging engine from which the mediation was invoked.
- `getSIDestinationConfiguration()` returns the SIDestinationConfiguration object associated with the destination that is specified by destinationName or destinationAddress.
- `receive()`, which receives an SIMessage from the service integration bus.
- `send()`, which sends a copy of an SIMessage to the service integration bus in addition to the message that is returned by the message interface.

SDO DataGraphs

A message published in one format, a Web services SOAP message for exampe, can be routed to a service provider that requires another format, such as Java beans, using the Java API for XML-based RPC (JAX-RPC). Equally, the routing could be in the other direction. If the message is operated on by a mediation as it passes through the bus, in either direction, the mediation must be able to operate on the message regardless of the underlying format. This is achieved by using a common message model for the data mediators to use. The model is called SDO DataGraph and it gives an abstract view of the message, allowing you to concentrate on the information being conveyed (such as the parameters of the request, the data of the response) without having to worry about the packaging of that information.

SDO is based on the concept of data graphs. In the data graphs architecture, a mediation retrieves a *data graph,* a collection of tree-structured or graph-structured data objects, from a message, transforms the data graph, and applies the data graph changes back to the data source.

In general, graphs that are generated from messages form a tree structure. The service presents a standard SDO data graph representation of the message payload, whatever the format of the incoming message‘s payload. A data object holds a set of named properties, each of which contains either a primitive-type value or a reference to another Data Object. The Data Object API provides a dynamic data API for manipulating these properties.

Routing paths

A *routing path* defines a sequential list of intermediary bus destinations that messages must pass through to reach a target bus destination. A routing path is used to apply the mediations configured on several destinations to messages sent along the path.

A forward routing path identifies a list of bus destinations that a message should be sent to from the producer to the last destination from which receivers retrieve messages. The reverse routing path is constructed automatically for request/reply messages, and identifies the list of destinations that any reply message should be sent to from the receiver back to the producer. Use of reverse routing path enables a reply message to take a different route back to the producer, and therefore have more mediations applied.

When a message arrives at a destination in the path, mediations can manipulate the entries in the forward routing path, to change the sequence of destinations through which messages pass. If a mediation manipulates the forward routing path, and the reverse routing path has been set (for a request message that expects a reply), then the mediation is responsible for making any corresponding changes to the reverse routing path.

A destination without mediations can be included in a routing path to provide a future option to apply a mediation assigned to that destination.

8

Business scenario and design

The concepts in this book are illustrated through the use of a fictional scenario for an imaginary company called ITSOMart. This chapter gives you an overall view of the scenario and application design we follow throughout the samples. It will also discuss why we chose the business and application patterns used, the technology options chosen, and the products selected for implementation.

8.1 ITSOMart overview

ITSOMart is a well established grocery chain that has been operating for the past 40 years. The target customers are the high income group. ITSOMart focuses on higher margin, luxury, and speciality products. It has 1000 stores nation-wide.

The business is currently geared toward two distinct customer types:

- Business customers
- Home residential customers

In the future, they would like to have additional lines of business, starting with Institutional service.

Product types, quantities, marketing strategy, and delivery services differ depending on the customer type. Although there is one common warehouse, the company has created two divisions to handle these customer types. Each division has its own account database with information specific to the customer.

8.1.1 Business goals

Market research has shown that there is a growing demand in the high income group for full-service, online home shopping. ITSOMart wants to capitalize on this demand by taking their store services and delivery online.

ITSOMart wants to put the Customer Management and Order systems online and make them accessible over the Web. In the process, they want to use, rather than replace, their significant investment in the existing CRM. They would also prefer to use a third-party credit rating service rather than implementing their own.

They anticipate doing this in several phases, starting with the online customer registration capability.

8.2 Customer registration scenario

The first step ITSOMart wants to take in making their services online is to allow customers to register online. They would prefer for potential customers to access the ITSOMart Web site and to register by entering data that can be used to contact the customer and to get an initial credit rating for them. The data will be stored in an existing system and an account number will be assigned.

8.2.1 Actors

In this scenario, there is one primary actor called Customer. Table 8-1 provides details on this actor.

Table 8-1 Customer Registration actors

Actor Name	Customer
Brief Description	Customer uses the self-service online Customer Management service to create a new account
Status	Primary
Relationships	
Association to use cases	001 Customer Registration

8.2.2 Use case

The Customer Registration use case is shown in Table 8-2.

Table 8-2 Customer Registration use case

Use Case Name	001 Customer Registration
Use Case overview	Customer enters the site and asks for the registration page. Once presented with the page, Customer enters the requested information and submits it. A credit check is performed on the customer. If the credit rating is acceptable, the Customer Management service is invoked to register the customer. The appropriate Delivery services, as selected by the customer, are invoked to assign account numbers. On completion, an e-mail is sent to the Customer indicating the status of the registration (success or failure). If the registration is successful, the account number or numbers are returned.
Preconditions	The Customer supplies the information required to create a new customer account and selects one or more account types (business, home, or both).
Termination Outcome 1	The Customer Management service registers the customer, creates the appropriate accounts, and sends a confirmation e-mail.

The use case model is shown in Figure 8-1.

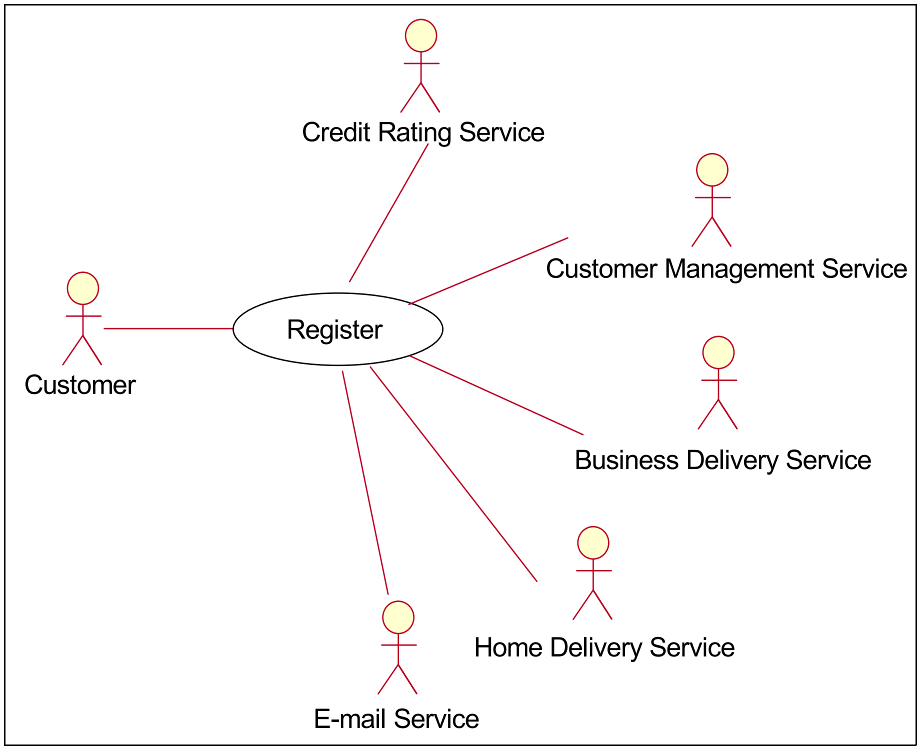


Figure 8-1 Customer Registration use case model

8.2.3 Self-Service pattern selection

The Self-Service business pattern addresses the following problem:

Organizations have services and capabilities that need to be surfaced for interested parties to access and manipulate information relevant to them. The organization or business needs to enable the interested party to work with this information.

This describes the situation for the ITSOMart customer, so they know they should focus on the Self-Service patterns. However, the requirements for the application must be defined before determining the application patterns that will be used.

8.3 Customer registration application design

The application design for the customer registration scenario is illustrated in this section by an activity diagram and a sequence diagram, followed by a step-by-step explanation. These diagrams were created using the modeling tools available in Rational Software Architect.

8.3.1 Activity diagram

The activity diagram, shown in Figure 8-2, is a UML diagram that provides a view of the behavior of a system by describing the sequence of actions in a process. ITSOMart created this diagram at the beginning of the project to clarify the flow of events required to accomplish their objectives.

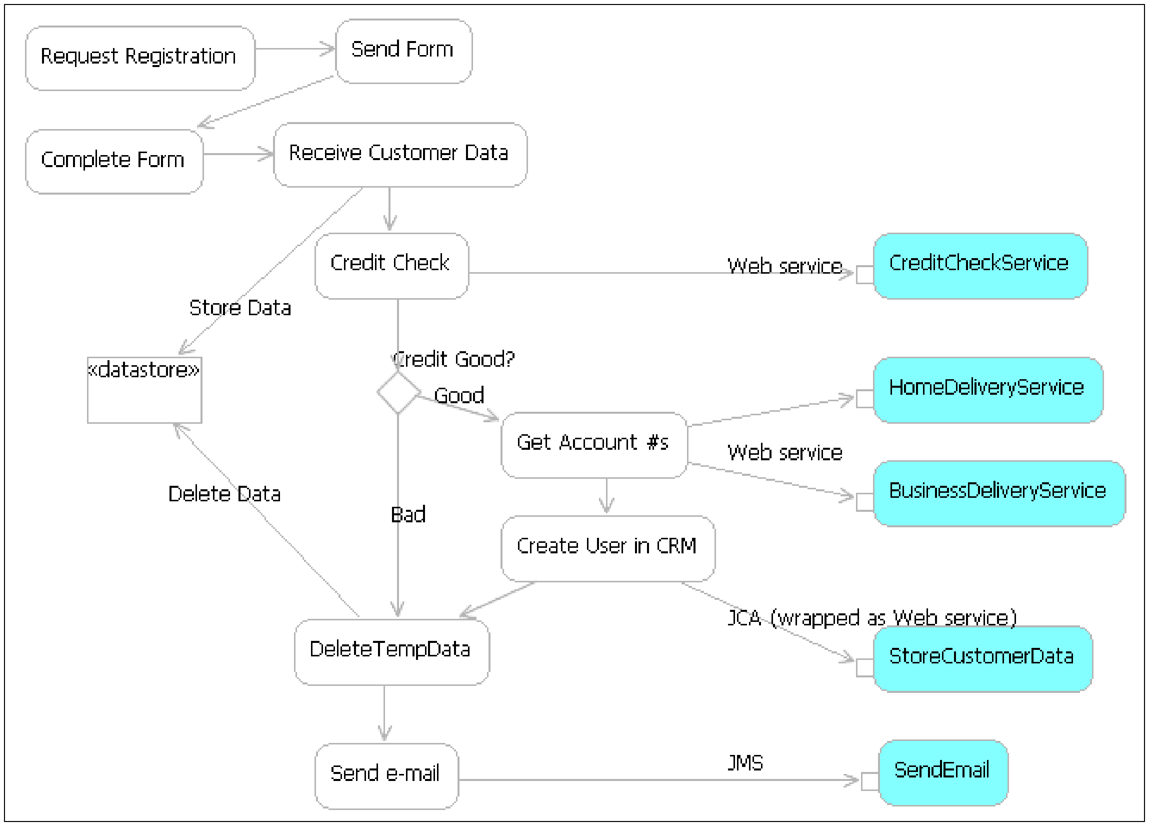


Figure 8-2 Activity diagram of ITSOMart application

Application pattern selection

The activity diagram makes the requirements of the application a little clearer. At this stage, we can see that several application patterns might be appropriate.

- Directly Integrated Single Channel

  There is one service provider for the credit check action and one for the e-mail action. These services will be accessed using a direct connection.

- Router

  The customer will have the opportunity to select the product and delivery type in which they are interested. The application will examine the choice selected by the customer and route the request to the appropriate application. The application will assign an account number to the new customer.

- Decomposition

  In the event that the customer selects multiple types, the application will send that request to each selected business division and collect the account numbers before responding to the client.

8.3.2 Sequence diagram

ITSOMart next used a sequence diagram to illustrate the chronological sequence of messages between objects in an interaction. The sequence diagram for the customer registration scenario is shown in Figure 8-3 on page 235.

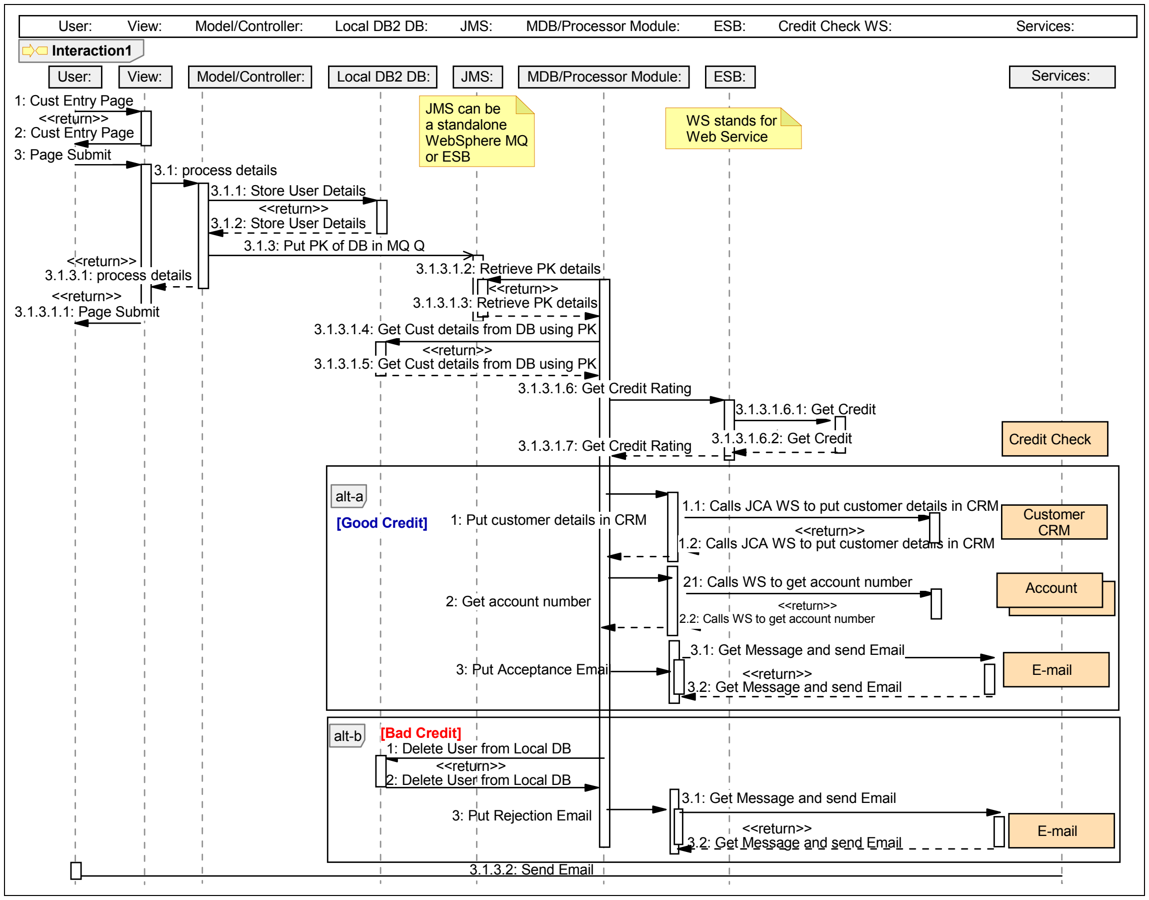

User:
View:
Model/Controller:
Local DB2 DB:
JMS:
MDB/Processor Module:
ESB:
Credit Check WS:
Services:
Interaction1
JMS can be a standalone WebSphere MQ or ESB
WS stands for Web Service
1: Cust Entry Page
<<return>>
2: Cust Entry Page
3: Page Submit
3.1: process details
3.1.1: Store User Details
<<return>>
3.1.2: Store User Details
3.1.3: Put PK of DB in MQ Q
<<return>>
3.1.3.1: process details
<<return>>
3.1.3.1.1: Page Submit
3.1.3.1.2: Retrieve PK details
<<return>>
3.1.3.1.3: Retrieve PK details
3.1.3.1.4: Get Cust details from DB using PK
<<return>>
3.1.3.1.5: Get Cust details from DB using PK
3.1.3.1.6: Get Credit Rating
3.1.3.1.6.1: Get Credit
3.1.3.1.6.2: Get Credit
3.1.3.1.7: Get Credit Rating
Credit Check
alt-a
[Good Credit]
1: Put customer details in CRM
1.1: Calls JCA WS to put customer details in CRM
<<return>>
1.2: Calls JCA WS to put customer details in CRM
Customer CRM
2: Get account number
21: Calls WS to get account number
<<return>>
2.2: Calls WS to get account number
Account
3: Put Acceptance Email
3.1: Get Message and send Email
<<return>>
3.2: Get Message and send Email
E-mail
alt-b
[Bad Credit]
1: Delete User from Local DB
<<return>>
2: Delete User from Local DB
3: Put Rejection Email
3.1: Get Message and send Email
<<return>>
3.2: Get Message and send Email
E-mail
3.1.3.2: Send Email

Figure 8-3 Sequence diagram of ITSOMart application

The flow of the ITSOMart application is as follows:

1) A customer accesses the Web site and clicks a button requesting to register. The request is processed by a JSP, which provides a form for user information.

2) The customer enters the requested data and clicks a Submit button. JSF is used to validate the data on the client-side. The data includes customer data and the type of account requested. One or more account types can be selected.

3.1.1) The JSP stores the data input by the customer into a DB2 database using CMP's.

3.1.3) Next, it places a message on a queue containing the primary key of the record stored in the database.

3.1.3.1.1) A response page is sent to the customer stating that the registration process is underway and he will receive an email confirmation when it is done.

3.1.3.1.2) The message placed on a JMS queue is retrieved by a message driven bean, initiating a serial process. The process invokes the back-end services.

3.1.3.1.4) Using the primary key in the message, the customer data is retrieved from the local database.

3.1.3.1.6) The customer data is sent to the Credit Service to get a quote. The Credit Service is available as a Web service and is accessed through the ESB (enterprise service bus).

alt-a) If the customer rating is acceptable, the customer data is stored in the existing CRM on CICS using JCA connectors. The CRM maintains the master copy of customer information. Each business application that corresponds to the customer type selected will be sent the information and will return an account number. Once the account numbers are returned, an e-mail is sent to the customer indicating the registration was accepted and the account numbers assigned.

alt-b) If the rating is not acceptable, the customer data is removed from the database and an e-mail is sent to the customer indicating the registration was rejected.

8.3.3 Technology and product selection

The application designers decided on the technology to use based on the existing back-end applications and the requirements of the new application. WebSphere Application Server was determined to be the best choice for the new application because it is a simple, service-oriented J2EE application design provided the functionality needed, while facilitating additions to the application as the business grows.

The user interface is implemented as one application while the driver for the registration process is implemented as a separate application. Future processes, such as an order process, can be implemented without affecting existing process applications. Simple changes to the user interface application can accommodate the new process. The processes required are invoked by placing a message on a queue to be picked up by a message-driven bean that starts the process.

Using a Web service interface to access the credit check service allows them to easily change credit check providers.

A Web service interface also works well for their home and business delivery services. Adding additional options, such as institutional delivery, are simplified by using a standard interface.

Their current preferred e-mail provider is accessed through a J2EE application that uses a messaging interface. The messaging infrastructure provides reliable, secure delivery of messages.

Their existing application that maintains the customer registration data is a CICS application. Wrapping the existing application as a Web service allowed them to build the new application without worrying that future alterations would be needed to accommodate changes in the CICS application.

The service integration technology in WebSphere Application Server was chosed for the implementation of the enterprise service bus. It provides transport capability for Web services, the default messaging provider, and its mediation capability is used to implement the router and decomposition aspects.

9

JSF front-end scenario

This chapter discusses the design and implementation of the Web-based front-end part of the ITSOMart application. In particular, the chapter focuses on the use of JavaServer Faces (JSF) technology. The chapter will use the practical example of the ITSOMart front-end, to illustrate many of the theoretical concepts introduced in Chapter 6, “Technology options” on page 101 and Chapter 7, “Application and system design guidelines” on page 139.

It is expected that you are familiar with the business scenario, a basic structure of the ITSOMart application, described in Chapter 8, “Business scenario and design” on page 229.

In addition to reviewing some of the key characteristics of a JSF Web-based application, we also focus on the use of Rational Software Architect to develop JSF applications.

Note: The Web development tools used to develop JSF applications are available in Rational Application Developer and Rational Web Developer as well.

9.1 Architectural overview

The architectural overview model shown in Figure 9-1, illustrates how the ITSOMart application was extended to communicate with an external enterprise system using Web services. The area we focus on in this chapter is highlighted. It represents the user interface portion of the application.

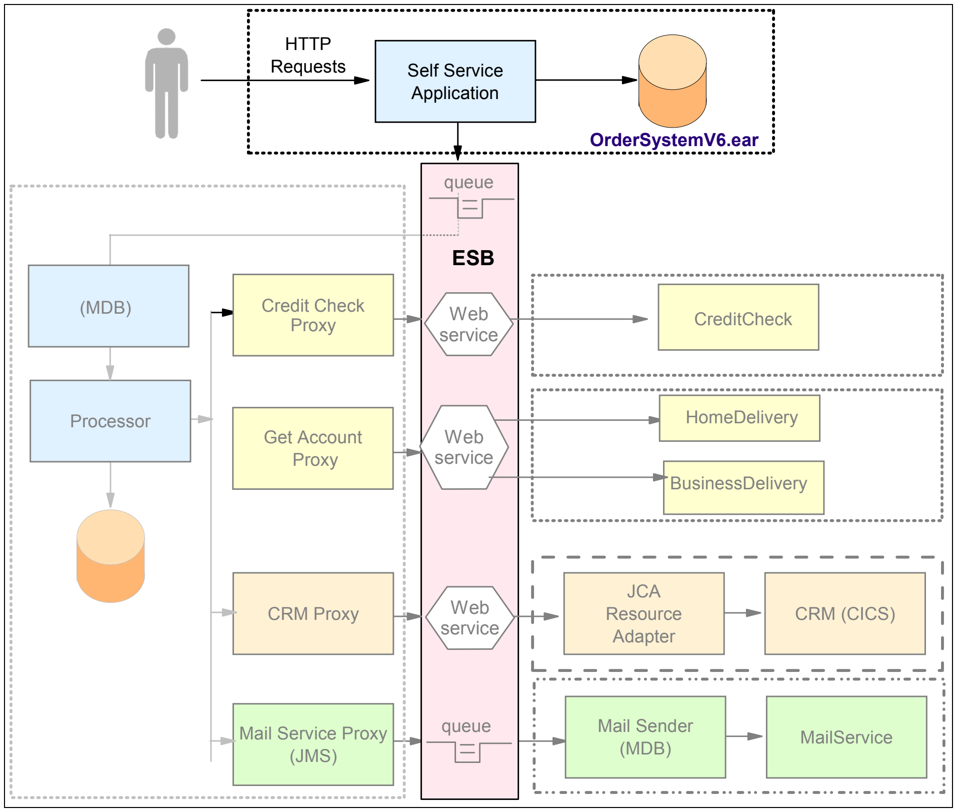


Figure 9-1 Architectural overview model: front-end application

The OrderSystemV6 application provides the interface to the customer. When a customer registers, OrderSystemV6 updates the local database and places a message on a queue containing the database record key. A message-driven bean takes the message out of the queue and initiates the rest of the processing.

9.2 System design overview

This section discusses the system design of the front-end of the ITSOMart application. Note that we have already extensively covered the topic of front-end application design on section 7.2, “Application structure” on page 141. However, the intent of this section is to use the real example of the ITSOMart application to illustrate how one would match the needs of an application to the design approaches previously discussed. We then reinforce the concepts by describing how the designed approach is specifically mapped to the ITSOMart application.

This section also tangentially illustrates the modeling capabilities of Rational Software Architect. All diagrams in this section were constructed using Rational Software Architect.

9.2.1 Design considerations

Before designing an application, it is critical to understand your guiding principles. Aside from the obvious goal of fullfilling the customer requirements, you need to identify the goals you are trying to achieve with your system design approach.

Goals can vary from application to application, so in this section we focus specifically in the ones we considered when designing the ITSOMart system. Nonetheless, these are general enough goals which are likely applicable to most self-service applications. We first examine these principles and goals below, then the component model and object model sections illustrate how they map to our selected design approach.

- Design must separate the presentation layer from business logic.

  The term *separate presentation from business logic* has been a staple in user interface design for decades now. The basic idea is that a good user interface design should keep the rendering of screens separate from the application logic which manages the information displayed.

  User interface design is about presenting information in an aesthetically pleasing fashion to end customers. Beyond a simple choice of technology, a graphical user interface (GUI) design might involve a wide variety of human factor guidelines, ranging from recommended types of menu bars, to which colors are psychologically more apt to grab a customer’s attention. Languages supporting GUI implementation, such as HTML or Java, generally support multiple ways of presenting data. The choice of the presentation should be driven by human factors guidelines.

  Conversely, the design of the of the application code responsible for implementing the business logic is influenced by more standard IT drivers such as performance, storage capacity, code reuse, reliability, and so forth.

Clearly the drivers for the presentation and business layers are very different, hence the need for a clean separation. It is essential that simple look and feel changes in the user interface do no propagate down to business logic, and vice versa. For instance, in the case of the ITSOMart application, if we changed the screen flow of the application, we would not want to impact the EJB layer supporting the customer administration functions. Similarly, if we chose to reorganize the customer database table, we would not want the user interface modules to be impacted.

- Design must leverage different skill sets.

  There are usually multiple technologies involved in the implementation of a self-service application. In the ITSOMart sample for instance, we are using JMS, Web services, Enterprise Service Bus (ESB), JCA, DB2, and EJB, in addition to whatever technology we chose for out front-end. Although the specific products and technologies may vary, such diversity is typical of most e-business applications.

  When designing the front-end we need an approach which will effectively leverage the skill set of our team. For example, we do not want the human factor specialist responsible for the look and feel of the front-end, to have to worry about the details of data storage in DB2. We do not need the developer responsible for the JSP or servlet code that handles the user request processing, to also be a specialist in the integration technologies needed to communicate with our back-end.

- Design should support multiple user interfaces.

  In today's e-business environment, there are multiple mediums in which to present customer information. In addition to Web-based applications over the Internet, wireless devices such as PDAs and cell phones are a rapidly growing market. Business-to-business interfaces can forgo the use of a user interface, and opt instead for an XML-based Web services interface.

  Looking at the ITSOMart application as an example, it is currently targeted for a Web browser interface. However, the business requirements could change in the future, such that a user could register from a cell phone. If that were the case, we would prefer our selected design approach to be such that it would minimize the impact to the overall application.

- Design approach should leverage on well defined design patterns.

  Patterns will improve the quality of the overall design because they represent the experience of proven successful solutions. Patterns can be applied at an overall architectural level, or as detailed as a single Java class. The use of patterns also improve productivity, as they facilitate the design process, and can even accelerate the implementation through the use of tooling. Rational Software Architect for instance, supports an extensive library of commonly used patterns, which can be easily applied to design models, and even automatically generate code.

- Design should favor industry-standard technology.

  It is often tempting to use unproven, or vendor-specific technology. For instance, a particular vendor might offer a proprietary package that perfectly fits the customer requirements. This solution might very well be cheaper and faster to purchase. There might also be a new open source framework, which appears to address many of our design goals.

  There are times when taking either of these approaches is a valid option, but the risks need to be well understood. Proprietary packages will generally limit the flexibility of your environment to change, because you will be dependent on the availability of new versions from the provider. New technology might sound appealing, but without careful analysis you can find yourself in the proverbial *bleeding edge*. New technology tends to evolve. Applications written using early versions can quickly become deprecated. In either scenario, staffing becomes an issue as well, because it might be difficult to quickly attain resources familiar with new or vendor-specific technologies. Therefore, all things being the same (not always an easy determination), we recommend the use of established industry-standard technology.

  For the ITSOMart front-end application in particular, there is no need of any new or specialized technology. Therefore, our design approach should rely on industry standard, preferably open standards, technology.

9.2.2 Component model

Now that we have discussed the various principles guiding our design process, we will discuss the different aspects of the design approach we selected. The component model will describe the overall structure of our application, in terms of its high level components.

Component diagram

The component diagram in Figure 9-2 on page 244 depicts the different software components which make up the front-end of the ITSOMart application. The following sections elaborate on the different components.

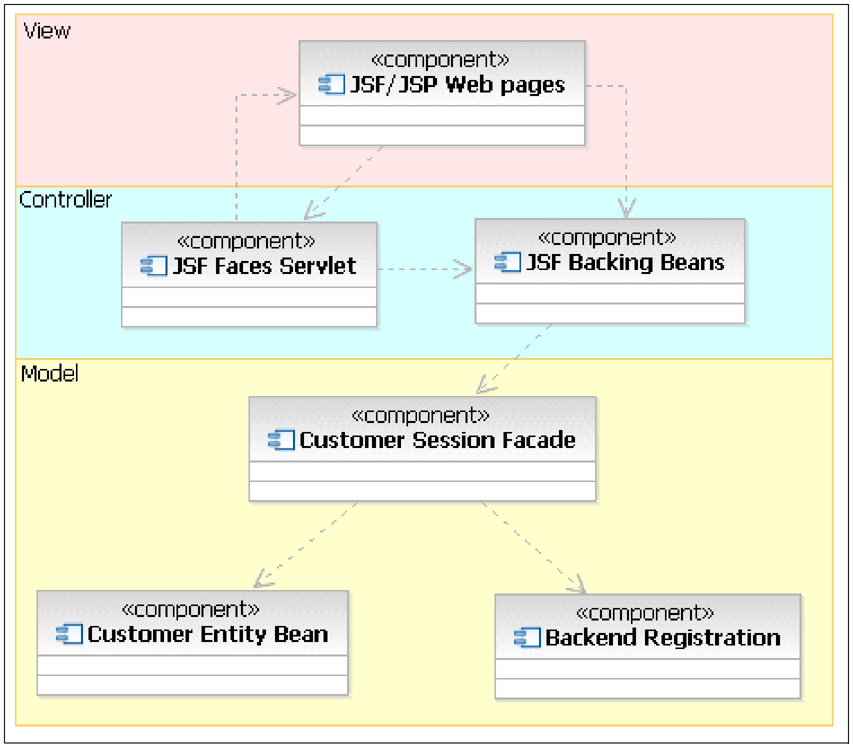


Figure 9-2 ITSOMart front-end component model

Model-View-Controller architectural pattern

Our component model is based on the Model-View-Controller (MVC) architectural pattern, widely used in user interface applications for many years. It is particularly well-suited for Web-based applications, making it an overwhelming choice across the industry. 7.2.1, "Model-View-Controller design pattern" on page 141, discusses the MVC pattern in detail, therefore we do not repeat it here. We simply review the main points so we can show how it is realized in the ITSOMart application. The MVC pattern divides the application in three components, indicated by differently shaded areas in Figure 9-2:

- The Model represents the data of the application, as well as the business rules and logic that govern the processing of the data. In the ITSOMart front-end, the Model is implemented through EJB classes that interact with the database and back-end.
- The View is a visual representation of the model. Multiple Views can exist simultaneously for the same Model. In the ITSOMart application, the View is implemented using JSF and JSP Web pages.

- The Controller decouples the visual representation from the underlying business data and logic by handling user interactions and controlling access to the Model. In the ITSOMart application, the Controller is implemented primarily by the JSF FacesServlet. The FacesServlet controls the flow by either forwarding requests to JSP and JSF pages, or invoking action calls on various managed beans associated with JSF components.

The MVC pattern addresses all of the design goals we identified for the ITSOMart application:

- It clearly supports the separation of presentation and business logic. The presentation layer is isolated to the View component, while the business logic is captured in the Model. Similarly, the separation of components also allows for leveraging the different skill sets.
- MVC is also obviously a good fit for supporting multiple user interfaces. The very definition of the View dictates that multiple Views can be associated with a single Model.
- As we mentioned earlier, the MVC pattern is certainly a well-established industry standard.

The MVC pattern describes a high-level architecture model, but does not specify the low level design for implementing it. There are multiple frameworks supporting the low level design and implementation an MVC-based application. For the ITSOMart application, we selected the JavaServer Faces (JSF) framework.

The JavaServer Faces framework

The ITSOMart application uses the JSF framework to implement the MVC pattern. 7.2.6, "Frameworks" on page 153, discusses the JSF framework in detail. The following are some of the reasons the JSF framework fits the design goals for our application:

- JSF provides a very clean separation of presentation and business logic. The presentation is contained in the JSP pages, built using HTML and JSF UI components. The business logic is implemented in separate Java beans, which are linked to the presentation through the JSF <faces-config> configuration file. This clear separation allows screen designers to operate strictly on the presentation layer, without needing the J2EE skills required of the application developers.
- Presenting content over multiple types of user interfaces is a basic design goal of JSF, due to its pluggable rendering capability. Although the current implementation of ITSOMart is rendered for a Web browser using HTML, a different rendering kit could present the same components using WML for a wireless device.

Note that a WML rendering kit is not a standard component of Rational Software Architect at this time. However the JSF framework can clearly accommodate this and other kits as they become available.

- JSF is a standard component of the J2EE 1.4 specification, and is rapidly gaining acceptance in the industry. JSF is now supported by all major IDEs and Web application servers, including the WebSphere and Rational family of products.

JSF versus Struts

A fair question at this point, would be to ask why we selected the JSF framework over Jakarta Struts? See 7.2.6, “Frameworks” on page 153 for more details about Struts. It is true that there is no overwhelming advantage for JSF at this time. The primary deciding factor was that JSF has now been adopted as a J2EE standard. JSF has been embraced rapidly by the IT industry, including all the major tool vendors. JSF incorporates most of the concepts which made Struts so popular over the last few years, while making improvements in various areas.

The general consensus at this time, is that if an application is already implemented in Struts, there is no great need to convert it to JSF. However, when writing a new application, as we are in this case, JSF is the preferred option.

Enterprise JavaBeans components

The last layer of our component model is composed of Enterprise JavaBeans (EJBs), which provide the Model portion of the MVC pattern, as well as the integration to the back-end services.

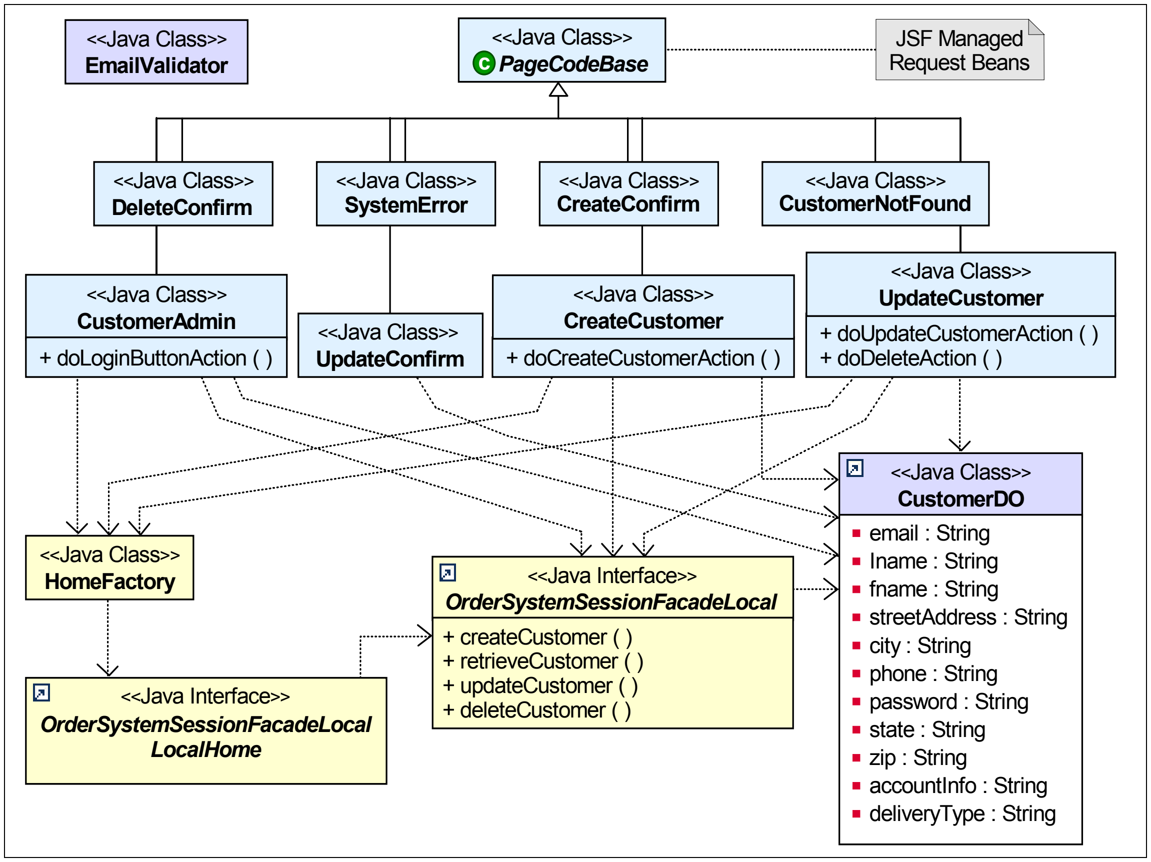
The Customer entity bean manages the data for one customer, storing it in a DB2 table. Following the J2EE session facade pattern, the access to this entity bean is encapsulated in an EJB session bean. The session facade also provides access to the back-end services supporting the ITSOMart application.

9.2.3 Object model

This section describes the object model for the ITSOMart front-end, through the use of various modeling diagrams.

Front-end class diagram

The class diagram in Figure 9-3 on page 247 shows the primary classes in the ITSOMart front-end. For the sake of readability, we restricted the visibility of the classes’ components strictly to those operations and data members we wanted to highlight. Note that given the nature of a JSF front-end, a formal UML class diagram might not provide the complete picture of the application. That is because the JSP pages, which are an integral component of the JSF framework, are not allowed in standard UML class diagram notation. However we illustrate the JSP relationships later in freeform diagrams, and Web flow diagrams.



<<Java Class>> EmailValidator
<<Java Class>> PageCodeBase
JSF Managed Request Beans
<<Java Class>> DeleteConfirm
<<Java Class>> SystemError
<<Java Class>> CreateConfirm
<<Java Class>> CustomerNotFound
<<Java Class>> CustomerAdmin
+ doLoginButtonAction ( )
<<Java Class>> UpdateConfirm
<<Java Class>> CreateCustomer
+ doCreateCustomerAction ( )
<<Java Class>> UpdateCustomer
+ doUpdateCustomerAction ( )
+ doDeleteAction ( )
<<Java Class>> CustomerDO
email : String
lname : String
fname : String
streetAddress : String
city : String
phone : String
password : String
state : String
zip : String
accountInfo : String
deliveryType : String
<<Java Class>> HomeFactory
<<Java Interface>> OrderSystemSessionFacadeLocal
+ createCustomer ( )
+ retrieveCustomer ( )
+ updateCustomer ( )
+ deleteCustomer ( )
<<Java Interface>> OrderSystemSessionFacadeLocal LocalHome

Figure 9-3 ITSOMart front-end class diagram

The following are some of the key classes and relationships depicted in the class diagram:

- The top portion of the diagram represents a layer of Java beans that support the Faces JSP pages. There is a bean assigned to each JSP page, and each bean is responsible for getting and setting the data in the JSF UI components of the page. Some beans also support the action methods which trigger the back-end requests. See 9.3.2, “JSF managed bean design” on page 257, for an extended discussion about JSF managed beans.
- The CustomerDO class is the data object used to encapsulate and exchange the customer information across the application.

- The OrderSystemSessionFacade related classes provide access to the ITSOMart application back-end. They provide a simple remote interface for performing the basic operations triggered by the front-end: create, retrieve, update, and delete. See 9.3.5, “Back-end interface” on page 262, for further discussion on the back-end interface
- The EmailValidator class provides a customer defined format validation for e-mail addresses. 9.3.3, “JSF input validation” on page 259, further describes JSF validation.

Create customer freeform diagram

The freeform diagram in Figure 9-4 on page 249 provides a less rigorous but more comprehensive overview of the ITSOMart front-end. To improve clarity, this pseudo-class diagram is restricted to only those classes and components which are related to the use case of creating a new customer profile. In the sections to come, we continue to focus on this create customer profile use case. This will provide an end-to-end view from high-level design, through low-level implementation.

Because this is not a formal UML class diagram, it also includes the JSP pages, the JSF FacesServlet, and the back-end beans responsible for storing the customer in the database.

The diagram also provides a detailed view of how the JSF framework in the ITSOMart front-end, maps into the MVC architectural pattern. This is similar to the high-level mapping presented in the component diagram in Figure 9-2 on page 244, but at a much lower level.

- The Model component is implemented through EJB session and entity beans. They store the customer information, and implement the business logic that triggers the credit check. The model data is stored and carried through the application in the CustomerDO data object.
- The View component is implemented using JSP Web pages which present the customer creation input form to the user (CreateCustomer.jsp), as well as the confirmation page (CreateConfirm.jsp). The JSP pages will use JSF UI components to render the information in the Web browser.
- The Controller component is implemented by the JSF FacesServlet, and the CreateCustomer JSF managed bean. FacesServlet is a standard J2EE class (in the javax.faces.webapp package), and is responsible for controlling the life cycle of a JSF request. This is more clear in the sequence diagram in the next section (Figure 9-5 on page 250).

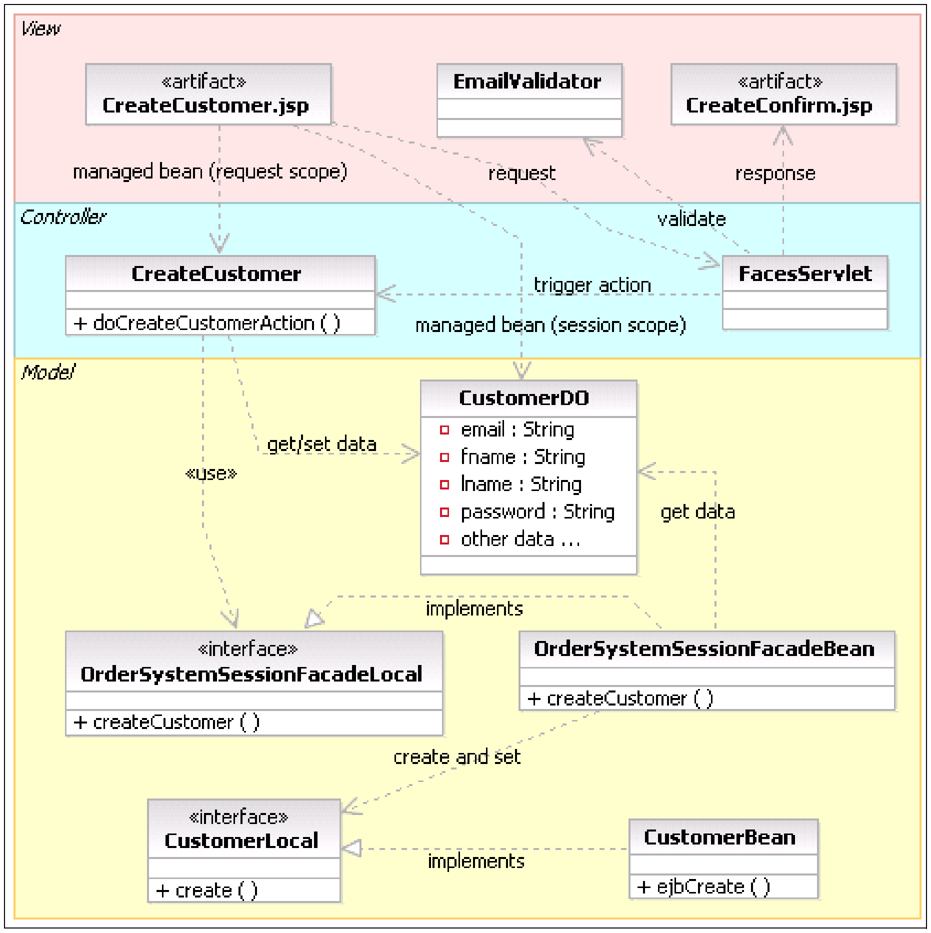


Figure 9-4 Create customer freeform diagram

Create customer sequence diagram

The relationship between the classes used to create a customer are further detailed in the sequence diagram in Figure 9-5 on page 250. In addition to presenting the logic flow for the ITSOMart front-end, the diagram also offers a good practical example of the role of FacesServlet in controlling the life cycle of the JSF request.

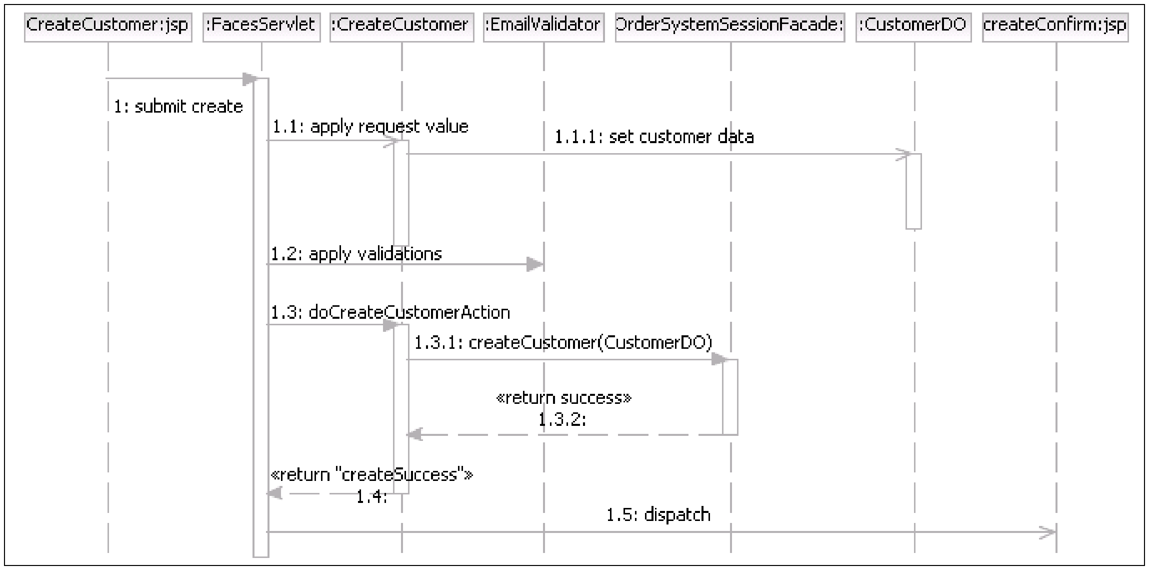
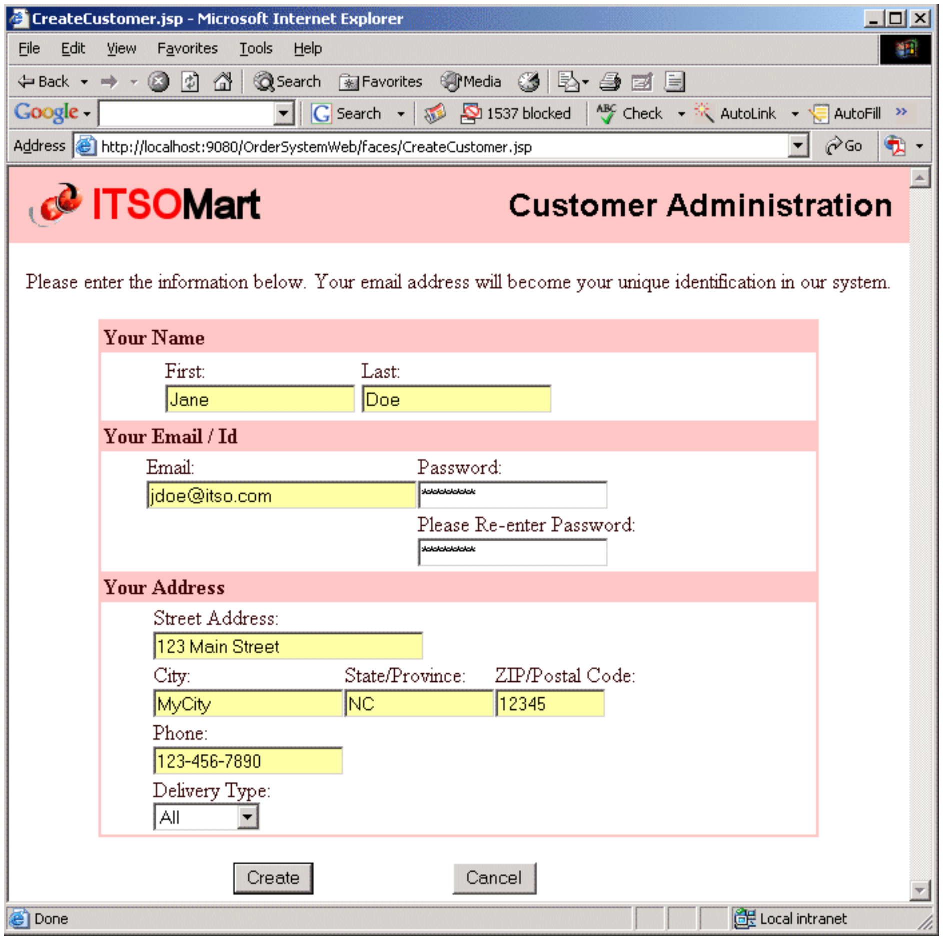


Figure 9-5 Create customer sequence diagram

The use case starts with the CreateCustomer.jsp page loaded onto the user's browser, as shown in Figure 9-6 on page 251.


CreateCustomer.jsp - Microsoft Internet Explorer

Address http://localhost:9080/OrderSystemWeb/faces/CreateCustomer.jsp

ITSOMart Customer Administration

Please enter the information below. Your email address will become your unique identification in our system.

Your Name

First: Jane
Last: Doe

Your Email / Id

Email: jdoe@itso.com
Password: ********
Please Re-enter Password: ********

Your Address

Street Address: 123 Main Street
City: MyCity
State/Province: NC
ZIP/Postal Code: 12345
Phone: 123-456-7890
Delivery Type: All

Figure 9-6 Create customer screen

After the user has completed the profile information, he or she clicks the **Create** button, which triggers the following flow:

1) The HTTP request is forwarded to the FacesServlet, registered in the Web server to handle all JSF-related requests.

1.1) FacesServlet obtains the request arguments and updates the CreateCustomer managed bean.

1.1.1) The CustomerDO object within CreateCustomer is loaded with the customer information.

1.2) FacesServlet now invokes any special validation functions. For the ITSOMart front-end, that means EmailValidator is called. In this case, we assume the request is valid. If it were not, the request would end here, and FacesServlet would return the same create input screen, with the appropriate validation errors added.

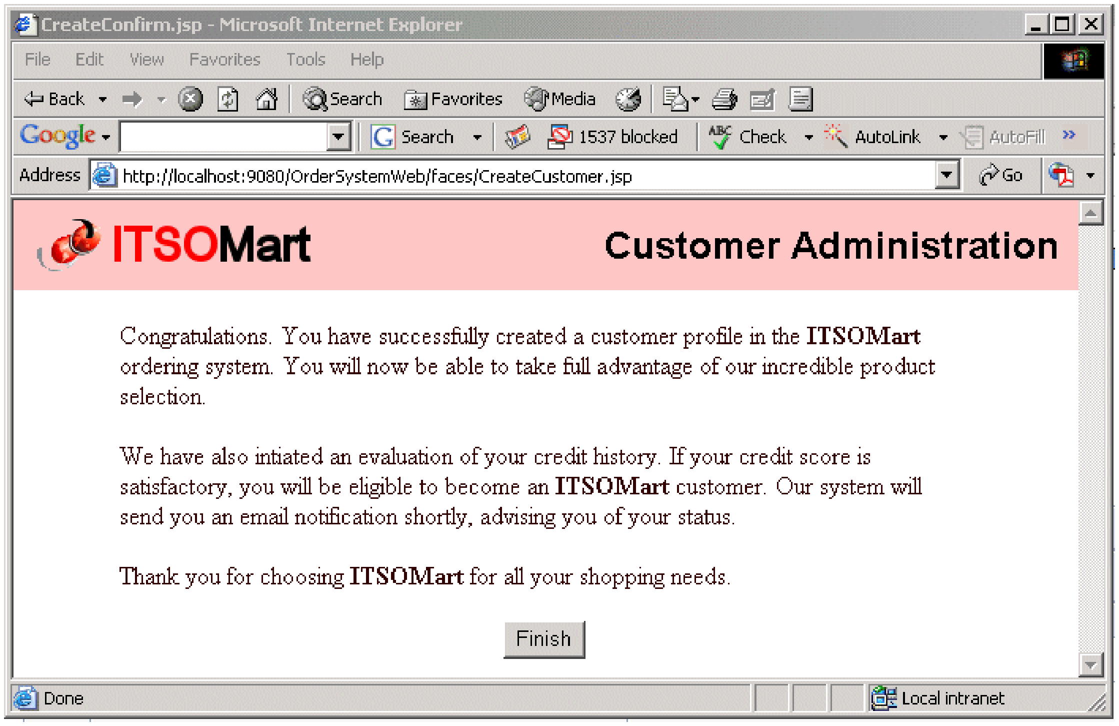
1.3) FacesServlet now invokes the doCreateCustomerAction method on the CreateCustomer managed bean, which is associated with the Create button in the JSP page.

1.3.1) doCreateCustomerAction obtains a reference to a OrderSystemSessionFacadeLocal object, and invokes the createCustomer method, passing in the CustomerDO object, which was loaded on step 1.1.1.

1.3.2) OrderSystemSessionFacadeLocal will get realized in the back-end, and perform the required steps to create and process the customer profile. In this case, we assume the request will be successful.

1.4) Upon receiving the successful creation status, the doCreateCustomerAction method returns the string “createSuccess”.

1.5) FacesServlet will match the “createSuccess” string with a navigation rule in the faces-config.xml configuration file. The rule identifies CreateConfirm.jsp as the target page (9.3.1, “ITSOMart Web diagram” on page 253, further describes this navigation concept). FacesServlet will then dispatch the response, which is displayed as seen in Figure 9-7 on page 253.


CreateConfirm.jsp - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Google Search 1537 blocked Check AutoLink AutoFill
Address http://localhost:9080/OrderSystemWeb/faces/CreateCustomer.jsp Go
ITSOMart
Customer Administration
Finish
Done
Local intranet

Congratulations. You have successfully created a customer profile in the **ITSOMart** ordering system. You will now be able to take full advantage of our incredible product selection.

We have also intiated an evaluation of your credit history. If your credit score is satisfactory, you will be eligible to become an **ITSOMart** customer. Our system will send you an email notification shortly, advising you of your status.

Thank you for choosing **ITSOMart** for all your shopping needs.

Figure 9-7 Create customer confirmation screen

9.3 Low level design

Now that we have discussed the overall design of the ITSOMart front-end, we will turn our attention to some of the low level design decisions associated with a JSF application. Once again, we focus these discussions on our own sample application.

9.3.1 ITSOMart Web diagram

The Web diagram in Figure 9-8 on page 254 (constructed using Rational Software Architect), presents the complete end-to-end flow of the ITSOMart application front-end. A Web diagram is a valuable tool for visualizing and constructing JSP/JSF-based applications.

In the diagram, the nodes represent the different JSP pages which make up the application presentation. The connections represent simple Web page links, or the outcome of JSF actions. Because the ITSOMart application uses strictly

JSF-based navigation, all connections in Figure 9-8 on page 254 represent JSF action results.

Using the Web diagram, a human factor expert can design the entire flow of the application, without having to look at a single line of code, or even a screen design. When a JSP page node is added, the corresponding .jsp file is created, but simply left blank. Adding a JSF outcome connection creates a JSF rule, which is stored in the faces-config.xml configuration file. Because JSF page navigation is isolated from the code with these external navigation rules, the designer can have the complete end-to-end flow planned out, without ever having to edit a JSP file. When the Web diagram is completed, there should be a complete framework of JSP files and navigation rules ready to be completed by a GUI developer.

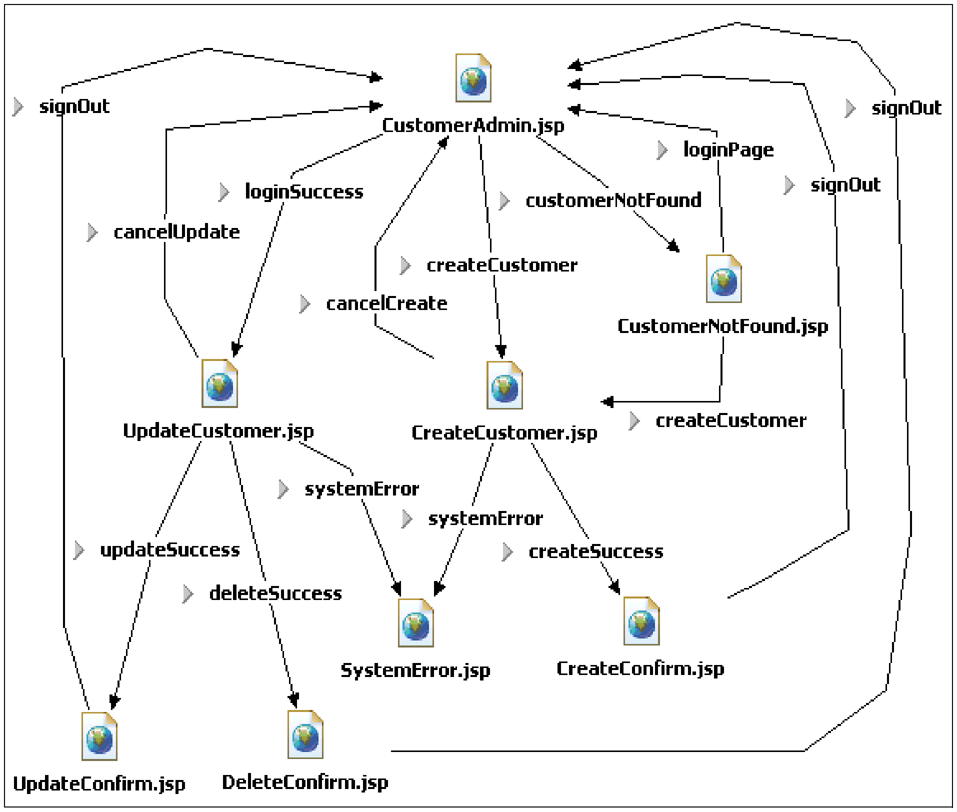


Figure 9-8 ITSOMart Web diagram

To better understand the Web diagram and the page design process, we return to the example of our familiar create customer use case. The following steps traverse the Web diagram top to bottom from the home page, through the creation of a customer profile.

1. The home page for the ITSOMart front-end is the CustomerAdmin.jsp page. From that page, if the user wants to create a customer, she should be redirected to the CreateCustomer.jsp page. Therefore, we add a JSF connection from CustomerAdmin.jsp to CreateCustomer.jsp, specifying the outcome to be createCustomer. See “Selecting the outcome string” on page 256. When we do that, a navigation rule is automatically added to the faces-config.xml file as in Example 9-1.

Example 9-1 Navigation rule for faces-fonfig.xml

```
<navigation-case>
   <from-outcome>createCustomer</from-outcome>
   <to-view-id>/CreateCustomer.jsp</to-view-id>
</navigation-case>
```

   Now when the CustomerAdmin.jsp is implemented, all we need to do is make sure that the link or button we use to trigger customer creation returns the string createCustomer.

2. From the CreateCustomer.jsp screen, we expect the user to enter the appropriate information, then press a create button. If the creation is successful, we would prefer the user to see a confirmation page, therefore we add a JSF connection with the outcome of createSuccess. We then link it to the CreateConfirm.jsp page.

   Note that the Web diagram also accounts for an error scenario. If there is an unexpected problem, and we cannot create the customer profile, we would prefer the user to see a standard error page. Therefore we add a JSF connection with a rule that points the systemError outcome to the SystemError.jsp page. Later, when implementing the code, all the developer needs to know is that in the case of an unexpected exception, the code should return the string systemError.

   As in the previous case, these JSF connections are automatically added by Rational Software Architect to the face-config.xml file. The navigation rules in Example 9-2 for both the success and failure cases, are saved in the configuration file.

Example 9-2 Navigation systemError outcome rules

```
<navigation-case>
   <from-outcome>createSuccess</from-outcome>
   <to-view-id>/CreateConfirm.jsp</to-view-id>
</navigation-case>
<navigation-case>
   <from-outcome>systemError</from-outcome>
```

```
        <to-view-id>/SystemError.jsp</to-view-id>
    </navigation-case>
```

3. From the create confirmation screen (CreateConfirm.jsp), we only allow the user to sign out. In our case, signing out redirects the customer to the home page (CustomerAdmin.jsp). This is represented with the signOut JSF connection from CreateConfirm.jsp to CustomerAdmin.jsp. As with the other connections, an appropriate entry is added to the face-config.xml file.

This example illustrates how the application flow can be designed through a Web diagram. Clearly the flow of the rest of the application can be inferred following the same process we have shown to create the customer use case.

Note that additional details can also be included in a Web diagram. Because we were designing the entire application flow, we chose to include only the JSP pages and navigation rules. However, a designer can also include the managed beans responsible for performing the JSF actions. Although we felt this would add too much clutter in the overall scenario, 9.4.5, “Implementing page navigation” on page 275, shows an example of a more detailed diagram for a specific case.

Selecting the outcome string

As we saw in the diagram, the outcome string is a key part of the navigation rule. When designing the application, we should decide on a strategy for selecting these strings consistently.

One approach sometimes used is to choose a string that is clearly associated with the target page. For instance, the strategy could be that the outcome string would match the name of the desired target JSP page. In other words, an outcome string of updateConfirm would always take us to the UpdateConfirm.jsp page.

The problem with such an approach, is that it defeats the purpose of an external configurable navigation framework. If the outcome strings are hardcoded to match exact JSP file names, we could have just as easily used a standard servlet forwarding mechanism to forward the request to the JSP file.

The approach we recommend and the one used by the ITSOMart application, is to choose outcome strings that generically represent the outcome of an action. Instead of expressing what the next page should be, the string should represent what the current page accomplished.

For example in our create customer use case, if the customer creation were successful, we would return an outcome of createSuccess. Currently createSuccess leads to the CreateConfirm.jsp page. However in the future, the flow could change, and after creating a profile, the customer might be forwarded to a different area to enter additional details. In that case, we could simply change the navigation rule on web-config.xml, such that createSuccess would now lead to a new page called CustomerDetail.jsp. This would be done without having to touch any of the JSP files, or Java classes.

9.3.2 JSF managed bean design

Once we have designed the end-to-end screen flow of our JSF application, we need to understand how data is exchange between screens, and how the back-end functions are accessed. In JSF, this is accomplished by the use Java beans managed by the JSF framework.

The managed beans can be declared with four different scopes:

- Request

  The bean is persisted between two pages, the current page and the target next page, or on a page reload. In that case, the bean is stored in the HTTPServletRequest.
- Session

  The bean is stored in the HTTPSession and exists for the life of a user's session.
- Application

  The bean is stored in the ServletContext and exists for the life cycle of the JSF application, with one instance for all users.
- None

  The bean is not stored anywhere, and can only be used as a temporary variable

From a logical perspective, managed beans are used in multiple ways in the JSF applications. In the following sections, we discuss the different applications of managed beans, and specifically how they are used in the ITSOMart front-end.

Backing beans

A typical Faces JSP page is composed of multiple input and output UI JSF components (text fields, output texts, selection lists, and so on). Backing beans are managed beans used to bind these UI components to Java objects that can be accessed by the application.

When using Rational Software Architect to create a Faces JSP page, a backing bean is automatically created, and a definition for it is added to the faces-config.xml file. This bean will be used to bind with the JSF UI components in the JSP page. Although technically speaking we could edit the configuration file and change this JSP to bean mapping, there is no particular reason to that. Therefore, following this default behavior, the design of the ITSOMart application is such that each JSP page will have a unique corresponding backing bean.

The backing beans are declared with request scope. Because the bean's purpose is strictly to interact with the one JSP file in particular, there is no reason its information needs to be stored beyond the request.

Action beans

In addition to rendering data, the other key operation in a Faces JSP page is to trigger an action. These actions cause the application to perform some operation and might include navigating to another page. They are triggered by JSF UI components such as command buttons and links.

From a JSF design perspective, we could declare a managed bean dedicated explicitly to handle actions. However, this is not strictly necessary. The default behavior of Rational Software Architect is to create an action handler method in the standard backing bean associated with the JSP page. In that case, the backing bean is logically also an action handling bean.

The ITSOMart front-end follows the simple approach of placing the action handler on the backing bean. There is little need for dedicated action classes, since the relevant operations are already clearly isolated in the OrderSystemSessionFacade object, as seen in the object model on 9.2.3, "Object model" on page 246.

For example, again using the create customer case, the pc_CreateCustomer backing bean has a doCreateCustomeAction method. This method is bound as the handler for the Create button (a JSF <hx:commandExButton> UI component) on the CreateCustomer.jsp page.

Data beans

Although backing beans get and set data on a specific page, they are not meant to carry data across multiple screens. Therefore, if an application has persistent data which needs to be carried through to multiple screens, a dedicated managed bean is generally used for that purpose.

The scope of the data bean depends on the requirement of the application. The most common scenario for e-business applications, is probably that a certain amount of data gets carried through a user session, the shopping cart being the classic case. In that case, the managed bean should be declared with session scope. The declaration of the data bean is done in the faces-config.xml configuration file.

In the case of the ITSOMart application, we define the CustomerDO class as the managed bean responsible for storing session data. The class stores the basic customer data carried through multiple screens in the application.

9.3.3 JSF input validation

The JSF framework provides a variety of ways of validating user input. This section discusses some of these options, and describes the ones employed in the ITSOMart application. Implementation details for the various validation options are presented later in 9.4.6, “Implementing input validation” on page 280.

Client side versus server side validation

Before looking into the various JSF validation options, we first discuss one of known limitations. JSF currently does not support client side validation. That means that even the most simple of input format checking requires validation to be performed on the server. This goes against the common practice for Web-based applications, where simple input validation is often done on the client side, using JavaScript.

The options in this case, are to stick with standard JSF validation, or manually program client side validation with JavaScript. The JSF-based validation will fit in naturally with the tooling and the framework, but the JavaScript client side validation reduces the server load. Either option is valid, and the choice will depend on the relative importance of reducing server load for the application in question.

In the ITSOMart front-end, we have chosen to use JSF-based validation to demonstrate this technique. Note that the specifications for the upcoming JSF2.0 release will include client-side validation, which should remedy the current shortcoming.

Standard component validation

Most JSF UI components include at least some level of input validation. These include checks for required values, string length, and date formats. These are not comprehensive, but they do offer some useful basic functionality. The ITSOMart front-end makes use of all appropriate standard validation provided by the JSF components.

Custom validation

The validation provided in the standard JSF components is limited to basic data types such as strings, numbers, and dates. Clearly this is not sufficient for most complex applications. However, JSF does provide a simple framework for specifying custom-defined validation functions. Once a special validation class is constructed, and the definition is added to the faces-config.xml configuration file, it can be reused to validate any applicable JSF component.

For example, in the case of the ITSOMart application, we have defined a custom validation class: EmailValidator. The EmailValidator class implements a special validation for e-mail addresses. Once defined and included in faces-config.xml, the function is reused for e-mail format validation in multiple screens.

9.3.4 Error handling

The error handling on the ITSOMart front-end is relatively simple. Because this is a sample application, we chose to spend more time on the sunny day scenario, rather than on the rainy day one. The following sections focus on the design considerations we contemplated when incorporating error handling to our JSF application.

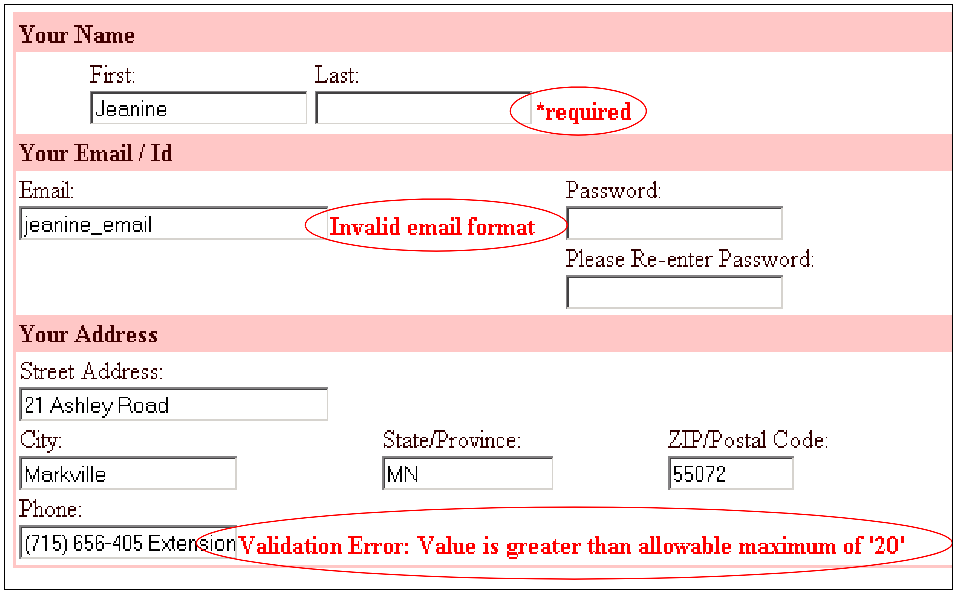
Validation errors

The previous section discusses how data is validated, but does not describe how validation errors are eventually presented to the user. There are a few options available in the JSF framework using error display UI components.

Although it is possible to use a separate JSP error page, from a human factors perspective, it is generally preferred to display input validation errors in the same page where they were entered. In that case, we can follow two different approaches. The validation errors can be shown specifically for each field, in which case the error message is positioned right next to the component being validated. The other option is to have a single error display field which displays all the errors of the page. JSF offers UI components which accommodate either approach.

In the ITSOMart application, we opted for the approach of associating validation errors to each individual component. The approach of grouping all validation errors in one place might have reduced screen clutter, but it requires some specialized coding and configuration to work in a reasonable way. Associating error display fields for each individual input component is relatively easy to accomplish using Rational Software Architect, and offers a clear indication of where the error occurred.

Figure 9-9 on page 261 shows a section of the Create Customer screen in the ITSOMart application, illustrating the messages displayed in case of validation errors.


Your Name

First: Jeanine

Last: *required

Your Email / Id

Email: jeanine_email Invalid email format

Password:

Please Re-enter Password:

Your Address

Street Address: 21 Ashley Road

City: Markville

State/Province: MN

ZIP/Postal Code: 55072

Phone: (715) 656-405 Extension Validation Error: Value is greater than allowable maximum of '20'

Figure 9-9 Create customer input validation

Application and system errors

There is nothing particularly unique about the way we handle application and system errors in the JSF framework. Errors from the back-end are captured with exceptions just as in any other Java-based application. What JSF does have in its favor, is the flexible screen navigation framework that makes it quite simple to redirect responses to the appropriate error pages. As we have seen before, JSF actions return a plain outcome string, which gets mapped separately into a target JSP page.

For example, assume our create customer action checks for the appropriate exceptions, and chooses to report on the following conditions: a customer already exists with that e-mail, there was a DB2 error inserting the data, or there was another system exception (the *catch all* case). In that case, our action handler would return three possible error strings: duplicateEmail, dbError, or systemError. Now all we would have to do is setup the navigation rules such that these strings would direct the application to an appropriate JSP error page. Better yet, many of the error pages and navigation rules can easily be reused for all other pages.

Note that although we could have implemented that level of detail in the create customer error handling, in reality we did not. As we mentioned earlier, we wanted to concentrate our effort on other aspects of the technology. As it is, in the ITSOMart application, all internal application or system errors map to the same outcome string: systemError. This string is then mapped to the generic error page, SystemError.jsp, with a global navigation rule. Although a single error page is not realistic for most complex applications, this does show how quickly we can implement basic error handling in a JSF application.

9.3.5 Back-end interface

There are a number of patterns that can be used to integrate the JSF front-end with the back-end of an application. These include the command bean pattern, the business delegate pattern, session facade pattern, and data object pattern, among others. In this redbook, we concentrate on the new JSF technology, as well as the integration technologies used in the back end such as ESB, JCA, JMS, and so forth. We did not want to spend much time on this particular layer, and therefore chose to reuse classes from a larger ITSOMart ordering system, of which our customer registration application is a subset.

The ITSOMart front-end to back-end interface follows the session facade design pattern. The OrderSystemSessionFacade class offers coarse-grained services to access the entity beans that store the customer data, and the message-based credit check subsystem. It provides a simple remote interface for performing the basic operations triggered by the front-end: create, retrieve, update, and delete.

9.4 Application development guidelines

Now that we have thoroughly discussed the high-level and low-level design issues associated with the ITSOMart front-end, we will turn our attention to the development aspects. Specifically, we highlight the use of Rational Software Architect in building the application.

Note that given the scope of this redbook, we cannot expect to provide a comprehensive user guide for building a JSF application using Rational Software Architect. What we do instead, is touch on some of the key features, and see how they are applied in the construction of our ITSOMart example. Instead of describing every aspect of JSF development, we focus on the specific choices and guidelines which drove the development of the ITSOMart front-end.

The complete ITSO front-end source code is available for download so we will not attempt to walk through the entire process of constructing it from scratch. See Appendix B, “Additional material” on page 485. We show, however, in depth examples of how some of the key construction steps are performed.

9.4.1 Rational Software Architect development environment

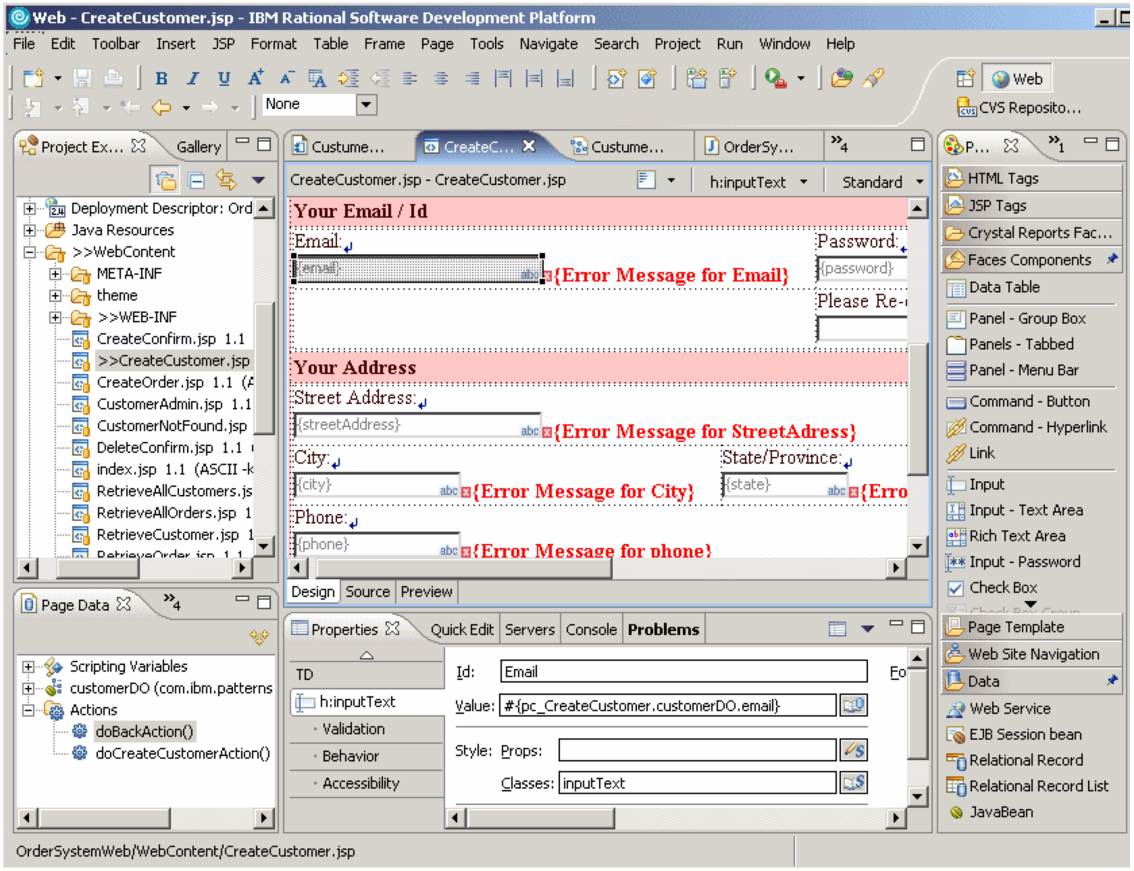
The development techniques we address in this chapter use the Web development tools available in Rational Software Architect, Rational Application Developer, and Rational Web Developer.

The OrderSystemWeb project

JSF applications should be developed under a dynamic Web project. In ITSOMart, the dynamic Web project for the ITSOMart front-end is called OrderSystemWeb. All JSP files, Java backing beans, page templates, CSS style sheets, and JSF configuration files are contained within that project. For the whole application to work, the project is deployed along with multiple other projects which make up the complete ITSOMart application. See “Description of application files” on page 454 for a description of the ITSOMart application project structure.

The Web perspective

Development of JSF applications is generally done under the Web perspective. Figure 9-10 on page 264 shows a bird’s eye view of the ITSOMart OrderSystemWeb project under the Web perspective.



Id: Email

Value: #{pc_CreateCustomer.customerDO.email}

Style: Props:

Classes: inputText

OrderSystemWeb/WebContent/CreateCustomer.jsp

Figure 9-10 Web perspective

The top center area shows the CreateCustomer.jsp file being edited in the Page Designer, and the bottom area shows the properties window. To the right is the palette containing the HTML, JSF and JSP UI components used to construct the front-end. The top left area is the standard project explorer, and the bottom left area shows the data associated with the CreateCustomer.jsp page.

9.4.2 Web page templates

Most Web applications have a standard look and feel that to all their pages. In addition to a standard Cascading Style Sheet (.CSS), which the ITSOMart application uses, pages often share common UI components, generally consisting of a header, footer, navigation map, and so on. Rational Software Architect facilitates the use of this feature by supporting page templates.

Creating the ITSOMart page template

The ITSOMart front-end application uses a page template. To construct the page template, we perform the following steps:

1. In the Project Explorer view, right-click the **OrderSystemWeb** dynamic Web project.
2. In the context menu select **New** → **Page Template File.**
3. When the New Page Template File wizard appears, enter the following:
   – Under File Name, enter `CustomerAdminTemplate`.
   – Under Model, select **Template Containing Faces Components.**
4. Click **Finish** and the template file is created and placed in the design canvas, ready to by constructed.
5. Complete the template by using the same HTML, JSP, and JSF UI components you would use on a regular Faces JSP page. For the ITSOMart template, we basically use HTML tables and graphic images to build the top banner which appears in all our screens.

The resulting template for the ITSOMart application is shown in Figure 9-11. The figure also shows a closer view of the palette used to add UI components. If we decide to update the template later, all pages which used it will automatically inherit the changes.

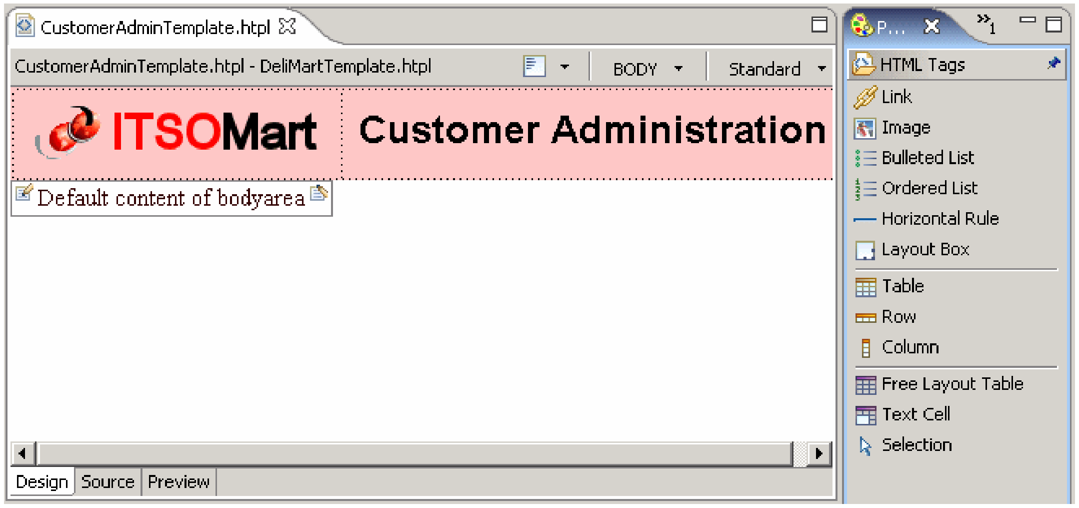


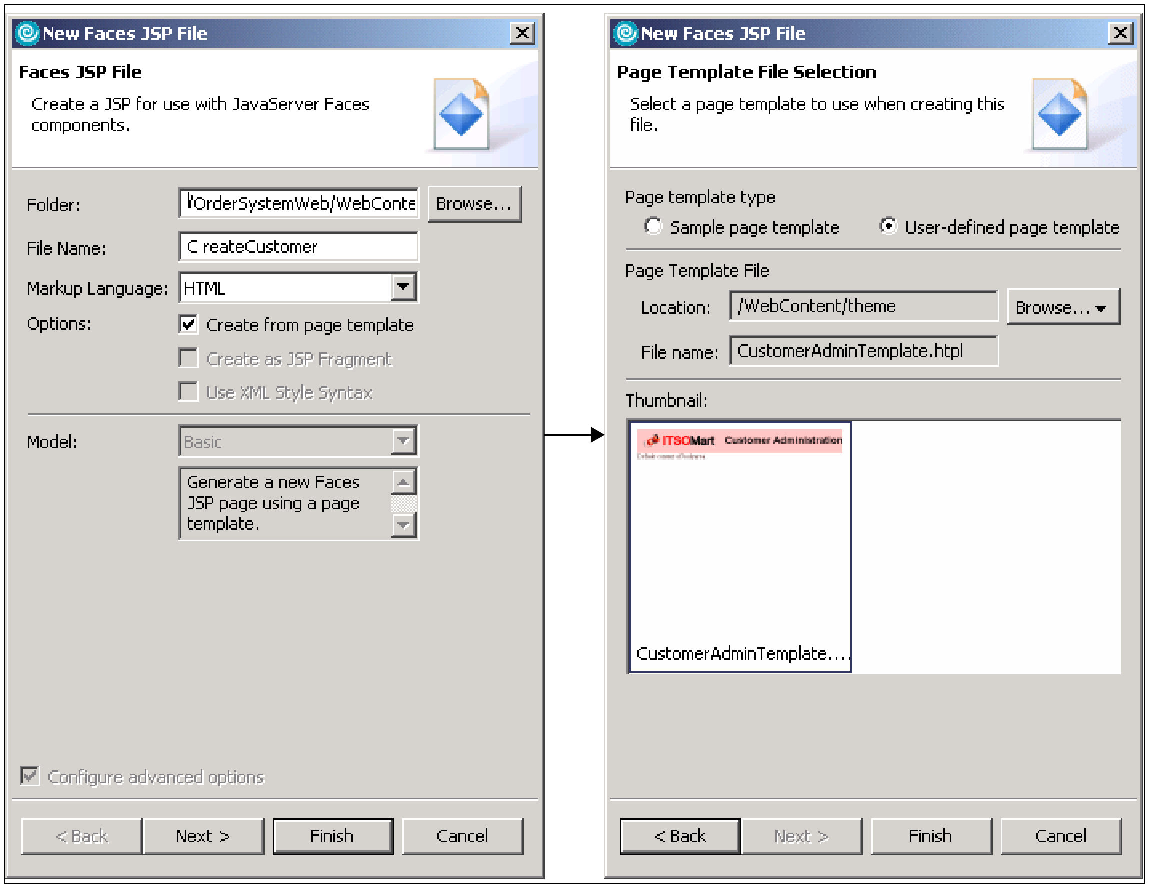
Figure 9-11 ITSOMart *page template*

Using the ITSOMart page template

Once a template is created, it can be used in two ways. We can apply it to an existing JSP page, or we can specify it to be used when we first create a JSP page.

For example, the following steps show how we applied the CustomerAdminTemplate template in the creation of the CreateCustomer.jsp Faces JSP file:

1. Right-click the **OrderSystemWeb** dynamic Web project.
2. In the context menu select **New** → **Faces JSP File**.
3. When the New Faces JSP Template File wizard appears, populate the information as shown in Figure 9-12 on page 267.
   a. Make sure to check the **Create from page template** option in the first screen.
   b. On the second screen, select **CustomerAdminTemplate** from the list.
4. Click **Finish** and the CreateCustomer.jsp file is created, using the template.


New Faces JSP File

Faces JSP File

Create a JSP for use with JavaServer Faces components.

Folder: /OrderSystemWeb/WebConte

File Name: CreateCustomer

Markup Language: HTML

Options: [✓] Create from page template

[ ] Create as JSP Fragment

[ ] Use XML Style Syntax

Model: Basic

Generate a new Faces JSP page using a page template.

[✓] Configure advanced options

New Faces JSP File

Page Template File Selection

Select a page template to use when creating this file.

Page template type

( ) Sample page template [✓ User-defined page template]

Page Template File

Location: /WebContent/theme

File name: CustomerAdminTemplate.htpl

Thumbnail:

CustomerAdminTemplate....

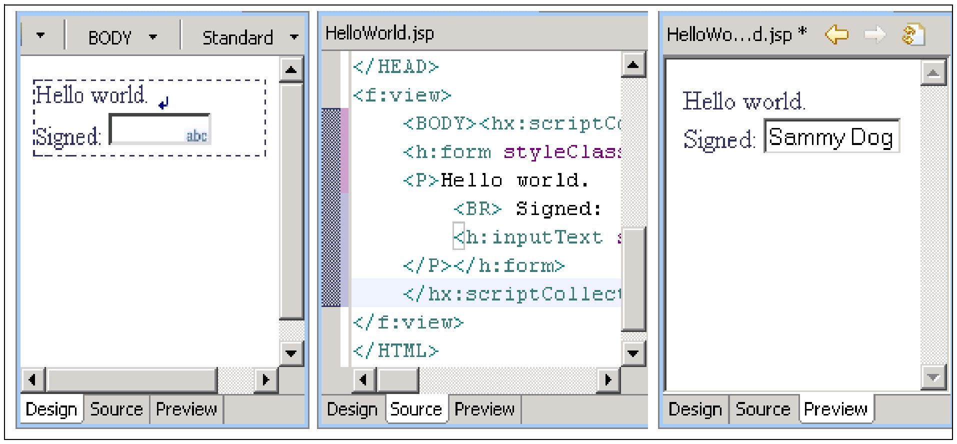
Figure 9-12   Creating JSP page specifying a template

9.4.3 Designing screens using the Page Designer

The Page Designer is the basic Rational Software Architect tool for designing Web pages. It is generally located in the center of the Web perspective. It provides three different tabs for editing pages:

- The Design tab provides a canvas where UI components are dropped from the palette (usually located to the right of the design panel), and arranged for the desired look and feel.
- The Source tab displays the HTML, JSP, and JSF source code.
- The Preview tab provides an approximation of the way the screen will look on a Web browser.

Figure 9-13 shows an example of the three tabs of the Page Designer. We deviate here from our standard create user use case, simply because those screens would be to wide to fit side by side in the page.



```
</HEAD>
<f:view>
    <BODY><hx:scriptCo
    <h:form styleClass
    <P>Hello world.
        <BR> Signed:
        <h:inputText s
    </P></h:form>
    </hx:scriptCollect
</f:view>
</HTML>
```

Figure 9-13 Page designer three tabs

In addition to the standard HTML and JSP tags, JSF pages have a number of JSF UI components available for screen design. Not only are there standard JSF tags from the reference J2EE implementation, but the framework also allows for easy incorporation of additional tags from other vendors. For instance, Rational Software Architect offers an IBM extension library which supports a number of additional tags.

All pages in the ITSOMart application are Faces JSP pages that use JSF UI components. In the subsequent sections, we show how these UI components interact with the JSF managed beans, perform standard validation, and are used to trigger actions.

9.4.4 Binding UI components to managed beans

As we discussed in 9.3.2, "JSF managed bean design" on page 257, managed beans are a critical part of a JSF application. This section will show how we create the managed beans used in the ITSOMart application, and how they are bound to the JSF UI components.

Creating backing beans

As previously noted in 9.3.2, "JSF managed bean design" on page 257, when using Rational Software Architect to create a Faces JSP page, a backing bean is automatically created, and a definition for it is added to the faces-config.xml file. The naming convention is such that the bean class will have the same base name as the JSP page, only varying in the extension (.java instead of .jsp). The managed bean is named `pc_<class>`, where `<class>` is the name of the backing bean class. For example, when the CreateCustomer.jsp file is created, the CreateCustomer.java bean class is automatically created. CreateCustomer.java will have methods to get and set each JSF UI component in the JSP file. The pc_CreateCustomer managed bean definition is added to the faces-config.xml file as in Example 9-3:

Example 9-3 The pc_CreateCustomer managed bean definition

```
<managed-bean>
   <managed-bean-name>pc_CreateCustomer</managed-bean-name>
   <managed-bean-class>pagecode.CreateCustomer</managed-bean-class>
   <managed-bean-scope>request</managed-bean-scope>
</managed-bean>
```

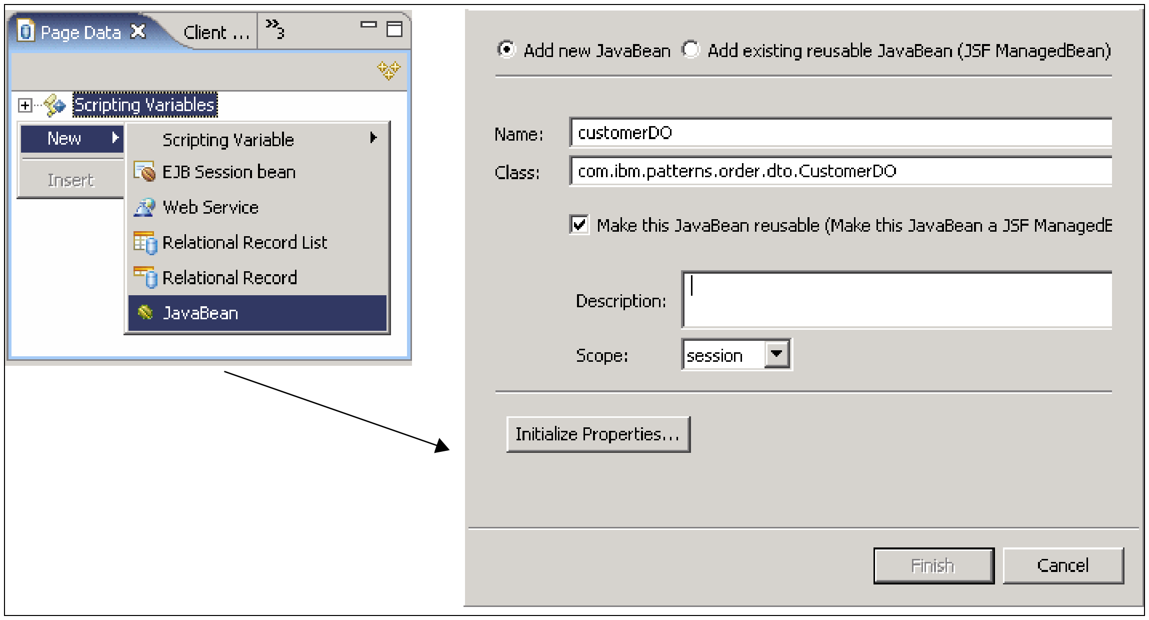
Creating the CustomerDO data bean

The customerDO data bean is used to store the customer data, and transfer it across the different pages.

The first step is to create the CustomerDO Java class. This is a standard class, which we create under the com.ibm.patterns.order.dto package. The class should have data members for each piece of customer information, such as first name, last name, e-mail, address, etc. The class will also have the standard get/set methods to set and retrieve the properties.

After the class is created, in order to declare a new data bean customerDO, we use the page data window as follows:

1. Open one of the JSP files where the data bean is to be used. In our case, we can use the CreateCustomer.jsp file we have used in other examples.
2. On the page data window, right-click and select **New** → **JavaBean.**
3. When the Add Java Bean wizard appears, enter the customerDO bean information as shown in Figure 9-14 on page 270. Note that we need to check the **Make this JavaBean reusable** check box, and set the scope to **session,** in order to make the data available for other pages.
4. Click **Finish** and the customerDO bean is created.



(● Add new JavaBean) ○ Add existing reusable JavaBean (JSF ManagedBean)

Name: customerDO

Class: com.ibm.patterns.order.dto.CustomerDO

[✓] Make this JavaBean reusable (Make this JavaBean a JSF ManagedE

Description:

Scope: session

Figure 9-14   Creating customerDO managed bean

When the bean is created through the wizard, the following lines in Example 9-4 are automatically added to the faces-config.xml:

Example 9-4   Creating the customerDO managed bean in faces-config.xml

```
   <managed-bean>
      <managed-bean-name>customerDO</managed-bean-name>
      <managed-bean-class>com.ibm.patterns.order.dto.CustomerDO</managed-bean-
      class>
      <managed-bean-scope>session</managed-bean-scope>

</managed-bean>
```

After the bean is created, in order to add it to other pages such as the UpdateCustomer.jsp page, the process is nearly the same. The only difference is that in the Add JavaBean wizard, we select **Add existing reusable Java Bean** instead. The customerDO bean will appear in a table where we can select it.

Once a bean has been added to a JSP page, it appears in its associated page data window. Figure 9-15 on page 271 shows the resulting page data for the CreateCustomer.jsp page after the customerDO bean is added to it.

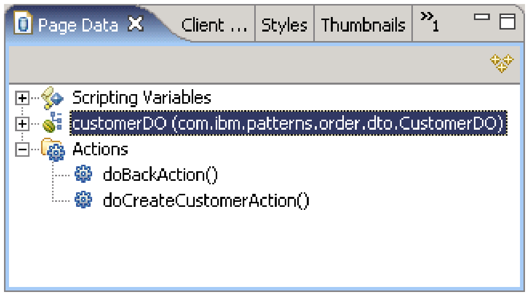
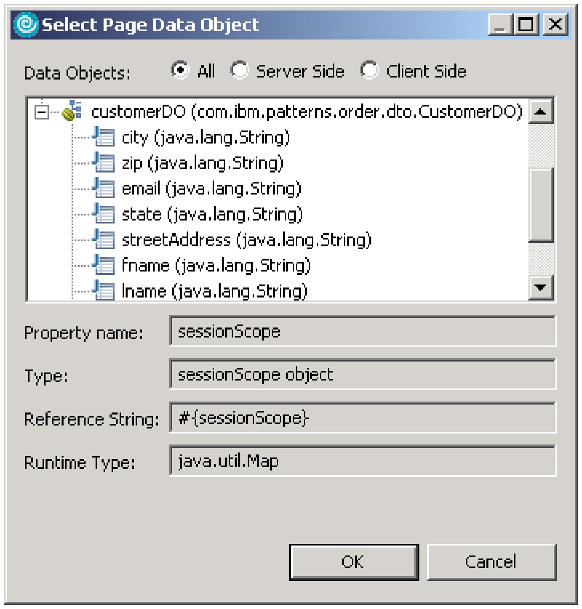


Figure 9-15 CreateCustomer.jsp page data window

Binding UI components to managed beans

Now that we have discussed the creation of both UI components and managed beans, we examine how to bind them. For example, the following steps describe the steps for binding the Last Name field in the CreateCustomer.jsp page to the lname property in customerDO.

1. Open CreateCustomer.jsp on the Design tab of the Page Designer.
2. Click the JSF **inputField** component used for the last name, which should populate the Properties window on the bottom. Select the **Properties** tab if it is not yet selected.
3. In the **Value** field, click the small icon to the right of the field. That brings up the Select Page Data Object wizard, as shown in Figure 9-16 on page 272.
4. Expand the **customerDO** bean, select the **lname** property, and click **Ok**.



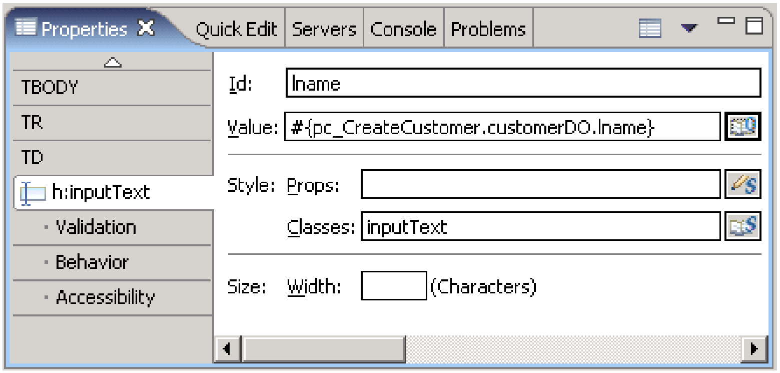
Data Objects: [✓ All] Server Side Client Side

```
customerDO (com.ibm.patterns.order.dto.CustomerDO)
    city (java.lang.String)
    zip (java.lang.String)
    email (java.lang.String)
    state (java.lang.String)
    streetAddress (java.lang.String)
    fname (java.lang.String)
    lname (java.lang.String)
```

Property name:	sessionScope
Type:	sessionScope object
Reference String:	#{sessionScope}
Runtime Type:	java.util.Map

Figure 9-16 Selecting bind variable property

5. Now the Properties tab should show the value bind expression as seen in Figure 9-17.



Id:	lname
Value:	#{pc_CreateCustomer.customerDO.lname}
Style: Props:	
Classes:	inputText
Size: Width:	(Characters)

Figure 9-17 Bind value populated

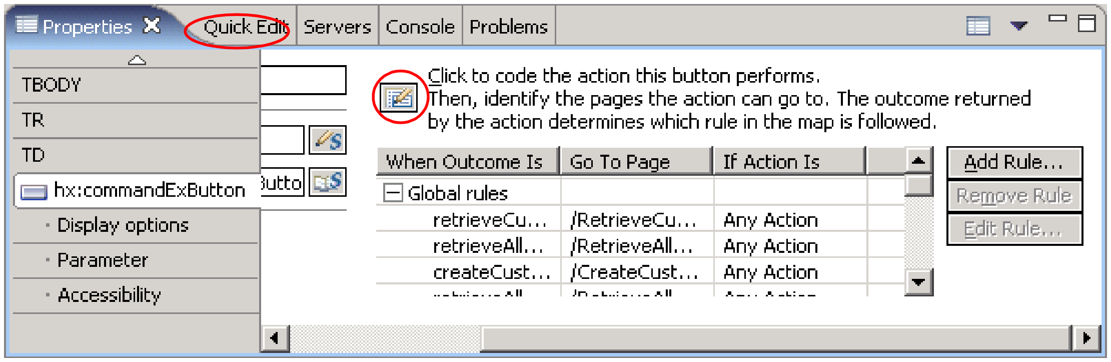
With the steps above completed, changes in the last name screen field will update the lname property in the customerDO session variable, and vice-versa.

Binding command buttons to action methods

To trigger actions from a Faces JSP page, we use JSF command buttons or command link UI components. When triggering an action, we must link the button with the code that will handle it. As we saw in 9.3.2, "JSF managed bean design" on page 257, these methods can be placed in dedicated action beans, but the standard behavior is to simply add them to the backing bean for the JSP page.

In our customer create use case, we used the following steps to bind the Create button to the appropriate method on the CreateCustomer.java backing bean.

1. Open **CreateCustomer.jsp** on the **Design** tab.
2. Click the **Create** commandButton JSF component, which should populate the Properties window on the bottom. Select the **Properties** tab if it is not yet selected.
3. Click the small icon on top right side of the Properties window, as circled in Figure 9-18. The icon is labeled, **Click to code the action this button performs**. This will take you to the Command Quick Edit screen. Alternatively, you may click the **Quick Edit** tab.

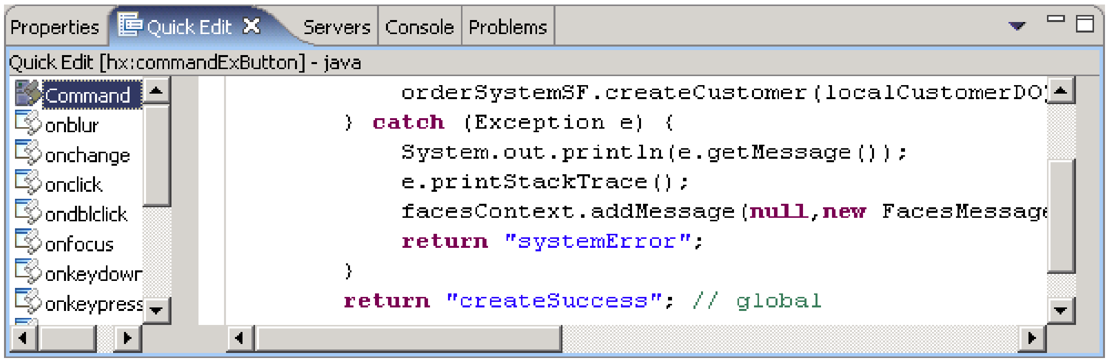


Click to code the action this button performs.
Then, identify the pages the action can go to. The outcome returned by the action determines which rule in the map is followed.

When Outcome Is	Go To Page	If Action Is
Global rules		
retrieveCu...	/RetrieveCu...	Any Action
retrieveAll...	/RetrieveAll...	Any Action
createCust...	/CreateCust...	Any Action

Figure 9-18 Command button properties

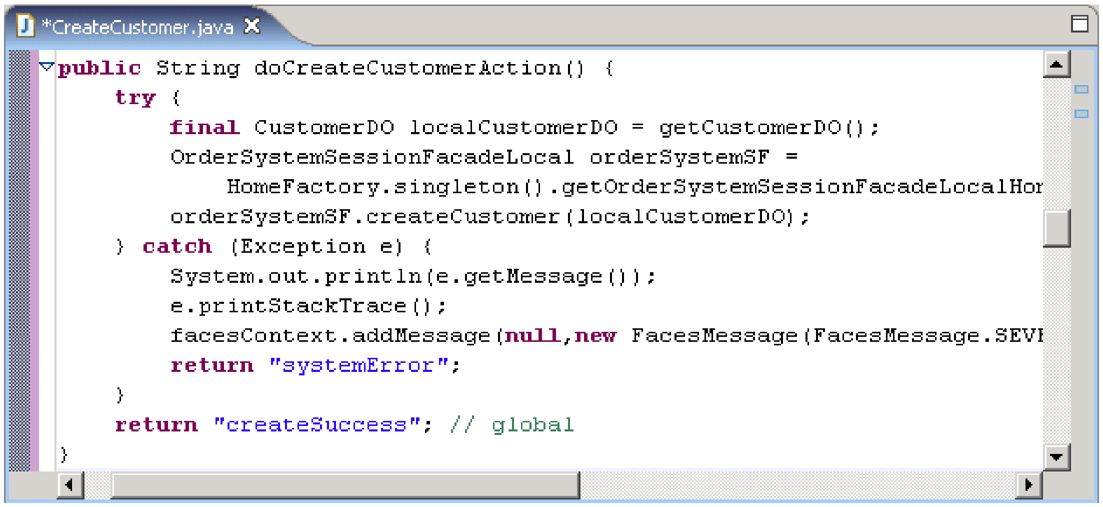
4. As shown in Figure 9-19 on page 274, enter the code you want to execute when the Create button is clicked.



```
            orderSystemSF.createCustomer(localCustomerDO
        } catch (Exception e) {
            System.out.println(e.getMessage());
            e.printStackTrace();
            facesContext.addMessage(null,new FacesMessage
            return "systemError";
        }
        return "createSuccess"; // global
```

Figure 9-19 Command Quick Edit window

5. Now if you check the CreateCustomer.java backing bean source code (see Figure 9-20), you see the method doCreateCustomerAction is created, using the code entered in the Quick Edit screen.



```
public String doCreateCustomerAction() {
    try {
        final CustomerDO localCustomerDO = getCustomerDO();
        OrderSystemSessionFacadeLocal orderSystemSF =
            HomeFactory.singleton().getOrderSystemSessionFacadeLocalHor
        orderSystemSF.createCustomer(localCustomerDO);
    } catch (Exception e) {
        System.out.println(e.getMessage());
        e.printStackTrace();
        facesContext.addMessage(null,new FacesMessage(FacesMessage.SEVI
        return "systemError";
    }
    return "createSuccess"; // global
}
```

Figure 9-20 CreateCustomer action method

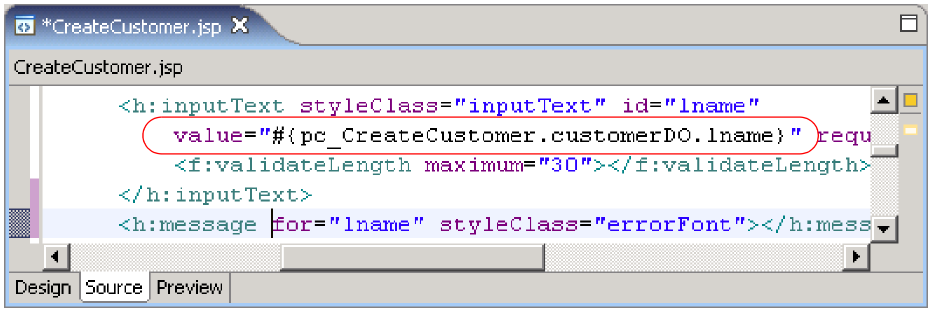
Now when you click the **Create** button, the doCreateCustomerAction method is called.

Code view of managed bean binding

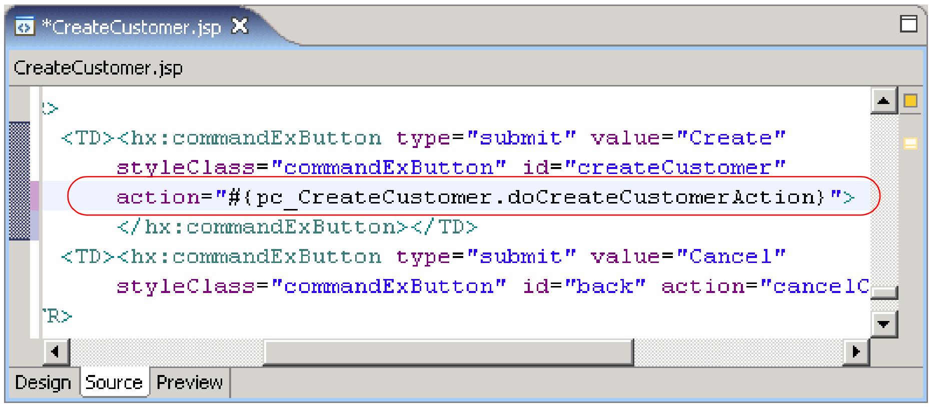
When using a sophisticated IDE such as Rational Software Architect, we may at times lose track of how the actions we performed using the different wizards, translate into the JSF/JSP code which will ultimately be deployed on the server.

Using the Source tab in the page designer, we can inspect the actual source code of the JSP files. Figure 9-21 shows the portions of source code corresponding to the two examples we described earlier. This is code which was generated by Rational Software Architect based on the information we entered in the properties window and the different wizards.

The first window shows the code binding the last name input field to the lname field in the customerDO managed bean. The second window shows the code binding the Create button to the doCreateCustomerAction method in the pc_CreateCustomer backing bean.



```
        <h:inputText styleClass="inputText" id="lname"
            value="#{pc_CreateCustomer.customerDO.lname}" requ
            <f:validateLength maximum="30"></f:validateLength>
        </h:inputText>
        <h:message for="lname" styleClass="errorFont"></h:mess
```



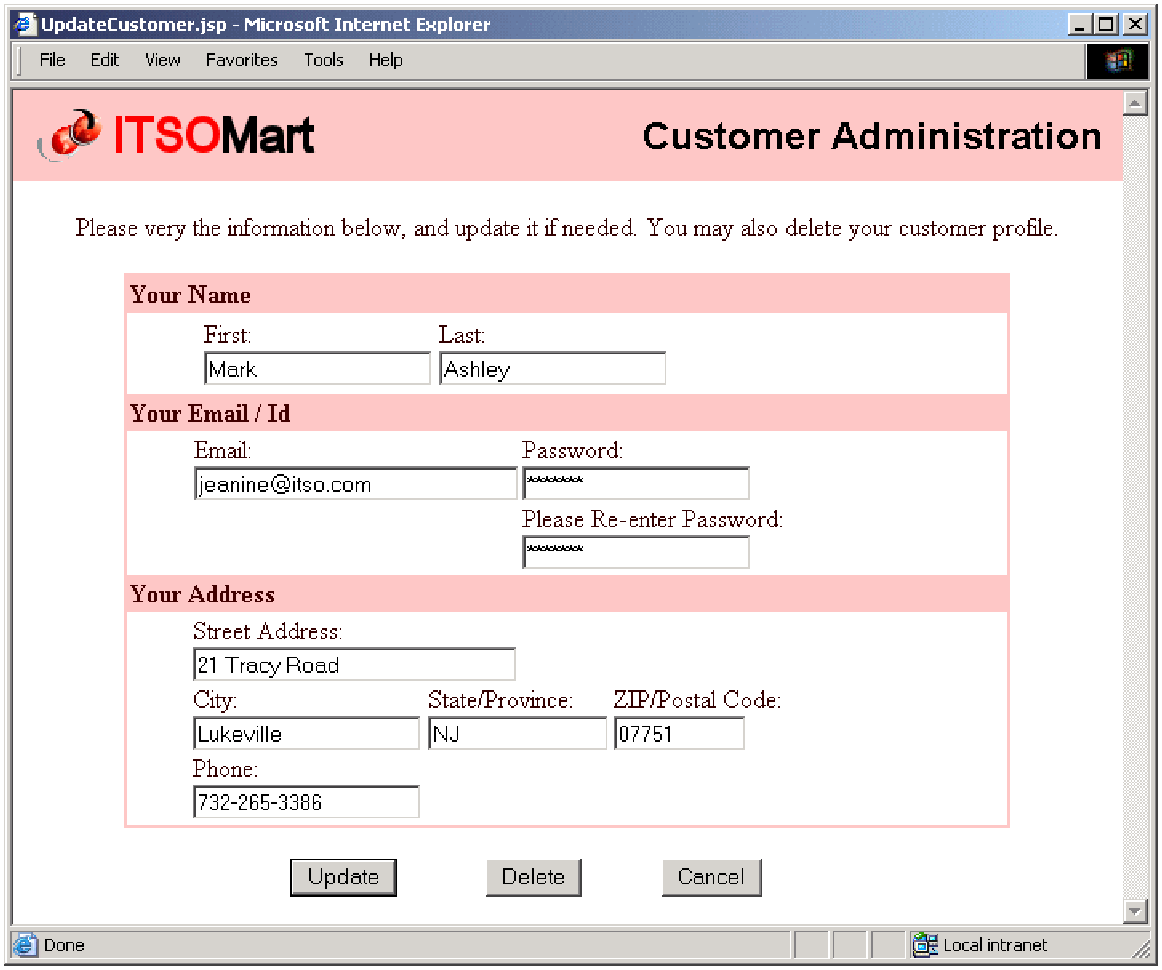
```
>
  <TD><hx:commandExButton type="submit" value="Create"
      styleClass="commandExButton" id="createCustomer"
      action="#{pc_CreateCustomer.doCreateCustomerAction}">
      </hx:commandExButton></TD>
  <TD><hx:commandExButton type="submit" value="Cancel"
      styleClass="commandExButton" id="back" action="cancelC
R>
```

Figure 9-21 CreateCustomer.jsp data binding source code

9.4.5 Implementing page navigation

Section 9.3.1, “ITSOMart Web diagram” on page 253, provides a very good overview of the page navigation in JSF, and specifically in the ITSOMart application. This section explains the low-level details of how we use Rational Software Architect to setup the navigation rules.

For this particular example, we depart from our usual create customer use case, and instead look at the update customer screen. The reason is simply because that screen offers more possible outcomes, which makes it a more interesting example to describe navigation. Figure 9-22 shows the ITSOMart update screen, allowing the user to update their profile information, delete, or cancel the operation.


UpdateCustomer.jsp - Microsoft Internet Explorer
File Edit View Favorites Tools Help
ITSOMart
Customer Administration

Please very the information below, and update it if needed. You may also delete your customer profile.

Your Name

First: Mark
Last: Ashley

Your Email / Id

Email: jeanine@itso.com
Password: *******
Please Re-enter Password: *******

Your Address

Street Address: 21 Tracy Road
City: Lukeville
State/Province: NJ
ZIP/Postal Code: 07751
Phone: 732-265-3386

Update
Delete
Cancel
Done
Local intranet

Figure 9-22 Update Customer Screen

Update customer Web diagram

Section 9.3.1, "ITSOMart Web diagram" on page 253 introduces the concept of Web diagrams. In that section however, the diagram is used to illustrate the complete end-to-end flow of the ITSOMart application and omits such details as the managed beans being invoked to trigger actions. Figure 9-23 on page 277 provides a more detailed Web diagram illustrating the screen flow design and navigation for the update customer use case.

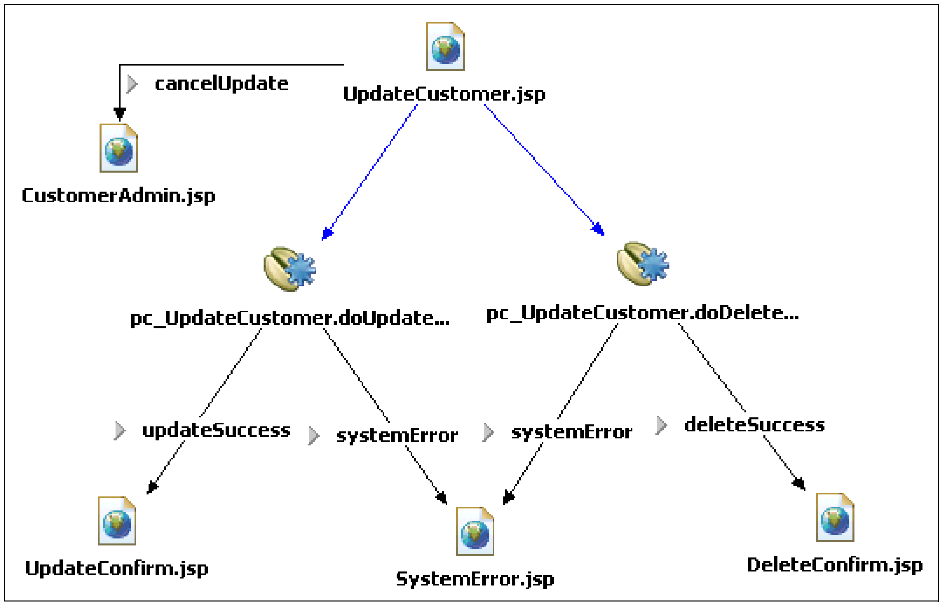


Figure 9-23 Update customer Web diagram

Reading the diagram, we can infer the following information about the screen flow of the update customer use case.

- From the UpdateCustomer.jsp page, there are three possible navigation scenarios:
  - The user can trigger an action to update the profile, with the doUpdateCustomerAction method of the pc_UpdateCustomer backing bean. Note that the method name is truncated by Rational Software Architect in the Web diagram.
  - The user can trigger an action to delete the profile with the doDeleteAction method of the pc_UpdateCustomer backing bean.
  - The user can cancel out of the update screen, and simply navigate back to the home page, CustomerAdmin.jsp.
- Depending on whether the update or delete actions are successful, the user application navigates to an appropriate confirmation screen, or to a generic system error page.

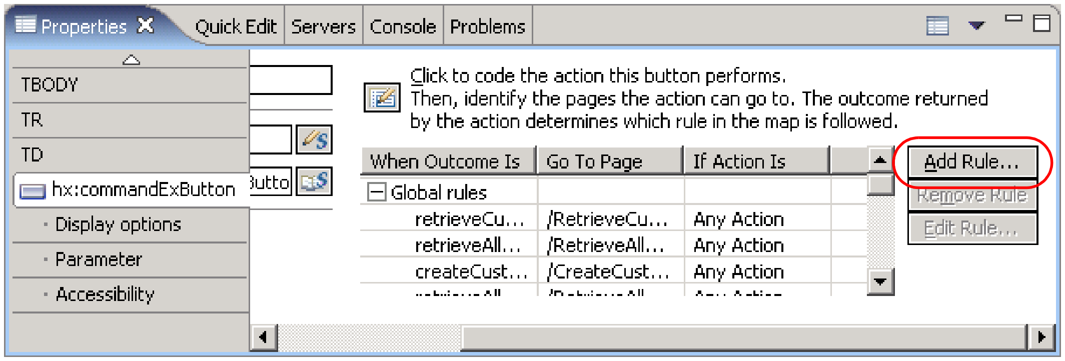
Implementing customer update navigation

Now that we understand how we want the use case to flow, we show how we implement the flow using Rational Software Architect. We begin by setting up a navigation rule for the Update button in the UpdateCustomer.jsp page.

Assume that before we start, we first create an action method, doUpdateCustomerAction, in the UpdateCustomer.java backing bean. This method is bound to the Update button in the UpdateCustomer.jsp page, following the steps previously described in “Binding command buttons to action methods” on page 273. The method will return the outcome string of updateSuccess, if the customer information is successfully updated. If there are any errors, the method returns systemError.

Now we can add the navigation rule as follows.

1. Open **UpdateCustomer.jsp** in the **Design** tab.
2. Click the **Update** commandButton JSF component, which should populate the Properties window on the bottom.
3. Scroll to the right portion of the Properties window, where you will find the navigation rules table, as shown in Figure 9-24. Click the **Add Rule** button.

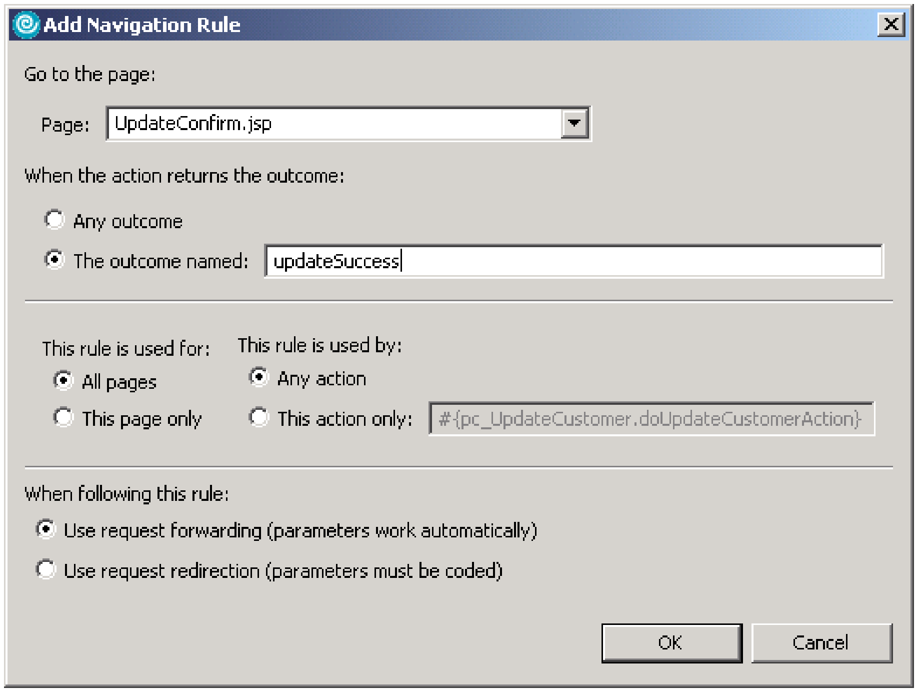


Click to code the action this button performs.
Then, identify the pages the action can go to. The outcome returned by the action determines which rule in the map is followed.

When Outcome Is	Go To Page	If Action Is
Global rules		
retrieveCu...	/RetrieveCu...	Any Action
retrieveAll...	/RetrieveAll...	Any Action
createCust...	/CreateCust...	Any Action

Figure 9-24 Navigation rules table

4. When the Add Navigation Rule wizard appears, enter the following, as shown in Figure 9-25 on page 279:


Add Navigation Rule

Go to the page:

Page: UpdateConfirm.jsp

When the action returns the outcome:

( ) Any outcome
(●) The outcome named: updateSuccess

This rule is used for:
(●) All pages
( ) This page only

This rule is used by:
(●) Any action
( ) This action only: #{pc_UpdateCustomer.doUpdateCustomerAction}

When following this rule:
(●) Use request forwarding (parameters work automatically)
( ) Use request redirection (parameters must be coded)

Figure 9-25 Add Navigation Rule wizard

   a. Under Page, select **UpdateConfirm.jsp.**
   b. Under **The outcome named**, enter `updateSuccess`.
   c. Check **All pages** and **Any action.**

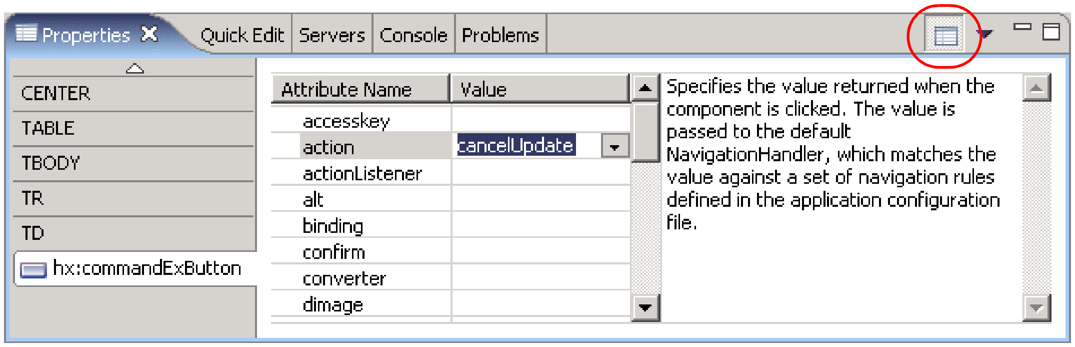
5. Click **OK**, and the navigation rule is created.
6. Clearly, adding the rule for the Delete button follows much the same procedure. The only difference is in the outcome string (deleteSuccess), and the target JSP (DeleteConfirm.jsp).
7. The rule directing errors to the SystemError.jsp page is even simpler, because the same rule applies to all pages in the ITSOMart application. It is a global rule that sets the target of SystemError.jsp for the systemError outcome, regardless of the current page or action.

Implementing a simple page redirection

In the previous example, the navigation is based on the result of some action being performed. In many cases however, the navigation is a simple static link to a different page. Those scenarios do not require an action method to be created in the backing bean.

One such case, is the cancel button in the UpdateCustomer.jsp page. In that case, we simply want to be redirected to the home page (CustomerAdmin.jsp). We can implement that as follows:

1. Following the directions in the previous section, we can create a navigation rule that maps the outcome string cancelUpdate to theUpdateCustomer.jsp page.
2. Click the **Cancel** commandButton JSF component, which should populate the Properties window on the bottom.
3. Click the **All Attributes** icon, as shown on the top right portion of Figure 9-26, to display the complete attribute list for the Cancel button.



Attribute Name	Value
accesskey	
action	cancelUpdate
actionListener	
alt	
binding	
confirm	
converter	
dimage	

Specifies the value returned when the component is clicked. The value is passed to the default NavigationHandler, which matches the value against a set of navigation rules defined in the application configuration file.

Figure 9-26 Cancel update command detailed attributes

4. Select or type the outcome string `cancelUpdate` in the **action** attribute.

Now the Cancel button will navigate to the UpdateCustomer.jsp directly, without any action method being created on the backing bean.

9.4.6 Implementing input validation

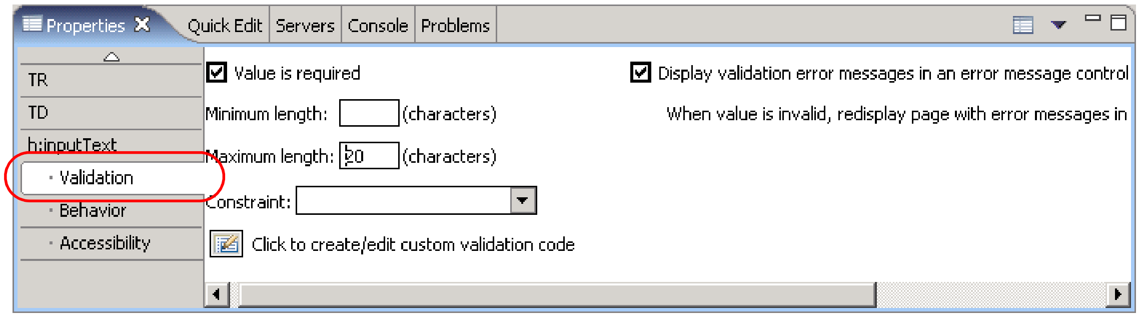
This section describes how input validation was implemented in the ITSOMart front-end. The application makes use of both standard validation available in the JSF components, as well as custom-defined validation for the e-mail format.

Implementing standard validation

Most JSF UI components include at least some level of standard input validation. These include checks for required values, string length, and date formats. For example, the following steps describe how to setup standard validation for the first name component of the CreateCustomer.jsp page.

1. Open CreateCustomer.jsp on the **Design** tab of the page designer.

2. Click the JSF **inputField** component used for the first name, which should populate the Properties window on the bottom. Select the **Properties** tab if it is not yet selected.
3. Select the **Validation** panel in the left navigation bar.
4. In the Validation panel, enter the options, as shown in Figure 9-27:
   - Check the **Value is required** option, to indicate first name is a mandatory field.
   - Set Maximum length to **20**.
   - Check the option **Display validation error messages in an error message control**. This will automatically place a Display Error JSF component adjacent to the first name input field. The field will display the appropriate validation errors when needed.



☑ Value is required

Minimum length: (characters)

Maximum length: 20 (characters)

Constraint:

☑ Display validation error messages in an error message control

When value is invalid, redisplay page with error messages in

Figure 9-27 Standard validation panel

Implementing custom validation

As we have just seen, standard validation is easy to implement in Rational Software Architect. However, it only supports simple data types. In the ITSOMart application, we make use of custom-defined validation to check the syntax of e-mail addresses.

Unlike the standard validation, custom validation requires a rather manual process. The following sections show how we implement the special e-mail format validation in the ITSOMart front-end.

Implement validator interface

The first step is to define a Java class to validate the e-mail address. This class must implement the javax.faces.validator.Validator, and specifically the validate method. For the ITSOMart application, we created the EmailValidator class. It provides a simple validation of the most common e-mail format (user@domain), using Java pattern matching. There are more sophisticated e-mail validation functions, but for the purpose of this example, this one should suffice. Example 9-5 shows the code for the EmailValidator class.

Example 9-5 EmailValidator class

```
public class EmailValidator implements Validator {

   public void validate(FacesContext arg0, UIComponent arg1, Object arg2)
         throws ValidatorException {
      System.out.println("EmailValidator start");
      UIInput field = (UIInput) arg1;
      String emailValue = (String)arg2;

      // use regular expression to identify a reasonable e-mail
      Pattern pat = Pattern.compile(".+@.+\\.[a-z]+");

      if (!(pat.matcher(emailValue).matches())) {
            FacesMessage errmsg = new
FacesMessage(FacesMessage.SEVERITY_ERROR,
               "Invalid e-mail format", "Invalid e-mail format");
         throw new ValidatorException(errmsg);
      }
   }
}
```

Registering the validator in the faces-config.xml file

Once we have created our validator class, we need to register it by adding the lines in Example 9-6 to the faces-config.xml file.

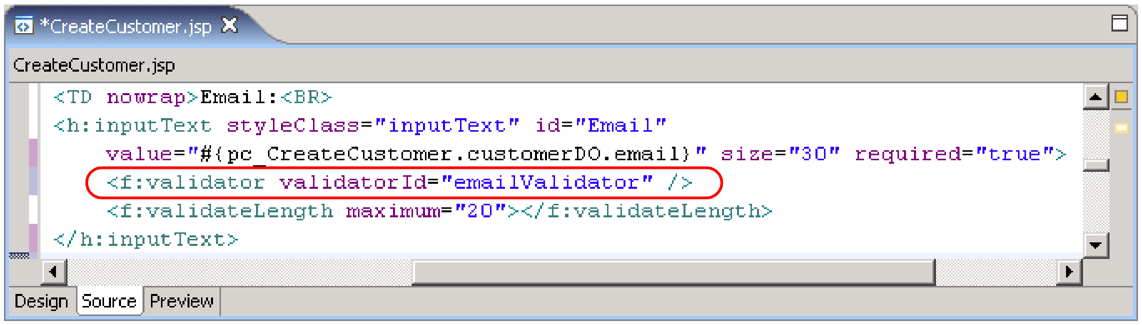
Example 9-6 Adding the validator class to faces-config.xml

```
<validator>
   <validator-id>emailValidator</validator-id>
   <validator-class>
      com.ibm.patterns.orderutil.web.EmailValidator
   </validator-class>
</validator>
```

Attach validator to UI component

Now that we have created and registered the validator class, we need to attach it to the e-mail UI component, using the *`<f:validator/>`* JSF tag. The component Properties window of Rational Software Architect does not support this tag, so we need to manually enter it in the source code.

Figure 9-28 on page 283 shows the source code for the validator being applied to the email field of the CreateCustomer.jsp page.



```
<TD nowrap>Email:<BR>
<h:inputText styleClass="inputText" id="Email"
    value="#{pc_CreateCustomer.customerDO.email}" size="30" required="true">
    <f:validator validatorId="emailValidator" />
    <f:validateLength maximum="20"></f:validateLength>
</h:inputText>
```

Figure 9-28 Using custom e-mail validator

Once the validator is associated with inputText, the validation behaves as the standard one would. You can associate a Display Error JSF component adjacent to the email field, and validation errors would be displayed in the same way standard errors were. In fact, looking back at Figure 9-9 on page 261, you can see that the standard validation error for last name has the same general appearance as the custom one we applied on the e-mail address.

9.4.7 Debugging applications in Rational Software Architect

Rational Software Architect provides a comprehensive debugging facility, which also supports JSF. Although a complete discussion of the debugging features would be out of the scope of this chapter, we do provide an introduction of how to perform a simple debugging task on a JSF application.

Using our customer creation example, we imagine a situation where the customer profile was not entered correctly in the database. In this scenario, we suspect the values entered by the user in the screen are being corrupted somewhere before they are inserted in the database. To debug this problem, we might want to see the values collected from the UI components, just before they are passed in to the session facade. The following subsections present the different steps we would take to perform that debugging task.

Starting the server in debug mode

Before we can perform debugging on the application server, we need to start it in debugging mode. To following steps describe that process:

1. In the workspace, open the **Servers** view, which is generally available as a one of the tabs in the bottom center area.

1. Right-click the server you are using. In our example, we used the WebSphere Application Server V6 test environment that comes integrated with Rational Software Architect.

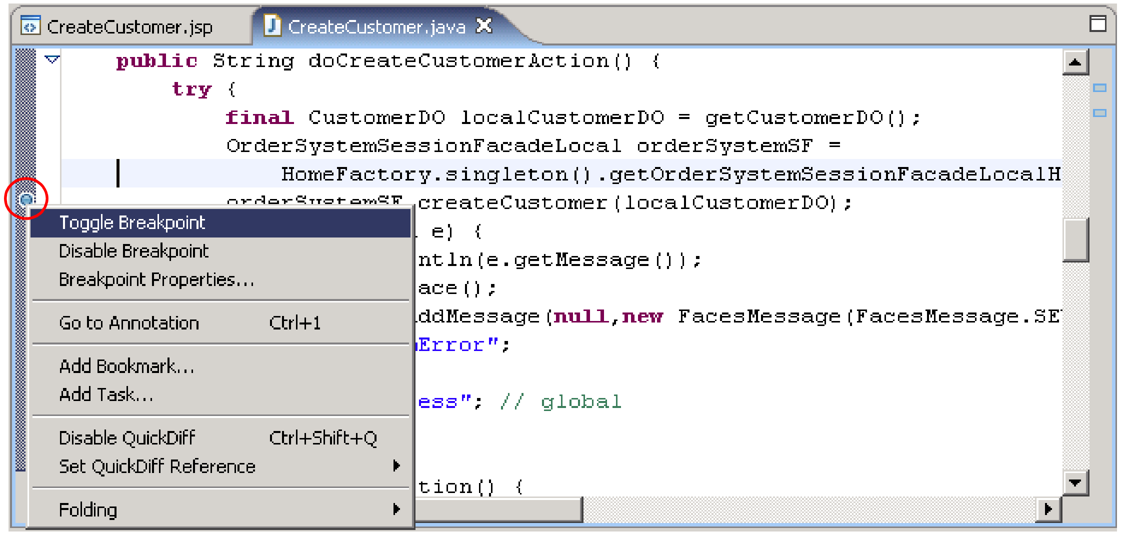
2. In the context menu select **Debug**. Alternatively, a short cut icon with the usual bug picture is also available on the top right side of the Servers view.

The WebSphere server starts in debug mode. Note that if the server was already running in normal mode, it would have to be restarted to operate in debug mode.

Setting breakpoints

The next step is to set a breakpoint to stop the code execution at the right spot. In our case, we want to stop inside the doCreateCustomerAction method of the CreateCustomer managed bean, just before we call the createCustomer method in the session facade. This is accomplished with the following steps:

1. Open the **CreateCustomer.java** file in the standard Java editor window.
2. Right-click the **marker bar** (vertical ruler) to the left of the line at which we want to stop the execution flow. This should bring up a context menu, as seen in Figure 9-29.



```
    public String doCreateCustomerAction() {
        try {
            final CustomerDO localCustomerDO = getCustomerDO();
            OrderSystemSessionFacadeLocal orderSystemSF =
                HomeFactory.singleton().getOrderSystemSessionFacadeLocalH
            orderSystemSF.createCustomer(localCustomerDO);
                                        e) {
                                       ntln(e.getMessage());
                                       ace();
                                       ddMessage(null,new FacesMessage(FacesMessage.SE
                                       Error";

                                       ess"; // global


                                       tion() {
```

Figure 9-29 Setting a breakpoint

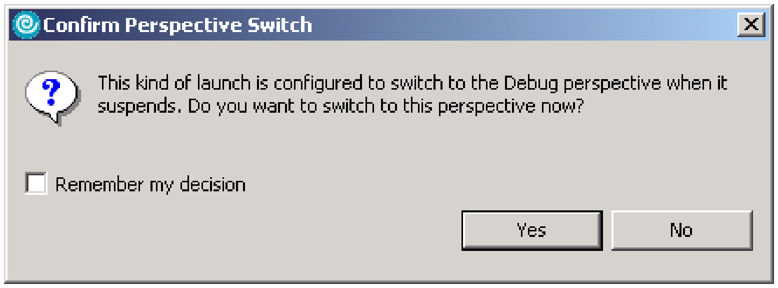
3. In the context menu select **Toggle Breakpoint.**

   The breakpoint is set. A small blue circle on the marker bar, indicates the breakpoint location.

Debug the application on a server

To begin the debugging session, start the application in debug mode on the server. We start at the home page CustomerAdmin.jsp as follows:

1. Right-click the **CustomerAdmin.jsp** in the Project Explorer view.
2. In the context menu select **Debug** → **Debug on Server ...**
3. The application opens on the standard Internet browser. Follow the customer creation use case, entering information about the user on the CreateCustomer.jsp page, and then clicking the **Create** button.
4. When the code reaches the breakpoint we set, it will stop execution, and prompt the user to switch to the debug perspective, as seen in Figure 9-30.


Confirm Perspective Switch

This kind of launch is configured to switch to the Debug perspective when it suspends. Do you want to switch to this perspective now?

Remember my decision

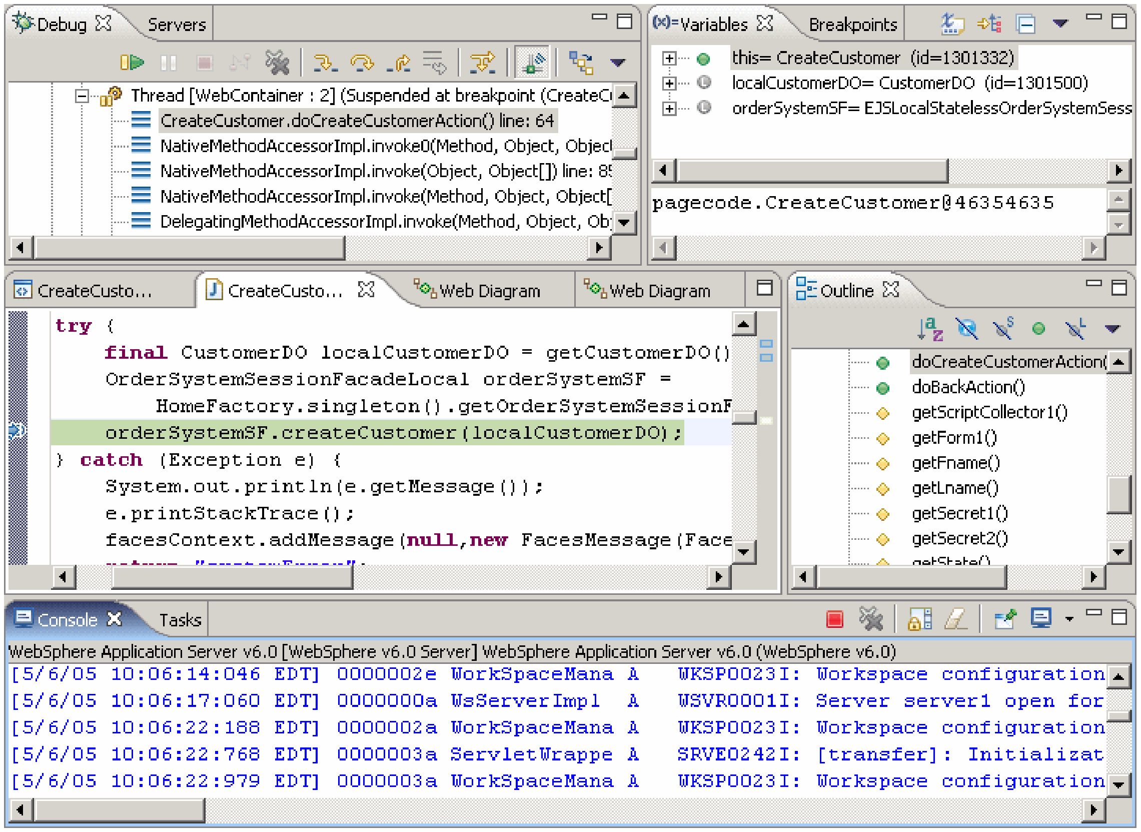
Figure 9-30 Confirm switch to debug perspective

5. Click **Yes**, and the workspace changes to the Debug perspective, which we discuss on the next section.

Debug perspective

Figure 9-31 on page 286 provides a bird's eye view of the debug perspective in Rational Software Architect. The default view shows the following panels:

- The Debug view at top left highlights the stack trace for the thread we are currently debugging.
- The Variable view at top right displays different application and system variables.
- The middle area shows the code where the execution was suspended, and the outline for the class. The exact line where the code stopped, is highlighted. In our case, this is the line on the CreateCustomer.java managed bean where we are calling the createCustomer method in the session facade.
- The Console view at the bottom shows the standard WebSphere console error messages.



```
Thread [WebContainer : 2] (Suspended at breakpoint (CreateCu
    CreateCustomer.doCreateCustomerAction() line: 64
    NativeMethodAccessorImpl.invoke0(Method, Object, Object
    NativeMethodAccessorImpl.invoke(Object, Object[]) line: 85
    NativeMethodAccessorImpl.invoke(Method, Object, Object[
    DelegatingMethodAccessorImpl.invoke(Method, Object, Ob
```

```
this= CreateCustomer  (id=1301332)
localCustomerDO= CustomerDO  (id=1301500)
orderSystemSF= EJSLocalStatelessOrderSystemSess

pagecode.CreateCustomer@46354635
```

```
try {
    final CustomerDO localCustomerDO = getCustomerDO()
    OrderSystemSessionFacadeLocal orderSystemSF =
        HomeFactory.singleton().getOrderSystemSessionF
    orderSystemSF.createCustomer(localCustomerDO);
} catch (Exception e) {
    System.out.println(e.getMessage());
    e.printStackTrace();
    facesContext.addMessage(null,new FacesMessage(Face
```

```
doCreateCustomerAction(
doBackAction()
getScriptCollector1()
getForm1()
getFname()
getLname()
getSecret1()
getSecret2()
getState()
```

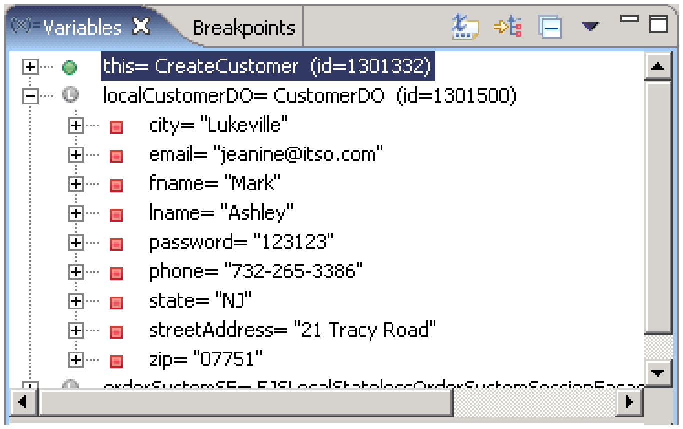
```
WebSphere Application Server v6.0 [WebSphere v6.0 Server] WebSphere Application Server v6.0 (WebSphere v6.0)
[5/6/05 10:06:14:046 EDT] 0000002e WorkSpaceMana A   WKSP0023I: Workspace configuration
[5/6/05 10:06:17:060 EDT] 0000000a WsServerImpl  A   WSVR0001I: Server server1 open for
[5/6/05 10:06:22:188 EDT] 0000002a WorkSpaceMana A   WKSP0023I: Workspace configuration
[5/6/05 10:06:22:768 EDT] 0000003a ServletWrappe A   SRVE0242I: [transfer]: Initializat
[5/6/05 10:06:22:979 EDT] 0000003a WorkSpaceMana A   WKSP0023I: Workspace configuration
```

Figure 9-31   Debug perspective overview

Displaying variable values

The basis of our debugging example is that we wanted to examine the values collected from the UI components, just before they are passed in to the session facade. Now that we have the debug perspective opened and the execution has stopped on the right breakpoint, this is a simple matter. As we see in the code, the variable being passed to the session facade is localCustomerDO. We can view the variable data as follows:

1. In the Variables view, locate the node for the **localCustomerDO** variable.
2. Expand the node so the member data is displayed as seen in Figure 9-32 on page 287.



```
this= CreateCustomer  (id=1301332)
localCustomerDO= CustomerDO  (id=1301500)
    city= "Lukeville"
    email= "jeanine@itso.com"
    fname= "Mark"
    lname= "Ashley"
    password= "123123"
    phone= "732-265-3386"
    state= "NJ"
    streetAddress= "21 Tracy Road"
    zip= "07751"
```

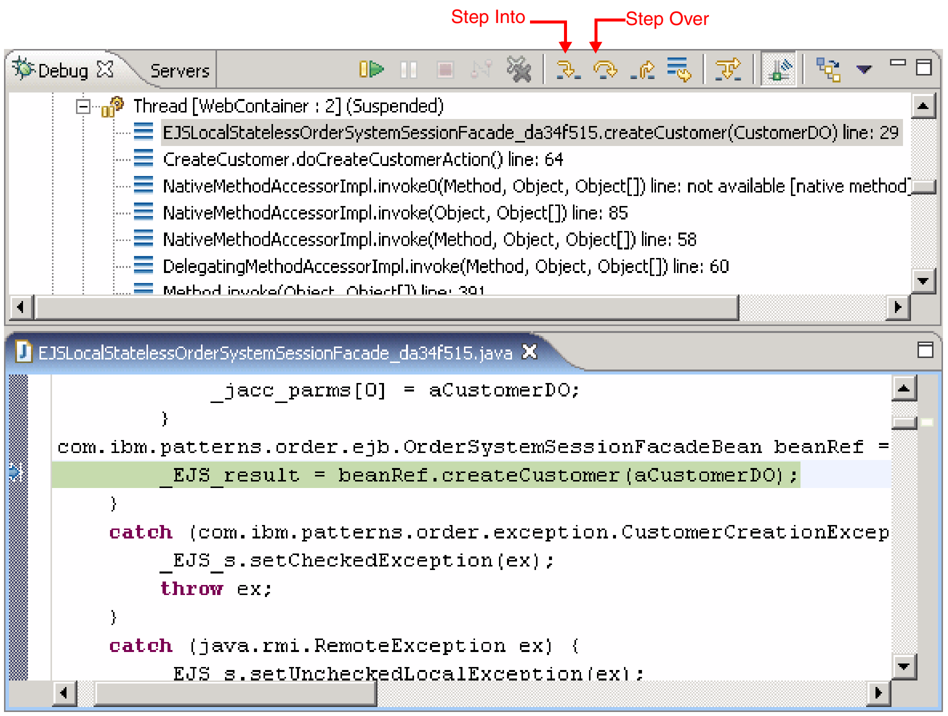
Figure 9-32 Display variable data

Now we can view the data that is about to be passed to the session facade. The data seems consistent with what was entered on the screen. This indicates there were no problems retrieving the screen values.

Stepping through the execution

Continuing with our debugging example now that we have determined that the screen values have been captured correctly, we want to step through the code to determine where the values get corrupted. This can be accomplished using the Debug view:

1. The small icons an the top of the Debug view allow you to step through the execution of the thread. Click the **Step Into** arrow icon, which will step the debug view into the session facade code. The Java editor in the center of the screen automatically opens the session facade class.
2. Click the **Step Over** icon to step to the next line of code. The Java editor continuously highlights the current line being executed. Continue clicking **Step Over** until you have reached the line where the we are about the execute the createCustomer method on the session bean. Figure 9-33 on page 288 shows the Debug view and the Java editor at this point in time.



```
Thread [WebContainer : 2] (Suspended)
    EJSLocalStatelessOrderSystemSessionFacade_da34f515.createCustomer(CustomerDO) line: 29
    CreateCustomer.doCreateCustomerAction() line: 64
    NativeMethodAccessorImpl.invoke0(Method, Object, Object[]) line: not available [native method]
    NativeMethodAccessorImpl.invoke(Object, Object[]) line: 85
    NativeMethodAccessorImpl.invoke(Method, Object, Object[]) line: 58
    DelegatingMethodAccessorImpl.invoke(Method, Object, Object[]) line: 60
    Method.invoke(Object, Object[]) line: 391
```

EJSLocalStatelessOrderSystemSessionFacade_da34f515.java

```
            _jacc_parms[0] = aCustomerDO;
        }
com.ibm.patterns.order.ejb.OrderSystemSessionFacadeBean beanRef =
        _EJS_result = beanRef.createCustomer(aCustomerDO);
    }
    catch (com.ibm.patterns.order.exception.CustomerCreationExcep
        _EJS_s.setCheckedException(ex);
        throw ex;
    }
    catch (java.rmi.RemoteException ex) {
        _EJS_s.setUncheckedLocalException(ex);
```

Figure 9-33 Stepping through code execution

Once again, at this point we would look at the Variables view (not shown in Figure 9-33) and inspect the value of the customer data. In the code above, we would look at the aCustomerDO variable. If the values were correct, we would continue and step into the createCustomer method. We would continue this process until we isolated the place where the values were corrupted.

9.5 Runtime guidelines

The JavaServer Faces framework requires no special runtime configuration in WebSphere Application Server. All the required JSP files, managed bean classes, utility classes, and configuration files are deployed into the server when we publish the Web project.

9.5.1 The web-config.xml configuration file

Just about all significant configuration information needed by the application server to support a JSF-based application is stored in the web-config.xml file.

Through out this chapter we have seen multiple examples of navigation rules, managed bean declarations, validator declarations, and so forth that are stored in the file. For the most part, Rational Software Architect takes care of creating and updating web-config.xml based on information we entered using the various wizards. However we encourage you to look at the examples we provide and understand the general structure of the file. It is not unusual for a designer or developer to make small manual changes to the file for things that are not supported by wizards. The custom e-mail validator in 9.4.6, “Implementing input validation” on page 280 is one such example.

9.6 System management

As we have explained, the JavaServer Faces framework does not require special resources to be allocated and maintained on the server. Moreover, once the JSP files, managed beans, and configuration information is processed by the application server, it breaks down into well known J2EE components such as servlets, java classes, HTTP session data, and so forth. The display code which eventually gets forwarded to the user browser, is a standard combination of HTML and JavaScript. Therefore, there are no significant system management considerations particular to the JSF technology.

9.7 For more information

For more information about the topics covered in this chapter see:

- *WebSphere Studio 5.1.2, JavaServer Faces and Service Data Objects,* SG24-6361
- *Rational Application Developer V6 Programming Guide, SG24-6449*
- *WebSphere Application Server V6 System Management and Configuration Handbook, SG24-6451*
- IBM developerWorks

  `http://www.ibm.com/developerWorks`
- Sun JavaServer Faces Technology Page

  `http://java.sun.com/j2ee/javaserverfaces/index.jsp`

10

Web services scenario

This chapter gives an overview of the use of Web services in an SOA environment and illustrates the three application patterns using the ITSOMart sample application:

It discusses how the design guidelines outlined in Chapter 7, “Application and system design guidelines” on page 139 were applied to the sample scenario. It also describes the process used to develop the Web services and the Web service clients as a way of illustrating development guidelines for Web services applications. The sample has been deployed on WebSphere Application Server V6 and we use this to illustrate runtime configuration tasks that must be done. Lastly, we discuss system management considerations for using Web services.

10.1 Architectural overview model

The architectural overview model, shown in Figure 10-1, illustrates how the ITSOMart application was extended to communicate with enterprise systems using Web services.

The highlighted portion of Figure 10-1 is the focus of this chapter. From the J2EE architecture point of view, this is the part of the application server business logic tier that retrieves some business data from one or more applications in the EIS tier. Results are retrieved and processed and sent back to the presentation tier.

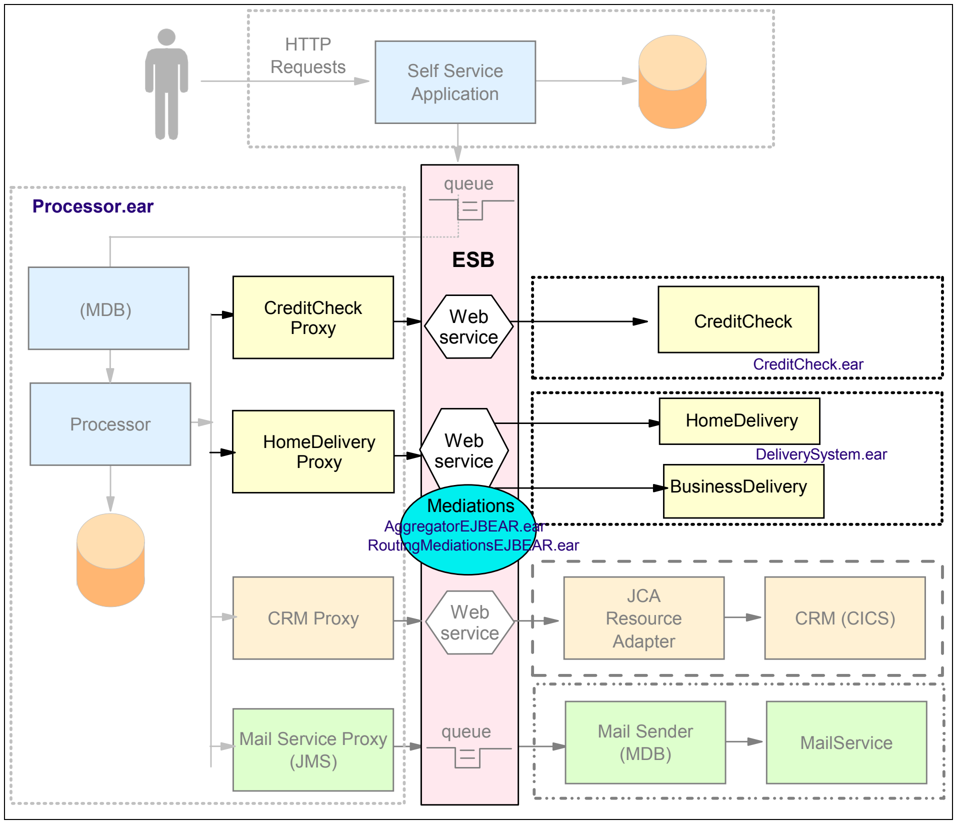


Figure 10-1 Architectural overview model: Web services

The components that participate in the Web Services Scenario are:

- CreditCheckProxy

  This proxy is used by the Processor application to call the CreditCheck Web service.

- HomeDeliveryProxy

  The Processor application uses this proxy to call the HomeDelivery Web service. The ESB intercepts the request and routes it to the appropriate services.

- Enterprise service bus (ESB)

  The ESB is implemented using the service integration bus. The bus acts as an intermediary for the Web service. Using an ESB insulates the application from changes in the Web service location and provides the opportunity to implement mediations or security measures in the future.

- CreditCheck

  The CreditCheck Web service returns a credit rating for the customer.

- DeliverySystem

  The DeliverySystem application has two Web services: HomeDelivery and BusinessDelivery. These delivery systems create account numbers for the new customer and return the information to the user.

This scenario illustrates the following three application patterns using Web services technology:

- Directly Integrated Single Channel application pattern:

  ITSOMart calls a Web service that returns a customer credit rating. The rating is used to determine whether to allow the customer to open an account.

- Router application pattern

  ITSOMart allows the user to select whether they want home delivery, business delivery, or both. If home or business delivery is chosen the request is routed to the appropriate service to assign an account number that is then returned to the user. The routing is performed through the use of a mediation in the ESB.

- Decomposition application pattern

  If the user selects both home delivery and business delivery, the request for an account number is sent to both services. The responses with the account numbers are aggregated into a single response for the customer. The decomposition of the message into multiple messages and the recomposition into a single response are performed through the use of a mediation in the ESB.

10.2 System design overview

As system designers, the first question we should ask ourselves is What is the business problem we are trying to solve? In the case of our sample ITSOMart application, we know that we want to obtain credit information for customers who register to use the application and we want to create an account number for the new customer. We know these services can be obtained through an existing enterprise that could reside potentially anywhere, on any platform, database, or application. The challenges are related to how we can provide seamless integration between these disparate systems and how we make this transparent to the user. Web services is one choice that we have as system designers for solving seamless integration problem.

For our example, we assume the following conditions:

- ITSOMart does not want to develop a CreditCheck application if there is one already available that they can use for their purpose of obtaining sensitive credit information about possible customers. They have been able to locate such an application that offers the service by means of a Web service.
- ITSOMart has existing applications maintained at each division headquarters that can assign customer numbers for new customers. These applications are J2EE applications hosted on a WebSphere Application Server and can be easily made available as Web services.
- The CreditCheck and DeliverySystem applications are the Web service providers. The ITSOMart application is the Web service requester. ITSOMart will initially use a static, rather than dynamic, publish and discovery approach, therefore no UDDI registry is needed. Using the interface and implementation information contained in the WSDL file provided by each service provider, ITSOMart will be able to communicate with the services.
- The DeliverySystem application has two Web services: one for home delivery and one for business delivery. When a request to create a new customer is entered, a Web service request is sent to the HomeDelivery system. A mediation attached to the HomeDelivery destination in the bus will determine (based on delivery type) whether the request will go to HomeDelivery, BusinessDelivery or both. In the event a request is sent to both services, the response is diverted to a destination where another mediation combines (or aggregates) the responses into one response and sends the new response back to the client.
- The CreditCheck and HomeDelivery applications can be located:
  - On the same server as the ITSOMart application
  - On another server on the same LAN as the ITSOMart application
  - On another server on the same intranet, or private network, as the ITSOMart application

The enterprise applications could also be located on another server accessible on the Internet. However, our focus is on intra-enterprise applications with the Self-Service business pattern. The Extended Enterprise (business-to-business) business pattern can be used to connect intra-enterprise applications.

10.2.1 Component model

A component model shows a breakdown of both the client (Web service requester) application server and the enterprise (Web service provider) application server. The component model diagram in Figure 10-2 illustrates the design for the CreditCheck service.

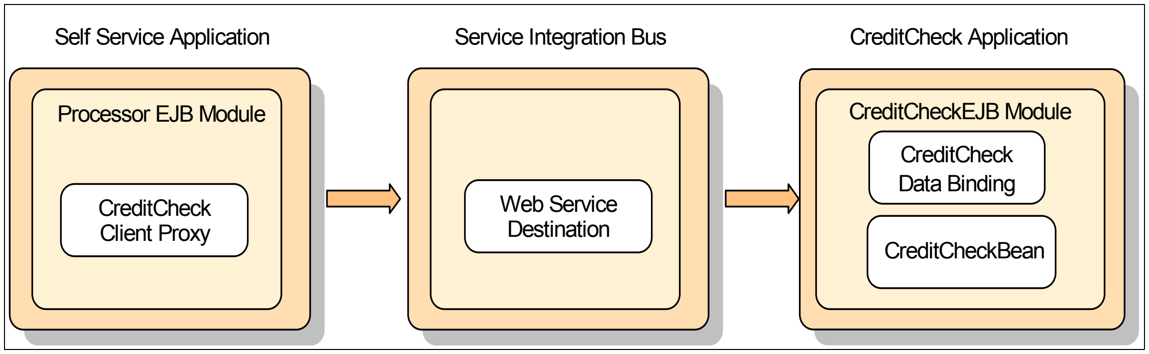


Figure 10-2 CreditCheck component model diagram

A session bean provides the connection to a Web services proxy. The proxy has the information required to send a message to the service provider using the SOAP and HTTP protocols. In this case, the proxy sends the request to an inbound service on the service integration bus. The bus makes the connection between the inbound service and the outbound service representing the CreditCheck service.

The component model diagram in Figure 10-3 on page 296 illustrates the design for the DeliverySystem service.

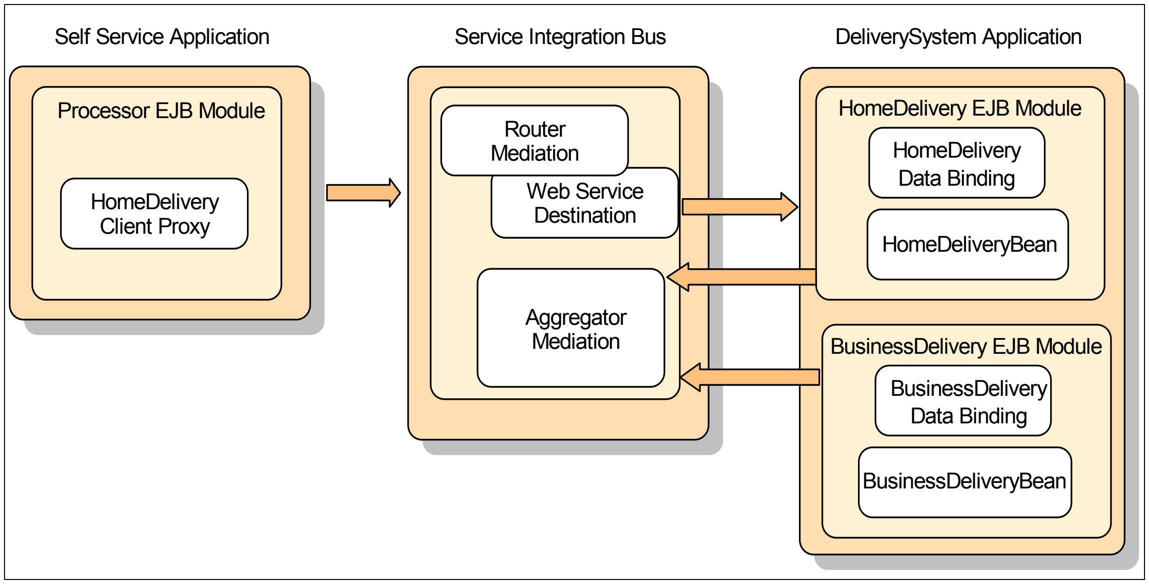


Figure 10-3 Component model diagram

A Web service call to HomeDelivery invokes the HomeDelivery inbound service on the bus. The destination representation for the HomeDelivery service has a mediation that looks at the request and makes a routing decision based on the delivery service requested by the client. The request could be routed to the HomeDelivery service, the BusinessDelivery service or both. If both services are called, a second mediation intercepts the responses and combines them into one response.

10.2.2 Object model

In this section we provide an object model for our Web services scenario.

Class diagram for CreditCheckClient

Figure 10-4 on page 297 shows a class diagram of the classes directly involved with providing Web services for the CreditCheckClient application.

On the requester side, the ProcessorWebService bean from the ITSOMart application requests the credit rating using the CCWSClientBean EJB. CCWSClientBean is the Web services requester bean, serving as the interface between the ITSOMart application and the CreditCheck Web service. As far as the application knows, it is simply calling a local method called returnSimpleQuote on this bean and expects a credit rating to be returned.

On the provider side, a stateless session EJB called CreditCheck serves as the Web services bean for the provider. In a real application, this bean would probably act as a facade to the actual enterprise business objects. It knows which enterprise objects to call in order to retrieve the information it needs to create a response to the Web services request.

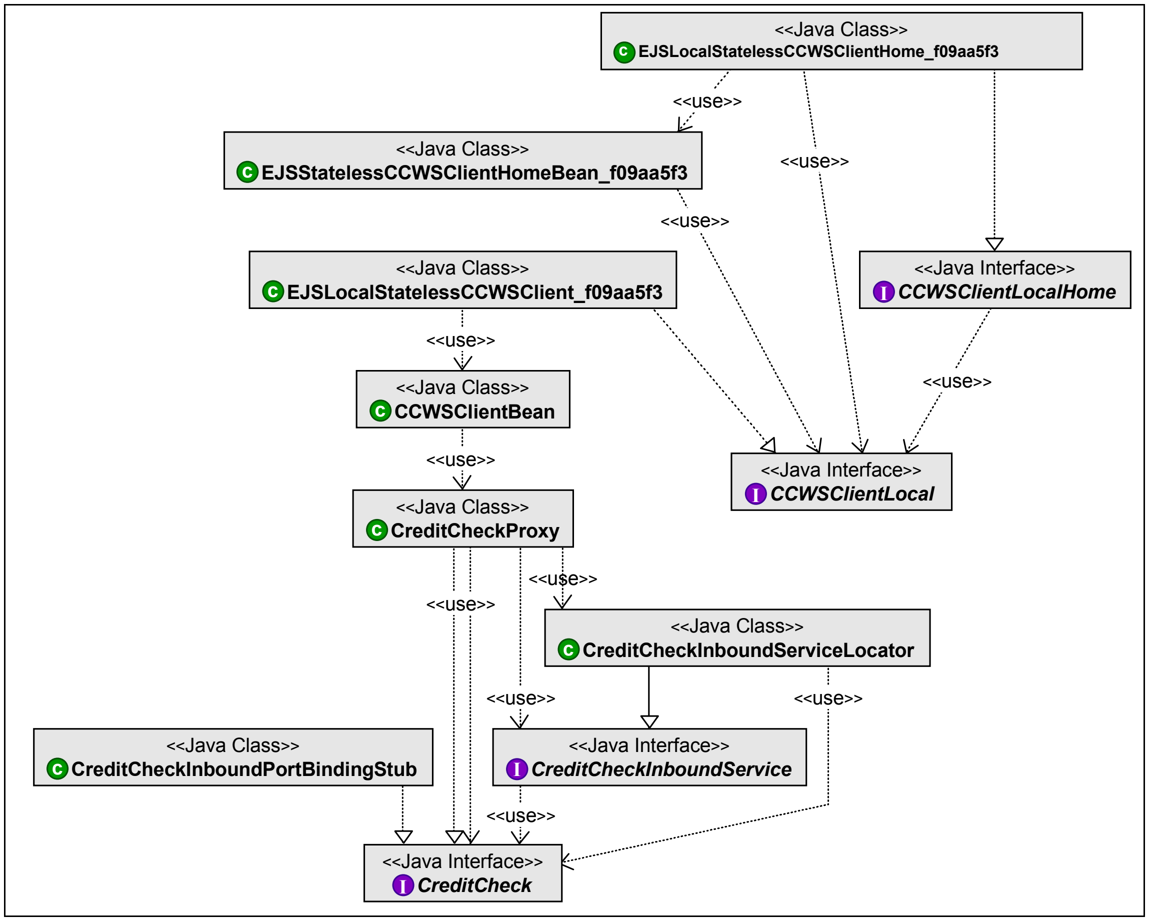


Figure 10-4 Class diagram of the CreditCheckClient application

Class diagram for ClientDelivery

Figure 10-4 on page 297 shows a class diagram of the classes directly involved with providing Web services for the ClientDelivery application. The flow is identical to that of the CreditCheckClient application.

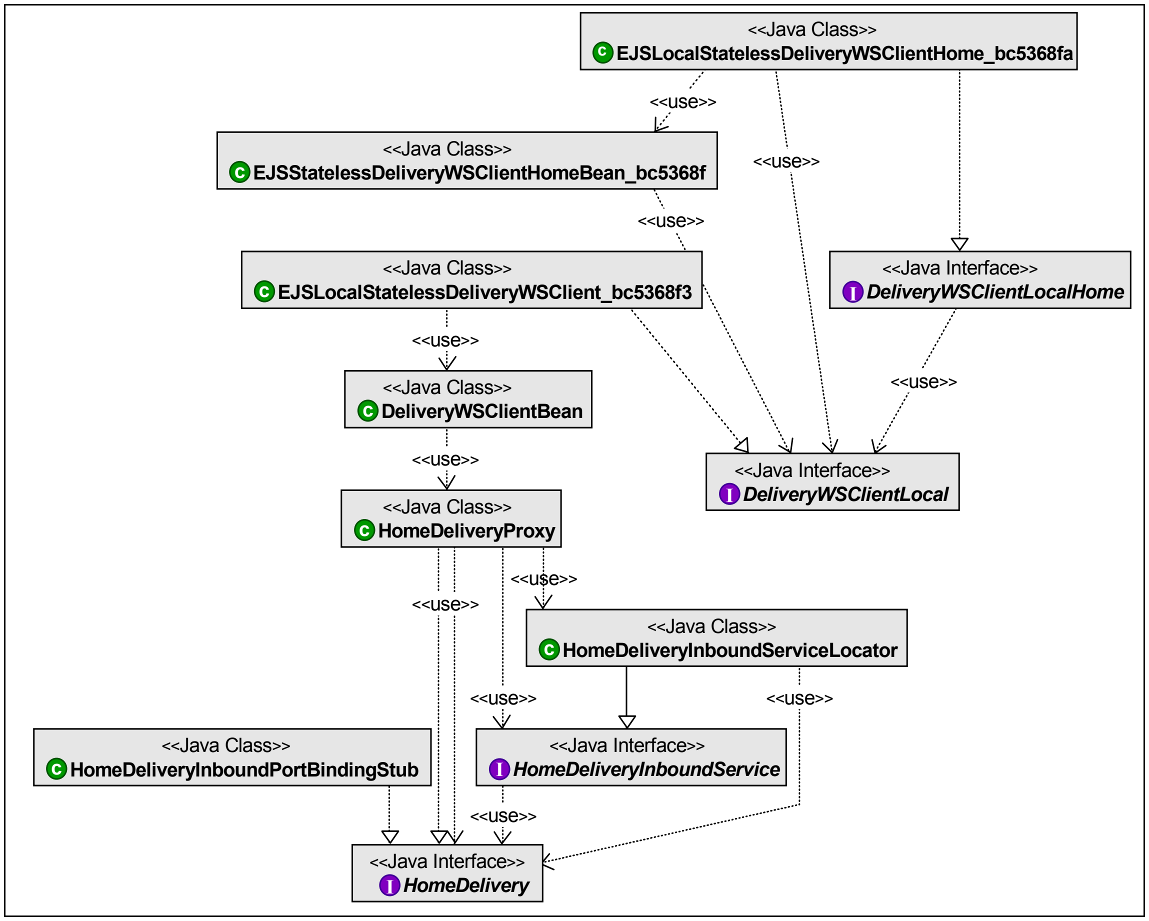


Figure 10-5 Class diagram of the Web service requester and provider

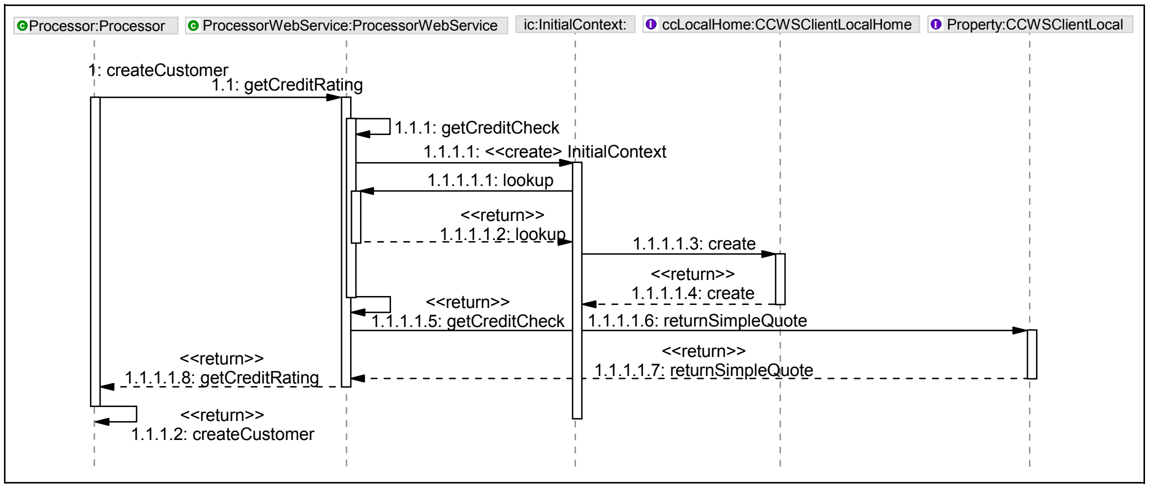
Sequence diagram

Two Web service calls are performed in this part of the scenario. Both calls are originated from the createCustomer() method in Processor.java. The first call is issued to get a credit rating for the customer from the CreditCheck Web service. The second call is to get an account number. The fact that the account number is created by one of two account services depending on the type of delivery account requested is transparent to the user and to the Processor application. Routing to the proper account service is done in the ESB.

For simplicity we will treat each of these Web service invocations as separate sequence diagrams. In reality, the Processor code performs several steps in sequence to create a customer, the first is to get the credit rating, and the second is to get the account number.

Get credit rating

The sequence diagram in Figure 10-6 shows the interaction of classes and the flow of control through the application as the credit rating for the customer is obtained from the CreditCheck service.



Processor:Processor
ProcessorWebService:ProcessorWebService
ic:InitialContext:
ccLocalHome:CCWSClientLocalHome
Property:CCWSClientLocal
1: createCustomer
1.1: getCreditRating
1.1.1: getCreditCheck
1.1.1.1: <<create> InitialContext
1.1.1.1.1: lookup
<<return>> 1.1.1.1.2: lookup
1.1.1.1.3: create
<<return>> 1.1.1.1.4: create
<<return>> 1.1.1.1.5: getCreditCheck
1.1.1.1.6: returnSimpleQuote
<<return>> 1.1.1.1.7: returnSimpleQuote
<<return>> 1.1.1.1.8: getCreditRating
<<return>> 1.1.1.2: createCustomer

Figure 10-6 Sequence diagram for the CreditCheck Web services requester and provider

1) The createCustomer() method in the Processor class is invoked.

1.1) createCustomer() invokes getCreditRating() method in the ProcessorWebService class, passing the customer information needed to obtain a credit rating.

1.1.1) getCreditRating() invokes the getCreditCheck() method to get the creditCheck information of a particular customer.

1.1.1.1) getCreditCheck() does a JNDI lookup for the CCWSClient bean, which acts as an interface to the Web service.

1.1.1.1.5) getCreditCheck() invokes the returnSimpleQuote() method of CCWSClient bean, which in turn, invokes the returnSimpleQuote() method of the proxy for the CreditCheck application. Control is then passed to the CreditCheck application on the Web service provider. The CreditCheck application returns the credit check reply message back through the chain to CCWSClient bean.

1.1.1.1.7) The credit check rating gets passed back through the chain to the requester side.

Get account

The sequence diagram in Figure 10-7 shows the interaction of classes and the flow of control through the application as the account number for the customer is obtained from one or both of the delivery services.

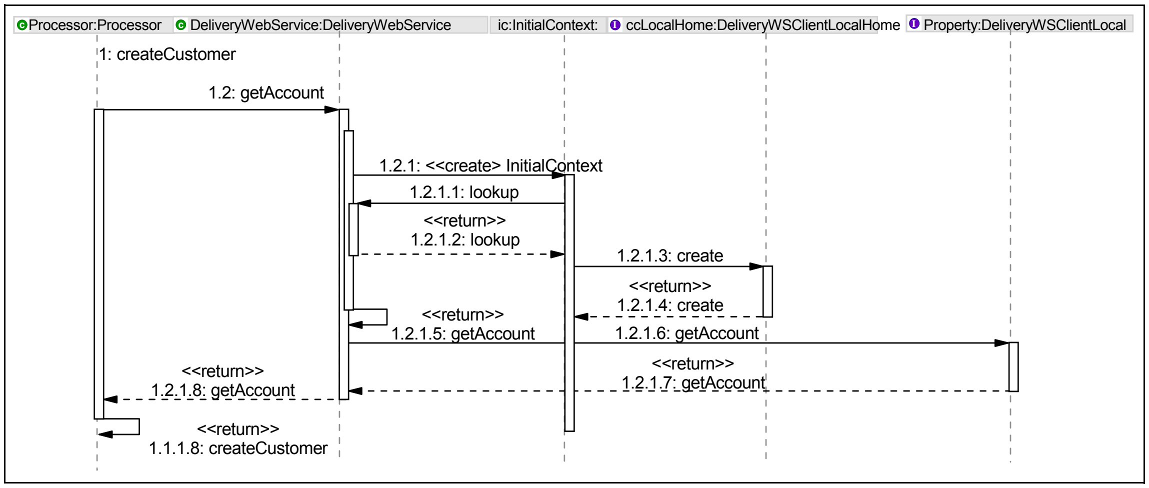


Figure 10-7 Sequence diagram for HomeDelivery Web services requester and provider

1)	The createCustomer() method in the Processor class is invoked.
1.2)	createCustomer() invokes the getAccount() method in the DeliveryWebService class, passing the customer information needed to obtain an account number.
1.2.1)	getAccount() does a JNDI lookup for the DeliveryWSClient bean, which acts as an interface to the Web service.
1.2.1.6)	getAccount() invokes the getAccount() method of DeliveryWSClient bean, which in turn, invokes the getAccount() method of the proxy for the HomeDelivery application. The ESB intercepts the Web service request and routes it to the HomeDelivery Web service provider, the BusinessDelivery Web service provider, or both depending on the type of delivery the customer has requested. The Web service application returns the account number reply message back through the chain to DeliveryWSClient bean.
1.2.1.8	The account number then gets passed back through the chain to the requester side.

10.3 Applying the design guidelines

In this section we describe the architecture or design decisions that went into building our sample Web services application. This sample application is an extension of the ITSOMart application. We use Web services to communicate between the ITSOMart and the CreditCheck and Delivery Enterprise system.

There are several architectural decisions that go into designing Web service providers and requesters. These include:

- Transmission pattern
- SOAP messaging mechanism
- Static or dynamic Web services discovery
- Synchronous versus asynchronous Web services
- Message structure
- Mediation

For a complete discussion of application design considerations when developing Web services applications, see 7.3, “Design guidelines for Web services” on page 163.

Transmission pattern

Our first task was to define a request-response transmission pattern. In this pattern, the Web service receives a single request, sends a single response, and then closes the session. This is an appropriate pattern for our case, where the ITSOMart application simply requests a credit rating from an enterprise system and receives a response with Web services.

SOAP messaging mechanism and synchronous pattern

The SOAP messaging mechanism we selected in our application is the RPC mechanism. Our choice was between using the Remote Procedure Call (RPC) mechanism or the message-oriented communication mechanism. We decided on RPC because this is the simplest and most straightforward method. It is also the most commonly used today. Also, we made the assumption that this would be a synchronous message transfer and we did not have a need for a delivery confirmation. The message-oriented communication mechanism is most appropriate for asynchronous or confirmed delivery types of scenarios. Since our application had no need for these features, the RPC messaging mechanism was the most appropriate.

Message structure

This scenario uses two simple text messages (request and a response) that are exchanged between the Web service requester and the Web service provider. These text messages contain customer details (the request) and the resulting credit rating (the response). This could be extended to use a more structured message approach in the form of XML messages.

Mediation

In this scenario, there are multiple delivery systems that can be called depending on the delivery type selected by the client. Mediation can be used to route the requests to one or more of the systems and if necessary, manage multiple responses.

10.4 Development guidelines for Web services

The development process for building a Web service is very similar to the development process of any other software. There are four main phases in developing a Web service: build, deploy, run, and manage.

1. The *build* phase includes development and testing of the Web service application, including the definition and functionality of the service.
2. The *deploy* phase includes publication of the service definition, the WSDL document, and deployment of the runtime code of the Web service.
3. The *run* phase includes finding and invoking the Web service.
4. The *manage* phase includes the management and administration of the Web service. This includes performance measurement and maintenance of the Web service.

Figure 10-8 depicts the complete development process. Using different problem domains, the terms used within this picture would change; however, the general view would not.

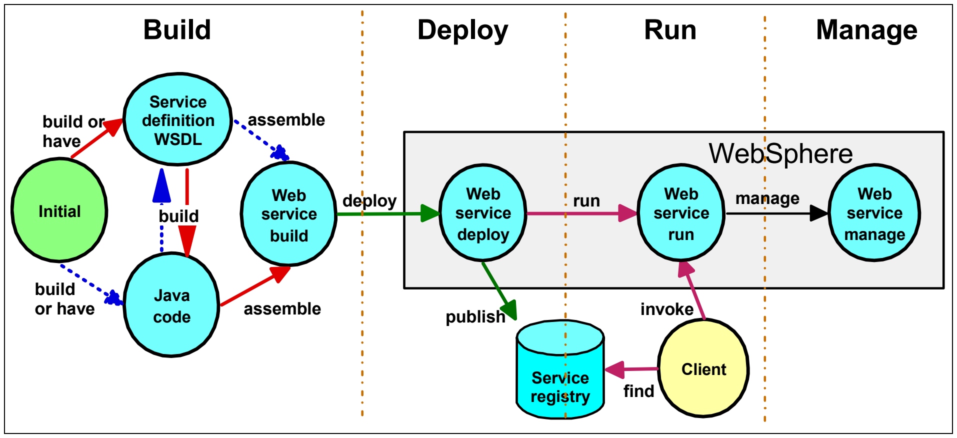


Figure 10-8 Web services development

The remainder of this section describes the four development phases in more detail.

Build phase

The build phase, which includes testing and debugging, is the first phase in the development process of a new Web service. Because Web services can be written from scratch and use already existing applications, there are two possible paths to be followed:

- The solid path

  From the initial state, we build or already have Java code. Using this Java code, we build the service definition (WSDL document) with the business methods that we want to expose. After we have generated the WSDL document, we assemble the Web service application. This approach is called bottom-up development. This is the approach used for developing our CreditCheck application. We first implement the CreditCheck application as a single application, and then expose these methods for use with a Web service.

- The dashed path

  From the initial state, we build or already have a service definition, a WSDL document. Using this WSDL document, we build or adapt the Java code to implement that service. After we have implemented the code, we assemble the Web service application. This approach is called top-down development.

Deploy phase

The second phase of a Web service development process is deployment. In this phase, we deploy the Web service to an application server. Deploying a Web service makes it accessible by clients. However, these clients have to be aware of the newly installed Web service, and thus, the next step in this phase is to publish the Web service. The publication can be done through a private or public UDDI registry, using a WSIL document, or by directly providing the information about the new service to consumers, for example, through e-mail. A combination of all these publishing methods is also possible. After the service has been published, it can be called by clients. In our example we do not publish the WSDL to a UDDI registry, rather we directly provide the information about the service directly.

Run phase

The third phase is the runtime. In this phase, the Web service is operative and is invoked by clients that require the functionality offered by the service. Our client is the ITSOMart application. Internally, ITSOMart has an EJB project containing a client that calls the CreditCheck Web service.

Manage phase

The final phase is the management phase where we cover all the management and administration tasks of the Web service.

The manage phase can include measurement tools that are used for monitoring the Web service and to accumulate performance data. In most real-life applications, clients would require a certain quality of service. Also, tools for authorization and statistics (billing) could be required.

10.5 Application development using Web services

The following sections illustrate the process of using Rational Software Architect, or Rational Application Developer if you prefer, to develop a Web service.

10.5.1 Implementation approach

The implementation tasks will be performed in two phases.

In the first phase, the CreditCheck and HomeDelivery Web services will be called directly without using an ESB. The clients are created using WSDL provided by the service provider. In this phase, a client is created for the HomeDelivery Web service. The BusinessDelivery Web service is not invoked.

In the second phase, the Web services are defined to the bus and new WSDL is generated by the bus. The clients are regenerated to use the new WSDL, thus routing the Web service calls through the bus.

Using two phases is done for development and testing purposes only. It is easier to ensure the application is working as designed before introducing the extra layer of the ESB.

The ITSOMart scenario uses three Web services: CreditCheck in CreditCheck.ear, HomeDelivery in DeliverySystem.ear, and BusinessDelivery in DeliverySystem.ear. Each Web service was built in a similar manner. This section illustrates how the CreditCheck service was built.

The process takes CreditCheck, an existing application, and creates a Web service from it. We guide you through the process of creating the WSDL document, the proxy classes, and testing the generated server code using the Web Services Explorer. We then go through the process of creating a Web service client that can be incorporated into a requester application to access the new Web service. We test again to ensure that the client and provider are working correctly. Last, we define the services to the service integration bus and regenerate the clients to call the service using the service definition on the bus.

Note: The following process ensures that the Web services developer capability is enabled in the workbench:

1. Select **Window** → **Preferences.**
2. Expand Workbench and select **Capabilities**.
3. Check the **Web Service Developer** box and click **OK**.

For information about downloading the sample application, see Appendix B, “Additional material” on page 485.

10.5.2 Creating a Web service from a session bean

In this section, we create a Web service using the bottom-up development method using CreditCheckEJB as input.

Preparation: This sample assumes that the CreditCheck application has been imported into the Rational Software Architect workspace. This is the existing application that returns a credit rating and contains an EJB called CreditCheckEJB.

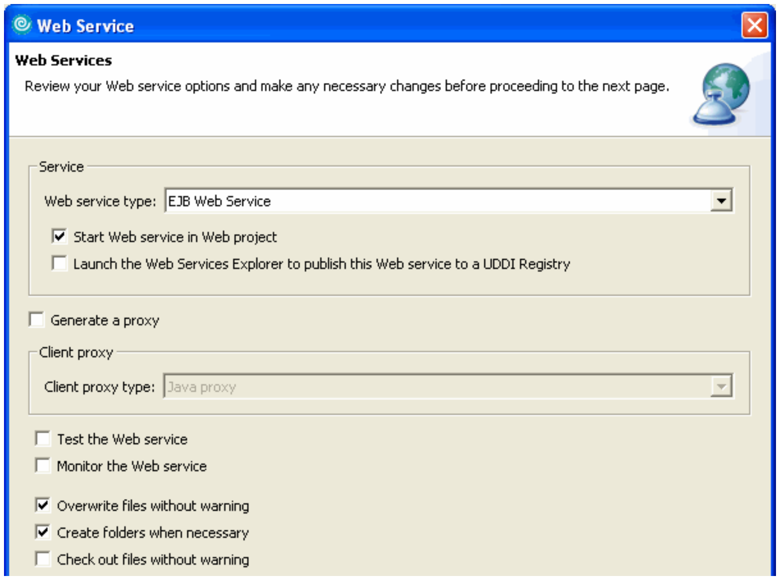
We use SOAP over HTTP as the transport mechanism. A Web router project is required with a servlet to route the client requests to the session EJB. Before starting this procedure, create a dynamic Web project called CreditCheckRouter and put it in CreditCheck. You will need to have this in place before starting the Web Service wizard.

To initiate the Web Service wizard on our EJB:

1. Switch to the J2EE Perspective.
2. In the Project Explorer view, navigate to **CreditCheckEJB/ejbModule/com.ibm.patterns.creditCheck/ CreditCheckBean.java.** Right-click to bring up the context menu.
3. Select **Web Services** → **Create Web Service.**

   Note: If you cannot find *Web Services* on the context menu, verify that the Web service capability is enabled.

   See Figure 10-9.


Web Service
Web Services
Review your Web service options and make any necessary changes before proceeding to the next page.

Service
Web service type: EJB Web Service
[✓] Start Web service in Web project
[ ] Launch the Web Services Explorer to publish this Web service to a UDDI Registry
[ ] Generate a proxy
Client proxy
Client proxy type: Java proxy
[ ] Test the Web service
[ ] Monitor the Web service
[✓] Overwrite files without warning
[✓] Create folders when necessary
[ ] Check out files without warning

Figure 10-9 Web Service wizard: Web services

On the Web Services page, select the following options (Figure 10-9):

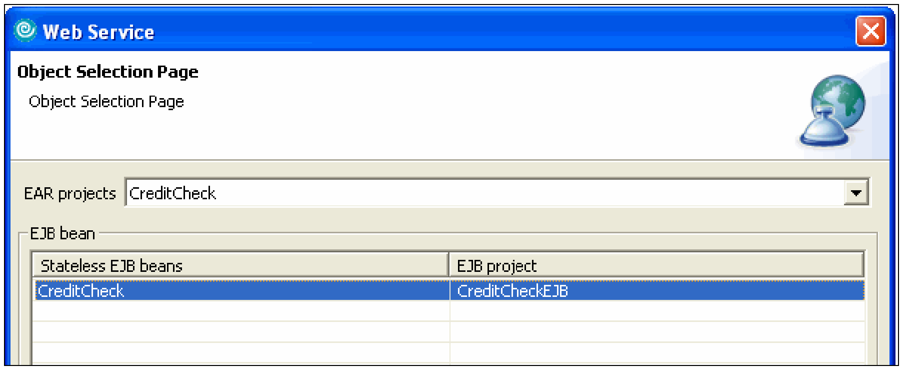
– Select the Web service type **EJB Web Service.**

– Select **Start Web service in Web project**.

– In our scenario, we cleared the **Launch Web Services Explorer to publish this Web service to a UDDI Registry**, because we are not using a UDDI in our test environment. Note that you can use the Web Services Explorer (included in Rational Software Architect) later to publish the Web service.

- Clear **Generate a proxy.** The wizard generates client proxy classes enabling simple method calls in a client program to call the Web service. These classes are generated based on the WSDL created for the Web service. Because we plan to use the bus as a destination for the Web service, new WSDL for the Web service will be created when we configure the bus. For now, we do not want to create a proxy for the client.
- Clear **Test the Web service**. This option lets you test the Web service using the Web Services Explorer before the proxy is generated. It also enables you to create a test client (a set of JSPs) in a client project.
- Clear **Monitor the Web service**. This option lets you monitor your Web service using the TCP/IP Monitor by routing the traffic through the monitor and configuring the monitor for the server on which the service is deployed.

4. Click **Next** to proceed to next page. On the Object Selection page (Figure 10-10), you can specify from which Java bean the Web service is generated.



EAR projects: CreditCheck

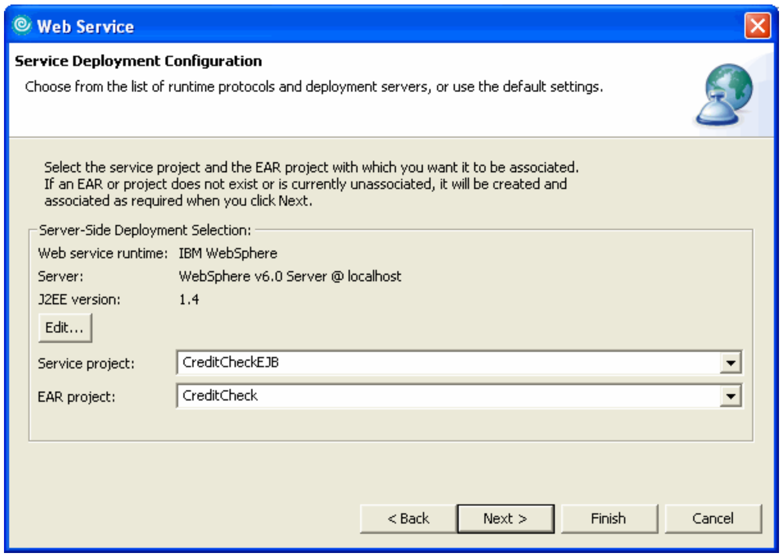
Stateless EJB beans	EJB project
CreditCheck	CreditCheckEJB

Figure 10-10 Web Service wizard: object selection

- Ensure that the EAR project selected is **CreditCheck.**
- Select the **CreditCheck** for the EJB beans and **CreditCheckEJB** Project from list.

Click **Next.**

5. On the Service Deployment Configuration page (Figure 10-11 on page 308), specify the deployment settings for the service and the generated test client, if you chose to create one.


Service Deployment Configuration

Choose from the list of runtime protocols and deployment servers, or use the default settings.

Select the service project and the EAR project with which you want it to be associated. If an EAR or project does not exist or is currently unassociated, it will be created and associated as required when you click Next.

Server-Side Deployment Selection:

Web service runtime: IBM WebSphere

Server: WebSphere v6.0 Server @ localhost

J2EE version: 1.4

Service project: CreditCheckEJB

EAR project: CreditCheck

Figure 10-11 Web Service wizard: service deployment configuration

To deploy the CreditCheck service for testing in the Rational Software Architect integrated test environment WebSphere Application Server V6, ensure that the following are selected:

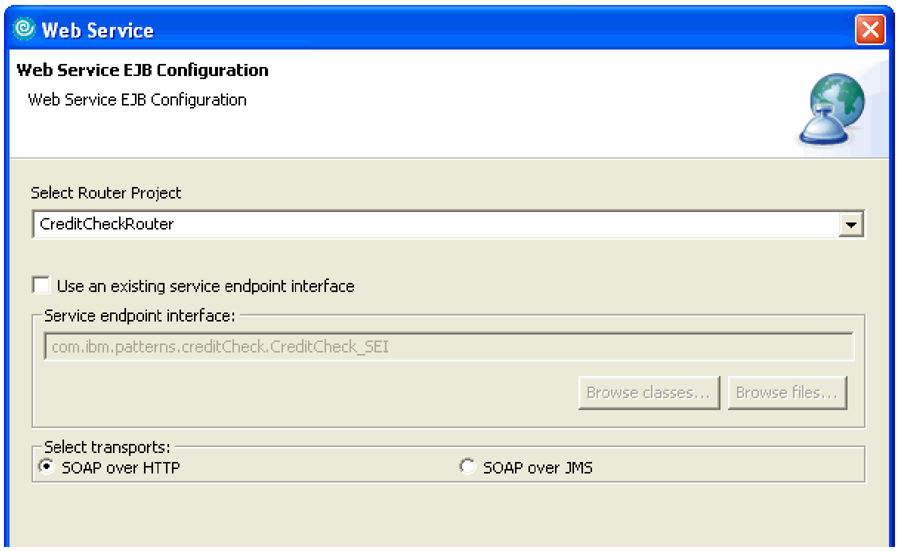
– Web service runtime: **IBM WebSphere**
– Server: **WebSphere v6.0 Server @ localhost**
– J2EE Version: **1.4**
– Service Project: **CreditCheckEJB**
– EAR Project: **CreditCheck**

Important: Always verify the generated project names. Rational Software Architect inserts default names that might not be your choice.

If you generate the client code into the wrong project (for example, a server project), it is very hard to back out of the changes unless you work in a team environment with a repository.

Click **Next.**

6. The Service Endpoint Interface Selection page allows you to specify the router project, the service endpoint interface, and the transport protocol used for the Web service. See Figure 10-12 on page 309.



Select Router Project: CreditCheckRouter

[ ] Use an existing service endpoint interface

Service endpoint interface: com.ibm.patterns.creditCheck.CreditCheck_SEI

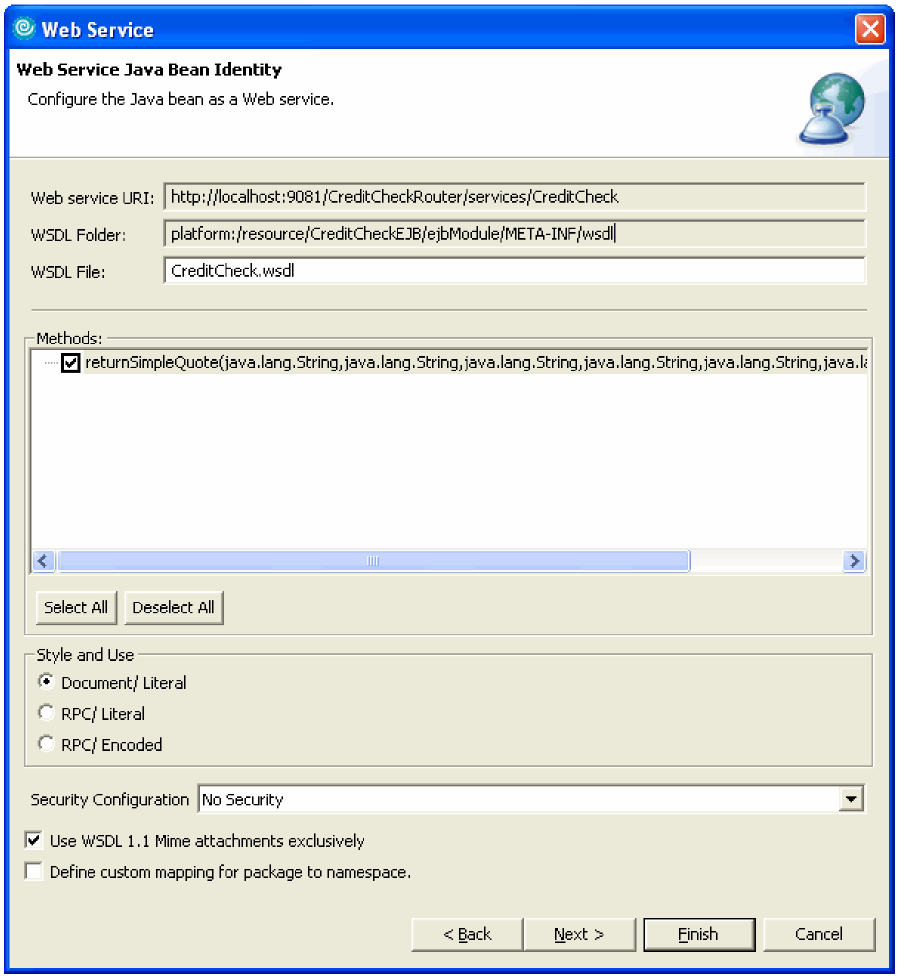
Select transports: [✓ SOAP over HTTP] SOAP over JMS

Figure 10-12 Web Service wizard: service endpoint interface selection

- Select router project: **CreditCheckRouter.**
- Clear the **Use an existing endpoint interface** check box.
- Select **SOAP over HTTP** as the transport protocol used for this Web service.

Click **Next**.

7. The next page contains options for creating the Web service. For our sample, we took the defaults as shown in Figure 10-13 on page 310.


Web Service
Web Service Java Bean Identity
Configure the Java bean as a Web service.

Web service URI: http://localhost:9081/CreditCheckRouter/services/CreditCheck
WSDL Folder: platform:/resource/CreditCheckEJB/ejbModule/META-INF/wsdl
WSDL File: CreditCheck.wsdl

Methods:
[✓] returnSimpleQuote(java.lang.String,java.lang.String,java.lang.String,java.lang.String,java.lang.String,java.la

Style and Use
[● Document/ Literal]
○ RPC/ Literal
○ RPC/ Encoded

Security Configuration: No Security

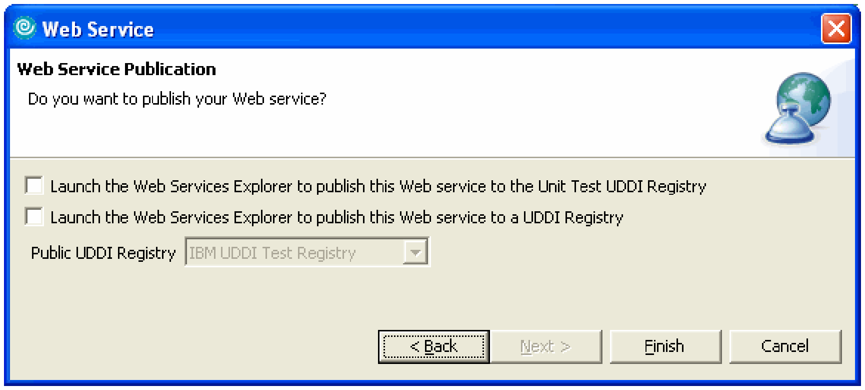
[✓] Use WSDL 1.1 Mime attachments exclusively
[ ] Define custom mapping for package to namespace.

Figure 10-13 Web Service wizard: Identity

- WSDL file: **CreditCheck.wsdl** (default).
- Methods: **returnSimpleQuote**
- Style and Use: **Document/Literal**
- Security Configuration: **No Security**

Click **Next**. The SEI, helper classes, the WSDL file, and the Web service deployment descriptor are generated into the service project.

8. Leave both options cleared on the Web Service Publication page because the CreditCheck Service will not be published to a UDDI registry (Figure 10-14 on page 311).


Web Service

Web Service Publication

Do you want to publish your Web service?

☐ Launch the Web Services Explorer to publish this Web service to the Unit Test UDDI Registry

☐ Launch the Web Services Explorer to publish this Web service to a UDDI Registry

Public UDDI Registry: IBM UDDI Test Registry

Figure 10-14 Web Service wizard: publication

Click **Finish** to complete the Web Service wizard.

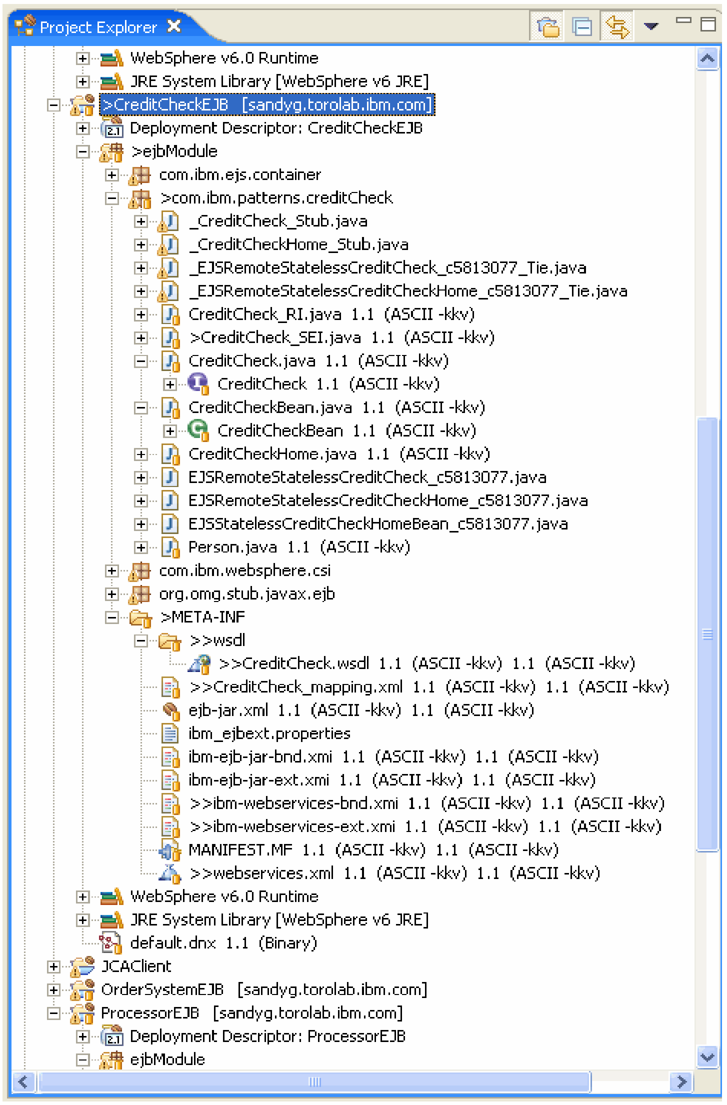
The Web Service wizard publishes the application to the server you specified and starts the server.

Generated files

According to the settings made during the run of the wizard, the following files in the creditCheckEJB/ejbModule project have been created:

- Service endpoint interface (SEI), `creditCheck_SEI.java`, is the interface defining the methods exposed in the Web service.
- The WSDL file, `/META-INF/wsdl/CreditCheck.wsdl`, describes the Web service.
- The deployment descriptor files, `webservices.xml`, `ibm-web services-ext.xml and ibm-Web services-bnd.xml` files describe the Web service according to the Web services for J2EE style (JSR 109). The JAX-RPC mapping is described in the `creditCheck_mapping.xml` file.

  See Figure 10-15 on page 312.



```
Project Explorer
WebSphere v6.0 Runtime
JRE System Library [WebSphere v6 JRE]
>CreditCheckEJB  [sandyg.torolab.ibm.com]
  Deployment Descriptor: CreditCheckEJB
  >ejbModule
    com.ibm.ejs.container
    >com.ibm.patterns.creditCheck
      _CreditCheck_Stub.java
      _CreditCheckHome_Stub.java
      _EJSRemoteStatelessCreditCheck_c5813077_Tie.java
      _EJSRemoteStatelessCreditCheckHome_c5813077_Tie.java
      CreditCheck_RI.java  1.1  (ASCII -kkv)
      >CreditCheck_SEI.java  1.1  (ASCII -kkv)
      CreditCheck.java  1.1  (ASCII -kkv)
        CreditCheck  1.1  (ASCII -kkv)
      CreditCheckBean.java  1.1  (ASCII -kkv)
        CreditCheckBean  1.1  (ASCII -kkv)
      CreditCheckHome.java  1.1  (ASCII -kkv)
      EJSRemoteStatelessCreditCheck_c5813077.java
      EJSRemoteStatelessCreditCheckHome_c5813077.java
      EJSStatelessCreditCheckHomeBean_c5813077.java
      Person.java  1.1  (ASCII -kkv)
    com.ibm.websphere.csi
    org.omg.stub.javax.ejb
    >META-INF
      >>wsdl
        >>CreditCheck.wsdl  1.1  (ASCII -kkv)  1.1  (ASCII -kkv)
      >>CreditCheck_mapping.xml  1.1  (ASCII -kkv)  1.1  (ASCII -kkv)
      ejb-jar.xml  1.1  (ASCII -kkv)  1.1  (ASCII -kkv)
      ibm_ejbext.properties
      ibm-ejb-jar-bnd.xmi  1.1  (ASCII -kkv)  1.1  (ASCII -kkv)
      ibm-ejb-jar-ext.xmi  1.1  (ASCII -kkv)  1.1  (ASCII -kkv)
      >>ibm-webservices-bnd.xmi  1.1  (ASCII -kkv)  1.1  (ASCII -kkv)
      >>ibm-webservices-ext.xmi  1.1  (ASCII -kkv)  1.1  (ASCII -kkv)
      MANIFEST.MF  1.1  (ASCII -kkv)  1.1  (ASCII -kkv)
      >>webservices.xml  1.1  (ASCII -kkv)  1.1  (ASCII -kkv)
  WebSphere v6.0 Runtime
  JRE System Library [WebSphere v6 JRE]
  default.dnx  1.1  (Binary)
JCAClient
OrderSystemEJB  [sandyg.torolab.ibm.com]
ProcessorEJB  [sandyg.torolab.ibm.com]
  Deployment Descriptor: ProcessorEJB
  ejbModule
```

Figure 10-15   Generated files for creditCheckEJB/ejbModule

HomeDelivery and BusinessDelivery Web services

The Web services for the HomeDelivery and BusinessDelivery EJBs are created in the same manner. Both EJBs are in the DeliverySystem EAR. Two router projects are required, HomeDeliveryRouter.war and BusinessDeliveryRouter.war.

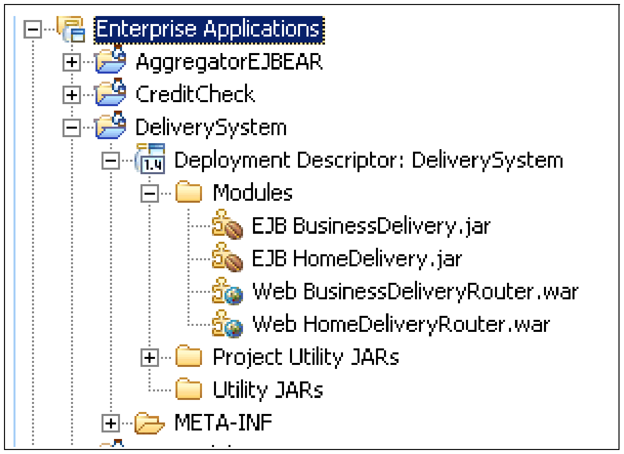

```
Enterprise Applications
  AggregatorEJBEAR
  CreditCheck
  DeliverySystem
    Deployment Descriptor: DeliverySystem
      Modules
        EJB BusinessDelivery.jar
        EJB HomeDelivery.jar
        Web BusinessDeliveryRouter.war
        Web HomeDeliveryRouter.war
      Project Utility JARs
      Utility JARs
    META-INF
```

Figure 10-16 DeliverySystem EAR file structure

The two EJBs each have the same structure. The Web services are created using HomeDeliveryBean and BusinessDeliveryBean. These beans were created in the com.ibm.patterns.Delivery package.

Figure 10-17 shows the structure of the BusinessDelivery EJB after the Web Service Wizard is run.

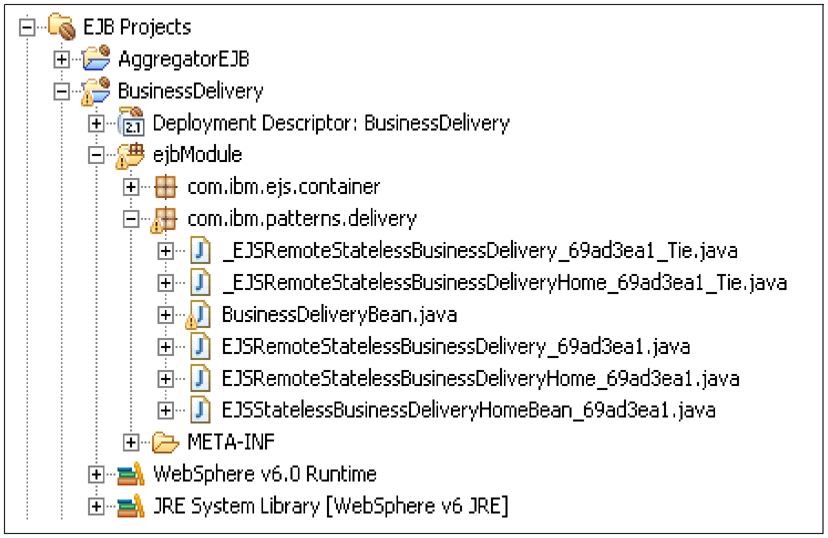
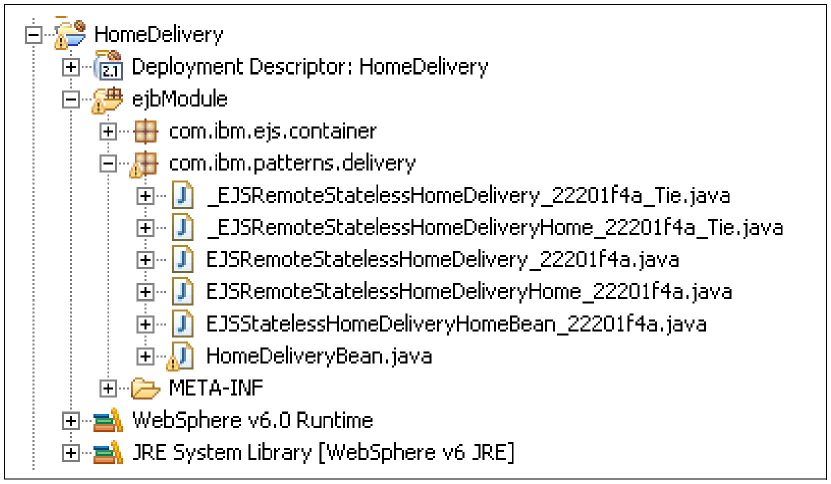

```
EJB Projects
  AggregatorEJB
  BusinessDelivery
    Deployment Descriptor: BusinessDelivery
    ejbModule
      com.ibm.ejs.container
      com.ibm.patterns.delivery
        _EJSRemoteStatelessBusinessDelivery_69ad3ea1_Tie.java
        _EJSRemoteStatelessBusinessDeliveryHome_69ad3ea1_Tie.java
        BusinessDeliveryBean.java
        EJSRemoteStatelessBusinessDelivery_69ad3ea1.java
        EJSRemoteStatelessBusinessDeliveryHome_69ad3ea1.java
        EJSStatelessBusinessDeliveryHomeBean_69ad3ea1.java
      META-INF
    WebSphere v6.0 Runtime
    JRE System Library [WebSphere v6 JRE]
```

Figure 10-17 BusinessDelivery EJB enabled as Web services

Figure 10-18 on page 314 shows the structure of the BusinessDelivery EJB after the Web Service Wizard is run.



```
HomeDelivery
  Deployment Descriptor: HomeDelivery
  ejbModule
    com.ibm.ejs.container
    com.ibm.patterns.delivery
      _EJSRemoteStatelessHomeDelivery_22201f4a_Tie.java
      _EJSRemoteStatelessHomeDeliveryHome_22201f4a_Tie.java
      EJSRemoteStatelessHomeDelivery_22201f4a.java
      EJSRemoteStatelessHomeDeliveryHome_22201f4a.java
      EJSStatelessHomeDeliveryHomeBean_22201f4a.java
      HomeDeliveryBean.java
    META-INF
  WebSphere v6.0 Runtime
  JRE System Library [WebSphere v6 JRE]
```

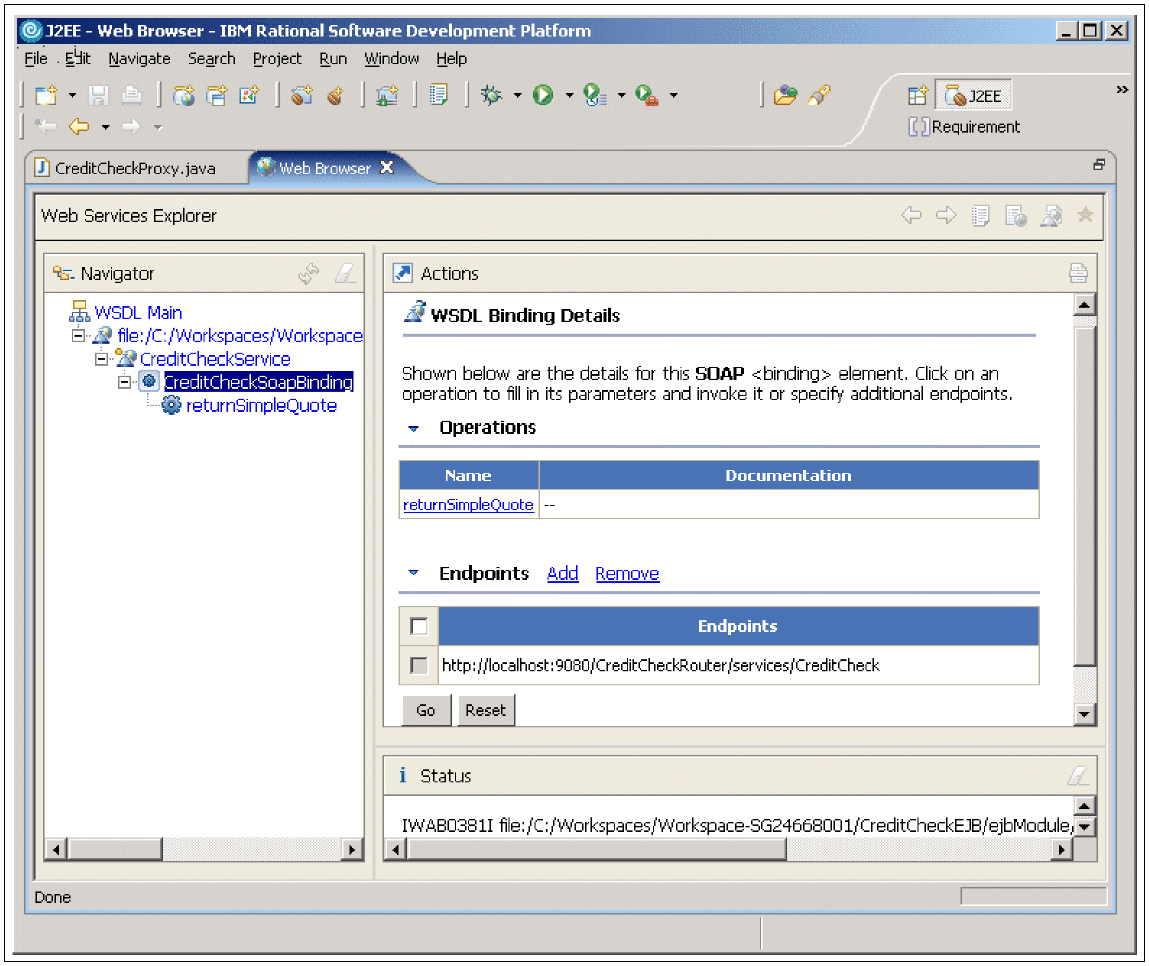
Figure 10-18 HomeDelivery EJB enabled as Web services

10.5.3 Testing with the Web Services Explorer

Once the Web service is installed and running in a server, it can be tested using the Web Services Explorer.

To start the Web Services Explorer, perform the following tasks:

1. Select the **CreditCheck.wsdl** file in `CreditCheckEJB/ejbModule/META-INF/wsdl`.
2. Right-click and select **Web Services** → **Test with Web Services Explorer**.
3. A Web Browser view opens with the WSDL file selected. It shows the operations (methods) that can be invoked and the endpoint (Figure 10-19).



WSDL Binding Details

Shown below are the details for this **SOAP** <binding> element. Click on an operation to fill in its parameters and invoke it or specify additional endpoints.

Operations

Name	Documentation
returnSimpleQuote	--

Endpoints Add Remove

☐	Endpoints
☐	http://localhost:9080/CreditCheckRouter/services/CreditCheck

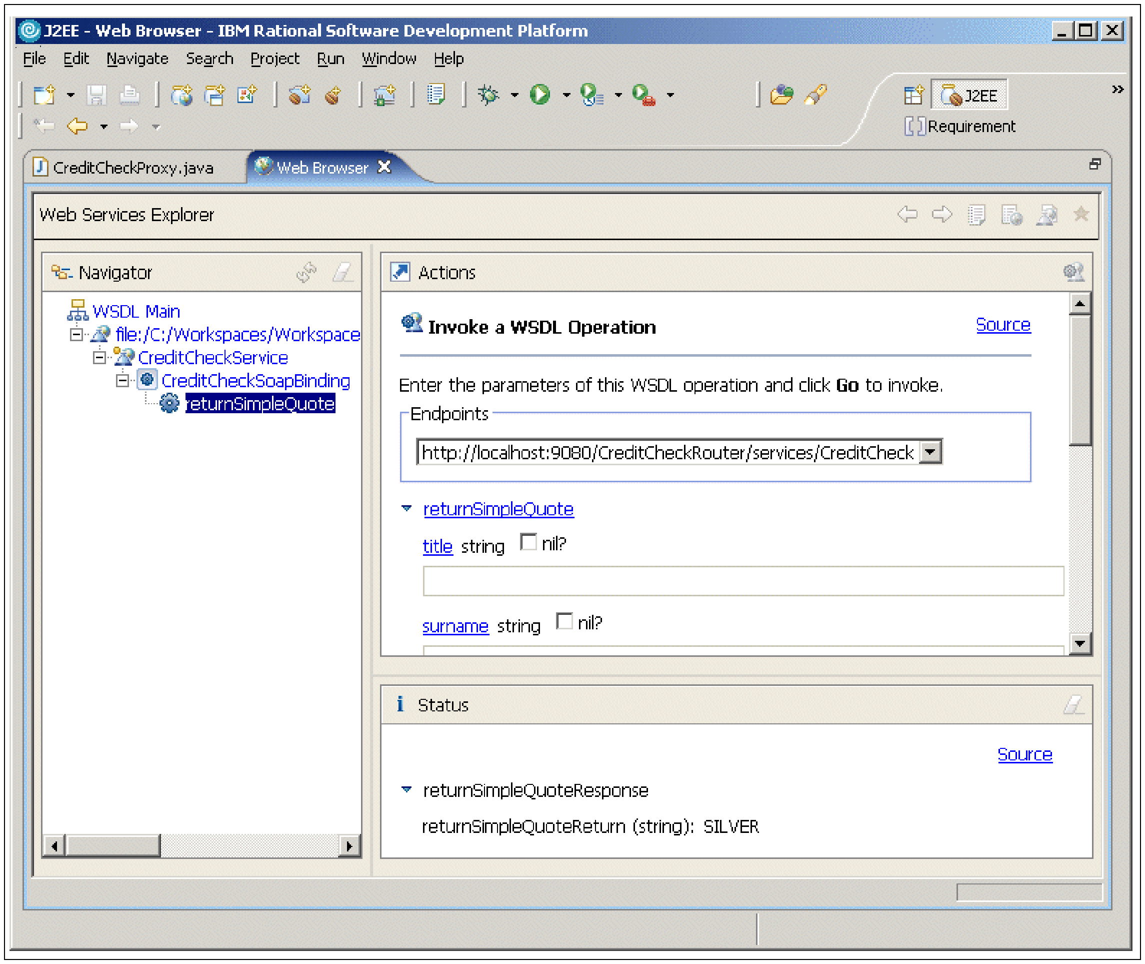
Go Reset

IWAB0381I file:/C:/Workspaces/Workspace-SG24668001/CreditCheckEJB/ejbModule/

Figure 10-19 Web Services Explorer: operations

To test the CreditCheck service, do the following:

1. Select the **returnSimpleQuote** operation. You are presented with input parameters for the service. In our example, no input values are necessary.
2. Click **Go** at the bottom of the Actions pane to execute the Web service. The Status pane displays the results (Figure 10-20 on page 316).



Invoke a WSDL Operation Source

Enter the parameters of this WSDL operation and click **Go** to invoke.

Endpoints

http://localhost:9080/CreditCheckRouter/services/CreditCheck

returnSimpleQuote

title string ☐ nil?

surname string ☐ nil?

Source

returnSimpleQuoteResponse

returnSimpleQuoteReturn (string): SILVER

Figure 10-20 Web Services Explorer: execution

3. Click **Source** in the Status pane to see the SOAP input and output messages. Double-click the Status pane header to maximize it.
4. A message from the CreditCheck application returns a credit assessment of either, GOLD, SILVER or BRONZE.

10.5.4 Creating Web service clients

Having the EJB Web service generated and running enables us to write a Web service client. The Web service client can be executed in different environments:

- Web: Servlets, JSPs, or Java beans invoked by a servlet or JSP
- EJB: Session EJBs or Java beans invoked by a session EJB

- Managed Java client: Java program running in an application client container.
- Stand-alone Java client: Java program running outside a container

For our example, we will create an EJB client.

Creating an EJB Web service client

Note for users of the service integration bus: The Web service client is built using WSDL files that describe the Web service. This section assumes you are using the WSDL generated by the Web Service wizard in the previous section. However, in our runtime configuration, we plan to use the WebSphere Application Server service integration bus as an intermediate destination point for the Web service. During the bus configuration, a new WSDL file for this Web service will be created pointing to the bus as the endpoint. We will show you how to update the client to use the new WSDL in 10.7.7, “Update the Web service clients to use the bus” on page 344.

ITSOMart uses two Web service clients, CreditCheckClient and ClientDelivery, both in Processor.ear. These clients are created in an identical manner. This section illustrates how CreditCheckClient is built, using the WSDL file from the CreditCheck Web service. A similar process would be used to create the ClientDelivery Web service client from the WSDL file of the HomeDelivery Web service.

Rational Software Architect provides the Web Services Client wizard to help you create a Web service client application.

Preparation: The client will be created in the Processor enterprise application. The Processor application contains an EJB module called CreditCheckClient that has a session bean called CCWSClientBean. This session bean will be used to call the Web service.

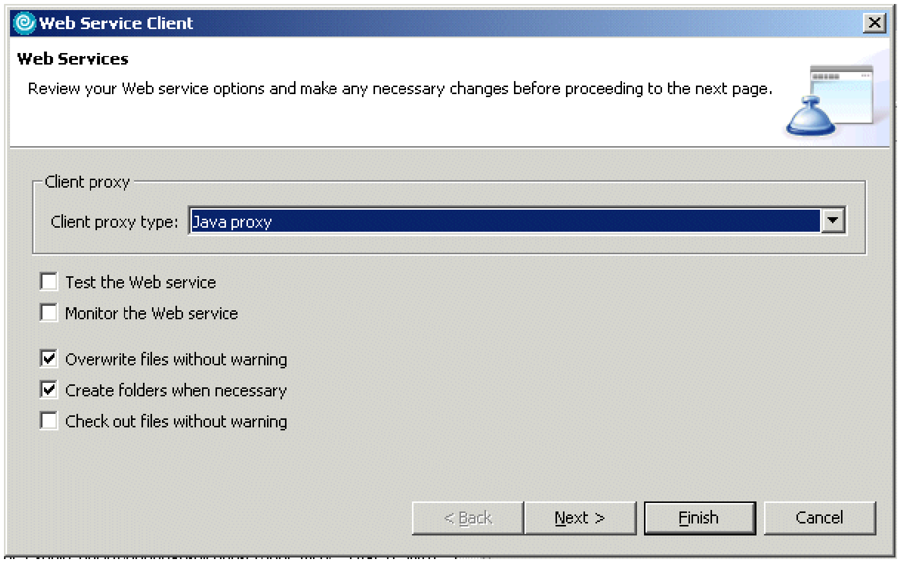
A new folder has been created in the CreditCheckClient module and the CreditCheck.wsdl file created previously (see “Generated files” on page 311) has been copied from the CreditCheckEJB module to the new folder, giving you the following:

```
CreditCheckClient/ejbModule/META-INF/wsdl/CreditCheck.wsdl
```

To create the Web service client, follow these steps:

1. Select **ProcessorEJB/ejbModule/META-INF/wsdl/CreditCheck.wsdl**.
2. Right-click and select **Web Services** → **Generate Client.** This starts the Web Service Client wizard.

3. Take the default values, shown in Figure 10-21, on the first panel.


Web Service Client

Web Services

Review your Web service options and make any necessary changes before proceeding to the next page.

Client proxy

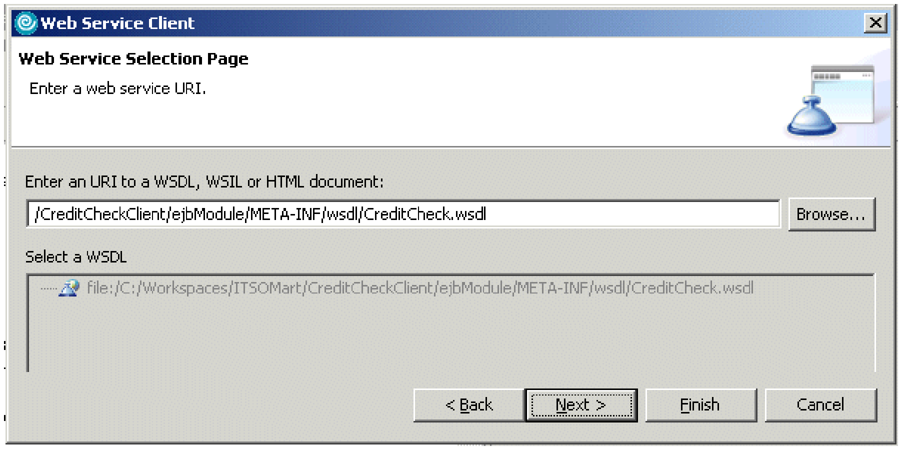
Client proxy type: Java proxy

- [ ] Test the Web service
- [ ] Monitor the Web service
- [x] Overwrite files without warning
- [x] Create folders when necessary
- [ ] Check out files without warning

Figure 10-21 Web Service Client wizard

Click **Next**

4. On Web Service Selection page (Figure 10-22), select the WSDL file:


Web Service Client

Web Service Selection Page

Enter a web service URI.

Enter an URI to a WSDL, WSIL or HTML document:

/CreditCheckClient/ejbModule/META-INF/wsdl/CreditCheck.wsdl

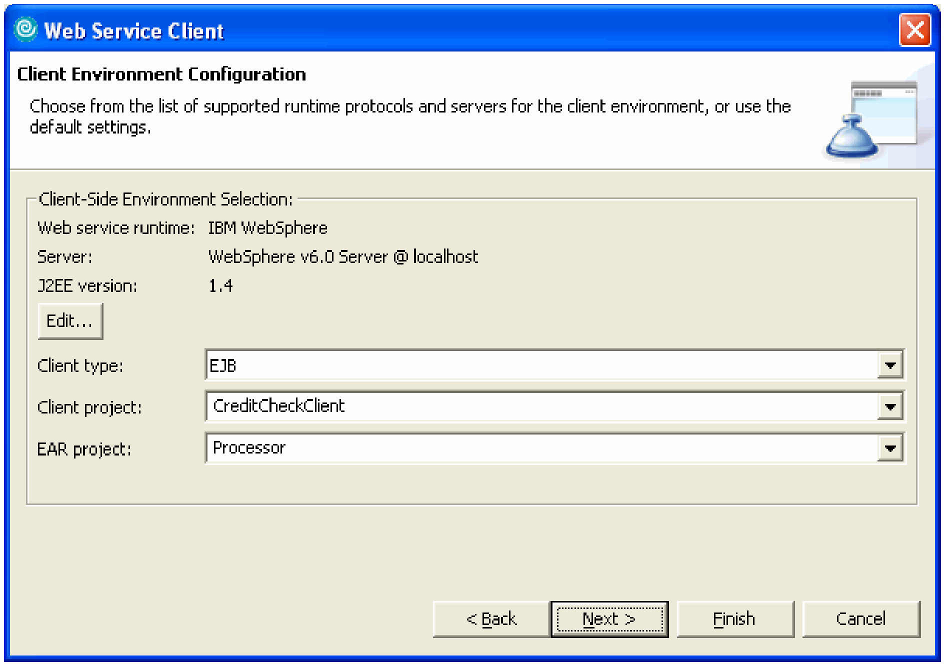
Select a WSDL

file:/C:/Workspaces/ITSOMart/CreditCheckClient/ejbModule/META-INF/wsdl/CreditCheck.wsdl

Figure 10-22 WSDL client definition

Click **Next**.

5. The settings on the Client Environment Definition page allow you to define the type of client to create and where to locate it. See Figure 10-23.



Client Environment Configuration

Choose from the list of supported runtime protocols and servers for the client environment, or use the default settings.

Client-Side Environment Selection:

Web service runtime:	IBM WebSphere
Server:	WebSphere v6.0 Server @ localhost
J2EE version:	1.4
Client type:	EJB
Client project:	CreditCheckClient
EAR project:	Processor

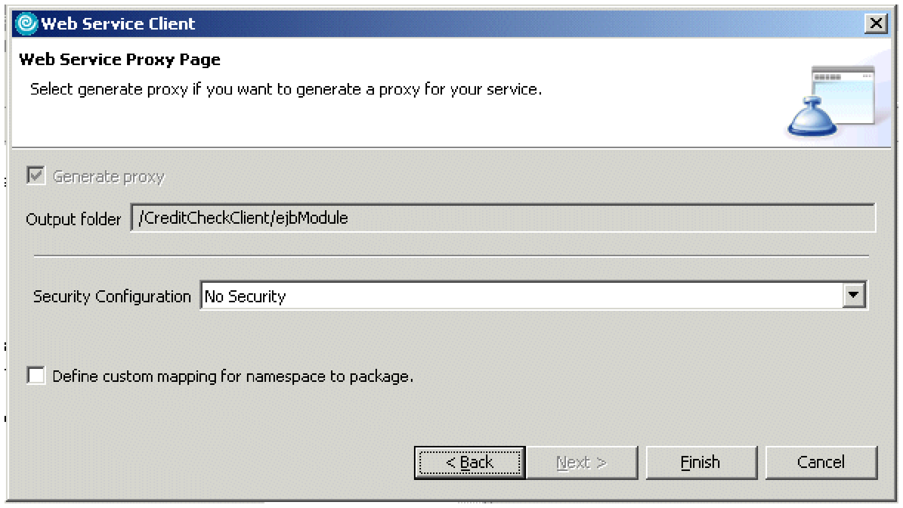
Figure 10-23 Client environment configuration

Select the following values:

– ClientType: **EJB**
– Client Project: **CreditCheckClient**
– EAR Project: **Processor**

Click **Next**.

6. The Client Proxy Page is displayed. See Figure 10-24 on page 320. Our sample application does not address security issues, so we take the defaults. This page also allows you to define custom namespace mappings. Ensure that this option is not selected.


Web Service Client

Web Service Proxy Page

Select generate proxy if you want to generate a proxy for your service.

☑ Generate proxy

Output folder /CreditCheckClient/ejbModule

Security Configuration No Security

☐ Define custom mapping for namespace to package.

Figure 10-24 Client proxy settings

7. Click **Finish**. The client code is generated and your Web service can now be accessed through it.

In addition to using the wizard to create the Web service, we now have to implement the returnSimpleQuote method in the CCWSClientBean. We use the proxy generated with the wizard in the previous steps as a means of locating the Web service and then calling the exposed returnSimpleQuote method. The following steps need to be followed:

1. Open the **CreditCheckClient** project.
2. Navigate to the **/ejbModule/com.ibm.patterns.creditCheck** package.
3. Open the **CCWSClientBean.java** and insert the code shown in Figure 10-30 on page 336 at the bottom of the bean. This method calls the proxy, which in turn, finds the CreditCheck Web service to obtain customer credit information from the CreditCheck application.

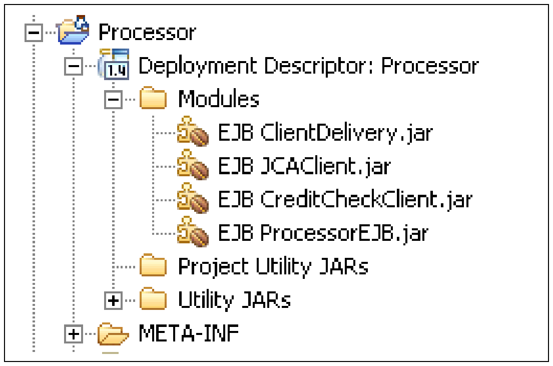
```
public String returnSimpleQuote(String title, String surname,
         String forename, String houseNo, String address1, String postcode,
         String country) {

      //implement the method, using the actual proxy
      String score = null;

      CreditCheckProxy ccProxy = new CreditCheckProxy();

      try {
         score = ccProxy.getCreditCheck().returnSimpleQuote(title, surname,
               forename, houseNo, address1, postcode, country);
      } catch (RemoteException e) {
         // TODO Auto-generated catch block
         e.printStackTrace();
      }

      return score;

   }
```

Figure 10-25 Implementation of exposed method in CCWSClientBean.java

4. Save your changes and deploy the changes to the server.

DeliveryClient

The Web service client for the DeliverySystem Web service is created in the ClientDelivery EJB. This EJB is also in the Processor EAR file (Figure 10-26).



```
Processor
   Deployment Descriptor: Processor
      Modules
         EJB ClientDelivery.jar
         EJB JCAClient.jar
         EJB CreditCheckClient.jar
         EJB ProcessorEJB.jar
      Project Utility JARs
      Utility JARs
   META-INF
```

Figure 10-26 Processor EAR file

A session bean called DeliveryWSClientBean has been created in the EJB. The WSDL file for the HomeDelivery Web service has been copied into the

ClientDelivery/ejbModule/META-INF/wsdl folder and is used to create the client. Note that only the WSDL for HomeDelivery is used. If the client request is actually for the BusinessDelivery Web service, the request is routed to that service by the ESB, not the client.

The code shown in Figure 10-27 is added to the DeliveryWSClientBean.

```
   public String getAccount(String type,String userId) throws
RemoteException
   {
      String score = null;

      HomeDeliveryProxy ccProxy = new HomeDeliveryProxy();

      try {
         score = ccProxy.getHomeDelivery().getAccount(type,userId);
      } catch (RemoteException e) {
         // TODO Auto-generated catch block
         e.printStackTrace();
      }

      return score;
   }
```

Figure 10-27 Implementation of exposed method in DeliveryWSClientBean

Incorporating the client into the application

The ITSOMart application calls the CreditCheck and DeliverySystem Web services from an EJB called ProcessorEJB in the Processor application. The code can be found in the createCustomer() method of com.ibm.patterns.serialProcess/Processor.java. Example 10-1 shows the code that invokes the credit check service.

Example 10-1 Code calling the credit check Web service

```
String creditRating = new
ProcessorWebService().getCreditRating(customerDetails);
```

Example 10-2 shows the code that calls the delivery system service.

Example 10-2 Code calling the delivery system Web service

```
String accountNo = new
DeliveryWebService().getAccount(deliveryType,customerDetails.getFname())
```

10.6 Creating the mediations

Mediations are implemented as mediation handlers. A mediation handler can be deployed, and each mediation handler executes some specific message processing at runtime (for example, transforming a message format or routing a message to a particular destination). A *mediation handler* is a Java program framework to which you add the code that performs the mediation function.

This section illustrates creation of two mediation applications:

1. A router mediation will receive messages at the destination created for the HomeDelivery outbound service. It will check the delivery type parameter send by client (Home, Business, or All) and based on the value it will send the message to the HomeDelivery outbound service, the BusinessDelivery outbound service or to both. If All is selected, the reverse routing path is changed to a queue that will be mediated with the aggregator mediation.
2. A mediation will aggregate responses from multiple services into one response. This situation will occur when a user elects to get both a home and a business account (All). This mediation will be attached to a queue destination that receives the responses from both the BusinessDelivery and HomeDelivery services.

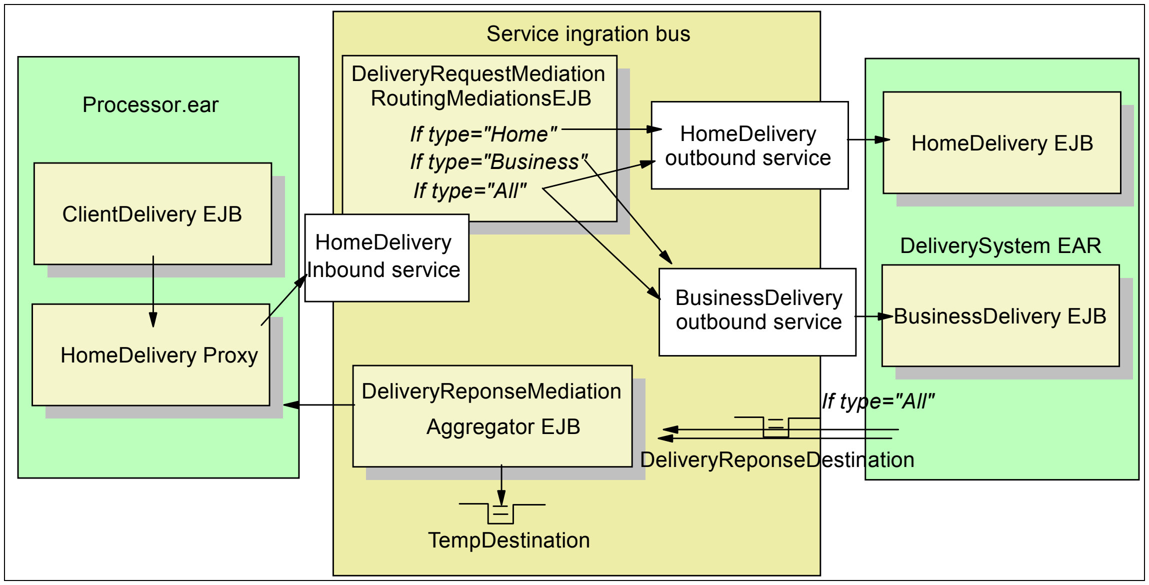


Figure 10-28 Mediation flow

Rational Application Developer provides support for developing mediation handler code and adding mediation handlers to the J2EE deployment descriptors.

10.6.1 Create the router mediation

The router mediation performs the following tasks:

- Examines the delivery type specified in the incoming message.
- If the type is Home, no action is required. The forward routing path of the message is already set to the HomeDelivery Web service.
- If the type is Business, it modifies the forward routing path of the message to send it to route it to the BusinessDelivery Web service.
- If the type is All, it:
  - Clones the message.
  - Modifies the forward routing path of one message to BusinessDelivery.
  - Modifies the reverse routing path of both messages to a queue destination (for aggregation).

Use the following steps to create the router mediation:

1. Build the code:
   a. Create a simple project called `RoutingMediations`.
   b. Right-click the RoutingMediations project and select **Add** → **Others** → **Java Class**. Name the class `RoutingMediation`.
   c. Implement the MediationHandler interface and add the routing code in the handle method (see "RouterMediation code" on page 325).
   d. Create a JAR file from the package and call it RoutingMediations.jar.
2. Create the mediation EJB:
   a. Create an EJB project called `RoutingMediationsEJB` with an EAR file name of `RoutingMediationsEJBEAR`.
   b. Drag and drop RoutingMediations.jar to the Utility JARS folder in `RoutingMediationsEJBEAR`.
3. Define the mediation handler in the EJB deployment descriptor:
   a. Expand the RoutingMediationsEJB project in the navigator and double-click **Deployment Descriptor**.
   b. Select the Mediation Handler tab.

c. Click the **Add** button and enter `RoutingHandler` as the name. Enter `myPackage.RoutingMediation` as the handler class.

d. Save the descriptor.

RouterMediation code

Note: The entire source code for the router mediation can be seen by downloading the sample application. The code is in the RoutingMediations.jar in the RoutingMediationsEJB project.

The handle() method (Example 10-3) is invoked by the arrival of a message on the mediated port. It gets a handle to the MessageContext interface, which provides methods to manage a property set. Message context properties enable handlers in a handler chain to share processing related state. The context properties used for the mediation (Table 10-3 on page 350) are defined when you prepare the runtime for mediation.

The handle() method casts the message context to SIMessageContext. This is the object that is required on the interface of a mediation handler. In addition to the context information that can be passed from one handler to another, it can return a reference to an SIMessage and an SIMediationSession. The SIMessage is the service integration bus representation of the message being processed by the MediationHandler. An SIMessage contains message properties, header contents, routing path, and the message body. The SIMediationSession is a handle to the runtime resources.

Example 10-3 RoutingMediation part 1

```
public class RoutingMediation implements MediationHandler {

   public boolean handle(MessageContext context) throws MessageContextException {

      String businessDestination=(String)context.getProperty("businessDestination");
      SIMessageContext ctx =(SIMessageContext)context;
      SIMediationSession session = ctx.getSession();
      String busName = session.getBusName();
      SIMessage message = ctx.getSIMessage();
      SIMessage newMessage=null;
      List frp=null;
      SIDestinationAddressFactory factory =
            SIDestinationAddressFactory.getInstance();
```

Next (Example 10-4 on page 326), we retrieve the DataGraph object from the message. The message is inspected to find the type of delivery the customer has requested-Home, Delivery, or All (both).

Example 10-4 RoutingMediation part 2

```
    try {

        DataGraph graph = message.getDataGraph();
        DataObject rootNode = graph.getRootObject();
        DataObject infoNode = rootNode.getDataObject("Info");

        // Get the body node
        DataObject bodyNode = infoNode.getDataObject("body");

        // Get the data object for the first part of the body
        DataObject part1Node = bodyNode.getDataObject("parameters");

        // Determine the delivery option requested by the customer (Home, Business, All)
        String ticker = part1Node.getString((String)context.getProperty("type"));
```

If the user has requested both home and business delivery (type=All), the next section (Example 10-5) clones the message.

The original message has a forward routing path that will send it to the HomeDelivery service. The forward routing path in the cloned message is set to send it to the BusinessDelivery service, using the businessDestination context property.

The reverse routing path for both messages is changed to the value of the DeliveryResponseDestination context property. In this case, the property is set to a destination queue defined on the bus. Once the forward and reverse routing paths are properly set, the new message is sent.

Example 10-5 RoutingMediation part 3

```
        if (ticker.equalsIgnoreCase((String)context.getProperty("ALL")))
        {
        newMessage = (SIMessage)message.clone();
         List frp1= newMessage.getForwardRoutingPath();
         newMessage.setApiMessageId(message.getApiMessageId());
         newMessage.setDataGraph(message.getDataGraph(),message.getFormat());

           SIDestinationAddress businessDestAddress = factory.
              createSIDestinationAddress(businessDestination,busName);

         frp1.add(businessDestAddress);
         newMessage.setForwardRoutingPath(frp1);
         List reverse1 = newMessage.getReverseRoutingPath();
         System.out.println("Reverse Routing Path --  "+reverse1);
         SIDestinationAddress deliveryResponseAddress = factory.
```

```
createSIDestinationAddress((String)context.getProperty("DeliveryResponseDestination"),busName);
            reverse1.clear();
            reverse1.add(deliveryResponseAddress);
            newMessage.setReverseRoutingPath(reverse1);

            List reverse = message.getReverseRoutingPath();
             reverse.clear();
             reverse.add(deliveryResponseAddress);
             message.setReverseRoutingPath(reverse);

             try {
                 System.out.println("sending Cloned Message--  "+newMessage);
               session.send(newMessage,false);
                System.out.println("Cloned Message sent ");
             } catch (SIMediationRoutingException e2) {
               e2.printStackTrace();
             } catch (SIDestinationNotFoundException e2) {
               e2.printStackTrace();
             } catch (SINotAuthorizedException e2) {
               e2.printStackTrace();
             }
           }
```

If the user has requested only business delivery, the next section (Example 10-6) sets the forward routing path of the original message to the BusinessDelivery service.

Example 10-6 RoutingMediation part 4

```
else if (ticker.equalsIgnoreCase((String)context.getProperty("BusinessDelivery")))
{
frp= message.getForwardRoutingPath();
   SIDestinationAddress businessDestAddress = factory.
     createSIDestinationAddress(businessDestination,busName);
 frp.add(businessDestAddress);
 message.setForwardRoutingPath(frp);
}
```

Finally (Example 10-7), if the user has requested only home delivery, no action is needed.

Example 10-7 RoutingMediation.class part 5

```
else if (ticker.equalsIgnoreCase((String)context.getProperty("HomeDelivery")))
{
System.out.println("****    End Of The Request    ****");
}
```

```
        }
        catch (Exception e) {
             e.printStackTrace();
           throw new MessageContextException();
        }
        return true;
    }
}
```

10.6.2 Create the Aggregator mediation

This mediation is associated with the reply queue destination. When an incoming message has the delivery type All, the request is sent to both the HomeDelivery and the BusinessDelivery services. The reverse routing path is modified to new reply queue destination. The response from both the messages is sent to this reply queue destination where the Aggregator mediation is invoked. The mediation aggregates the responses from both services into one response and sends it back to the client.

The following are the steps to create the aggregator mediation:

1. Build the code:
   a. Create a simple project called `Aggregator`.
   b. Right-click the Aggregator project and select **Add** → **Others** → **Java Class**. Name the class `AggregatorMediation`.
   c. Implement the MediationHandler interface and add the code required for aggregating the responses in the handle method (see “Aggregator code” on page 329).
   d. Create a JAR file from the package and call it `Aggregator.jar`.
2. Create the mediation EJB:
   a. Create an EJB project called `AggregatorEJB` with an EAR file name of `AggregatorEJBEAR`.
   b. Drag and drop Aggregator.jar to the Utility JARS folder in `AggregatorEJBEAR`.
3. Define the mediation handler in the EJB deployment descriptor:
   a. Expand the `AggregatorEJB` project in the navigator and double-click **Deployment Descriptor**.
   b. Select the Mediation Handler tab.

c. Click **Add** and enter `AggregatorHandler` as the name. Enter `myPackage.AggregatorMediation` as the handler class.

d. Save the descriptor.

Aggregator code

Note: The entire source code for the aggregator mediation can be seen by downloading the sample application. The code is in the Aggregator.jar file in the AggregatorEJB project.

The handle() method (Example 10-8) is responsible for the message handling and aggregation of the response messages, and sending the aggregated response back to the original requester. The handle() method is invoked by the arrival of a message on the queue that it has been configured to mediate, DeliveryResponseDestination.

The handle() method casts the message context to an SIMessageContext and retrieves the forward routing path for the message.

Example 10-8 AggregatorMediation code part 1

```
public class AggregatorMediation implements MediationHandler {

   public boolean handle(MessageContext context) throws MessageContextException {

      SIMessageContext ctx =(SIMessageContext)context;
       String busName = ctx.getSession().getBusName();
        SIMessage message = ctx.getSIMessage();
        List frp = message.getForwardRoutingPath();
        SIMediationSession session = ctx.getSession();
        SIMessage otherMessage=null;
        SIDestinationAddressFactory factory =
           SIDestinationAddressFactory.getInstance();
```

Next (Example 10-9), it retrieves the name of a temporary storage queue it will use to hold received messages. The code checks to see if it can receive a message from the temporary queue. If no message is there, then the message that has arrived on the mediated queue is the first response. If it is the first response, the forward routing path is set to the temporary queue and the message is stored there.

Example 10-9 AggregatorMediation code part 2

```
        try {
        otherMessage = session.receive((String)context.getProperty("TempDataDestination"));
     if (otherMessage == null)
```

```
      {
         SIDestinationAddress queue= factory.
   createSIDestinationAddress((String)context.getProperty("TempDataDestination"),busName);
          frp.clear();
           frp.add(queue);
           message.setForwardRoutingPath(frp);
      }
```

If there is a message on the temporary queue, this response is the second message (Example 10-10).

The first message (otherMessage) is retrieved from the temporary queue and the account number information (getAccountReturn) is extracted. Next, the account information is extracted from the second message (message). Then the two responses are combined and written back into the current (second) message.

Example 10-10 AggregatorMediation code part 3

```
      else
      {
         DataGraph graph = null;
         try {
            graph = otherMessage.getDataGraph();
            DataObject rootNode = graph.getRootObject();
             DataObject infoNode = rootNode.getDataObject("Info");
             DataObject bodyNode = infoNode.getDataObject("body");
              DataObject part1Node = bodyNode.getDataObject("parameters");
              String ticker = part1Node.getString("getAccountReturn");

              DataGraph graph1 = message.getDataGraph();
                DataObject rootNode1 = graph1.getRootObject();
                 DataObject infoNode1 = rootNode1.getDataObject("Info");
                 DataObject bodyNode1 = infoNode1.getDataObject("body");
                  DataObject part1Node1 = bodyNode1.getDataObject("parameters");
                  String ticker1 = part1Node1.getString("getAccountReturn");

              part1Node1.setString("getAccountReturn",
                ticker+","+ticker1);
```

Lastly (Example 10-11), the forward routing path is changed so the response will be sent back to the client.

Example 10-11 AggregatorMediation code part 4

```
          String newDestination =
(String)context.getProperty("InboundServiceClientResponseDest");

           SIDestinationAddress newAddress = factory.
```

```
                createSIDestinationAddress(newDestination,busName);
            frp.clear();
             frp.add(newAddress);
             message.setForwardRoutingPath(frp);
        }
        return true;
    }
}
```

10.6.3 Extending the mediations

The techniques in this simple example illustrate how to manipulate messages in a mediation. The code as written will work if you only expect two responses.

You can extend this example to manage more than two responses by adding a queue to contain control data that indicates how many messages were sent out. This information is used by the aggregator to know how many responses to expect. For an example of this, see the broker scenario in *Patterns: SOA with an Enterprise Service Bus in WebSphere Application Server V6,* SG24-6494.

Another element to consider is how long to wait for a response from each service. This example waits indefinitely, but it would be reasonable include code that handles a situation where not all responses are received within a time limit.

10.7 Runtime guidelines for Web services

This section describes how to configure the service integration bus for Web services. We first look at the Web services support provided in WebSphere Application Server. We then go into specific configuration tasks required to run the sample application.

10.7.1 Web services support in WebSphere Application Server V6

WebSphere Application Server can act as both a Web service provider and as a requester. As a provider, it hosts Web services that are published for use by clients. As a requester, it hosts applications that invoke Web services from other locations. WebSphere Application Server supports SOAP-based Web service hosting and invocation.

WebSphere Application Server V6 provides J2EE 1.4 support, including Web services 1.1.

- JAX-RPC v1.0 for J2EE 1.3, v1.1 for J2EE 1.4
- JSR 109 (Web services for J2EE)

- WS-I Basic Profile 1.1.2 support
- WS-I Simple SOAP Binding Profile 1.0.3
- WS-I Attachments Profile 1.0
- SAAJ 1.2
- UDDI V2 and V3
- JAXR
- WS-TX (transactions)
- SOAP 1.1
- WSDL 1.1 for Web services
- WSIL 1.0 for Web services
- OASIS Web Services Security: SOAP Message Security 1.0 (WS-Security 2004)
- OASIS Web Services Security: UsernameToken Profile 1.0
- OASIS Web Services Security X.509 Certificate Token Profile

WebSphere Application Server also provides an integrated private UDDI V3 registry and in the Network Deployment package, integrated Web Services Gateway function.

Web services and the service integration bus

Also new, is the ability to use the service integration bus as an intermediary between service requestors and service providers, allowing control over the flow, routing, and transformation of messages through mediations and JAX-RPC handlers. The bus provides a flexible way to expose and call services located in an intranet from the Internet (and vice versa), while also providing mechanisms for protocol switching and security.

The use of Web services with the bus is an evolution of the Web Services Gateway provided in WebSphere Application Server V5. Whereas the Web Services Gateway was a stand-alone application in V5, the bus is more tightly integrated into the application server, enabling users to take advantage of WebSphere Application Server administration and scalability options, and also build on top of the asynchronous messaging features provided by WebSphere Application Server.

The bus enables users to specify a level of indirection between service requestors and providers by exposing existing services at new destinations. It also provides options for managing these services through *mediations*, which can access and manipulate incoming and outgoing message content, or even route the message to a different service. Support for JAX-RPC handlers is also included in the bus, as is Web services gateway functionality.

While all WebSphere Application Server packages can use the bus as a destination point for Web services, the Web Services Gateway function is only available in the Network Deployment package.

Figure 10-29 illustrates a basic bus configuration and how it can enable Web services clients in an intranet to access an Internet-based Web service. Clients would use the bus-generated WSDL to access the service, and the specified mediations could be used for message logging, transformation, routing, or other purposes.

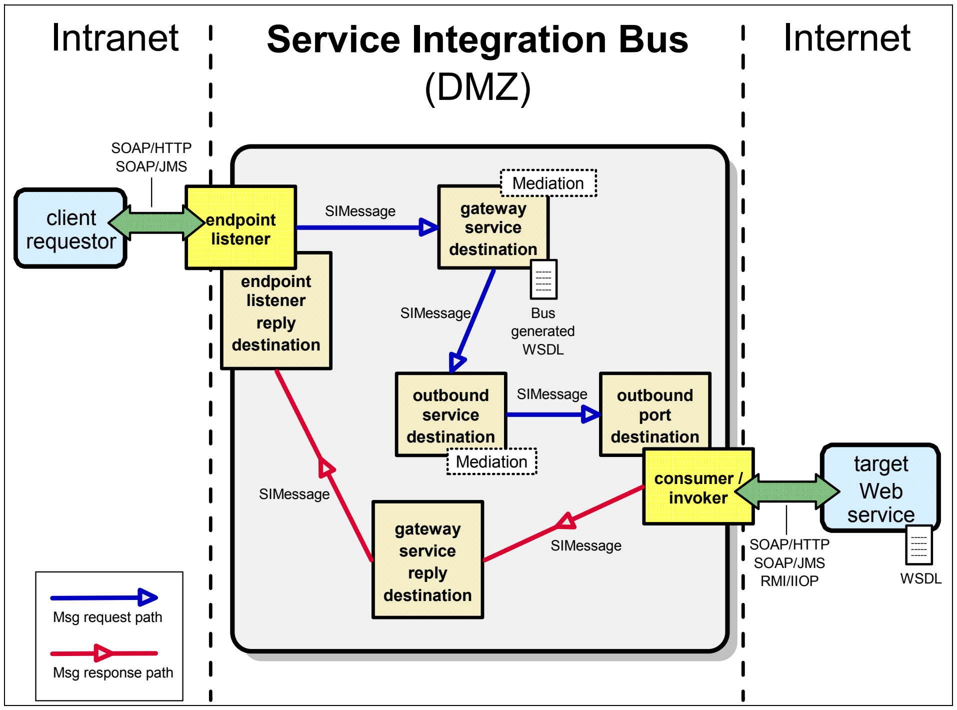


Figure 10-29 Exposing Web services through the bus

The following terms apply to the Web services support in the bus:

- *Endpoint listeners* are entry points to the bus for Web services clients. Endpoint listeners control whether clients connect over SOAP/HTTP or SOAP/JMS. They are associated with inbound services and gateway resources.
- *Inbound services* are destinations within the bus exposed as Web services. Inbound services define how Web service consumers communicate with the bus.
- *Outbound services* are destinations within the bus that represent external Web services. Outbound services define how the bus communicates with the Web service providers.

  An inbound and outbound service needs to be defined for each Web service routed through the bus.

- A *Gateway instance* enables a user to create gateway and proxy services.
- A *Gateway service* exposes external Web services as bus-managed Web services.
- A *Proxy service* exposes external Web services as bus-managed Web services, but with the added feature of allowing runtime control over the target service endpoint that is called.
- A *Mediation* is a stateless session EJB attached to a service destination that can apply processing to messages that pass through it, for example, logging or message transformation.
- *JAX-RPC handler* is a J2EE standard for intercepting and manipulating Web services messages.
- *JAX-RPC handler list* is used to manage JAX-RPC handlers by determining the order in which they are executed. These lists can be associated with bus services.
- The *UDDI reference* contains configuration information for UDDI registries to which the bus is able to connect.

10.7.2 Configuration tasks

In the ITSOMart scenario, we have chosen to use the service integration bus as our implementation of the ESB. The configuration tasks that need to be done in order to run the credit check portion of the application are as follows:

1. Create an endpoint listener.
2. Create the outbound services.
3. Create the inbound services.
4. Generate and export new WSDL for the services.
5. Update the Web service clients to use the bus.
6. Configure the router mediation.

Preparation: Our scenario assumes that you have done the following:

1. Created a bus and added the application server as a member. The bus definition is fundamental to most other configuration tasks. It is not possible to begin creating or configuring any services or gateway resources until a bus object has been created. One bus is normally sufficient to handle all application servers in a cell. In our scenario, we use one bus called ITSOMartBus.

   For information about creating the bus and adding the members, see “Create a service integration bus” on page 463.
2. Installed the Web services support for the bus. For information about how to do this, see “Install Web services support for the bus” on page 464.

10.7.3 Create an endpoint listener

Endpoint listeners are required if you want to expose destinations on the bus to clients connecting over SOAP/HTTP and SOAP/JMS. They are the entry points to the bus for these protocols, carrying requests between Web services clients and buses, and are used by both inbound services and gateway services. An endpoint listener acts as the ultimate receiver of an incoming SOAP message to the bus and passes on the data and context of that message.

Note: Before creating endpoints, make sure you have the CreditCheck and DeliverySystem applications deployed and running.

To define a new endpoint listener using the administrative console, perform the following steps:

3. Expand **Servers** and click **Application Servers**.
4. Click the server name.
5. Under Additional Properties click **Endpoint Listeners**.
6. Click **New**.
7. For our CreditCheck example using SOAP/HTTP, enter in the following values:
   - Name

     This is the name of the endpoint listener. It should have a value of `SOAPHTTPChannel1`.
   - URL root

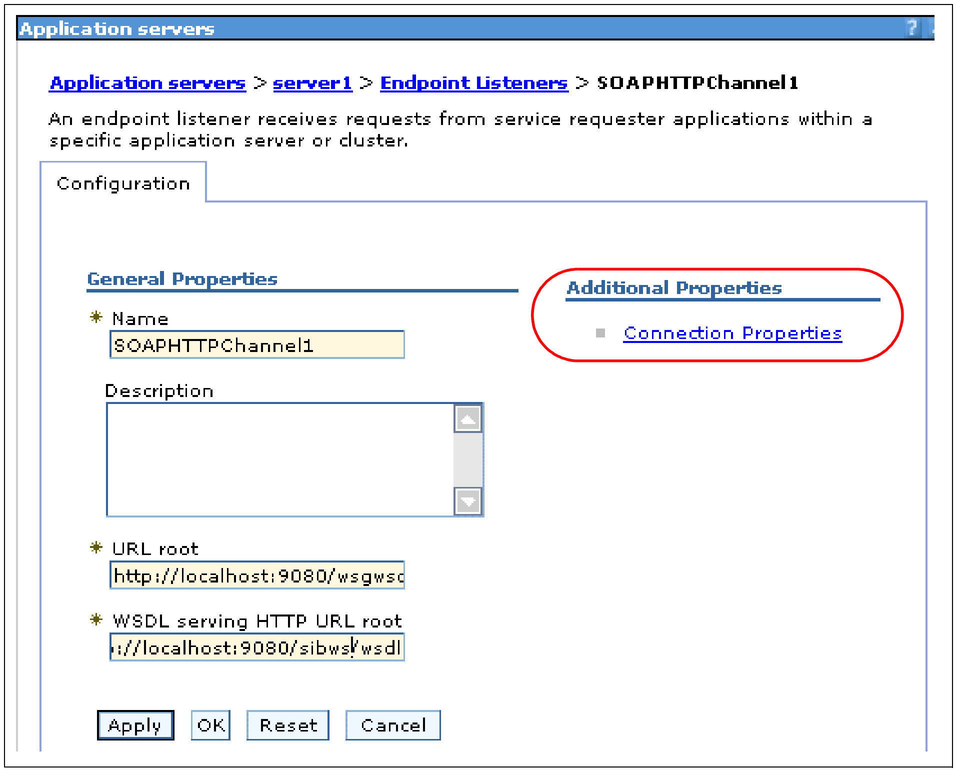
     This is the base URL for Web service requests into this endpoint listener. The URLs used for making Web service requests to the service integration bus will have this at the beginning. Set this to:

     `http://localhost:9080/wsgwsoaphttp1`

     Where `localhost` can be replaced with the server's host name and `9080` may be replaced with the correct port number for your server setup.
   - WSDL serving HTTP URL root

     The location of the HTTP URL that is serving your Web service WSDL. Enter a value of:

     `http://localhost:9080/sibws/wsdl`
8. Click **Apply**. This saves the General Properties and makes the Additional Properties available (Figure 10-30 on page 336).


Application servers

Application servers > server1 > Endpoint Listeners > SOAPHTTPChannel1

An endpoint listener receives requests from service requester applications within a specific application server or cluster.

Configuration

General Properties

* Name: SOAPHTTPChannel1

Description:

* URL root: http://localhost:9080/wsgwsc

* WSDL serving HTTP URL root: ://localhost:9080/sibws/wsdl

Additional Properties

- Connection Properties

Figure 10-30 Additional properties of Endpoint

9. From the Additional Properties list, click **Connection properties** then **New**.
10. A drop-down list of available buses appears to which you can connect the endpoint listener. Select the bus you want to use, click **OK**, and then click **save** to save your changes.

This completes the configuration of your endpoint listener. By connecting it to a bus, we make it available for use with inbound services.

Note: The URL root values supplied for the SOAP/HTTP endpoint listeners assume that you have used the defaults supplied in the endpoint listener applications. If you have changed these, you also have to modify the values you supply when creating the endpoint listener.

10.7.4 Create the outbound services

Outbound services define Web service requests that leave the service integration bus and are received by a service provider. See Figure 10-31.

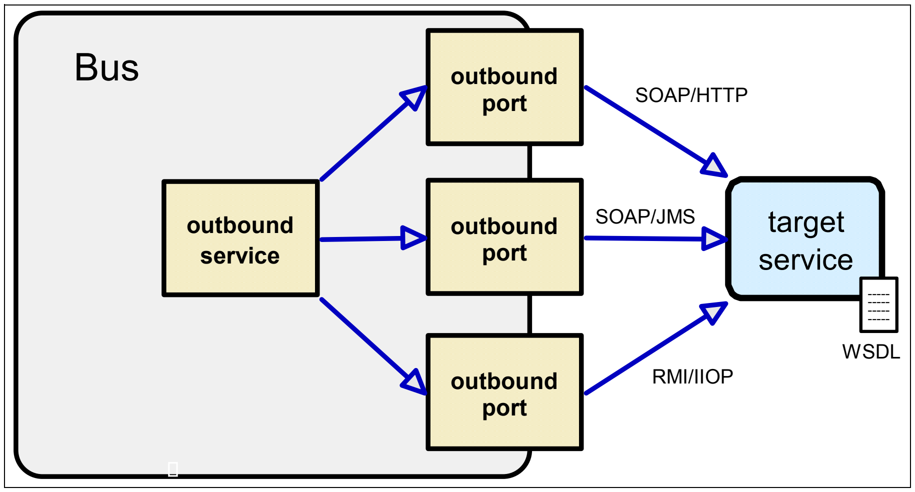
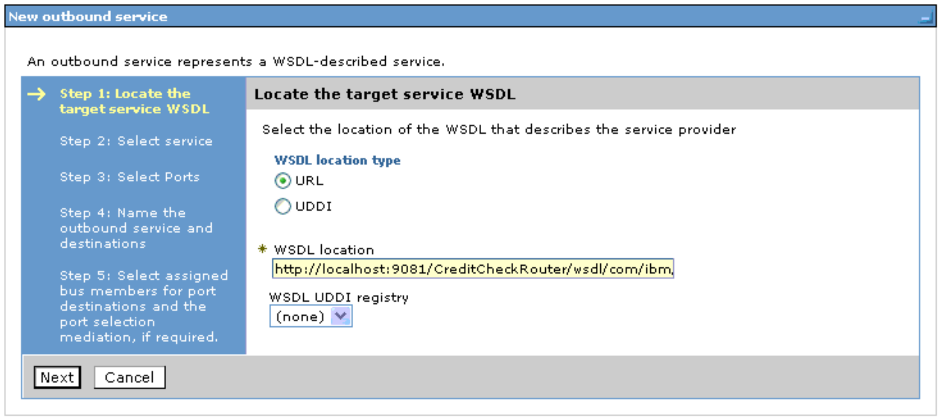


Figure 10-31 A typical outbound service configuration

In our scenario, we need to define an outbound service definition for the CreditCheck, HomeDelivery, and BusinessDelivery Web services. Start with the CreditCheck service:

1. From the administrative console, expand **Service integration** and click **Buses**.
2. Click **ITSOMartBus**.
3. Under Additional Properties click **Outbound Services**.
4. Click **New**.
5. The first page of the wizard (Figure 10-32 on page 338) requires you to specify a URL or UDDI repository where a WSDL definition of the service can be found. In our case, we are using a URL. The URL options allows you to specify an HTTP URL or a file system path. Enter the following URL then click **Next**.

```
http://localhost:9080/CreditCheckRouter/wsdl/com/ibm/patterns/creditCheck/
CreditCheck.wsdl
```


New outbound service

An outbound service represents a WSDL-described service.

→ Step 1: Locate the target service WSDL

Step 2: Select service

Step 3: Select Ports

Step 4: Name the outbound service and destinations

Step 5: Select assigned bus members for port destinations and the port selection mediation, if required.

Locate the target service WSDL

Select the location of the WSDL that describes the service provider

WSDL location type

● URL

○ UDDI

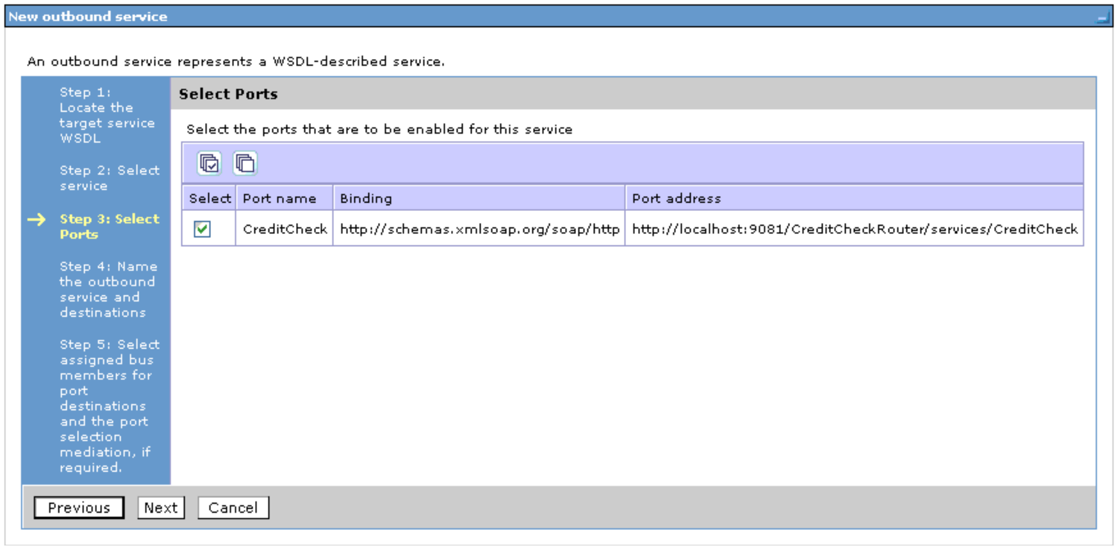
* WSDL location: http://localhost:9081/CreditCheckRouter/wsdl/com/ibm,

WSDL UDDI registry: (none)

Next | Cancel

Figure 10-32 Service destination and template WSDL settings page

6. The next page displays the available services defined in the WSDL file. This page allows you to select which service you wish to create an outbound service for. In our case there is only one service to select **CreditCheckService**. Click **Next**.
7. The next page (Figure 10-33) displays the ports defined for the selected service. There is only one port in our service, so check **CreditCheck** and click **Next**.


New outbound service

An outbound service represents a WSDL-described service.

Step 1: Locate the target service WSDL

Step 2: Select service

→ Step 3: Select Ports

Step 4: Name the outbound service and destinations

Step 5: Select assigned bus members for port destinations and the port selection mediation, if required.

Select Ports

Select the ports that are to be enabled for this service

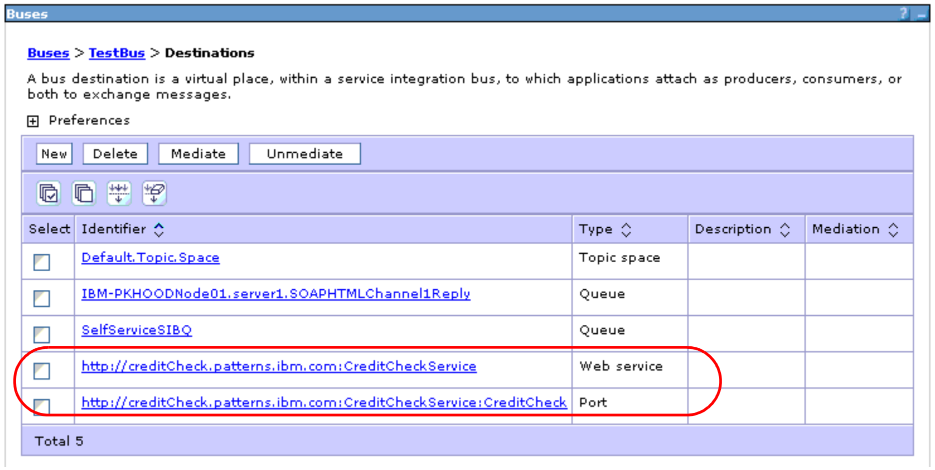
Select	Port name	Binding	Port address
✓	CreditCheck	http://schemas.xmlsoap.org/soap/http	http://localhost:9081/CreditCheckRouter/services/CreditCheck

Previous | Next | Cancel

Figure 10-33 Port sections page

8. The next page allows you to change the name of the outbound service, service destination name and port destination name. It also allows you to specify a port selection mediation. Accept the defaults and click **Next**.
9. The final page allows you to select the bus member to which to assign the outbound service. Accept the defaults and click **Finish**. The outbound service will be created.

   A Web service destination and port destination are created for each outbound service. You can see these destinations by clicking on **Destinations** in the bus details page. See Figure 10-34 for details.


Buses

Buses > TestBus > Destinations

A bus destination is a virtual place, within a service integration bus, to which applications attach as producers, consumers, or both to exchange messages.

⊞ Preferences

Select	Identifier	Type	Description	Mediation
☐	Default.Topic.Space	Topic space		
☐	IBM-PKHOODNode01.server1.SOAPHTMLChannel1Reply	Queue		
☐	SelfServiceSIBQ	Queue		
☐	http://creditCheck.patterns.ibm.com:CreditCheckService	Web service		
☐	http://creditCheck.patterns.ibm.com:CreditCheckService:CreditCheck	Port		

Total 5

Figure 10-34 Newly created outbound Web service and port destination

10. Save the changes.

HomeDelivery and BusinessDelivery outbound services

Repeat these steps and create two additional outbound services for HomeDelivery and BusinessDelivery using the following WSDL locations:

- HomeDelivery:

  `http://localhost:9080/HomeDeliveryRouter/wsdl/com/ibm/patterns/delivery/HomeDelivery.wsdl`

- BusinessDelivery:

  `http://localhost:9080/BusinessDeliveryRouter/wsdl/com/ibm/patterns/delivery/BusinessDelivery.wsdl`

10.7.5 Create the inbound services

Inbound services define Web service requests that are received by the bus. These requests are then routed to the appropriate outbound service.

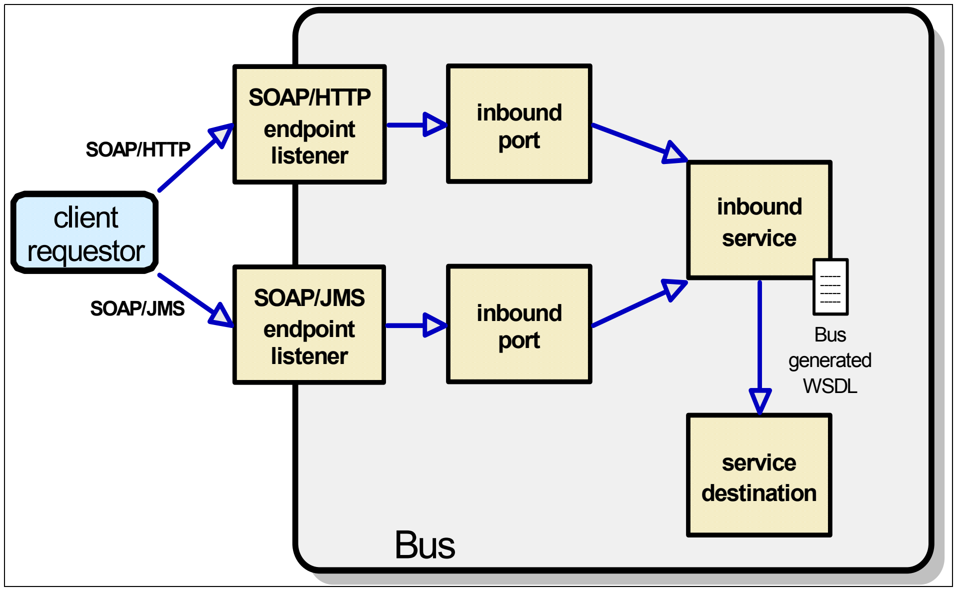
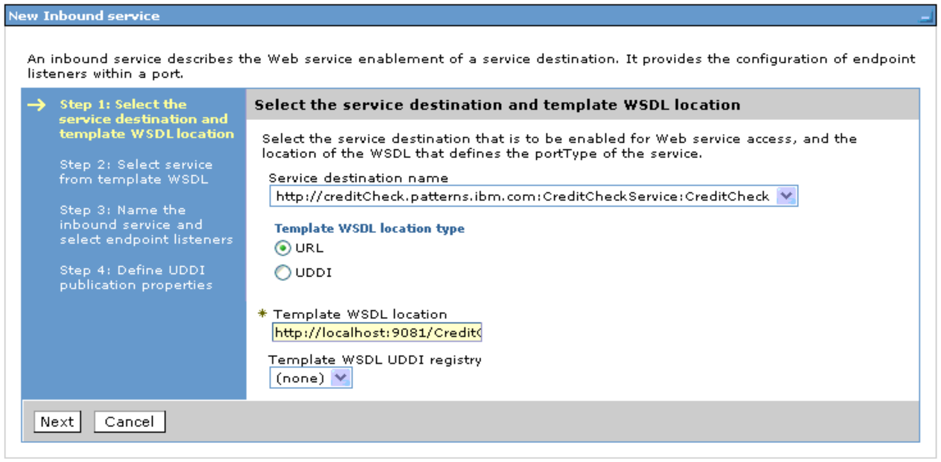


Figure 10-35 A typical inbound service configuration

We need to define inbound services for the CreditCheck service and HomeDelivery service. Start with the CreditCheck inbound service:

1. From the bus details page under Additional Properties, click **Inbound Services**.
2. Click **New**.
3. The first page of the wizard, shown in Figure 10-36 on page 341, is used to select the service destination name and supply the template WSDL service definition.


New Inbound service

An inbound service describes the Web service enablement of a service destination. It provides the configuration of endpoint listeners within a port.

→ Step 1: Select the service destination and template WSDL location
Step 2: Select service from template WSDL
Step 3: Name the inbound service and select endpoint listeners
Step 4: Define UDDI publication properties

Select the service destination and template WSDL location

Select the service destination that is to be enabled for Web service access, and the location of the WSDL that defines the portType of the service.

Service destination name: http://creditCheck.patterns.ibm.com:CreditCheckService:CreditCheck

Template WSDL location type: [✓ URL] UDDI

* Template WSDL location: http://localhost:9081/Credit(

Template WSDL UDDI registry: (none)

Figure 10-36 Service destination and template WSDL settings page

– Service destination name

Use the drop-down to select the destination the inbound service requests should be placed on. In our case we want to specify the Web service destination that was created for the CreditCheck outbound service.

Select the value of:

http://creditCheck.patterns.ibm.com:CreditCheckService:CreditCheck

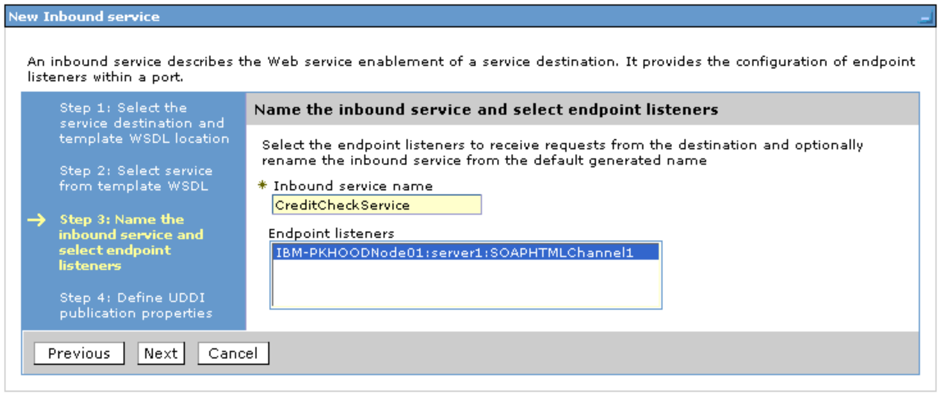
– Template WSDL location

This field is for specifying the WSDL definition of the Web service to be invoked. While the WSDL that will be used by client applications will be slightly different, it will be based on this WSDL. In our scenario we will specify the WSDL of the Web service endpoint that will ultimately be invoked once the request has been routed through the bus:

Enter a value of:

```
http://localhost:9080/CreditCheckRouter/wsdl/com/ibm/patterns/
creditCheck/CreditCheck.wsdl
```

4. Click **Next**.
5. The next page asks which service in the template WSDL should be used. Our WSDL has only one entry, so accept the default and click **Next**.
6. The next page, shown in Figure 10-37 on page 342, allows for you to rename the inbound service and to specify which endpoint listener is to be used.


New Inbound service

An inbound service describes the Web service enablement of a service destination. It provides the configuration of endpoint listeners within a port.

Step 1: Select the service destination and template WSDL location
Step 2: Select service from template WSDL
Step 3: Name the inbound service and select endpoint listeners
Step 4: Define UDDI publication properties

Name the inbound service and select endpoint listeners

Select the endpoint listeners to receive requests from the destination and optionally rename the inbound service from the default generated name

* Inbound service name: CreditCheckService

Endpoint listeners: IBM-PKHOODNode01:server1:SOAPHTMLChannel1

Figure 10-37 Specify inbound service name and endpoint listener.

– Inbound Service name

This name will be the name of the service in the WSDL and affects the code that is generated by Rational Software Architect. By default it is based on the service destination name with InboundService at the end.

Enter `CreditCheckInboundService`.

– Endpoint listener

The endpoint listener defines what mechanism will be used to get Web service requests into the inbound service. Select the endpoint created in 10.7.3, “Create an endpoint listener” on page 335 (SOAPHTTPChannel1).

7. Click **Next**.
8. The final page allows UDDI-specific properties to be specified. Because we are not using UDDI accept the defaults and click **Finish**.

   The default port name for the inbound service is based on the endpoint listener name followed by the phrase InboundPort, so in our case the inbound port name will be SOAPHTTPChannel1InboundPort. Because our clients are calling a port called CreditCheckInboundPort, the clients will be unable to invoke the service. This can be rectified with the following steps.
9. From the Inbound service listing page click **CreditCheckInboundService**.
10. Under Additional Properties click **Inbound Ports**.
11. Click the port named **SOAPHTTPChannel1InboundPort**.
12. The next page allows you to modify the inbound port name. Change the name to **CreditCheckInboundPort** and click **OK**.

HomeDelivery inbound service

Repeat this process to create the inbound service for HomeDelivery.

Table 10-1 HomeDelivery inbound service property values

PROPERTY NAME	PROPERTY VALUE
Template WSDL location	`http://localhost:9080/HomeDeliveryRouter/wsdl/com/ibm/patterns/delivery/HomeDelivery.wsdl`
Service destination name	`http://delivery.patterns.ibm.com:HomeDeliveryService:HomeDelivery`
Endpoint Listener	SOAPHTTPChannel1
Inbound Service Name	HomeDeliveryInboundService
Inbound Port Name	HomeDeliveryInboundPort

10.7.6 Generate and export new WSDL for the services

The enterprise applications need to be modified to point to our service integration bus inbound services rather than directly to the Web service.

To do this, we must create new WSDL files for the Web services, and export these WSDL files from WebSphere Application Server into ZIP files. We can then give the ZIP files to an application developer who can import them into Rational Software Architect and change the enterprise applications client code accordingly.

Perform the following steps to download a ZIP file for each inbound service:

1. In the administrative console, locate the inbound services list. Click **CreditCheckInboundService**.
2. Under Additional Properties click **Publish WSDL files to ZIP file**.
3. Click **CreditCheckInboundService.zip** and save the file to disk.
4. The ZIP file can now be provided to the application developers to regenerate the Web service clients.
5. Repeat these steps to download a ZIP file for HomeDeliveryInboundService .

Schemas: When the inbound and outbound services are created, the WSDL is automatically imported into the SDO repository. Unfortunately any schemas used are not similarly imported and must be imported manually. The ITSOMart application contains no XSD definitions so this is not necessary.

10.7.7 Update the Web service clients to use the bus

This section describes how to regenerate the Web service client code we generated earlier in "Creating an EJB Web service client" on page 317, this time using WSDL generated for the Web service at the bus.

This changes the following:

- Endpoint address

  The endpoint address used by Web service consumers to invoke a Web service provider will change to invoke the inbound service on the bus rather than the provider directly.

- WSDL name space

  By going through the service integration bus, the WSDL definition namespace changes. This requires that we regenerate the Web service client stubs and the service references in our J2EE applications. It does not affect the in-transit SOAP message.

When an inbound service is exported from WebSphere Application Server, it is packaged in a ZIP file. The ZIP file contains four WSDL files. Each WSDL file describes a portion of the service.

Assuming that our sample bus is called ITSOMartBus and the service we are looking at is called CreditCheckInboundService, these files are named:

- ITSOMartBus.CreditCheckInboundServiceBindings.wsdl

  Defines the binding and transport for each operation in the service.

- ITSOMartBus.CreditCheckInboundServicePortTypes.wsdl

  Imports the port type, operations, and messages for the service from an HTTP server.

- ITSOMartBus.CreditCheckInboundServiceService.wsdl

  Defines the service and port for the service.

- ITSOMartBus.CreditCheckInboundServiceNonBound.wsdl

  Imports a port type and defines a non-specific binding and service. Not used in this scenario.

Note: Ensure the WebSphere Application Server V6 unit test environment is started prior to completing these steps. Check this by examining the status of the server in the Servers view of the J2EE perspective.

Import the generated WSDL

The WSDL files generated by WebSphere Application Server for the inbound services have to be imported into your workspace.

To import the CreditCheckInboundService WSDL, perform the following tasks:

1. In the workbench select **File** → **Import.**
2. The import dialog box is shown. Scroll down. select **Zip file,** and click **Next**.
3. In the next panel:
   a. Specify the location of the zip file containing the WSDL by either entering the path into the pull-down box labelled From zip file or by clicking **Browse** and selecting the file form the file dialog box.
   b. In the right panel uncheck the box labelled **ITSOMartBus.CreditCheckInboundServiceNonbound.wsdl**.
   c. Select the project into which to import the WSDL files. Click **Browse** and expand **CreditCheckClient** → **ejbModule** → **META-INF**, select **wsdl**, and click **OK**.
   d. Click **Finish** to import the files.
4. Repeat these steps to import the HomeDeliveryInboundService WSDL into the ClientDelivery project.

Create the namespace mapping files

When you generate a Web service client, Rational Software Architect creates Java class implementations of all the XSD components defined by the Web service. The name of each Java class is derived from the name of the XSD component. The package name of each Java class is derived from the namespace of the XSD component.

The Java package name is based on the host part of the namespace name. For example, an XSD component that belongs to the package:

```
http://www.ibm.com/CreditCheck.wsdl
```

Would be stored in a Java package called:

```
www.ibm.com
```

Note: Certain characters that are valid in a host part of a URI are not valid in a package name, such as a dash (**-**) character. In these cases those characters will be mapped to a different character.

It is common to have multiple namespaces defined with a common host part. Examples include `www.ws-i.org`, `www.w3c.org` and `schemas.xmlsoap.org`. It is possible in these cases that some schemas will define elements with the same name. Without namespace mapping, these elements generate classes with the same name and in the same package. This would result in one of the generated classes overwriting the other. Namespace mapping allows a namespace to be mapped to an arbitrary package name, eliminating this problem.

Namespace mapping can be used to map XSD components from multiple namespaces into a single Java package. This is useful for mapping the namespaces defined in the bus-generated WSDL files to the package name for the code (for, example `com.ibm.patterns.creditCheck`).

There are three namespaces to map:

- The targetNamespace, which is the same for all service definitions generated by the service integration bus.
- The sibusbinding namespaces, which are unique to each Web service with which we are working.
- The other namespaces used by XSD components.

To do this, perform the following tasks:

1. Determine the namespaces defined by the CreditCheck Web service. Open the WSDL file **ITSOMartBus.CreditCheckInboundServiceService.wsdl** and examine the namespace definitions defined in the `<definitions>` tag (Example 10-12).

Example 10-12 Definitions attribute of a service WSDL file

```
<wsdl:definitions
   targetNamespace=

"http://www.ibm.com/websphere/sib/webservices/IBM-PKHOODNode03Node01Cell/ITSOMartBus/Service"
   xmlns:sibusbinding=
      "http://www.ibm.com/websphere/sib/webservices/IBM-PKHOODNode03Cell/ITSOMartBus/
         CreditCheckService/Binding"
   xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
```

2. From this file, we can determine the value of the targetNamespace is:

   ```
   http://www.ibm.com/websphere/sib/webservices/IBM-PKHOODNode03Cell/
   ITSOMartBus/Service
   ```

   We can also determine the value of the sibusbinding namespace is:

   ```
   http://www.ibm.com/websphere/sib/webservices/IBM-PKHOODNode03Cell/ITSOMa
   rtBus/CreditCheckService/Binding
   ```

Notice that these namespaces incorporate the cell and name of the bus.

3. Create a namespace mapping file to map these two namespaces to the Java package `com.ibm.com.patterns`. Perform the following tasks:

   a. Select the **CreditCheckClient** project and go to the **ejbModule/META-INF** folder.

   a. Select **File** → **New** → **Other** from the main menu.

   b. In the New dialog box select **Simple** → **File** then click **Next**.

   c. Set the parent folder to **namespace mappings**. Then enter `nsmappings.properties` in the File name text field and click **Finish**. This will create the file and open and editor for it.

   d. The file is formatted so the namespace comes first, followed by equals, then the package name, all on one line. Multiple lines can be specified. The namespace mappings will be unique to your system, so you must determine the namespace names yourself by examining ITSOMartBus.CreditCheckInboundServiceService.wsdl. We used the mappings shown in Example 10-13.

Example 10-13 Namespace mapping file for CreditCheck

```
http\://www.ibm.com/websphere/sib/webservices/IBM-PKHOODNode03Cell/ITSOMartBus/Service=com.ibm.
patterns.creditCheck
http\://www.ibm.com/websphere/sib/webservices/IBM-PKHOODNode03Cell/ITSOMartBus/CreditCheckInbou
ndService/Binding=com.ibm.patterns.creditCheck
```

Note: Although one of the namespace mappings shown in Example 10-13 spans multiple lines, it should each be entered on a single line in the namespace mapping file. This also applies to Example 10-12 on page 346.

Also note that the namespace must be modified from using `http://` to using `http\://` in the namespace mapping file.

   e. Save the changes to **nsmapping.properties**.

   f. Next, create an nsmappings.properties file in the ClientDelivery project and perform the same mapping for the HomeDelivery inbound service (See Example 10-14).

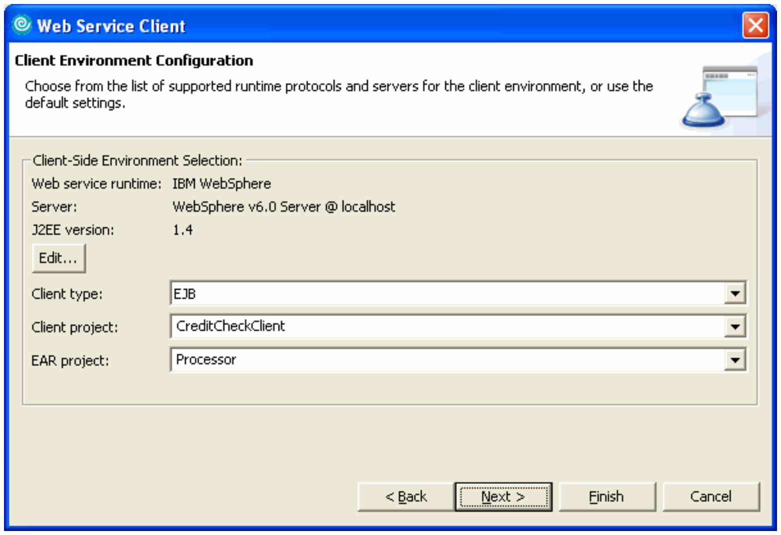
Example 10-14 Namespace mapping file for HomeDelivery

```
http\://www.ibm.com/websphere/sib/webservices/T23734WLZ2Node01Cell/ITSOMartBus/Service=com.ibm.
patterns.delivery
http\://www.ibm.com/websphere/sib/webservices/T23734WLZ2Node01Cell/ITSOMartBus/HomeDeliveryInbo
undService/Binding=com.ibm.patterns.delivery
```

Generate the Web service clients

We are now ready to generate (or regenerate, because ours already exists) the Web service clients for the CreditCheck and HomeDelivery services. Their respective clients will point to the inbound service on the bus rather than directly to the Web service. Perform the following tasks:

1. Select **File** → **New** → **Other**. Then select **Web Services** → **Web Service Client** and click **Next**.
2. We want to create a Java proxy client, so ensure Client proxy type is set to **Java proxy** then click **Next**.
3. Use the browse button to locate **CreditCheckClient/ejbModule/META-INF/wsdl/ITSOMartBus.CreditCheckServiceService.wsdl**. Click **OK**, then **Next**.
4. On the page shown in Figure 10-38, specify the information about how the Web service client will be generated.



Client Environment Configuration

Choose from the list of supported runtime protocols and servers for the client environment, or use the default settings.

Client-Side Environment Selection:

Web service runtime: IBM WebSphere

Server: WebSphere v6.0 Server @ localhost

J2EE version: 1.4

Client type: EJB

Client project: CreditCheckClient

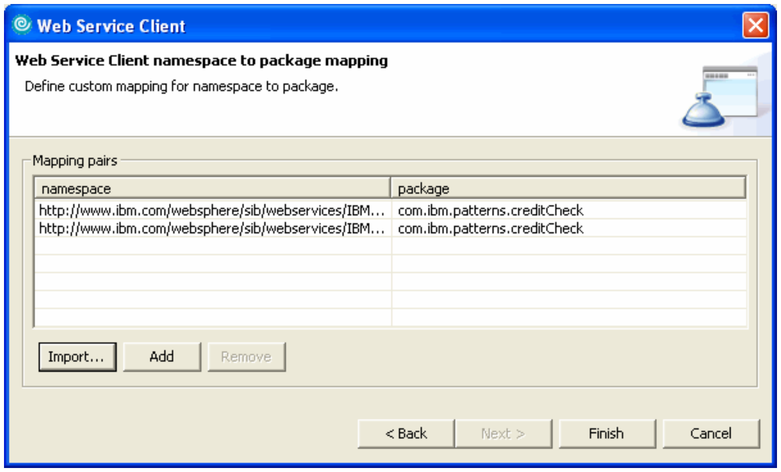
EAR project: Processor

Figure 10-38 Specify Web service client type

   – Client type specifies the type of the Web service client to generate. Select **EJB**. The other options are Web, Application client, and Java.
   – Client Project specifies the project where the Web service client will be generated. Select **CreditCheckClient**.
   – EAR Project specifies the EAR project which the Web service client will be associated with. Select **Processor**.

   Click **Next** to move to the next page.

5. The next page allows security information to be specified. It also has a check box labelled **Define custom mappings for namespace to package**. Select this check box then click **Next**.
6. Import the namespace mappings file. Click **Import**, expand **CreditCheckClient** to find the **namespace.mapping** folder, highlight **nsmappings.properties**, and click **OK**. This populates the Mapping pairs table with the relevant information shown in Figure 10-39.


Web Service Client
Web Service Client namespace to package mapping
Define custom mapping for namespace to package.
Mapping pairs

namespace	package
http://www.ibm.com/websphere/sib/webservices/IBM...	com.ibm.patterns.creditCheck
http://www.ibm.com/websphere/sib/webservices/IBM...	com.ibm.patterns.creditCheck

Import...
Add
Remove
< Back
Next >
Finish
Cancel

Figure 10-39 Namespace mappings

7. Click **Finish** and the Web service client is generated. Acknowledge any warning messages you receive during the Web service client generation.
8. Repeat these steps to regenerate the client for the HomeDelivery Web service. See Table 10-2.

Table 10-2 HomeDelivery Web service

Proper Name	Property Value
WSDL Location	`ClientDelivery/ejbModule/META-INF/wsdl/ITSOMartBus.HomeDeliveryServiceService.wsdl`
Client Project	ClientDelivery
EAR Project	Processor
Namespace Mappings File	`Expand ClientDelivery to find the namespace.mapping folder, select nsmappings.properties.`

Test the application

Now that we have configured the Web service client to go through the bus, it is time to test it.

By regenerating the client, the generated proxy is reconfigured to point to the new inbound service. Therefore no further changes need to be performed to the ITSOMart application as it uses the proxy to locate the Web service. You can test with the Web Services Explorer (10.5.3, "Testing with the Web Services Explorer" on page 314. Note that without the mediations in the bus, testing the ClientDelivery will always return a home delivery account number.

10.7.8 Configure the router mediation

The following steps take you through the installation and configuration process for the router mediation created in 10.6.1, "Create the router mediation" on page 324.

Define the mediation

1. Install the mediation application, RoutingMediationsEJBEAR.
2. Locate the bus definition by selecting **Service integration** → **Buses**. Click the bus name (**ITSOMartBus**) to open it.
3. Under **Additional Properties,** click **Mediations**.
4. Click **New**.
5. Enter the following values:
   - Mediation name: `DeliveryRequestMediation`
   - Handler list name: `RoutingHandler`

     The handler list name must match the handler list name defined in the EJB deployment descriptor.
6. Add the following context properties in Table 10-3 to the mediation.

Table 10-3 :DeliveryRequestMediation context properties

Name	Data Type	Value
ALL	String	All
BusinessDelivery	String	BusinessDelivery
type	String	type
DeliveryResponseDestination	String	DeliveryResponseDestination
HomeDelivery	String	HomeDelivery

Name	Data Type	Value
businessDestination	String	http://delivery.patterns.ibm.com:BusinessDeliveryService:BusinessDelivery

Click **OK**.

Mediate the destination

To mediate the destination, perform the following steps:

1. Locate the bus definition by selecting **Service integration** → **Buses**. Click the bus name (**ITSOMartBus**) to open it.
2. Under **Additional Properties** click **Destinations**.
3. In the list of destinations, check the box to the left the port name. In this case, the Web service port is:

   `http://delivery.patterns.ibm.com:HomeDeliveryService:HomeDelivery.`

   Click the **Mediate** button.
4. In the next screen, select DeliveryRequestMediation as the mediation to apply to the destination.

   Click **Next**.
5. In the next screen, select the bus and click **Next**.
6. Click **Finish**.

Save your changes.

10.7.9 Configure the aggregator mediation

The following steps take you through the installation and configuration process for the aggregator mediation created in 10.6.2, “Create the Aggregator mediation” on page 328.

Create the queues

Two queue destinations on the bus are used to aggregate the responses from multiple services.

1. From the bus details page for ITSOMartBus under **Additional Properties** click **Destinations**.
2. Click **New**.
3. For the destination type, accept the default of **Queue** and click **Next**.

4. The first page of the wizard, asks for an identifier and description to be entered. The identifier is the queue name. Enter `DeliveryResponseDestination` and click **Next**.
5. The next page allows you to specify to which bus member to assign the destination. There is only one bus member in our scenario so accept the default and click **Next**.
6. The final page is just a summary, click **Finish** and the destination is created.
7. Repeat this process to create a queue destination called `TempDestination`.

Define the mediation

The following steps take you through the configuration process for the mediation application.

1. Install the mediation application, AggregatorEJBEAR.ear.
2. Locate the bus definition by selecting **Service integration** → **Buses**. Click the bus name (ITSOMartBus) to open it.
3. Under Additional Properties click **Mediations**.
4. Click **New**.
5. Enter the following values:
   - Mediation name: DeliveryResponseMediation
   - Handler list name: AggregatorHandler

     The handler list name must match the handler list name defined in the EJB deployment descriptor.
6. Add the following context properties in Table 10-4 to the mediation.

Table 10-4 Aggregator mediation values

Name	Data Type	Value
InboundServiceClientResponseDest	String	*<your_node>*.server1.SOAPHTTPChannel1Reply
TempDataDestination	String	TempDestination
getAccountReturn	String	getAccountReturn

Click **OK**.

Mediate the destination

To mediate the destination, perform the following steps:

1. Locate the bus definition by selecting **Service integration** → **Buses**. Click the bus name (**ITSOMartBus**) to open it.

2. Under Additional Properties click **Destinations**.
3. In the list of destinations, check the box to the left the queue name. In this case, the queue is **DeliveryResponseDestination**.

   Click the **Mediate** button.
4. In the next screen, select **DeliveryResponseMediation** as the mediation to apply to the destination.

   Click **Next**.
5. In the next screen, select the bus and click **Next**.
6. Click **Finish**.

Save your changes.

10.8 System management for Web services

Planning for systems management in a Web services environment is similar to the planning needed for any other distributed system, but we face some unique challenges in Web services technology.

Web services are based on plain text XML messaging that could potentially be vulnerable to interception. In addition, Web services potentially allow transactions beyond firewalls and enable external entities to invoke internal applications or access sensitive information.

Another problem is that since there is no confirmed delivery (at least not explicitly built into the Web services specification) what happens if we do not get a response to a particular request? How do we know if the Web service provider is not working, just slow, or just did not receive our request?

We have no explicit way of determining if a service on a remote server (not under our control) is working. This becomes an availability issue on our local server. At the application level, we need a way to respond to unavailable systems.

Some other areas of system management to consider with Web services include:

- Firewall considerations

  From the perspective of the firewall, an RPC router is just another servlet that is accessed through HTTP or HTTPS.
- Load-balancing considerations
  - The same considerations and techniques as for normal servlets apply, with one exception. By default, there is no hot failover for session type Web services. If failover is a requirement, it should be handled on the next

layer; for example, in a servlet acting as Web service implementation, or by using a redundantly configured EJB session facade.

  - A stateless request-style Web service instance can be pooled and load-balanced because it is a Java object in the JVM of the application server.
  - Failover is also supported for request-style Web services.
  - If session style is used, scalability is best. However, the Web service implementation class must be serializable and small.
  - If application style is used, load balancing is not an issue, because there is only one instance whose lifetime is identical to the one of the application server hosting it.

- Other Quality of Service considerations
  - Clearly define and document achievable QoS standards or requirements.
  - Service providers should try to make key measurements available online.
  - Use existing standards or proposed specifications as design guidelines.

10.8.1 Security considerations for Web services

Web services security is one of the bigger challenges in implementing Web services-based systems. With WebSphere Application Server V6, you have the following options for securing Web services:

- Message-level security using Web services security (WS-Security)
- Transport-level security using TLS/SSL

Figure 10-40 provides an overview of Web services security.

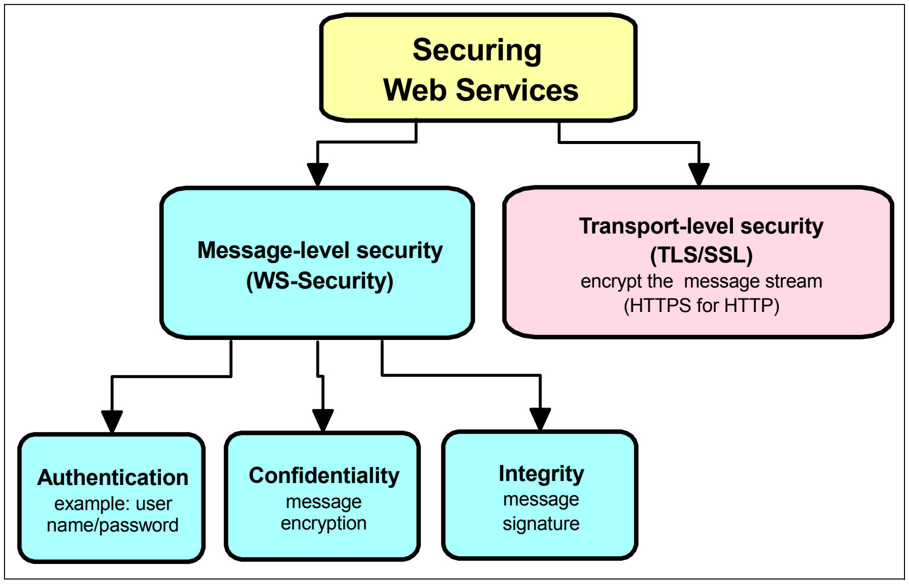

Securing Web Services
Message-level security (WS-Security)
Transport-level security (TLS/SSL) encrypt the message stream (HTTPS for HTTP)
Authentication example: user name/password
Confidentiality message encryption
Integrity message signature

Figure 10-40 Securing Web services

For a full discussion of Web services security and implementation examples, see *WebSphere Version 6 Web Services Handbook Development and Deployment,* SG24-6461.

WS-Security

The WS-Security specification defines message-level security that provides for message content integrity and confidentiality. Unlike SSL, WS-Security can provide end-to-end message-level security. This means that the message security can be protected even if the message goes through multiple services, called intermediaries. WS-Security is independent of the transport layer protocol and can be used for any Web service binding (for example, HTTP, SOAP, RMI).

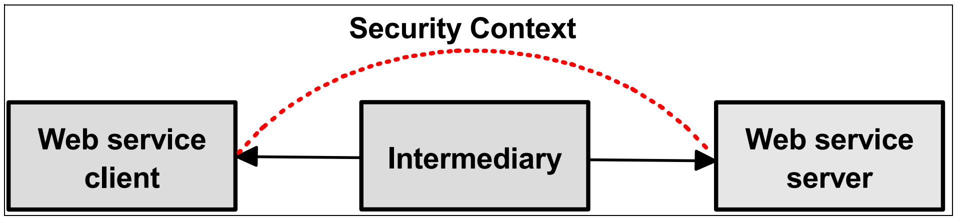


Figure 10-41 End-to-end message-level security

WebSphere Application Server Version 6.0 supports the WS-Security 2004 specification and two token profiles (UsernameToken 1.0, X.509 Certificate

Token 1.0). The level of the security specification supported in WebSphere Application Server Version 6 is above these specifications, with the described changes in the OASIS erratas:

- Web Services Security: SOAP Message: Errata 1.0

  http://www.oasis-open.org/committees/download.php/9292/oasis-200401-wss-soap-message-security-1%200-errata-003.pdf

- Web Services Security: UsernameToken Profile: Errata 1.0

  http://www.oasis-open.org/committees/download.php/9290/oasis-200401-wss-username-token-profile-1.0-errata-003.pdf

- Web Services Security: X.509 Token Profile: Errata 1.0

  http://www.oasis-open.org/committees/download.php/9287/oasis-200401-x509-token-profile-1.0-errata-003.pdf

Here are some simple guidelines as to when WS-Security should be used:

- Multiple parts of message can be secured in different ways.

  You can apply multiple security requirements, such as integrity on the security token (user ID and password) and confidentiality on the SOAP body.

- Intermediaries can be used.

  End-to-end message-level security can be provided through any number of intermediaries.

- Non-HTTP transport protocol is used.

  WS-Security works across multiple transports (also change of transport protocol) and is independent of the underlying transport protocol.

- User authentication is possible.

  Authentication of multiple party identities is possible.

WS-Security represents only one of the layers in a complex, secure Web services solution design. A more general security model is required to cover other security aspects, such as logging and non-repudiation. The definition of those requirements is defined in a common Web services security model framework. For more information, see:

- *Security in a Web Services World: A Proposed Architecture and Roadmap*, proposed by IBM and Microsoft.

  `http://www-106.ibm.com/developerworks/webservices/library/ws-secmap/`

Transport-level security

To secure HTTP, transport-level security can be applied. *Transport-level security* is a well-known and often used mechanism to secure HTTP Internet and intranet communications. Transport-level security is based on Secure Sockets Layer (SSL) or Transport Layer Security (TLS) that runs beneath HTTP.

HTTPS allows client-side and server-side authentication through certificates, which have been either self-signed or signed by a certification agency.

For Web services bound to the HTTP protocol, HTTPS/SSL can be applied in combination with message-level security (WS-Security).

Unlike message-level security, HTTPS encrypts the entire HTTP data packet. There is no option to apply security selectively on certain parts of the message. SSL and TLS provide security features including authentication, data protection, and cryptographic token support for secure HTTP connections.

SOAP/HTTP transport-level security

Although HTTPS does not cover all aspects of a general security framework, it provides a security level regarding party identification and authentication, message integrity, and confidentiality. It does not provide authentication, auditing, and non-repudiation. SSL cannot be applied to other protocols, such as JMS. To run HTTPS, the Web service port address must be in the form https://.

Even with the WS-Security specification, SSL should be considered when thinking about Web services security. Using SSL, a point-to-point security can be achieved.

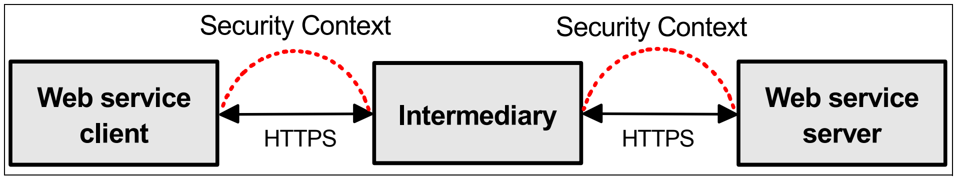


Figure 10-42 Point-to-point security with HTTPS

Here are a few simple guidelines to help decide when transport-level security should be used:

- No intermediaries are used in the Web service environment.

  With intermediaries, the entire message has to be decrypted to access the routing information. This would break the overall security context.

- The transport is only based on HTTP.

  No other transport protocol can be used with HTTPS.

- The Web services client is a stand-alone Java program.

  WS-Security can only be applied to clients that run in a J2EE container (EJB container, Web container, application client container). HTTPS is the only option available for stand-alone clients.

Bus security

The service integration bus provides facilities for secure communication between service requestors and the bus (inbound to the bus), and between the bus and any target Web services (outbound from the bus). Security in the bus can be applied at a number of different levels.

- Web services security (WS-Security) in the bus
- HTTP endpoint listener authentication
- Operation-level authorization
- Using HTTPS with the bus
- Proxy server authentication

For more details on how to implement the above security levels in the bus, see Chapter 22 of *WebSphere Version 6 Web Services Handbook Development and Deployment,* SG24-6461.

10.8.2 Web Services Gateway

If you are deploying the application using Network Deployment, you have the option to deploy your Web services through IBM's Web Services Gateway. This option is not available for standalone server environments.

A gateway service is the Web interface for an underlying service (the target service) that is either provided internally (hosted so as to be directly available at a service destination), or provided externally (as an external Web service). The gateway service is made available at a different location to the target service, so you can replace or relocate the target service without changing the details for the associated gateway service. You can also have more than one target service (that is, more than one implementation of the same logical service) for each gateway service.

It is important to note that IBM's Web Services Gateway is now integrated within the bus. It is therefore possible to control and monitor access to your gateway services in the following ways:

- You can control which groups of users can access a particular gateway service by making the service available only through a particular gateway instance.
- You can associate JAX-RPC handler lists with ports, so that the handlers can monitor activity at the port, and take appropriate action depending upon the sender and content of each message that passes through the port.
- You can set the level of security to be applied to messages (the WS-Security binding). The security level can be set independently for request and response messages.

10.9 More information

For more information, see:

- *WebSphere Version 6 Web Services Handbook Development and Deployment,* SG24-6461
- *Understand Enterprise Service Bus scenarios and solutions in Service-Oriented Architecture, Part 1*, available at:

  `http://www.ibm.com/developerworks/webservices/library/ws-esbscen/`
- *WebSphere Application Server Version 6.0 Information Center*, available at:

  `http://publib.boulder.ibm.com/infocenter/ws60help/index.jsp`
- The XML Signature workgroup home page is available at:

  `http://www.w3.org/Signature/`
- The XML Encryption workgroup home page is available at:

  `http://www.w3.org/Encryption/`
- The WS-Security specification 1.0 is available at:

  `http://www.ibm.com/developerworks/library/ws-secure/`

- *Security in a Web Services World: A Proposed Architecture and Roadmap*, proposed by IBM and Microsoft.

  `http://www-106.ibm.com/developerworks/webservices/library/ws-secmap/`

- OASIS WS-Security 1.0 and token profiles is available at:

  `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss`

- Web Services Security: SOAP Message: Errata 1.0

  `http://www.oasis-open.org/committees/download.php/9292/oasis-200401-wss-soap-message-security-1%200-errata-003.pdf`

- Web Services Security: UsernameToken Profile: Errata 1.0

  `http://www.oasis-open.org/committees/download.php/9290/oasis-200401-wss-username-token-profile-1.0-errata-003.pdf`

- Web Services Security: X.509 Token Profile: Errata 1.0

  `http://www.oasis-open.org/committees/download.php/9287/oasis-200401-x509-token-profile-1.0-errata-003.pdf`

There are several commercial and non-commercial information sources that cover more general subjects, such as SSL encoding and the HTTPS protocol.

11

JMS scenario

In this chapter, we give an overview of the use of JMS in an SOA environment. The ITSOMart sample application has been extended to use JMS to place a message on a queue. That message is handled by a message-driven bean that starts the process of sending the message to a mail server for processing.

This chapter discusses how the design guidelines outlined in Chapter 7, “Application and system design guidelines” on page 139 were applied to the sample scenario. It also describe the process used to develop the messaging application as a way of illustrating development guidelines for JMS. The sample has been deployed on WebSphere Application Server V6 and we use this to illustrate runtime configuration tasks that must be done. Lastly, we discuss system management considerations for using JMS.

11.1 Architectural overview model

The following diagram shows the ITSOMart application. The portion highlighted, the ESB, is the piece we discuss in this chapter. When a customer registers with ITSOMart, the application uses the Mail Service Proxy to send an e-mail to the customer. The proxy builds a JMS message, and sends it to MailService through the ESB. For the rest of the chapter, we focus only on these components.

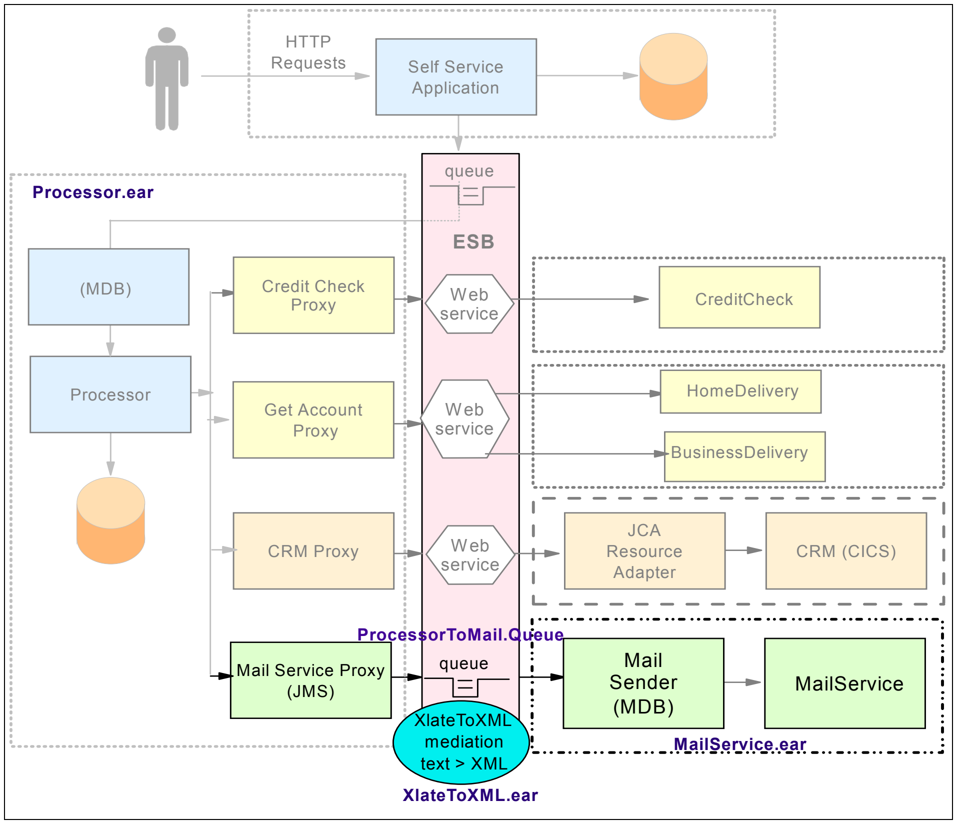


Figure 11-1 Architectural overview model: JMS

The components that participate in the JMS scenario are:

- Mail Service Proxy

  This component is used by the Processor application to deliver a JMS message that holds the information to send to the customer.

- Enterprise Service Bus

  The ESB is implemented using the service integration bus. The bus provides the transport mechanism for the default messaging provider. Messages sent to MailService are placed on a queue defined to the bus. A mediation will be added to transform the format of the message payload from text to XML. This mediation activity is transparent to the applications. If, in the future, a different mail service is used and it requires something other than XML, the only change required is in the mediation activity on the bus.
- Mail Sender

  This component uses an EJB message-driven bean to receive the message from the queue and the Java Mail API to send the e-mail to MailService.
- MailService

  Our SMTP server, that sends the e-mail to the customer.
- XlateToXML

  This component is a mediation application. When a message arrives at the bus, this application transforms the message. The Mail Sender MDB receives the transformed message.

This scenario illustrates the following application pattern using messaging technology:

- Directly Integrated Single Channel application pattern

  ITSOMart places a message on a queue. Mediation takes place to transform the message into a new format. The Mail service retrieves the message and sends an e-mail to the user.

11.2 System design overview

In this scenario, the ITSOMart application will be connected to an enterprise-tier application (MailService) using JMS.

11.2.1 Component model

The ITSOMart will send a notification message to the customer when the registration process is complete. The application does this by generating a JMS message for MailService. The message contains the e-mail text to be sent. MailService is expecting XML format so a mediation is added to the bus that transforms the message to the proper format.

Figure 11-2 on page 364 shows the component model for the scenario.

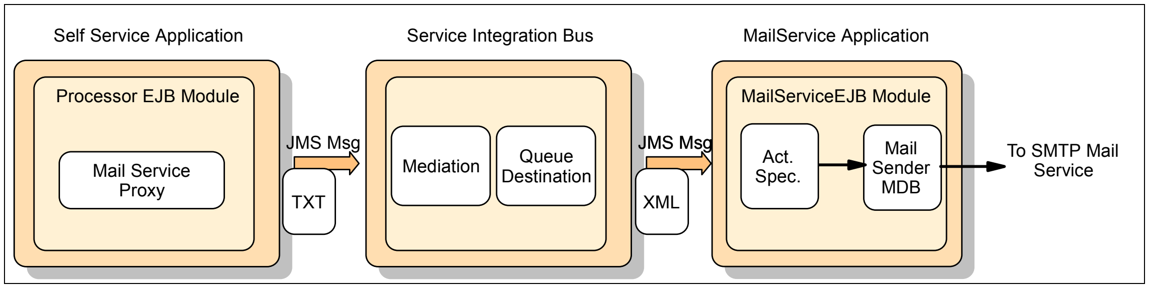


Figure 11-2 Using mediation in the bus

11.2.2 Object model

The sequence diagram in Figure 11-3 shows how the Mail Service proxy sends the e-mail message to MailService using JMS. Note that the application is unaware that the ultimate receiver of the message requires XML format. The use of the ESB makes the format transformation transparent to the sender.

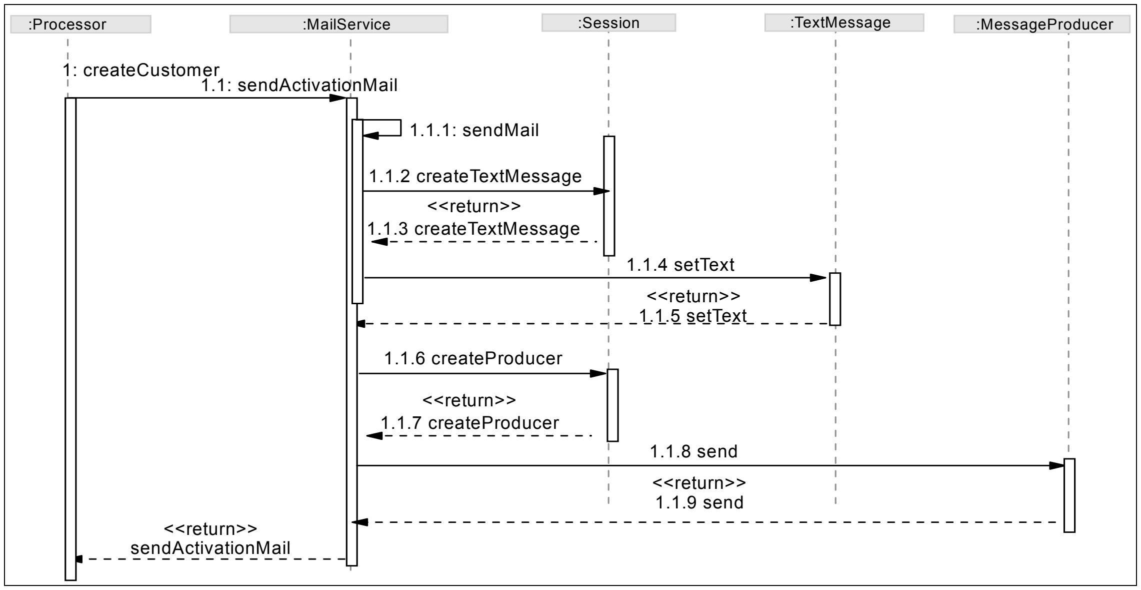


Figure 11-3 Sequence diagram for sending e-mail message to MailService

1) The createCustomer() method in the Processor class is invoked.

1.1) createCustomer() invokes the sendActivationMail() method in the MailService class, passing the customer information from the registration.

1.1.1)	sendActivationMail() invokes sendMail(), passing the text of the message and e-mail address.
1.1.2)	sendMail() creates a connection to the message service provider, a session for sending and receiving messages, and an empty text message.
1.1.4)	sendMail() sets the text for the message.
1.1.6)	sendMail() creates a message producer to create a message.
1.1.8)	sendMail() sends the message.

11.3 Applying the design guidelines

In the ITSOMart application, the following applies:

- The Mail Service Proxy acts as the JMS client.
- A JMS message of type *text* is used to send the e-mail information to the MailService.
- The default messaging provider in WebSphere Application Server V6 is used as the JMS provider.

11.3.1 Point-to-point messaging model

The messaging model used for this example is the JMS point-to-point model. In this model, a client typically sends a message to a specific queue and receives a message from a specific queue. The ITSOMart application connects to a single messaging application at the back-end and therefore point-to-point is the most suitable model.

The front-end application uses the send-and-forget messaging pattern. The back-end application uses the message consumer pattern to process requests and the send-and-forget pattern to deliver the reply.

For further reference on JMS point-to-point and publish/subscribe, refer to the JMS Specification available at:

`http://java.sun.com/products/jms/docs.html`

11.3.2 JMS resource lookups using JNDI

The sample application uses a utility class named JNDILookup to look up the JMS resources. JMSLookup adds some useful functionality, such as caching the JMS objects references. See Example 11-1 on page 366.

Example 11-1 JNDI lookup of JMS Connection Factory

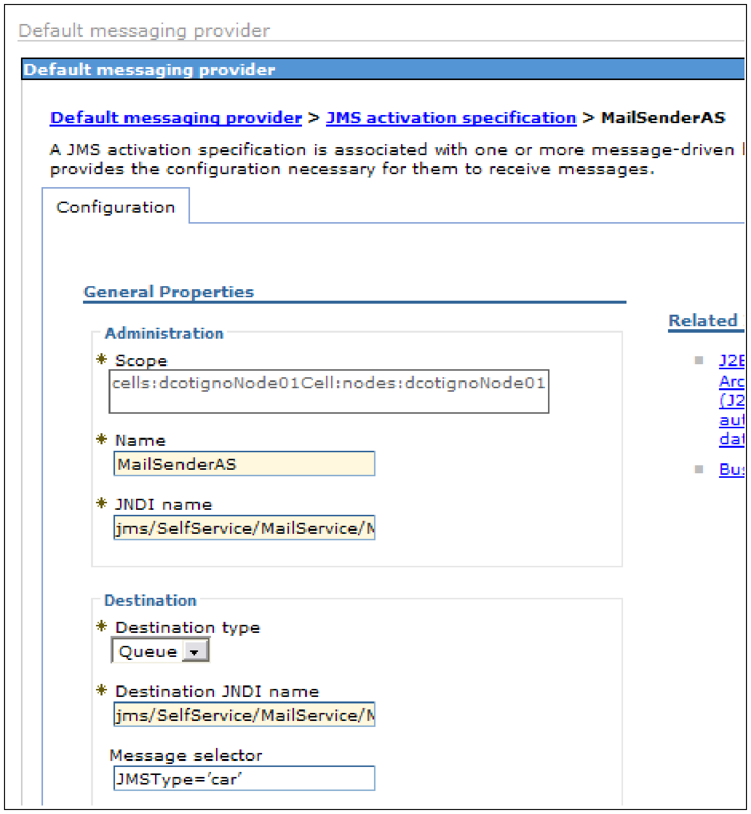
```
/*
 * Created on Apr 17, 2005
*/
package com.ibm.patterns.util;

import java.util.HashMap;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

/**
 * @author sandyg
 *
 * TODO To change the template for this generated type comment go to
 * Window - Preferences - Java - Code Style - Code Templates
 */
public class JNDILookup {

   private static JNDILookup instance = new JNDILookup();
   private Context ic = null;
   private HashMap objectNames=new HashMap();

   public static JNDILookup singleton() {
      return instance;
   }

   private void getInitialContext() throws NamingException {
         ic = (Context)new InitialContext().lookup("java:comp/env");
   }

   public Object getJndiObject(String jndiName) throws NamingException{

      //Check if object exists in HashMap if so return
      Object retreived=null;
      if (objectNames.containsKey(jndiName)) {
         retreived = objectNames.get(jndiName);
         return retreived;
      }
//The object was not found, do a lookup
      retreived = findObject(jndiName);
      return retreived;
   }

   private Object findObject(String jndiName) throws NamingException{
      Object retreived=null;
      //hmm.. object was not found, this is the first run we need to lookup
and put in HashMap
```

```
      if (ic==null) {
         //System.out.println("ic was null, it was probably the first time
this was called");
         getInitialContext();
      }
      retreived=ic.lookup(jndiName);
      objectNames.put(jndiName,retreived);
      return retreived;
   }

}
```

The JMS resource reference is bound to the JMS resource JNDI name during application assembly or deployment.

11.3.3 Message selectors

An activation specification defined at runtime triggers the message-driven bean when a message arrives on the queue. When defining the activation specification, you can limit the messages that trigger the MDB by specifying a message selector. See Figure 11-4 on page 368.


Default messaging provider

Default messaging provider

Default messaging provider > JMS activation specification > MailSenderAS

A JMS activation specification is associated with one or more message-driven provides the configuration necessary for them to receive messages.

Administration

* Scope
cells:dcotignoNode01Cell:nodes:dcotignoNode01

* Name
MailSenderAS

* JNDI name
jms/SelfService/MailService/M

Destination

* Destination type
Queue

* Destination JNDI name
jms/SelfService/MailService/M

Message selector
JMSType='car'

Figure 11-4 Specifing a message selector for a message-driven dean

The selector string can refer to fields in the JMS message header and fields in the message properties. Message selectors cannot reference message body values.

11.3.4 Message time-to-live

The default messaging provider provides a mechanism through JMS to set the *time-to-live* of a message. The code in Example 11-2 shows how we can set time-to-live within this scenario to 20 seconds.

Example 11-2 Setting the JMS message time-to-live

```
producer.send(reqMsg,DeliveryMode.NON_PERSISTENT,0,20000);
```

Setting the message time-to-live is an important best practice to avoid large numbers of unconsumed messages on queues.

JMSExpiration

JMSExpiration is the sum of the time-to-live and current GMT. If, however, time-to-live is set to zero, then JMSExpiration is set to zero.

If it is determined that GMT is now greater than the JMSExpiration value, then the message should be destroyed by the messaging provider. JMS, however, does not automatically notify the messaging provider. It is always good to check the JMSExpiration manually in the code.

11.3.5 Persistent versus non-persistent messages

A persistent or durable message is an obvious performance issue. However, if the operation is too critical to the business needs, then it becomes a necessary choice to ensure or guarantee delivery.

`PERSISTENT` is the default setting. We can explicitly set the use of non-persistent messages. Alternatively, it can be set within the send method provided by the MessageProducer, as shown in Example 11-3.

Example 11-3 Setting JMS message delivery mode

```
producer.send(reqMsg,DeliveryMode.NON_PERSISTENT,0,20000);
```

11.3.6 Mediation

In this scenario, the XlateToXML mediation will transform the message from text to XML. The destination is the queue destination used by the Mail Service Proxy.

11.4 Development guidelines for JMS

In this section we cover the basics of creating a JMS application from Java coding steps to the involvement of various tools that can aid development. The aim is to provide a pointer and not necessarily complete walkthroughs of the tools.

11.4.1 JMS development

Rational Application Developer also provides a WebSphere Application Server V6 test environment. You need to create the required JMS resources in the test environment.

11.4.2 Creating a JMS client application

In this section we consider the steps necessary to add JMS connectivity to your application. To create the Java application, perform the following steps:

1. Configure the JMS provider and destination or destinations in your Integrated Development Environment (IDE). In our environment, we used Rational Software Architect and the WebSphere Application Server V6 unit test environment, which includes the default messaging provider.
2. Get an instance of JMS javax.jms.ConnectionFactory, usually through a JNDI lookup.
3. Get an instance of JMS Connection from the ConnectionFactory.
4. Start the JMS Connection.
5. Get a JMS Session from the Connection object.
6. Using the Session object, create either a producer (QueueReceiver) or consumer (QueueSender) on a specified destination.
7. Use this producer or consumer to access the Destination.
8. Close the message consumer, session, and connection. Closing the connection will close the session and the message producers and consumers associated with it. See Example 11-4.

Example 11-4 Sending a JMS text message

```
Connection con = qcf.createConnection();
Session session = con.createSession(false, Session.AUTO_ACKNOWLEDGE);
TextMessage msg = session.createTextMessage();
msg.setText(message);
MessageProdcer producer = session.createProducer(queue);
producer.send(msg);
producer.close();
session.close();
con.close();
```

Tip: Remember to manage the JMS resources (opening and closing) properly and also to handle timeout properly if using EJBs as message consumers.

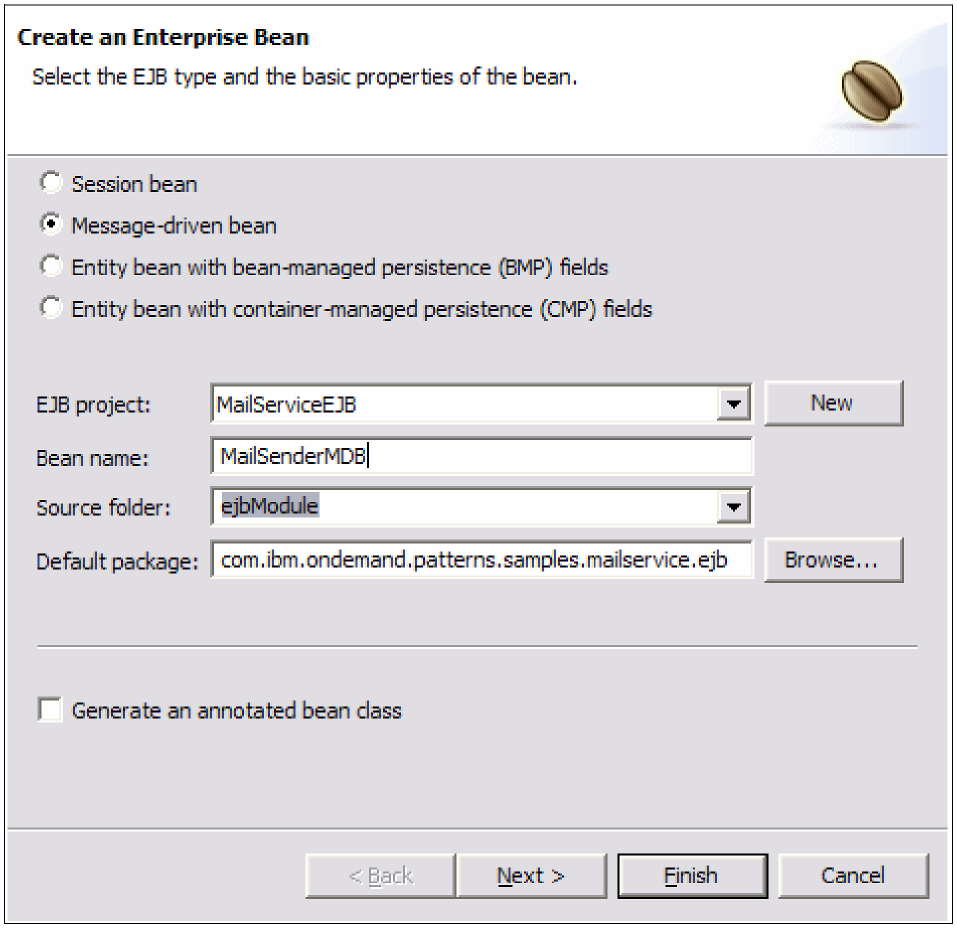
11.4.3 Creating a message-driven bean

To create a message-driven bean, perform the following steps:

1. Create the message-driven bean definition and class.
2. Configure the JCA activation specification.
3. Create the message-driven bean implementation.

Creating the message-driven definition and class

To create the message-driven bean, launch and follow the EJB Create wizard in Figure 11-5.


Create an Enterprise Bean

Select the EJB type and the basic properties of the bean.

- ( ) Session bean
- (●) Message-driven bean
- ( ) Entity bean with bean-managed persistence (BMP) fields
- ( ) Entity bean with container-managed persistence (CMP) fields

EJB project: MailServiceEJB

Bean name: MailSenderMDB

Source folder: ejbModule

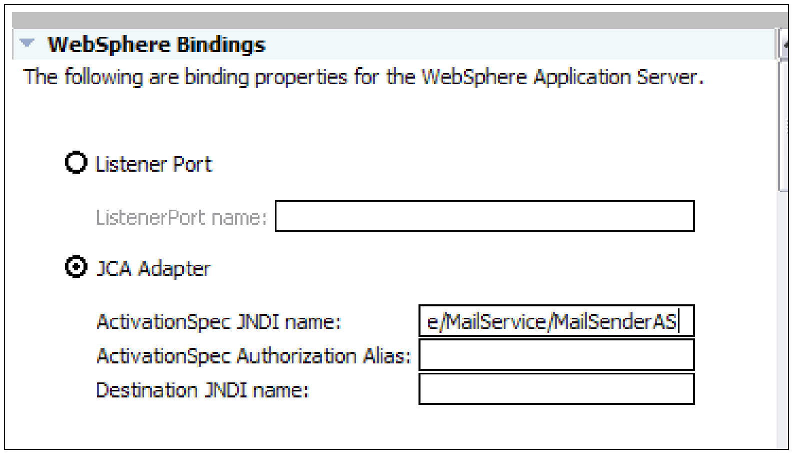
Default package: com.ibm.ondemand.patterns.samples.mailservice.ejb

[ ] Generate an annotated bean class

Figure 11-5 Creating a message-driven bean

Configuring the JCA activation specification

After you create the message-driven bean definition, you can use the EJB Deployment Descriptor’s editor to specify the JNDI name of the ActivationSpec bean. Select your message-driven bean from the **Bean** tab, then specify the JNDI name, as in Figure 11-6.


WebSphere Bindings

The following are binding properties for the WebSphere Application Server.

○ Listener Port

ListenerPort name:

[✓ JCA Adapter]

ActivationSpec JNDI name: e/MailService/MailSenderAS

ActivationSpec Authorization Alias:

Destination JNDI name:

Figure 11-6 Specify the JNDI name of the ActivationSpec Bean

Creating the message-driven bean implementation

To create the the message-driven bean implementation, just implement the method onMessage(javax.jms.Message) of the message-driven bean class, as in Example 11-5.

Example 11-5 Implementing the message-driven bean

```
   public void onMessage(javax.jms.Message msg) {
      try {
         if (msg instanceof TextMessage) {
            TextMessage txtMsg = (TextMessage) msg;
            String buffer = txtMsg.getText();
            Document document =
DocumentBuilderFactory.newInstance().newDocumentBuilder().
                  parse(new InputSource(new StringReader(txtMsg.getText())));
            Element mailElement = document.getDocumentElement();
            Element destinationElement =
(Element)mailElement.getElementsByTagName("destination").item(0);
            String to = destinationElement.getAttribute("address");
            Element messageElement =
(Element)mailElement.getElementsByTagName("message").item(0);
            String body = messageElement.getAttribute("body");
            sendMail(to, body);
         } else {
            System.out.println("Wrong format. Mail not sent.");
         }
      } catch (Exception e) {
         System.out.println(e.getMessage());
         e.printStackTrace();
      }
   }
```

11.4.4 Creating a mediation

This section will take you through the following steps to create a mediation:

- Create a mediation handler class
- Define a mediation handler list

Create a mediation handler class

The following steps outline how to create a mediation handler class:

1. Create a new EJB project.
2. Create a mediation handler class by implementing the com.ibm.websphere.sib.mediation.handler.MediationHandler interface, as in Example 11-6.

Example 11-6 TransformMediator medation handler class

```
package com.ibm.patterns.esb.ejb;

import javax.xml.rpc.handler.MessageContext;
import com.ibm.websphere.sib.mediation.handler.MediationHandler;
import com.ibm.websphere.sib.mediation.handler.MessageContextException;

public class TransformMediator implements MediationHandler {

   public boolean handle(MessageContext ctxt) throws MessageContextException {
      ...
   }
}
```

3. Add functional code to the mediation handler that will, for example, transform or route messages. Within a mediation you can:
   - Work with message context.

     The SIMessageContext interface has a superinterface of MessageContext. Methods in MessageContext allow you to manage a set of message properties (name/value pairs) within the message context. This allows handlers in a handler chain to share a processing-related state. You can get and set properties (getProperty, setProperty), view the names of the properties, and remove a property.

– Work with message properties.

  There are two different types of message properties: system properties (including JMS headers, JMSX properties, and JMS_IBM_properties) and user properties.

  Working with message properties allows you to affect how a message is processed downstream. The rule set in the selector field during mediation configuration tests values in the message properties.

  Properties are accessed using the SIMessage interface. You can get properties (getMessageProperty, getUserProperty), set properties (setMessageProperty, setUserProperty), delete and clear properties, and view property names.

– Work with the message header.

  The message header contains fields that affect message routing, priority, reliability, expiration, and so forth. These fields are most often used to change the routing of a message.

  The SIDestinationAddress API gives the mediation access to the bus name and destination. The SIDestinationAddressFactory API can be used to create a new SIDestinationAddress for the message.

– Work with the message payload.

  To work with the contents of the message, you will use the SIMessage and SIMessageContext APIs. The SIMediationSession API gives the mediation access to the bus so that the mediation can send and receive messages.

The code in Example 11-7 shows you the implementation code of the mediation handler class for our sample scenario. This mediation retrieves the payload of the JMS text message as a Java string value and transforms it to XML format. The message is retrieved from the SIMessage instance as a datagraphs. In data graphs representing JMS messages, the root data object contains a property named `data`, and that data object in turn contains a property named `value`.

Example 11-7 Mediation handler class used in out sample scenario

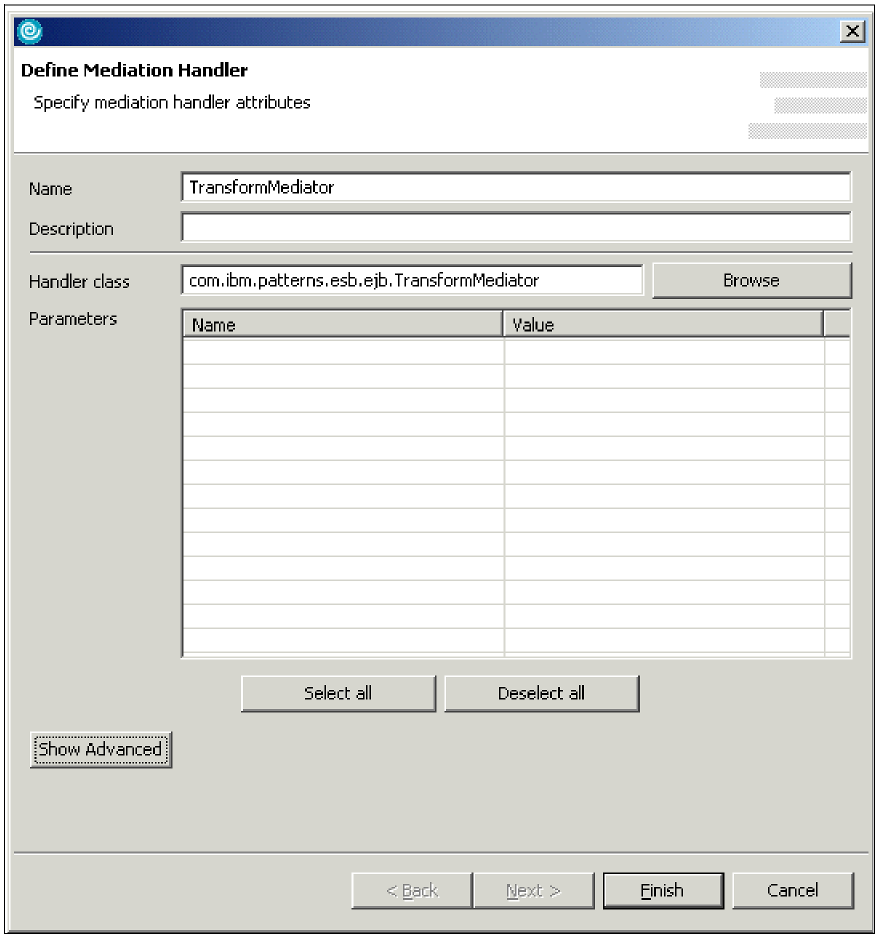
```
public boolean handle(MessageContext ctxt) throws MessageContextException {
   try {
      System.out.println("Transform Mediator...");
      SIMessageContext siMessageCtxt = (SIMessageContext)ctxt;
      SIMediationSession siSession = siMessageCtxt.getSession();
      SIMessage siMessage = siMessageCtxt.getSIMessage();

      String format = siMessage.getFormat();
      if (format.equals("JMS:text")) {
        DataGraph graph = siMessage.getDataGraph();
```

```
            String payload = graph.getRootObject().getString("data/value");
            graph.getRootObject().setString("data/value", formatJMSMessage(payload));
         } else {
            System.out.println("JMS Message is not a JMS Text Message");
         }
         System.out.println("Message transformed");
         return true;
      } catch (Exception e) {
         System.out.println("Error transforming message: "+e.getMessage());
         e.printStackTrace();
      }
      return false;
   }
   private String formatJMSMessage(String txtMessage){
      String newMessage = "";
      int index = txtMessage.indexOf("|");
      if (index!=-1) {
         String destination = txtMessage.substring(0,index);
         String message = "";
         if (index>message.length()) {
            message = txtMessage.substring(index+1);
         }
         try {
            DocumentBuilder docBuilder =
DocumentBuilderFactory.newInstance().newDocumentBuilder();
            Document document = docBuilder.newDocument();
            Element mailElement = document.createElement("mail");
            document.appendChild(mailElement);
            Element destinationElement = document.createElement("destination");
            destinationElement.setAttribute("address",destination);
            mailElement.appendChild(destinationElement);
            Element messageElement = document.createElement("message");
            messageElement.setAttribute("body",message);
            mailElement.appendChild(messageElement);
            newMessage = DOMWriter.nodeToString(document);
            System.out.println("Formated message: "+newMessage);
         } catch (Exception e){
            System.out.println("Error formating message: "+e.getMessage());
            e.printStackTrace();
            return txtMessage;
         }
      }
      return newMessage;
   }
}
```

Define a mediation handler list

To define a mediation handler list, you can use the EJB Deployment Descriptor's editor. Here, we are defining a list with only one mediation handler:

1. Open the EJB Deployment Descriptor.
2. Click the **Mediation Handlers** tab.
3. Click **Add...**
4. Specify a name for the mediation handler and the mediation handler class, as in Figure 11-7.



Name: TransformMediator
Description:
Handler class: com.ibm.patterns.esb.ejb.TransformMediator

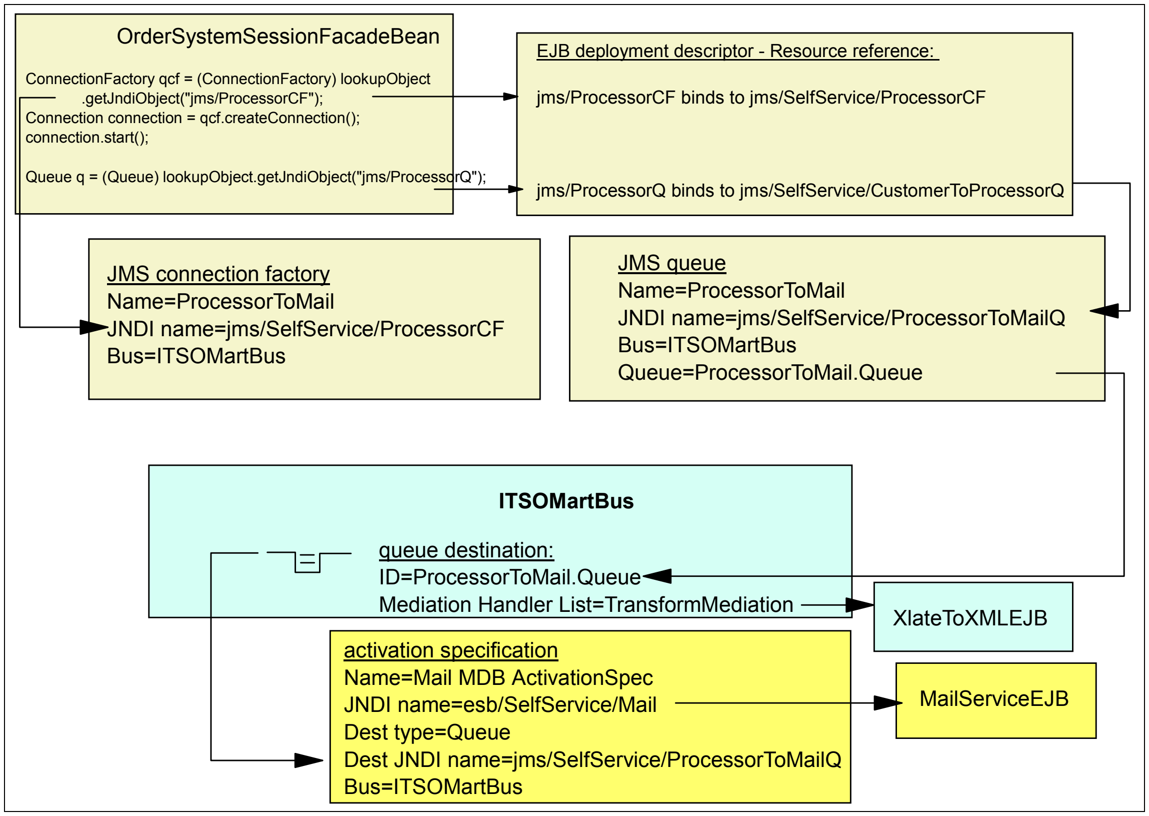
Figure 11-7 Define a mediation handler list

5. Click **Finish**.

11.5 Runtime configuration for JMS

This section discusses the runtime configuration required to run a JMS application in a WebSphere Application Server V6 environment and using the default messaging provider. See Figure 11-8.

Note: This configuration assumes that you have created a service integration bus and added your application server as a bus member. See “Create a service integration bus” on page 463 for information.



OrderSystemSessionFacadeBean
ConnectionFactory qcf = (ConnectionFactory) lookupObject .getJndiObject("jms/ProcessorCF");
Connection connection = qcf.createConnection();
connection.start();
Queue q = (Queue) lookupObject.getJndiObject("jms/ProcessorQ");
EJB deployment descriptor - Resource reference:
jms/ProcessorCF binds to jms/SelfService/ProcessorCF
jms/ProcessorQ binds to jms/SelfService/CustomerToProcessorQ
JMS connection factory
Name=ProcessorToMail
JNDI name=jms/SelfService/ProcessorCF
Bus=ITSOMartBus
JMS queue
Name=ProcessorToMail
JNDI name=jms/SelfService/ProcessorToMailQ
Bus=ITSOMartBus
Queue=ProcessorToMail.Queue
ITSOMartBus
queue destination:
ID=ProcessorToMail.Queue
Mediation Handler List=TransformMediation
XlateToXMLEJB
activation specification
Name=Mail MDB ActivationSpec
JNDI name=esb/SelfService/Mail
Dest type=Queue
Dest JNDI name=jms/SelfService/ProcessorToMailQ
Bus=ITSOMartBus
MailServiceEJB

Figure 11-8 JMS application configuration

11.5.1 Create a queue destination

The queue destination provides the queue for point-to-point messaging. The application places a message on this queue.

1. Locate the bus definition by selecting **Service integration** → **Buses**. Click the bus name (**ITSOMartBus**) to open it.
2. Under **Additional Properties** click **Destinations**.
3. Click **New**.
4. For the destination type, accept the default of **Queue** and click **Next**.
5. The first page of the wizard asks for an identifier and description to be entered. The identifier is the name by which the destination will be exposed to applications. Enter `ProcessorToMail.Queue` and click **Next**.
6. The next page allows you to specify which bus member to which to assign the destination. There is only one bus member in our scenario, so accept the default and click **Next**.
7. The final page is just a summary, click **Finish** and the destination will be created.
8. Save the configuration.
9. Click **Finish**.
10. Save your changes.

11.5.2 Create the JMS connection factory

The next step is to create the JMS connection factories that allow the application to send a JMS message.

1. From the administrative console expand **Resources** → **JMS Providers** and click **Default messaging**.
2. Under Connection Factories click **JMS connection factory.**
3. Click **New**.
4. The next page allows you to specify the properties for the JMS connection factory. Take the defaults for everything but the following:
   – Name

     Enter a value of `ProcessorToMail`.
   – JNDI Name

     This is where the application resource reference will be bound to. Enter a value of `jms/SelfService/ProcessorCF`.
   – Bus name

     Select **ITSOMartBus** in the pull-down.
5. Click **OK** and save the changes.

11.5.3 Create JMS queue

Now we need to create the JMS queue.

1. From the administrative console expand **Resources** → **JMS Providers** and click **Default messaging**.
2. Under **Destinations** click **JMS queue**.
3. Click **New**.
4. The next page allows you to specify the values for the queue.
   - Name

     Enter a value of `ProcessorToMail`.
   - JNDI Name

     This is the name to which the application's message reference will be bound. Enter a value of `jms/SelfService/ProcessorToMailQ`.
   - Bus name

     Select the value of **ITSOMartBus**. This will cause the page to be reloaded with the Queue names list already poplulated.
   - Queue name

     This field specifies the service integration bus queue type destination that will be used to store the messages sent to this JMS queue. Select the value of **ProcessorToMail.Queue**.
5. Click **OK**.
6. Save the configuration.

11.5.4 Create JMS activation specification

Now we need to create the activation specification that will trigger MailServiceEJB when a message arrives on the queue.

1. From the admin console expand **Resources** → **JMS Providers** and click **Default messaging**.
2. Under **Activation Specifications** click **JMS activation specification**.
3. Click **New**.
4. The next page allows you to specify the values for the activation specification. Most of the values can keep their default values. Described here are the ones of most interest.

- Name

  This field is an administrative name used for locating the JMS activation specification. Enter a value of `Mail MDB Activation Spec`.
- JNDI name

  This is where the application's message-driven beans will be bound to for message delivery. Enter a value of `esb/SelfService/Mail`.
- Destination type

  The means the type of the JMS destination (queue or topic) that will be used to deliver messages to the message-driven bean. Accept the default of **Queue**.
- Destination JNDI name

  This field is the location in JNDI of the JMS destination from which to receive messages. Enter a value of `jms/SelfService/ProcessorToMailQ`.
- Bus name

  The name of the bus from which the JMS destination will receive messages. This is not required, but for consistency select **ITSOMartBus.**

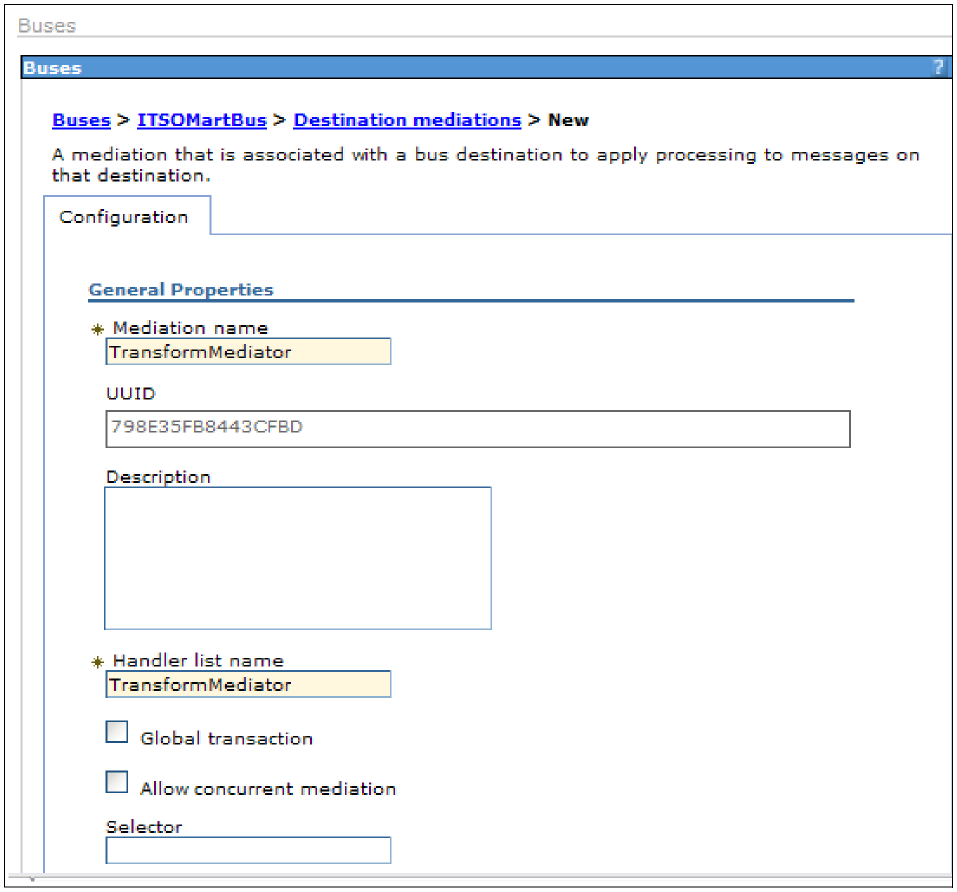
5. Click **OK**.
6. Save the changes

11.5.5 Mediation configuration

The following steps take you through the configuration process to add the mediation.

1. Install the application that will perform the mediation. In our sample, the application is the **XlateToXML** application.
2. Locate the bus definition by selecting **Service integration** → **Buses**. Click the bus name (**ITSOMartBus**) to open it.
3. Under **Additional Properties** click **Mediations**.
4. Click **New**.

See Figure 11-9 on page 381.


Buses

Buses

Buses > ITSOMartBus > Destination mediations > New

A mediation that is associated with a bus destination to apply processing to messages on that destination.

Configuration

General Properties

* Mediation name: TransformMediator

UUID: 798E35FB8443CFBD

Description:

* Handler list name: TransformMediator

☐ Global transaction

☐ Allow concurrent mediation

Selector:

Figure 11-9 Define a mediation

Enter the following values:

– Mediation name: `TransformMediator`
– Handler list name: `TransformMediator`

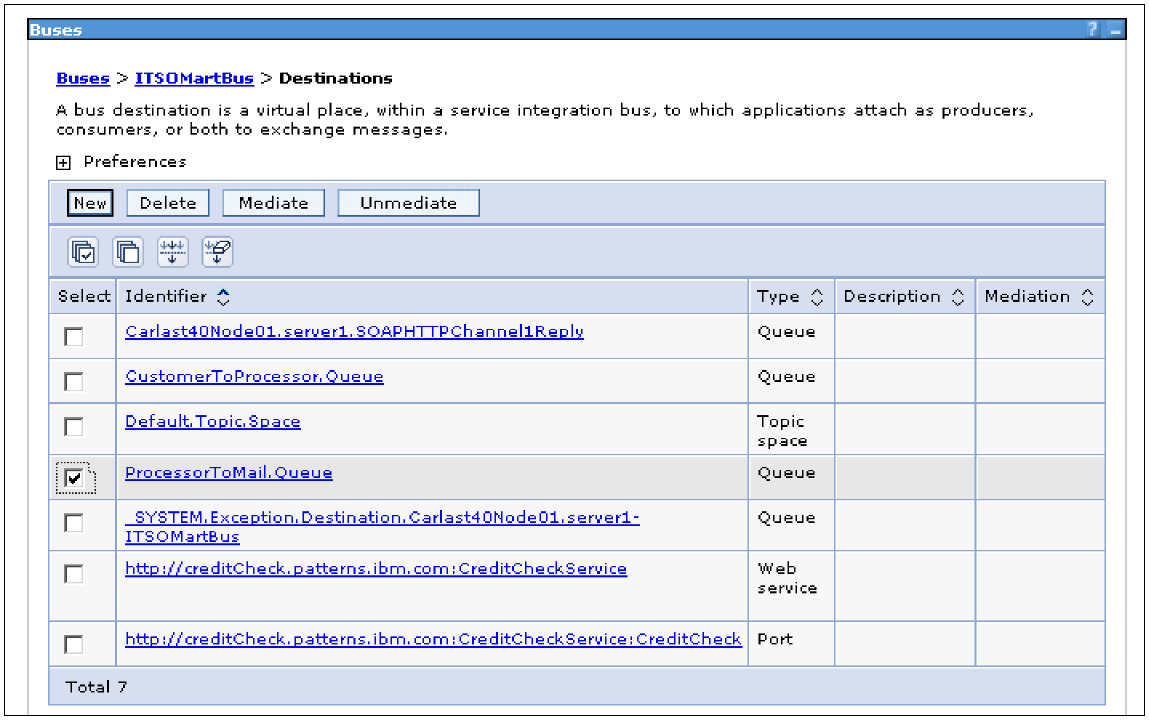
The handler list was defined in the EJB deployment descriptor for the application. See “Define a mediation handler list” on page 376.

Click **OK**.

Mediate the destination

1. Locate the bus definition by selecting **Service integration** → **Buses**. Click the bus name (**ITSOMartBus**) to open it.
2. Under **Additional Properties** click **Destinations**. See Figure 11-10 on page 382.

3. In the list of destinations, check the box to the left the queue name. In this case, the queue is **ProcessorToMail.queue**.


Buses
Buses > ITSOMartBus > Destinations

A bus destination is a virtual place, within a service integration bus, to which applications attach as producers, consumers, or both to exchange messages.

Preferences
New Delete Mediate Unmediate

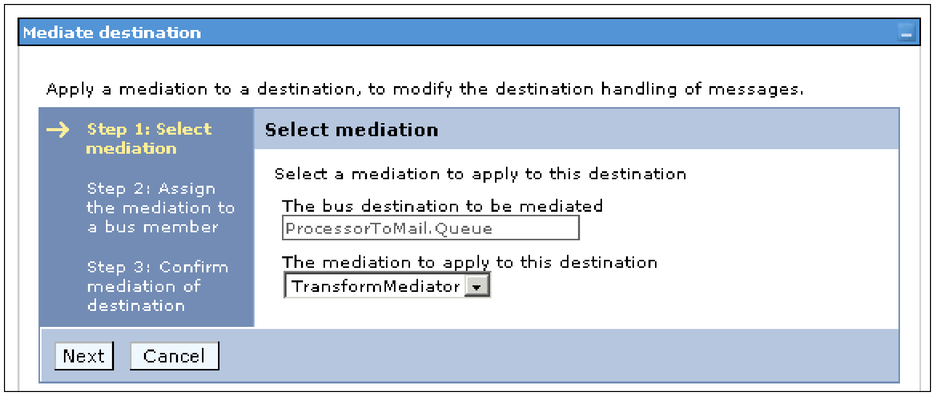
Select	Identifier	Type	Description	Mediation
☐	Carlast40Node01.server1.SOAPHTTPChannel1Reply	Queue		
☐	CustomerToProcessor.Queue	Queue		
☐	Default.Topic.Space	Topic space		
☑	ProcessorToMail.Queue	Queue		
☐	_SYSTEM.Exception.Destination.Carlast40Node01.server1-ITSOMartBus	Queue		
☐	http://creditCheck.patterns.ibm.com:CreditCheckService	Web service		
☐	http://creditCheck.patterns.ibm.com:CreditCheckService:CreditCheck	Port		

Total 7

Figure 11-10 Select a destination to mediate

Click the **Mediate** button.

4. In the next screen (Figure 11-11), select the mediation. Click **Next**.



Apply a mediation to a destination, to modify the destination handling of messages.

→ Step 1: Select mediation
Step 2: Assign the mediation to a bus member
Step 3: Confirm mediation of destination
Select mediation

Select a mediation to apply to this destination

The bus destination to be mediated: ProcessorToMail.Queue

The mediation to apply to this destination: TransformMediator

Figure 11-11 Assign the mediation to the destination

5. In the next screen, select the bus and click **Next**.
6. Click **Finish**.
7. Save your changes.

11.5.6 Test the application

When you use the ITSOMart application to register a customer, an e-mail is generated and sent to the e-mail address you enter in the registration.

View the message on the queue

If there is some question about the format of the message or whether it is arriving on the queue, you can view the message on the queue by performing the following steps:

1. Stop the MailService application.
2. Select **Service integration** → **buses**.
3. Click the bus name.
4. Under **Additional Properties**, click **Destinations**.
5. In the list of destinations, any mediations for the destination are listed in the table. Click the queue destination.
6. Under Message points, click **Queue points**.
7. Select the queue point.
8. Click the **Runtime** tab.
9. Click **Messages**.

Starting and stopping mediation processing

If you think there might be a problem in the mediation application or configuration, you can stop mediation processing by performing the following steps:

1. Select **Service integration** → **buses**.
2. Click the bus name.
3. Under **Additional Properties**, click **Destinations**.
4. In the list of destinations, any mediations for the destination are listed in the table. Click the queue destination.
5. Under Message points, click **Mediation points**.
6. Check the box to the left of the mediation and click the **Stop** button.

The message will be processed, but no mediation will occur.

11.6 System management for JMS

This section takes a brief look at system management issues when using JMS applications.

11.6.1 JMS performance issues

Some issues that play a role in JMS messaging performance are:

- Generic versus specific message structure

  Making the message structure more generic requires more translation and interpretation time at the sender and receiver ends. Making a message too specific reduces flexibility for even small changes in the message structure. Remember to create an error queue for messages that cannot be validated.
- Message persistence

  Using persistent messages requires writing the messages to disk, which takes time, reducing performance.
- Request/reply scenario

  In a request/reply scenario, it is important that the issue of blocking calls is dealt with correctly. Essentially, EJBs should only be used with appropriate request/reply timeouts and retries.
- Message-driven bean

  Minimize the time spent in a message-driven bean processing the message. This will make message-driven bean processing faster. Let the pool of message-driven beans depend on the number of messages that arrive at the queue.
- Optimization with connection

  Start the connection when appropriate so that consumers are ready to consume messages before the producers are started. Also process messages concurrently using a server session pool for the processing of the messages. Close the connection when you are finished consuming messages.

11.6.2 Performance monitoring for mediations

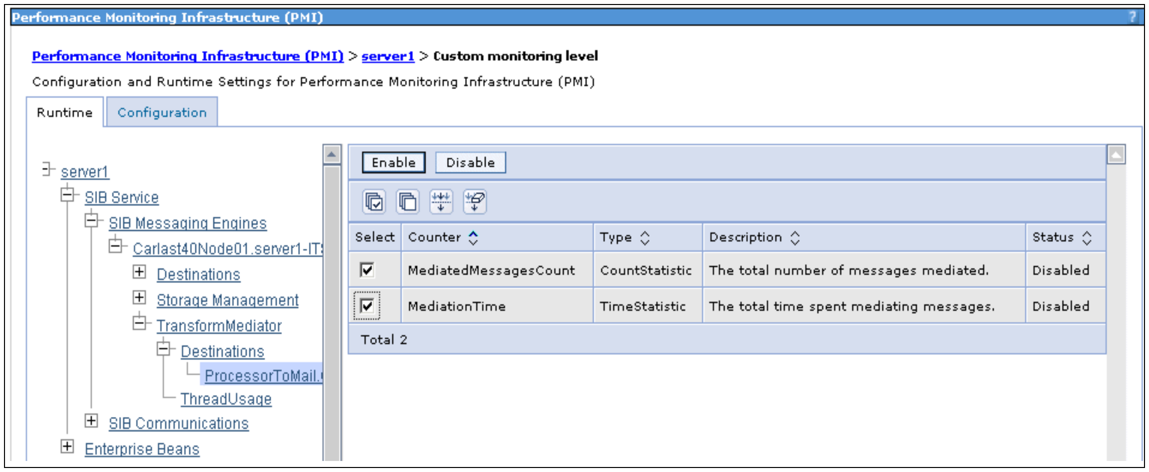
When PMI is enabled, the following information is collected for each mediated destination:

- *Mediation time* is the time taken to perform the mediation.
- *Messages mediated* is the number of messages mediated at this destination.

- *Thread count* is the number of threads in the mediation thread pool performing work for the mediation.

To enable these counters, perform the following tasks:

1. Select **Monitoring and Tuning** → **Performance Monitoring Infrastructure (PMI)**.
2. Click the server name.
3. Click the **Runtime** tab.
4. Click **Custom**.
5. Expand **SIB Service** → **SIB Messaging Engines** → ***<messaging engine>*** → ***<mediation_name>*** → **Destinations**.
6. Click the queue destination. See Figure 11-12.
7. Check the boxes to the left of the counters and click **Enable**.


Performance Monitoring Infrastructure (PMI)
Performance Monitoring Infrastructure (PMI) > server1 > Custom monitoring level
Configuration and Runtime Settings for Performance Monitoring Infrastructure (PMI)
Runtime
Configuration
server1
SIB Service
SIB Messaging Engines
Carlast40Node01.server1-ITS
Destinations
Storage Management
TransformMediator
Destinations
ProcessorToMail.
ThreadUsage
SIB Communications
Enterprise Beans
Enable
Disable

Select	Counter	Type	Description	Status
✓	MediatedMessagesCount	CountStatistic	The total number of messages mediated.	Disabled
✓	MediationTime	TimeStatistic	The total time spent mediating messages.	Disabled
Total 2				

Figure 11-12 Enable the mediation counters

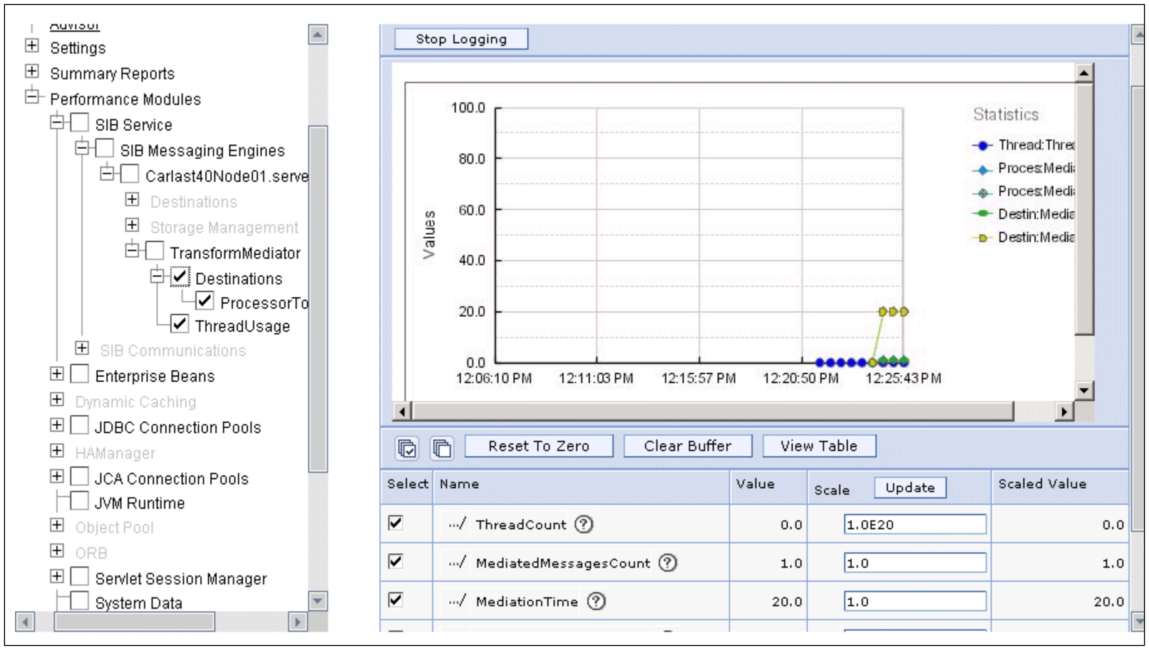
8. Click **Thread Usage** under Destinations.
9. Check the box to the left of **ThreadCount** and click **Enable**.

To view the counters, perform the following steps:

1. Select **Monitoring and Tuning** → **Performance Viewer** → **Current activity**.
2. Click the server name.
3. Expand **Performance Modules** → **SIB Service** → **SIB MessagingS Engines** → ***<message_engine>*** → ***<mediation_name>***.

   See Figure 11-13 on page 386.

4. Check the boxes to the left of the queue **Destinations** and the **ThreadUsage**.



Select	Name	Value	Scale	Scaled Value
✓	ThreadCount	0.0	1.0E20	0.0
✓	MediatedMessagesCount	1.0	1.0	1.0
✓	MediationTime	20.0	1.0	20.0

Figure 11-13 Monitoring mediation counters

11.6.3 Security considerations

The JMS specification does not specify any features for controlling message integrity or authentication. It is expected that the JMS provider will provide these services. Security is considered to be a JMS provider-specific feature that is configured by an administrator, rather than controlled with the JMS API by clients.

Message-driven bean security

Messages arriving at a destination being processed by a message-driven bean have no client credentials associated with them; the messages are anonymous. Security depends on the role specified by the RunAs Identity for the message-driven bean as an EJB component.

Security considerations for the default messaging provider

You can enable bus security so that access to the bus itself and to all destinations on the bus must be authorized. For bus security to be enabled, WebSphere global security must also be enabled.

When a bus is created, an initial set of authorization permissions is created. These permissions grant all authenticated users access to the bus and to all local destinations.

When bus security is enabled, you must set the Inter-engine authentication alias property to control the authentication of messaging engines joining the bus and for secure communication between messaging engines. Similarly, the Mediations authentication alias property is used for mediations that access the bus.

You can use secure transport connections (SSL or HTTPS) to ensure confidentiality and integrity of messages in transit between application clients and messaging engines and between messaging engines. This is achieved by defining transport chains and then referencing the transport chain name as follows:

- For application client connections: from the connection factory administered objects
- For connections to foreign buses: from the Target inbound transport chain property of the service integration bus link
- For connections to WebSphere MQ: from the Transport chain property of the WebSphere MQ link
- For connections between messaging engines: from the Inter-engine transport chain property of the bus

In the routing definitions for connections to foreign buses, the user ID applied to messages entering or leaving the foreign bus can be replaced by values specified by the Inbound user ID and Outbound user ID properties.

The Authentication alias property of the service integration bus link is used for authentication of access to a foreign bus.

Mediations security considerations

When an application sends a message to the bus, the identity of the sender application is associated with the message. When bus security is enabled, any new messages sent by a mediation will have the mediation identity versus the original sender identity. In this case, the mediation identity will require access to the destination. By default, a mediation inherits its identity from the messaging engine. You can change the identity for a mediation handler by specifying a RunAS role using the assembly tools.

12

J2EE Connector Architecture scenario

This chapter gives an overview of the J2EE Connector Architecture (JCA) and discuss its place in an SOA environment. The ITSOMart sample application has been extended to access a back-end CRM system using a resource adapter. A Web service that wraps the functionality accessing the CRM is used to access the CRM through the ESB.

This chapter discusses how the design guidelines outlined in Chapter 7, “Application and system design guidelines” on page 139 were applied to the sample scenario. It also describes the process used to develop the sample application as a way of illustrating development guidelines for JCA applications. The sample has been deployed on WebSphere Application Server V6 and we use the sample to illustrate runtime configuration tasks that must be performed. Lastly, we discuss system management considerations for JCA applications.

12.1 Architectural overview model

The architecture overview model in Figure 12-1 shows the overall structure of the ITSOMart application. The highlighted area shows the section of the application on which this chapter focuses, the ESB.

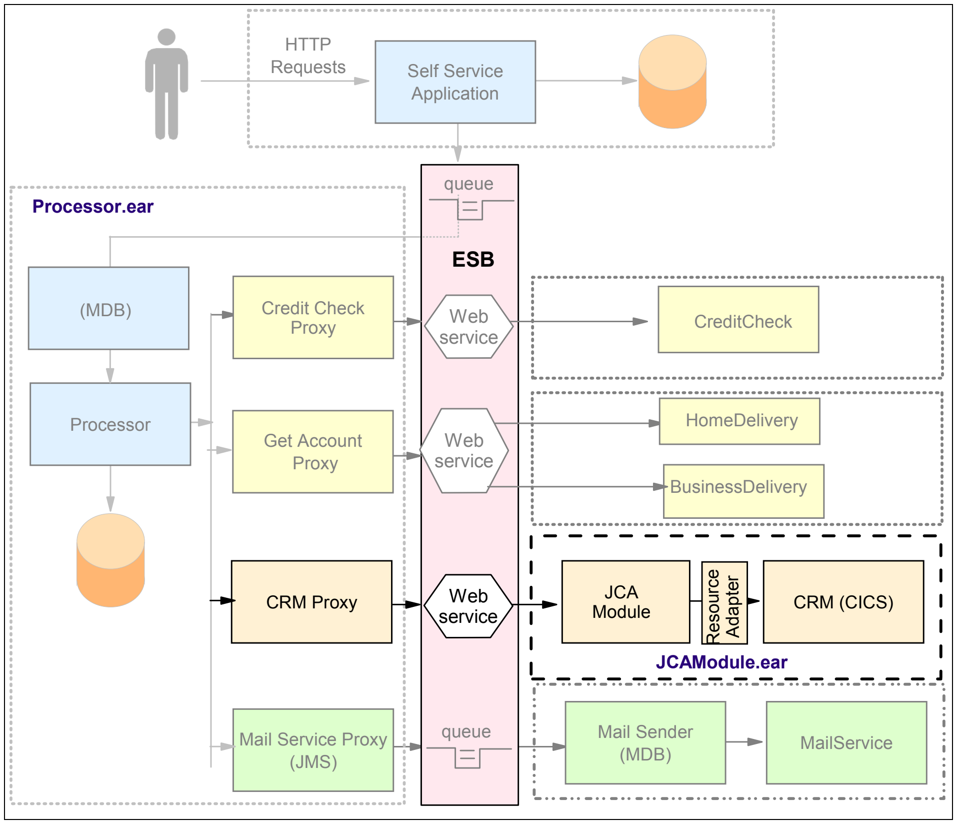


Figure 12-1 Architectural model for ITSOMart application

From the J2EE architecture point of view, this is the part of the business logic tier that creates, updates, and deletes business data from the single application in the Enterprise Information System (EIS) tier.

The components that participate in the JMS Scenario are:

- CRM Proxy

  This component is used by the Processor application to call the CRM Web service.

- Enterprise Service Bus (ESB)

The ESB is implemented using the service integration bus. The bus acts as an intermediary for the Web service. Using an ESB insulates the application from changes in the Web service location and provides the opportunity to implement mediations or security measures in the future.

- JCA Module

  This component sends the requests to create, update, or delete customer data to the CICS application.

- Resource Adapter

  The CICS ECI resource adapter is used to access the back-end CICS transaction.

- CRM

  A CICS transaction that manages the customer database.

This scenario illustrates the following application pattern using messaging technology:

- Directly Integrated Single Channel application pattern:

  ITSOMart sends the customer information directly to the back-end application. The connection to the CICS system is done using a J2C Java bean that has been deployed as a Web service.

12.2 System design overview

In this scenario (Figure 12-2 on page 392) we connect the ITSOMart application to a back-end CICS application called MartAcct.ccp using the CICS ECI resource adapter. The ECI resource adapter is one of the resource adapters that allows Java applications to connect to CICS. It uses the External Call Interface (ECI) of the CICS Transaction Gateway (CICS TG) to communicate with CICS. CICS TG can link to a CICS enterprise-tier, or back-end application passing data in a buffer called the COMMAREA. The CICS application receives customer info through the COMMAREA from the ITSOMart application.

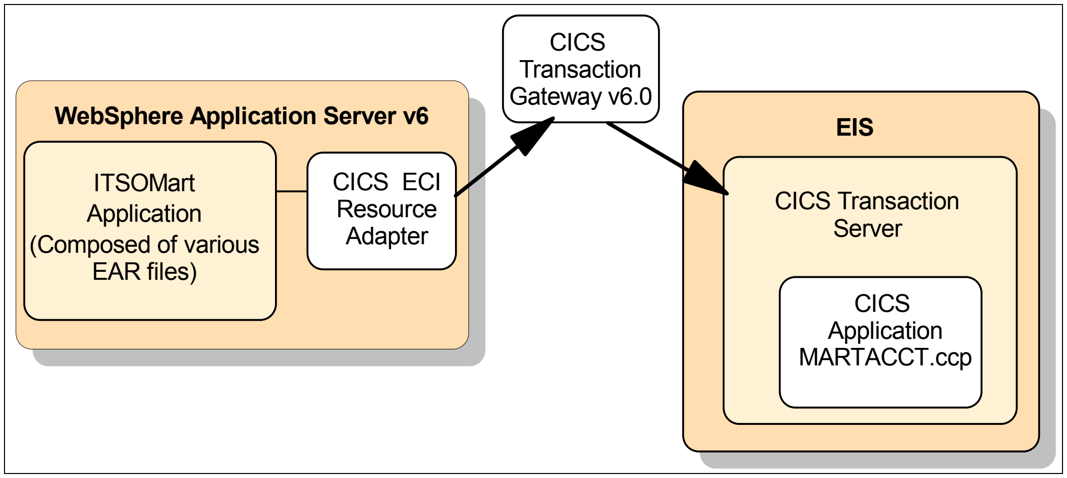

CICS Transaction Gateway v6.0
WebSphere Application Server v6
ITSOMart Application (Composed of various EAR files)
CICS ECI Resource Adapter
EIS
CICS Transaction Server
CICS Application MARTACCT.ccp

Figure 12-2 System design overview

The following assumptions apply to the scenario:

- Of the several resource adapters supported for CICS, we are using a CICS ECI resource adapter for the ITSOMart system.
- Our enterprise-tier application for ITSOMart is a COMMAREA-based CICS program, which resides in a CICS transaction server.

12.2.1 Component model

Figure 12-3 shows the component model for this portion of the ITSOMart application. The component model is described from an application developer's point of view rather than a user's point of view.

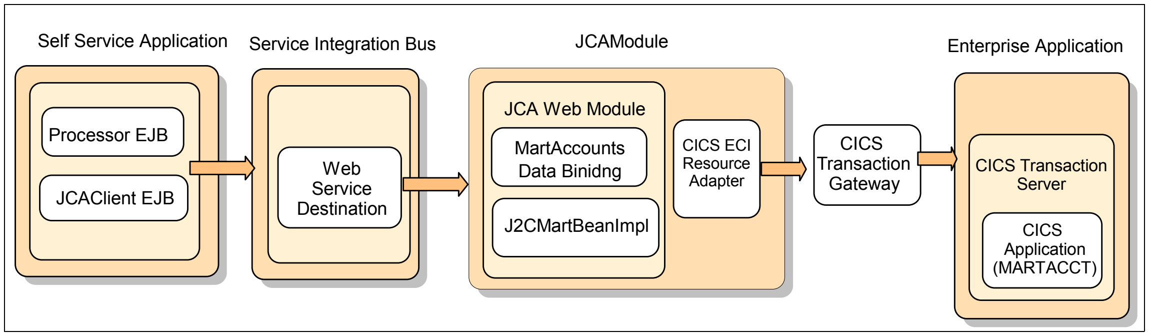


Figure 12-3 Component model

The components involved in the JCA scenario are:

- Processor EJB

  When a customer uses the registration process, the front-end portion of the application generates a message and places it on a queue. The processor EJB is an MDB that receives this message. It triggers the JCA Web service proxy to perform the appropriate action (add, update, or delete) on the CRM (CICS)

- JCA CICS client

  This EJB client is generated from the WSDL representing the CICS Web service. The client calls the exposed methods in the JCA Web service proxy.

- Service integration bus

  The bus enables users to specify a level of indirection between service requestors and providers by exposing existing services at new destinations.

- JCAModule

  The MartAccounts data binding bean maps the back-end application structure to Java.

  The J2CMartBeanImpl implementation bean is created from the CICS application. It uses the Common Client Interface (CCI) classes to send a request to a CICS back-end application via a CICS ECI resource adapter. The reply is sent back from the resource adapter with the appropriate success or failure message, which is eventually sent back to the Processor bean. All the interactions with the CICS component are performed in asynchronous manner,

  The CICS ECI resource adapter provides the CCI for the JCA service. It receives a request from the J2C Impl bean and passes it to the CICS application using the CICS ECI-specific protocol. The reply from the CICS application is passed back to the J2C Impl bean via the resource adapter asynchronously. The CICS ECI resource adapter has an interface to an application server that provides the system contacts defined in the J2EE Connector Architecture.

  The application server provides system and component contracts to both the J2C Impl bean and the CICS ECI resource adapter.

- Enterprise application

  The CICS application receives a request from the CICS ECI resource adapter containing the information about the customer using COMMAREA. It creates a new record for each new customer with the information provided. It can also update and delete customer information.

12.2.2 Object model

In this section, we provide an object model for our JCA scenario.

Class diagram

Figure 12-4 on page 394 shows a class diagram for the JCAModule portion of the application. We focus on this piece because it provides the actual interface to the back-end CRM system. The diagram shows the static relationships between the classes.

MartAccounts and MartJ2CBeanImpl were developed for the ITSOMart application. The rest of the classes are provided by the ECI resource adapter or the J2EE packages.

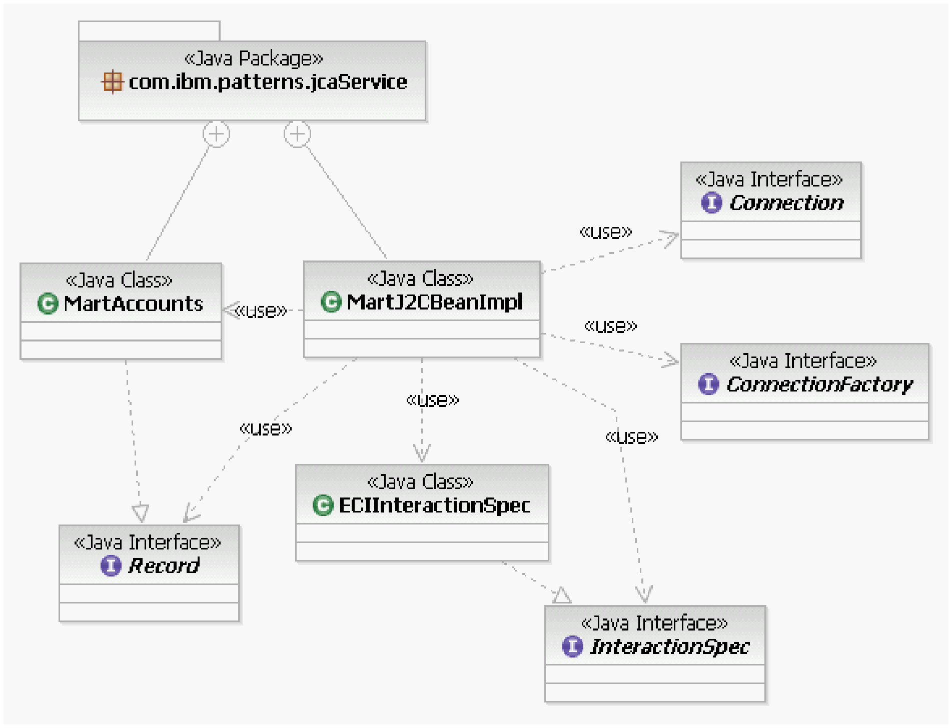


Figure 12-4 Class diagram for J2EE Connector scenario

The com.ibm.patterns.jcaService package includes MartJ2CBeanImpl which uses MartAccounts to update a customer record in the enterprise tier.

The MartAccounts class provides a byte array for the CICS COMMAREA, which is used to pass the information to the back-end application. The MartAccounts class is an implementation of the Record class provided by J2EE Connector class library.

The ECIInteractionSpec class is the specification of an interaction for CICS ECI. The superclass of ECIInteractionSpec is InteractionSpec. The various kinds of resource adapters extend InteractionSpec.

InteractionSpec and Record are the only classes that need to be specific to a particular resource adapter. If an application needs to interact with another resource adapter, the only changes that need to be made are to use the new Record class and InteractionSpec class. The use of the other classes, such as ConnectionFactory, Connection, and Interaction, remains the same. For example, if the application needs to have an interface with the IMS Resource adapter, the application needs a new Record for IMS and it needs to create and set an IMS InteractionSpec, which is passed to the generic interaction object.

Interaction diagram

The interaction diagram in Figure 12-5 shows the sequence of the message flow for the JCA call within the ITSOMart application. Once the customer credit rating is checked, the customer registration information is sent to the CRM application and an e-mail is sent to the customer.

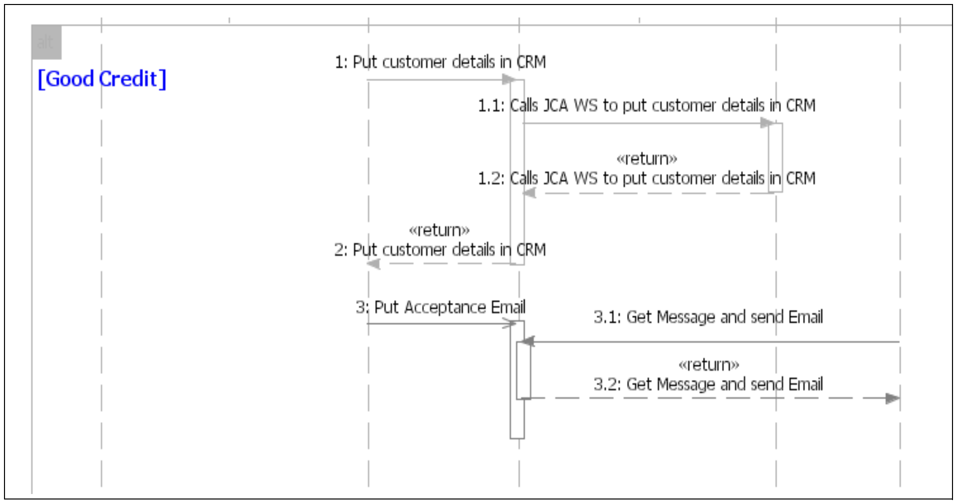

[Good Credit]
1: Put customer details in CRM
1.1: Calls JCA WS to put customer details in CRM
«return»
1.2: Calls JCA WS to put customer details in CRM
«return»
2: Put customer details in CRM
3: Put Acceptance Email
3.1: Get Message and send Email
«return»
3.2: Get Message and send Email

Figure 12-5 Interaction diagram for JCA functionality in ITSOMart scenario

Figure 12-6 on page 396 shows the message interactions of the classes.

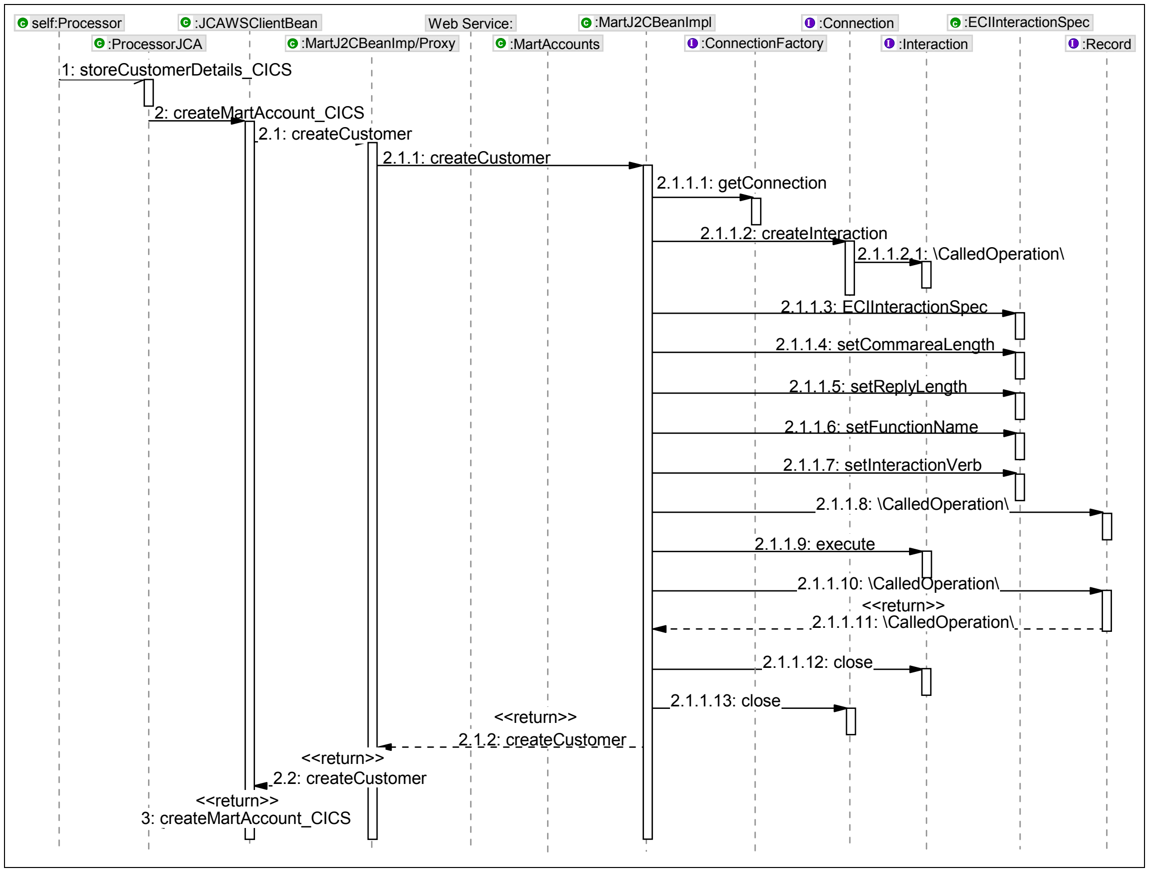


Figure 12-6 Sequence diagram for createCustomerJ2EE Connector scenario

1) Processor invokes the storeCustomerDetails_CICS() method of ProcessorJCA.

2) ProcessorJCA invokes the createMartAccount_CICS method of JCAWSClientBean to create the customer with the information provided in CustomerDetails object.

2.1) JCAWSClientBean invokes the createCustomer() method of MartJ2CBeanImplProxy.

2.1.1) MartJ2CBeanImplProxy invokes the createCustomer() method of MartJ2CBeanImpl.

2.1.1.1) MartJ2CBeanImpl gets a connection using a connection factory. The runtime connection attributes are configured from the application server connection factory properties. This may not mean a physical connection to the enterprise has been opened. A physical connection may have been opened by another connection instance and pooled by an application server. An application program only needs to be concerned with the logical connection given by the ConnectionFactory, no matter if the physical connection is opened or not.

2.1.1.2) MartJ2CBeanImpl invokes the connection createInteraction() method.

2.1.1.3) MartJ2CBeanImpl creates a CICS ECI interaction spec.

2.1.1.4) MartJ2CBeanImpl sets the COMMAREA length in CICS ECI interaction spec.

2.1.1.5) MartJ2CBeanImpl sets the reply length in CICS ECI interaction spec.

2.1.1.6) MartJ2CBeanImpl sets the CICS program name in CICS ECI interaction spec.

2.1.1.7) MartJ2CBeanImpl sets the interaction verb in CICS ECI interaction spec for asynchronous calling.

2.1.1.8) MartJ2CBeanImpl creates a MartAccounts record and sets it with the customer information.

2.1.1.9) MartJ2CBeanImpl invokes the execute method of the interaction passing the interaction spec and the record. The input and output record is the same object in this scenario.

2.1.1.10) MartJ2CBeanImpl creates the record.

2.1.1.12) MartJ2CBeanImpl closes the interaction.

2.1.1.13) MartJ2CBeanImpl closes the connection. This may not mean that a physical connection to the enterprise is closed, but an application program should close the logical connection every time an interaction is completed.

12.3 Applying the design guidelines

In this section, we discuss some of the low-level design considerations made while adding J2EE Connector support to the ITSOMart application.

12.3.1 Creating the input and output record

The first step is in building the call to the back-end application using JCA is to create an input and output record structure that corresponds to the enterprise application. A record is the Java representation of a data structure used as input or output to an EIS function. Once a connection to the enterprise tier is established, we should be able to pass and request the required data to the EIS for the business operations we need. In the case of ECI, the record is a representation of the COMMAREA.

We look at how you can create these records with two different approaches:

- By using Rational Software Architect to import the COMMAREA structure from an enterprise application written in C or COBOL
- By implementing the javax.resource.cci.Record interface to make a custom record

Using Rational Software Architect

Rational Software Architect provides the capability to create CCI custom records in a Java bean class known as a data binding bean. This class implements javax.resource.cci.Record class (included in <*was_install*>/lib j2ee.jar) and is a representation of the COMMAREA for an ECI connection.A J2C Java bean class uses this data binding record class as a data structure for both input and output data from EIS to application server. You will see how to do this later in "Step 1: Create the data binding class" on page 410.

Example 12-1 shows the code found in our sample MartAccounts class. This class implements Record and uses a simple getter-setter design pattern for its field values for a generic ECI application.

Example 12-1 Implementation of Record

```
public class MartAccounts implements javax.resource.cci.Record,
      javax.resource.cci.Streamable, com.ibm.etools.marshall.RecordBytes {
   /**
    * @generated
    */
   private byte[] buffer_ = null;

   /**
    * @generated
    */
   private int bufferSize_ = 0;

   /**
    * @generated
    */
```

```
private static byte[] initializedBuffer_ = null;

/**
 * @generated
 */
private static java.util.HashMap getterMap_ = null;
/**
 * @generated
 */
private java.util.HashMap valFieldNameMap_ = null;
/**
 * constructor
 * @generated
 */
public MartAccounts() {
   initialize();
}
/**
 * constructor
 * @generated
 */
public MartAccounts(java.util.HashMap valFieldNameMap) {
   valFieldNameMap_ = valFieldNameMap;
   initialize();
}
/**
 * @generated
 * initialize
 */
public void initialize() {
   bufferSize_ = 719;
   buffer_ = new byte[bufferSize_];
   if (initializedBuffer_ == null) {
      String ci__tyInitialValue = " ";
      MarshallStringUtils.marshallFixedLengthStringIntoBuffer(
            ci__tyInitialValue, buffer_, 401, "ISO-8859-1", 100,
            MarshallStringUtils.STRING_JUSTIFICATION_LEFT, " ");
      String street__addInitialValue = " ";
      MarshallStringUtils.marshallFixedLengthStringIntoBuffer(
            street__addInitialValue, buffer_, 301, "ISO-8859-1", 100,
            MarshallStringUtils.STRING_JUSTIFICATION_LEFT, " ");
      String pho__neInitialValue = " ";
      MarshallStringUtils.marshallFixedLengthStringIntoBuffer(
            pho__neInitialValue, buffer_, 501, "ISO-8859-1", 20,
            MarshallStringUtils.STRING_JUSTIFICATION_LEFT, " ");
      String l__nameInitialValue = " ";
      MarshallStringUtils.marshallFixedLengthStringIntoBuffer(
            l__nameInitialValue, buffer_, 201, "ISO-8859-1", 100,
            MarshallStringUtils.STRING_JUSTIFICATION_LEFT, " ");
```

```
        String e__mailInitialValue = " ";
        MarshallStringUtils.marshallFixedLengthStringIntoBuffer(
              e__mailInitialValue, buffer_, 1, "ISO-8859-1", 100,
              MarshallStringUtils.STRING_JUSTIFICATION_LEFT, " ");
        String pass__wordInitialValue = " ";
        MarshallStringUtils.marshallFixedLengthStringIntoBuffer(
              pass__wordInitialValue, buffer_, 521, "ISO-8859-1", 100,
              MarshallStringUtils.STRING_JUSTIFICATION_LEFT, " ");
        String mart__msgInitialValue = " ";
        MarshallStringUtils.marshallFixedLengthStringIntoBuffer(
              mart__msgInitialValue, buffer_, 625, "ISO-8859-1", 94,
              MarshallStringUtils.STRING_JUSTIFICATION_LEFT, " ");
        String mart__flagInitialValue = " ";
        MarshallStringUtils.marshallFixedLengthStringIntoBuffer(
              mart__flagInitialValue, buffer_, 0, "ISO-8859-1", 1,
              MarshallStringUtils.STRING_JUSTIFICATION_LEFT, " ");
        String f__nameInitialValue = " ";
        MarshallStringUtils.marshallFixedLengthStringIntoBuffer(
              f__nameInitialValue, buffer_, 101, "ISO-8859-1", 100,
              MarshallStringUtils.STRING_JUSTIFICATION_LEFT, " ");
        byte[] bytes = new byte[bufferSize_];
        System.arraycopy(buffer_, 0, bytes, 0, bufferSize_);
        initializedBuffer_ = bytes;
     } else {
        System.arraycopy(initializedBuffer_, 0, buffer_, 0, bufferSize_);
     }
   }
```

Creating a custom record manually

You can create a custom record that is specific to an enterprise application by extending the Record interface. In Example 12-2 the GenericRecord class uses a simple getter-setter design pattern for its field values for a generic ECI application. If you are not going to use a development tool for the record generation, this generic record provides a simple interface to communicate with COMMAREA-based enterprise applications.

Example 12-2 Implementation of Custom Record

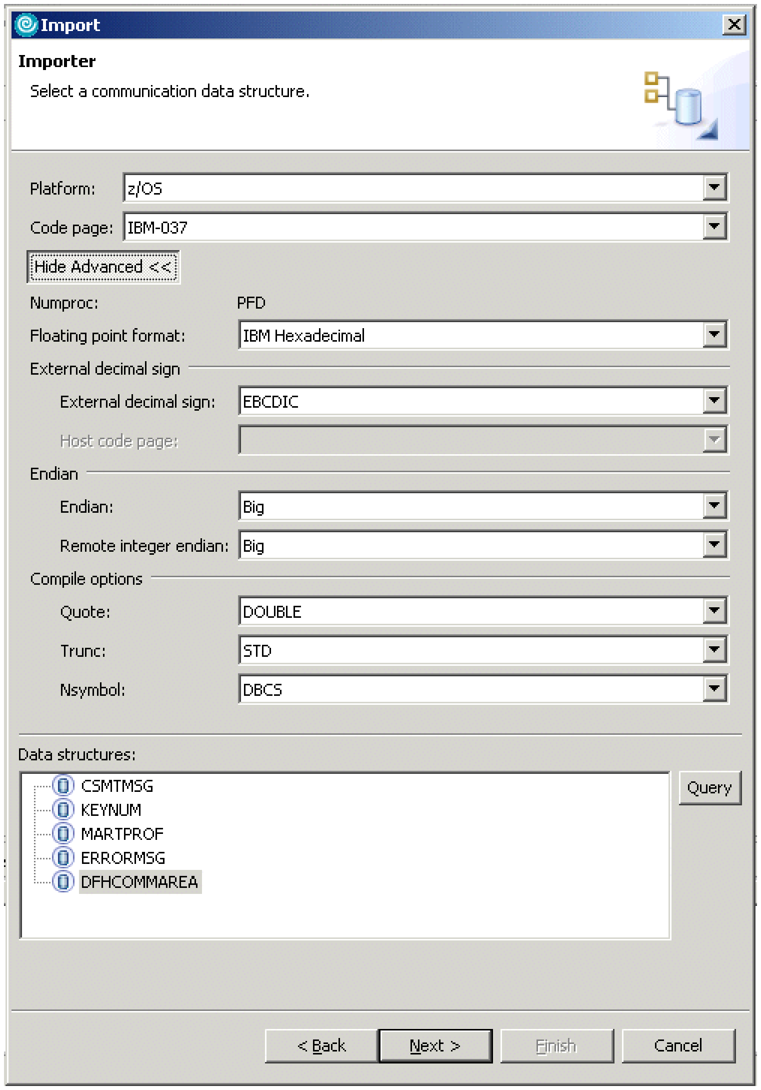
```
public class GenericRecord implements javax.resource.cci.Record,
javax.resource.cci.Streamable {

   private byte commarea[]=null;

   public GenericRecord() {
      super();
   }

   public GenericRecord(byte[] comm) {
      setCommarea(comm);
   }

   public Object clone() throws CloneNotSupportedException{
      return super.clone();
   }

   public void setCommarea(byte[] comm) {
      try {
         read(new java.io.ByteArrayInputStream(comm));
      }catch (java.io.IOException ioe) {
      }
}
```

12.3.2 Data conversion

An enterprise tier might use a different code set from the client application. Typically the enterprise tier resides in a mainframe using the EBCDIC code set, and the client application is written in Java using Unicode. Similarly, encoding of an integer in the PC world is different from that in the UNIX and mainframe worlds. There are various options to performing data conversion between different systems.

Data conversion with Rational Software Architect

If you are using Rational Software Architect, the J2EE Connector Tools provide the tools to do the automatic conversion. Using the tools, you can import the Java message structure and conversion code from the C or COBOL COMMAREA definition used in the CICS application. The panel shown in Figure 12-7 on page 402 allows you to select the required options while creating a data binding record class.



Import

Importer

Select a communication data structure.

Platform: z/OS

Code page: IBM-037

Hide Advanced <<

Numproc: PFD

Floating point format: IBM Hexadecimal

External decimal sign

External decimal sign: EBCDIC

Host code page:

Endian

Endian: Big

Remote integer endian: Big

Compile options

Quote: DOUBLE

Trunc: STD

Nsymbol: DBCS

Data structures:

- CSMTMSG
- KEYNUM
- MARTPROF
- ERRORMSG
- DFHCOMMAREA

Figure 12-7 Data conversion options

Data conversion manually

If you create a record manually using a generic record, you need to perform code conversion yourself. Example 12-3 on page 403 shows how the String account name is converted to EBCDIC using the String getBytes method, and then used to set the input record.

Example 12-3 Data conversion (Unicode to EBCDIC)

```
rateRecord = new GenericRecord(account.getBytes("IBM037"));
```

Example 12-4 shows how the entire output COMMAREA returned from the enterprise tier is converted from EBCDIC to a String format using a String constructor.

Example 12-4 Data conversion (EBCDIC to Unicode)

```
String commarea_rate = new String(rateRecord.getCommarea(),"IBM037");
```

12.3.3 Connection management

Once you have a mapping record structure in Java from COBOL or C code, you are ready to make a connection and supply the required data. This section describes the things that need to be done by the application program to connect to CICS using a J2EE connector.

A connection can be *managed* or *nonmanaged*. A managed connection is managed by the application server. A non-managed connection is obtained directly through the resource adapter.

The class diagram in Figure 12-4 on page 394 shows a managed connection. The application program uses ConnectionFactory and Connection to manage the connection. A reference to ConnectionFactory is obtained from the JNDI name space using InitialContext, and an instance of Connection is obtained from ConnectionFactory. The physical aspects of connection management are handled by the following components:

- Application server and resource adapter

  Connections from the application server to the CICS Transaction Gateway through the ECI resource adapter can utilize the application server pool manager to reuse free connections as they become available. This applies both to network connections when using a remote Gateway daemon and to local connections when using a local CICS Transaction Gateway. Similarly, terminals in use with the EPI resource adapter are also pooled, although connections are not pooled for the EPI resource adapter.

- Resource manager

  Some resource managers have their own connection management functions. The CICS Transaction Gateway internally manages the connections from the Client daemon to the attached CICS regions. This is not visible to the application server or resource adapter.

Rational Software Architect generates another Java bean called a J2C Java bean that implements connection management for us. The intializeBinding() method handles the JNDI look up for managed connections. A resource reference to the CICS ECI resource adapter connection factory, called MartAccountCFRef, is defined in the deployment descriptor to map to the actual JNDI name. The value of the JNDI reference can be changed at deployment time.

JNDI lookup

The ITSOMart application looks up a connection factory instance with a managed connection using the JNDI interface. Example 12-5 shows this lookup.

Example 12-5 Acquiring a connection factory using JNDI lookup

```
protected void initializeBinding() throws ResourceException {
      ConnectionFactory cf = null;
      String jndiName = "MartAccountCFRef";
      javax.naming.Context ctx = null;
      try {
         ctx = new javax.naming.InitialContext();
         if (ctx != null)
            cf = (ConnectionFactory) ctx.lookup("java:comp/env/" + jndiName);
         if (cf == null)
            throw new javax.naming.NamingException();
      } catch (javax.naming.NamingException exception) {
         try {
            if (ctx != null)
               cf = (ConnectionFactory) ctx.lookup(jndiName);
         } catch (javax.naming.NamingException exception1) {
         }
      }
```

For non-managed connections, the intializeBinding() method first creates a managedConnectionFactory instance and sets the required values. Once the connection properties are set, it creates the connection from the managedConnectionFactory instance. Example 12-6 shows acquiring a connection factory using managedConnectionFactory in a non-managed environment.

Example 12-6 Acquiring a connection factory

```
com.ibm.connector2.cics.ECIManagedConnectionFactory mcf = new
com.ibm.connector2.cics.ECIManagedConnectionFactory();
         mcf.setConnectionURL("sourcecode.rtp.raleigh.ibm.com");
         mcf.setServerName("gifty");
         mcf.setUserName("ANUPAG");
```

```
          mcf.setPassword("anpag");
          mcf.setTraceLevel(Integer.valueOf("1"));
          cf = (ConnectionFactory) mcf.createConnectionFactory();
```

Once the connection is acquired, whether managed or non-managed, it is set using setConnectionFactory:

```
setConnectionFactory(cf);
```

Dealing with a connection

The application component invokes the getConnection method on the connection factory to get a CICS connection, as shown in Example 12-7. The returned connection instance represents an application-level handle to an underlying physical connection.

Example 12-7 Acquiring a connection

```
// use connection factory to get a connection handle
conn = cf.getConnection();
```

You can also specify a user ID and password in a connection spec, as shown in Example 12-8.

Example 12-8 Acquiring a connection specifying user ID and password

```
// create a connection spec
ECIConnectionSpec connspec = new ECIConnectionSpec(userid, password);
// use connection factory to get a connection handle
conn = cf.getConnection(connspec);
```

The enterprise system uses the user ID and password for authentication and authorization. A connection specification is unique to a resource adapter, so the application needs to use the required connection specification, such as ECIConnectionSpec. Creating ConnectionSpec is optional because the authentication details can be specified when deploying the application.

After the component finishes with the connection, it closes the connection using the close method, as shown in Example 12-9:

Example 12-9 Closing a connection

```
// close a connection
conn.close();
```

If an application component fails to close an allocated connection after its use, that connection is considered an unused connection. The application server manages the clean up of unused connections. When terminating a component instance, the container cleans up all connections used by that component instance.

12.3.4 Executing the enterprise application

An *interaction* (Example 12-10) enables an application program to execute functions provided by the enterprise tier. The execute method takes an input record, output record, and an InteractionSpec. This method executes the enterprise tier function represented by the InteractionSpec and updates the output record.

An *Interaction instance* is created from a Connection and is required to maintain its association with the Connection instance. The close method releases all resources maintained by the resource adapter for the interaction. Closing an Interaction instance should not close the associated connection instance.

Example 12-10 Interaction

```
// use connection to create an interaction object
Interaction interaction = conn.createInteraction()
```

In Example 12-11, the code creates the InteractionSpec that executes the interaction on a CICS ECI resource adapter.

Example 12-11 IneractionSpec

```
// Create and setup the CICS ECI interation spec
InteractionSpec iSpec = interactionSpec;
```

While creating the Java methods in the J2C Java BeanImpl bean, Rational Software Architect allows you to set properties in InteractionSpec. In the ITSOMart scenario, we are using an asynchronous ECI call and set the FunctionName as MARTACCT. See Example 12-12,

Example 12-12 Create record and execute

```
if (iSpec == null)
{
iSpec = new com.ibm.connector2.cics.ECIInteractionSpec();
((com.ibm.connector2.cics.ECIInteractionSpec)iSpec).setFunctionName("MARTACCT")
;
}
...
//Create Record
```

```
com.ibm.patterns.jcaService.MartAccounts output = new
com.ibm.patterns.jcaService.MartAccounts();

// Invoke the program
interaction.execute(iSpec, input, output);
```

12.3.5 Transaction management

The two-phase commit capability of CICS ECI resource adapters is only supported with WebSphere Application Server for z/OS. However, we would still recommend that you use container-managed transactions in distributed platform environments, and allow the container to optimize the transaction using an only-agent optimization or last participant support.

For more information, see *IBM WebSphere Developer Technical Journal: Transactional integration of WebSphere Application Server and CICS with the J2EE Connector Architecture* at:

http://www-128.ibm.com/developerworks/websphere/techjournal/0408_wakelin/0408_wakelin.html

12.3.6 Security

The user ID and password required to access the back-end system can be specified in the resource adapter properties in the WebSphere Application Server runtime environment. The user ID and password can also be specified using a component-managed authentication alias, a container-managed authentication alias, or programatically using the ConnectionSpec.

12.4 Development guidelines for JCA

This section describes the implementation of a JCA connection in the ITSOMart sample application. The sample was developed using Rational Software Architect and deployed on WebSphere Application Server V6. The service integration bus was used to provide the destination endpoints for the Web service.

J2C Connector Tools: To do these steps you need v6.0.0.1 of Rational Software Architect and the optional J2C Connector Tools. To upgrade and add the tool, use the IBM Rational Product Updater. You can access this from the workbench by selecting **Help** → **Software Updates** → **IBM Rational Product Updater**. Click the **Updates** tab to find the V6.0.0.1 upgrade. The J2C Connector Tools can be found under the **Optional Features** tab.

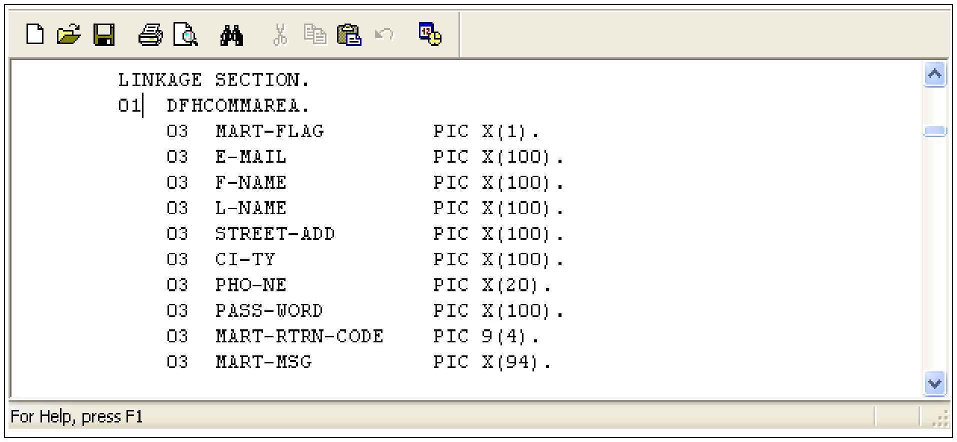
The following are the steps used to build a J2C application that interfaces with CICS transactions using ECI

1. Prepare the enterprise application, in this case a CICS-COBOL program.
2. Use the J2C Wizard to create a data binding bean and a J2C Java bean.
3. Deploy the J2C Java bean as a Web service.
4. Create inbound and outbound services for the bus.
5. Generate the client for the inbound Web service.
6. Integrate the J2C beans and the client in the ITSOMart application.

12.4.1 The CICS enterprise application

Note: This section assumes that you have an existing COBOL or COBOL/CICS program and that you can access the source code from the workspace.

In the ITSOMart scenario, a CICS COBOL enterprise application program named martacct.ccp exists on the back-end CICS system. It is designed to manage customer information on a database using the E-MAIL field as the primary key. The DFHCOMMAREA code snippet is shown in Figure 12-8 on page 409.

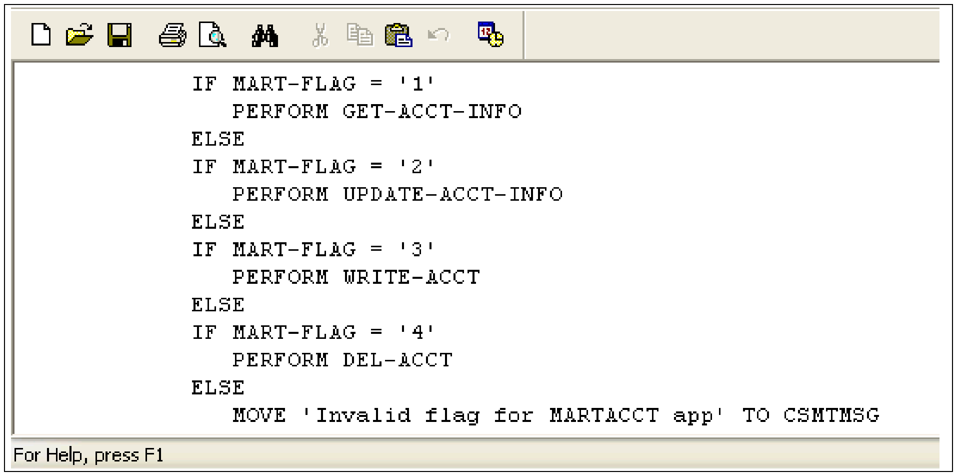


```
LINKAGE SECTION.
01  DFHCOMMAREA.
    03  MART-FLAG         PIC X(1).
    03  E-MAIL            PIC X(100).
    03  F-NAME            PIC X(100).
    03  L-NAME            PIC X(100).
    03  STREET-ADD        PIC X(100).
    03  CI-TY             PIC X(100).
    03  PHO-NE            PIC X(20).
    03  PASS-WORD         PIC X(100).
    03  MART-RTRN-CODE    PIC 9(4).
    03  MART-MSG          PIC X(94).
```

For Help, press F1

Figure 12-8 DFHCOMMAREA from CICS COBOL program

The martacct program file provides add, update, delete, and retrieve capability to the database. To separate these functions, a flag called MART-FLAG will be sent with every enterprise function call. Figure 12-9 shows the functions available in the program.



```
IF MART-FLAG = '1'
   PERFORM GET-ACCT-INFO
ELSE
IF MART-FLAG = '2'
   PERFORM UPDATE-ACCT-INFO
ELSE
IF MART-FLAG = '3'
   PERFORM WRITE-ACCT
ELSE
IF MART-FLAG = '4'
   PERFORM DEL-ACCT
ELSE
   MOVE 'Invalid flag for MARTACCT app' TO CSMTMSG
```

For Help, press F1

Figure 12-9 Functions available in CICS-COBOL program

12.4.2 Create a JCA application to access the enterprise application

Rational Software Architect includes wizards that assist you in creating JCA applications.

The steps to create the application are as follows:

1. Create a data binding class that maps the COBOL structures to Java.
2. Create a J2C Java bean that will have the EIS connection information and the Java methods that call the EIS functions. A J2C Java bean will use the data binding bean class as the input and output to the available functions.
3. Optionally: Deploy the created bean as a JSP, EJB, or Web service. In this scenario, we will deploy the J2C Java bean as a Web service.

Preparation: Before working with JCA applications in Rational Software Architect, be sure to do the following:

1. Ensure the J2C capability is enabled in your workspace:
   a. Open the workspace and select **Window** → **Preferences**.
   b. Expand **Workbench** and click **Capabilities**. Make sure the box to the left of the Enterprise Java folder is checked.
   c. Click **Apply** and then **OK**.
2. Open the J2EE perspective.

This sample assumes that you have created the following:

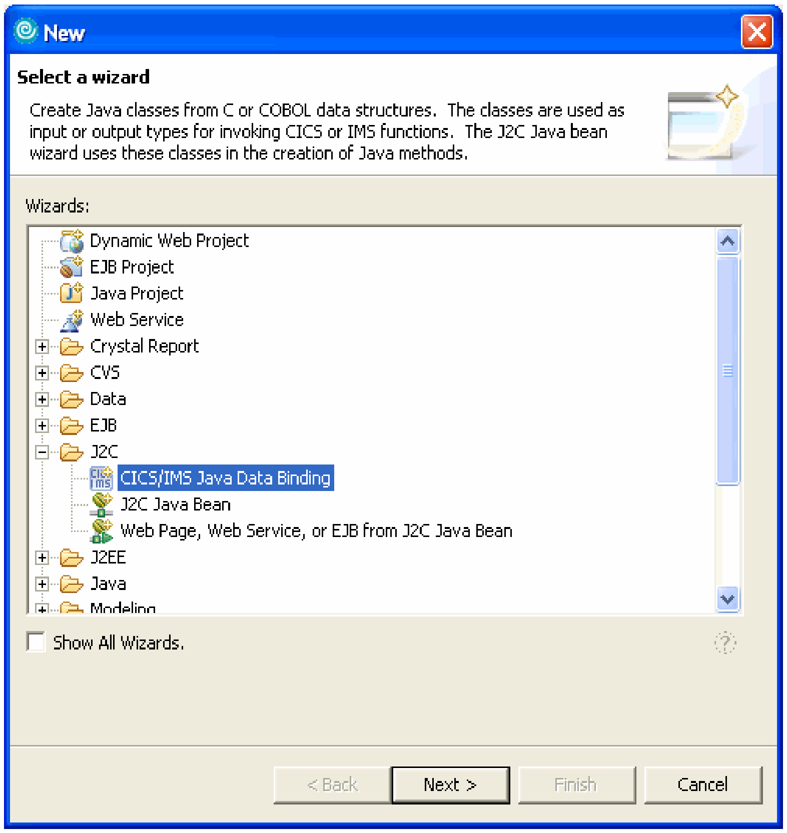
- An enterprise application project called **JCAModule**. The J2EE properties for the project specify J2EE version 1.4. The server properties specify WebSphere Application Server V6.0 as the target server.
- A dynamic Web project in JCAModule called JCAWeb. The context root is JCAWeb. The project properties specify Servlet version 2.4 and WebSphere Application Server V6.0 as the target server.

Note that to test the application you need to have the COBOL application running on the CICS server.

Step 1: Create the data binding class

Perform the following steps:

1. In the workbench, select **File** → **New** → **Other** → **J2C** in the tool bar.
2. Expand J2C and select **CICS/IMS Java Data Binding**, as shown in Figure 12-10 on page 411.



Create Java classes from C or COBOL data structures. The classes are used as input or output types for invoking CICS or IMS functions. The J2C Java bean wizard uses these classes in the creation of Java methods.

Figure 12-10 Create the J2C data binding

Click **Next**.

3. The next screen gives you the following data binding options:
   - C MPO to Java
   - COBOL to Java
   - C to Java
   - COBOL MPO to Java

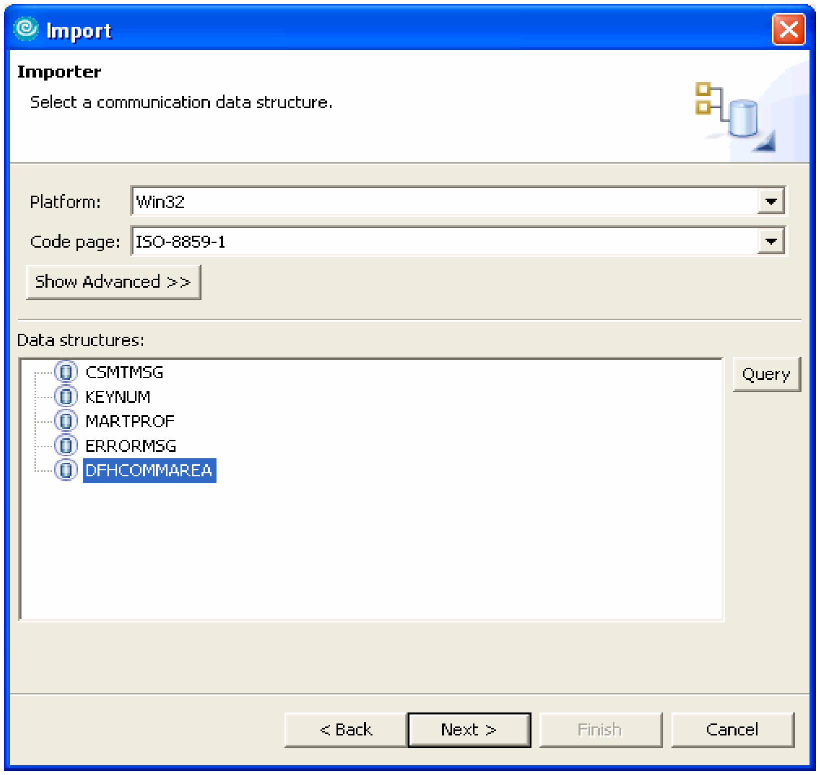
   Select **COBOL to Java** from the drop-down.

   Browse to the COBOL program file, **martacct.ccp**, and click **Next**.

4. In the next panel, select the platform the CICS server is running on and provide the code page settings.

   Click **Query** to display the data structures available in enterprise application program and select the appropriate settings.

   In our example, we used the settings shown in Figure 12-11 on page 412.


Importer

Select a communication data structure.

Platform: Win32

Code page: ISO-8859-1

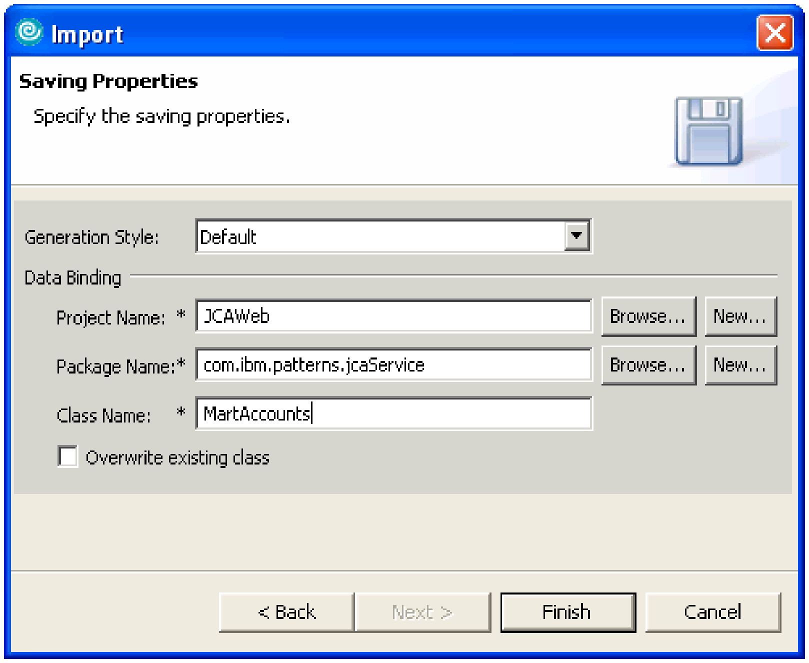
Data structures:

- CSMTMSG
- KEYNUM
- MARTPROF
- ERRORMSG
- DFHCOMMAREA

Figure 12-11 Import settings

Click **Next**.

5. Figure 12-12 on page 413 allows you to specify the values to use in creating and saving the new file.


Import

Saving Properties

Specify the saving properties.

Generation Style: Default

Data Binding

Project Name: * JCAWeb

Package Name:* com.ibm.patterns.jcaService

Class Name: * MartAccounts

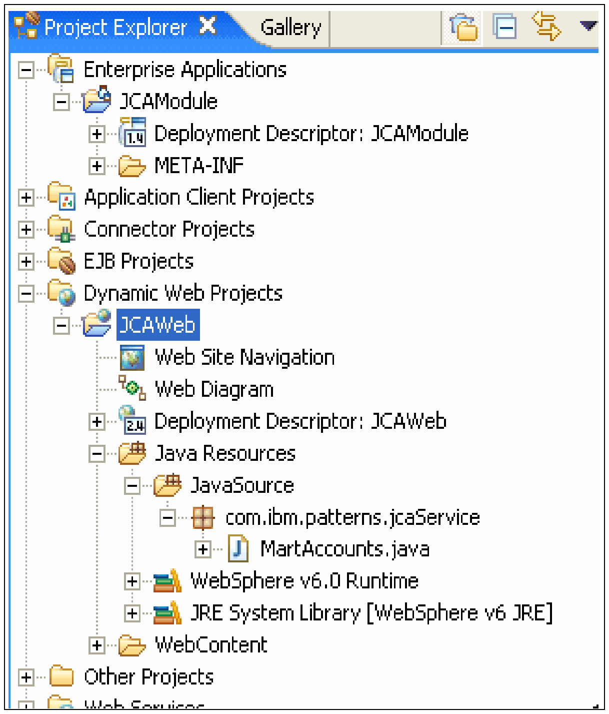
☐ Overwrite existing class

Figure 12-12 Import settings ,Data Binding name: MartAccounts

- Choose **Default** for the generation style.
- In the Data Binding section, provide the Web project name, `JCAWeb`.
- Enter the package name `com.ibm.patterns.jcaService`.
- The class name defaults to the one chosen in the Importer panel (Figure 12-11 on page 412). We chose DFHCOMMAREA. Rename it to `MartAccounts`.

Click **Finish**.

You can see the new file in the Project Explorer view (Figure 12-13 on page 414).



```
Enterprise Applications
  JCAModule
    Deployment Descriptor: JCAModule
    META-INF
Application Client Projects
Connector Projects
EJB Projects
Dynamic Web Projects
  JCAWeb
    Web Site Navigation
    Web Diagram
    Deployment Descriptor: JCAWeb
    Java Resources
      JavaSource
        com.ibm.patterns.jcaService
          MartAccounts.java
      WebSphere v6.0 Runtime
      JRE System Library [WebSphere v6 JRE]
    WebContent
Other Projects
```

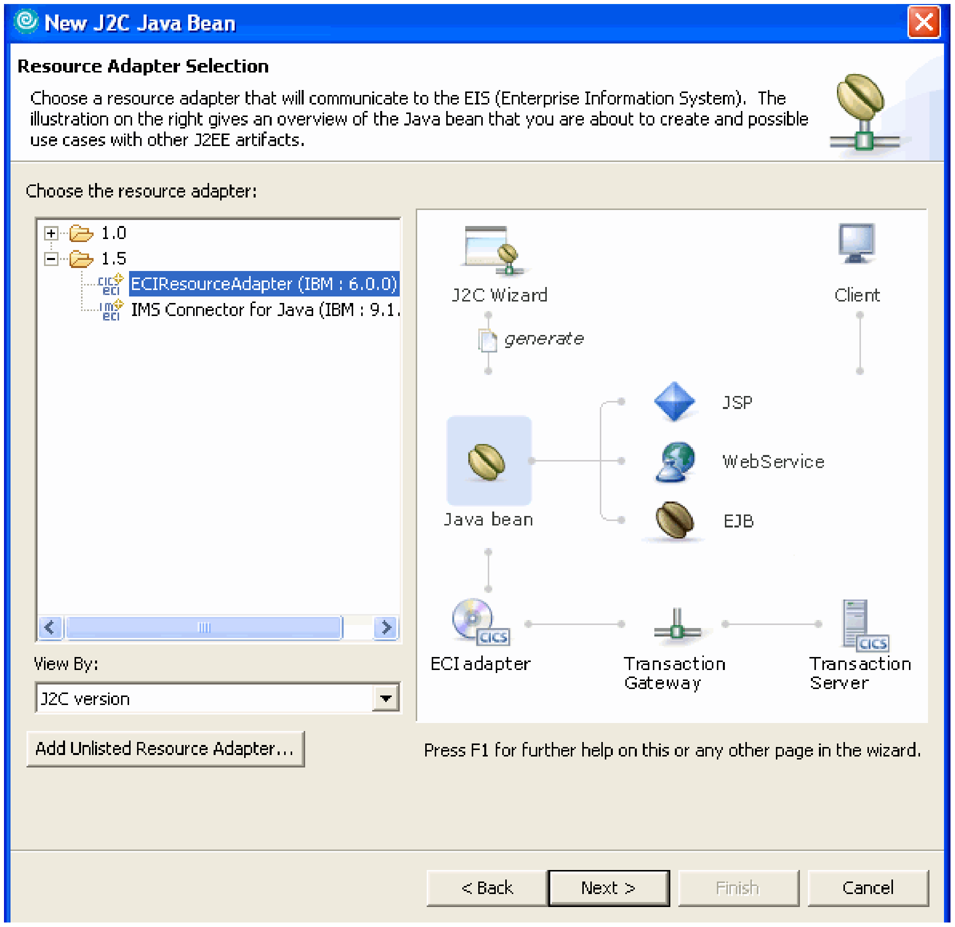
Figure 12-13 Project Explorer showing the data binding bean

Step 2: Create the J2C Java bean

In this step, we create a J2C Java bean that contains the methods to be exposed for the client from the COBOL program. This bean will also:

- Contain the connection information for the CICS Transaction Gateway.
- Use the data binding bean class we just created for the input and output record structure.

1. On tool menu select **File** → **New** → **Other** → **J2C** (Figure 12-10 on page 411).
2. Expand J2C and select **J2C Java Bean**. Click **Next**.
3. The next screen, shown in Figure 12-14 on page 415,allows you to select the resource adapter you will use to connect to the EIS system.


New J2C Java Bean

Resource Adapter Selection

Choose a resource adapter that will communicate to the EIS (Enterprise Information System). The illustration on the right gives an overview of the Java bean that you are about to create and possible use cases with other J2EE artifacts.

Choose the resource adapter:

- 1.0
- 1.5
  - ECIResourceAdapter (IBM : 6.0.0)
  - IMS Connector for Java (IBM : 9.1.

View By: J2C version

Add Unlisted Resource Adapter...

Press F1 for further help on this or any other page in the wizard.

Figure 12-14 Resource adapter selection.

In the View By field, select **J2C version**. In the resulting list of resource adapters, expand the 1.5 category and select **ECIResourceAdapter (IBM 6.0.0)**. Click **Next**.

4. In the next screen, two connection options are available:

   a. Managed Connection

      These connections are created by a connection factory and are managed by the application server. The application accesses a connection factory using JNDI.

      For this option, provide the JNDI name of the connection factory.

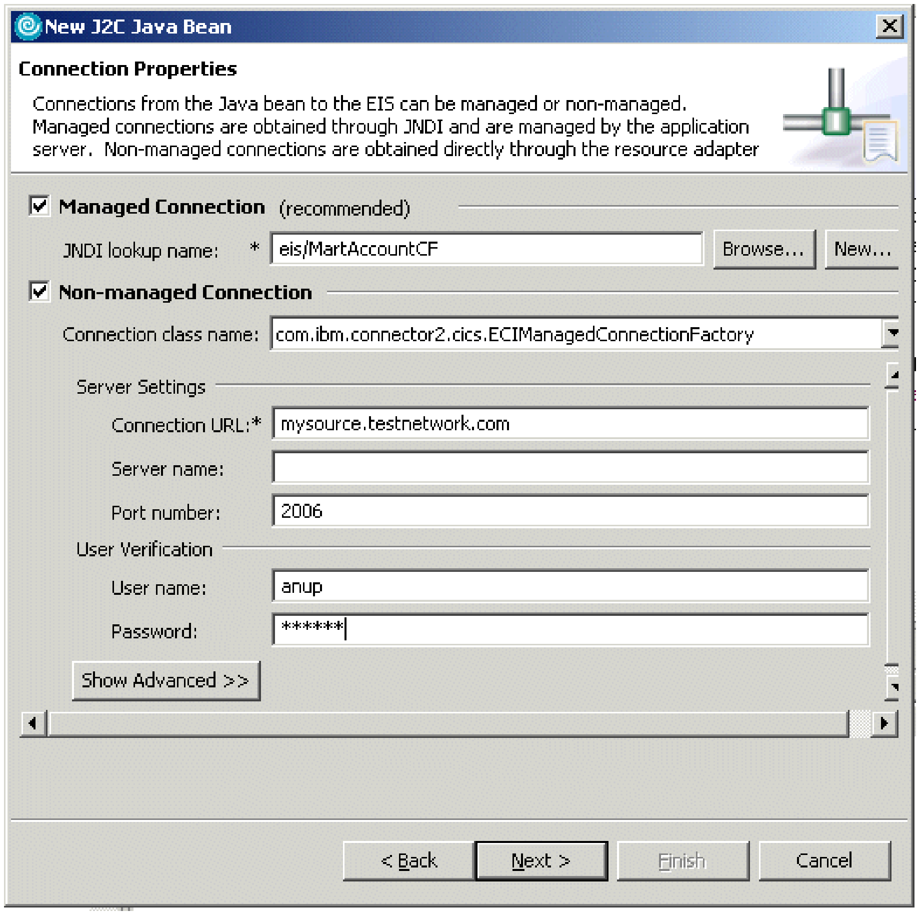
Best practice: Managed connections are recommended as best practice. The application server's connection manager works with the resource adapter to manage connections efficiently, providing JCA quality of service.

b. Non-managed Connection:

   These are obtained directly through the resource adapter and are not managed by the application server.

Select both connection types. The information required for our example is shown in Figure 12-15 on page 417.

Note: When both types of connection are selected the server will first try to find the JNDI name. Only if the managed connection cannot be located, will the non-managed connection be tried.


New J2C Java Bean

Connection Properties

Connections from the Java bean to the EIS can be managed or non-managed. Managed connections are obtained through JNDI and are managed by the application server. Non-managed connections are obtained directly through the resource adapter

☑ Managed Connection (recommended)

JNDI lookup name: * eis/MartAccountCF

☑ Non-managed Connection

Connection class name: com.ibm.connector2.cics.ECIManagedConnectionFactory

Server Settings

Connection URL:* mysource.testnetwork.com

Server name:

Port number: 2006

User Verification

User name: anup

Password: ******

Figure 12-15 J2C Java bean connection information for our application

The following is the list of input properties for non-managed connections:

– Connection class name: **com.ibm.connector2.ECIManagedConnectionFactory**.

– Connection URL: Provide the server address (URL) of the CICS ECI server.

– Server name: Provide the name of CICS server.

– Port Number: Provide the port number to be used to communicate with CICS TG.

– User name: Provide the user name for the connection.

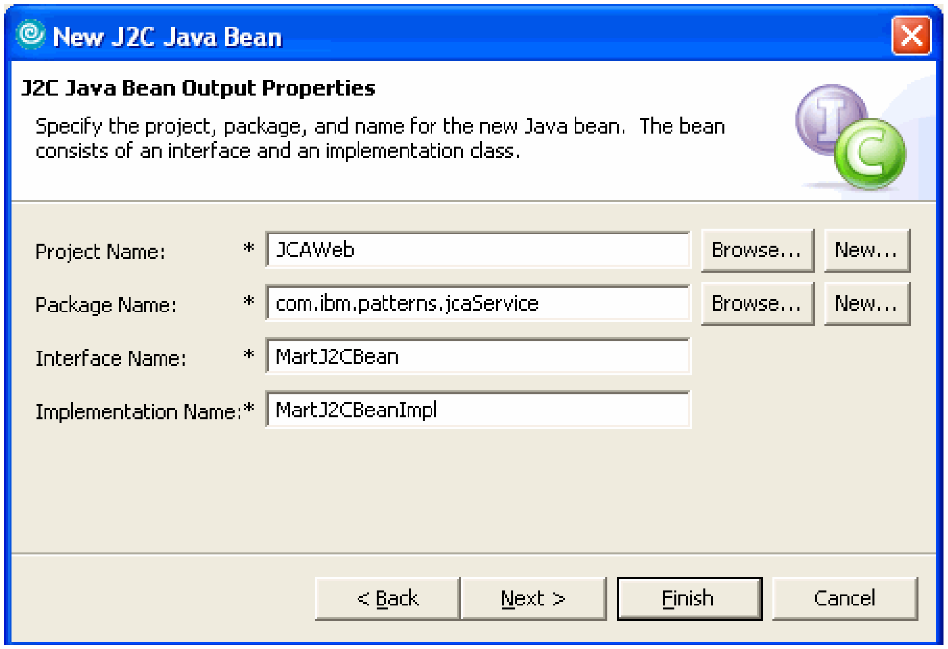
– Password: provide the password for the connection

To access the advanced CICS properties, click **Show Advanced,**

– Client security: Provide the client security class name.
– Server security: Provide the server security class name.
– Trace Level: Identify the level of information to be traced

Click **Next**.

5. The next screen (Figure 12-16) allows you to specify the output properties for the new J2C Java bean.



J2C Java Bean Output Properties

Specify the project, package, and name for the new Java bean. The bean consists of an interface and an implementation class.

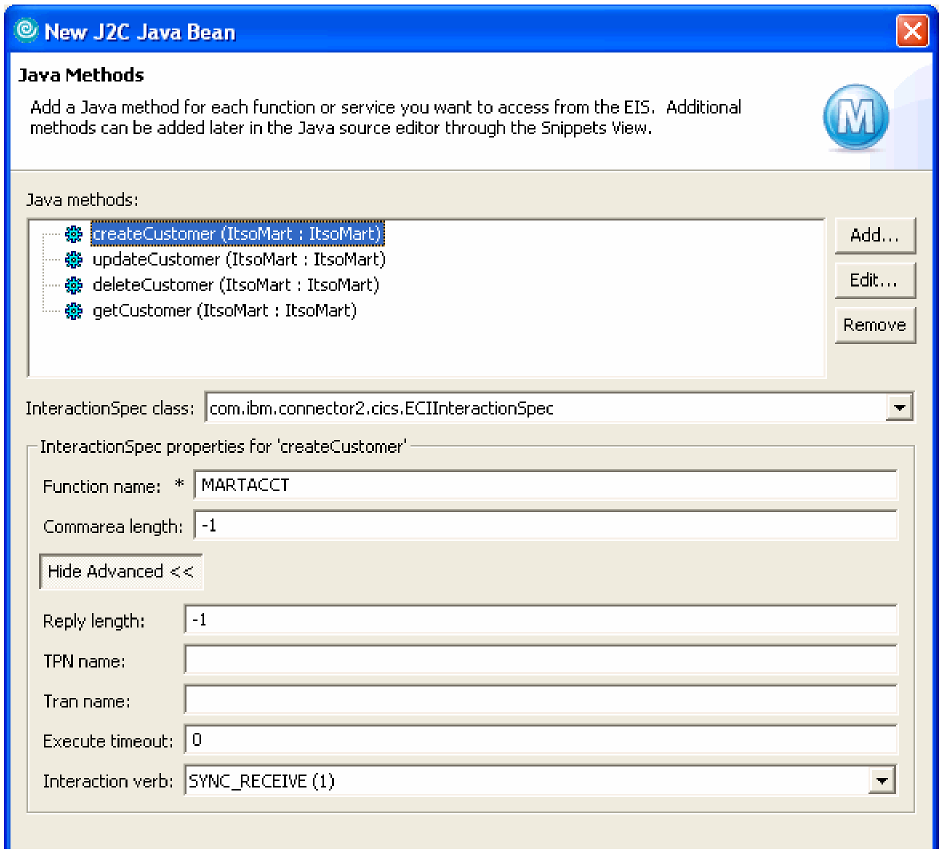
Project Name: * JCAWeb
Package Name: * com.ibm.patterns.jcaService
Interface Name: * MartJ2CBean
Implementation Name:* MartJ2CBeanImpl

Figure 12-16 J2C Bean Output properties panel

– Project name: **JCAWeb**
– Package name: **com.ibm.patterns.jcaService**
– Interface name: `MartJ2CBean`
– The implementation name will default to the interface name concatenated with `BeanImpl`.

Click **Next**.

6. The next screen (Figure 12-17 on page 419) allows you to select the Java methods for each function or service you want to access in the EIS. It will use the COBOL importer to map the data types between the COBOL source and the data in the Java methods.


New J2C Java Bean

Java Methods

Add a Java method for each function or service you want to access from the EIS. Additional methods can be added later in the Java source editor through the Snippets View.

Java methods:

- createCustomer (ItsoMart : ItsoMart)
- updateCustomer (ItsoMart : ItsoMart)
- deleteCustomer (ItsoMart : ItsoMart)
- getCustomer (ItsoMart : ItsoMart)

Add... Edit... Remove

InteractionSpec class: com.ibm.connector2.cics.ECIInteractionSpec

InteractionSpec properties for 'createCustomer'

Function name: * MARTACCT

Commarea length: -1

Hide Advanced <<

Reply length: -1

TPN name:

Tran name:

Execute timeout: 0

Interaction verb: SYNC_RECEIVE (1)

Figure 12-17 Java Methods created for our application

Repeat the following process four times to add these methods:

– **createCustomer**
– **deleteCustomer**
– **updateCustomer**
– **getCustomer**

a. Click **Add**.

b. Enter the method name and click **Next**.

c. Specify the input type by browsing to and selecting the data binding class. In each case for our sample we select **MartAccounts**.

d. Check the **Use input type for output** box.

e. Click **Finish**.

f. This will bring you back to the original screen where you can specify values related to the use of the method. For our sample, each method uses the following settings:

- interactionSpec class: **com.ibm.connector2.cics.ECIInteractionSpec**
- Function name: `MARTACCT`
- Commarea length : **-1**

Click **Next**.

7. The next screen contains deployment information. Leave the settings as they are and click **Finish**.

You can see the new Java bean in the Project Explorer view (Figure 12-18).

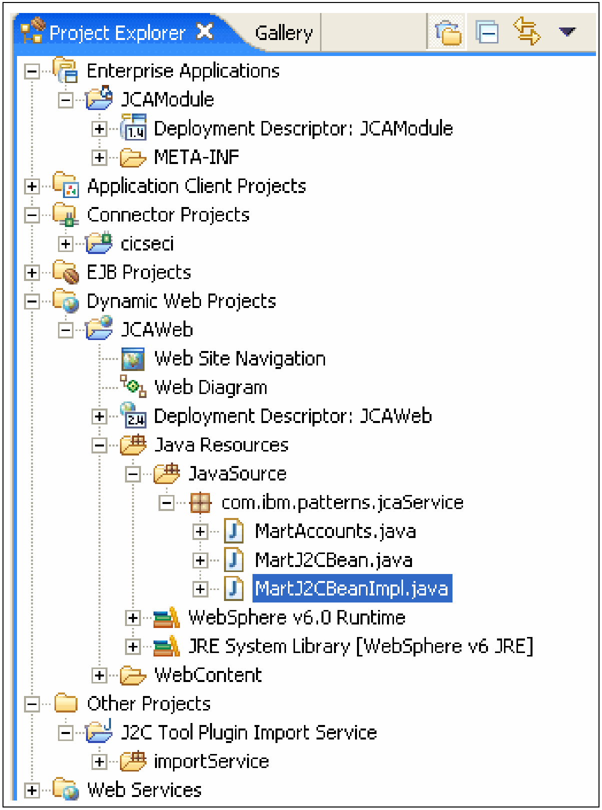


Figure 12-18 Project Explorer showing J2C Java bean

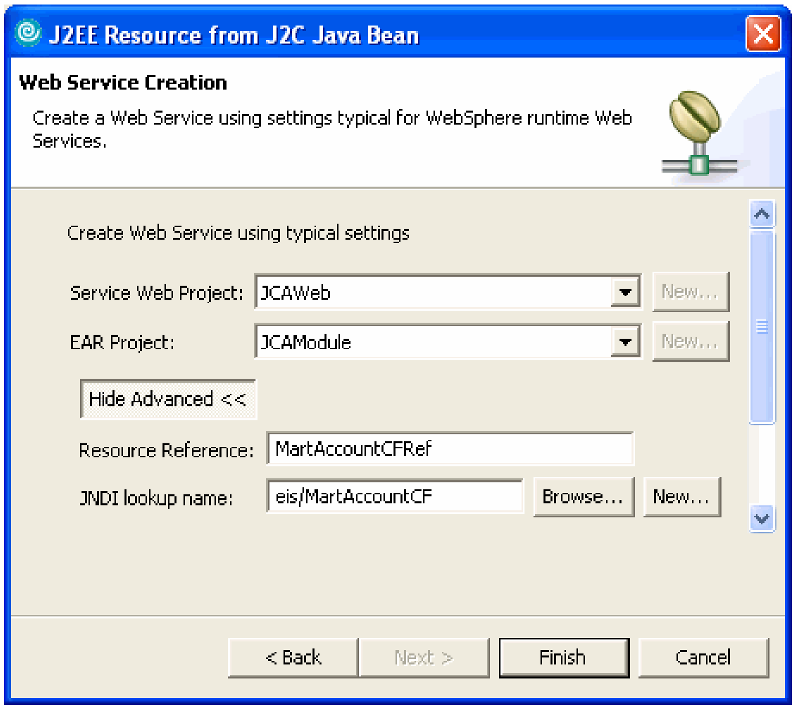
Step 3: Deploy the J2C Java bean as a Web service

This step deploys the J2C Java bean as a Web service.

1. On tool menu select **File** → **New** → **Other** → **J2C** (Figure 12-10 on page 411).
2. Expand J2C and select **Web page, Web Service, or EJB from J2C Java Bean**. Click **Next**,
3. In the next window provide the J2C bean Impl file name we created in the previous step:

   `/JCAWeb/JavaSource/com/ibm/patterns/jcaService/MartJ2CBeanImpl.java`

   Click **Next**.
4. Select **Web Service** as the deployment option.
5. Click **Advanced** and select **Configure resource Adapter Deployment.** Click **Next.** See Figure 12-19.


J2EE Resource from J2C Java Bean

Web Service Creation

Create a Web Service using settings typical for WebSphere runtime Web Services.

Create Web Service using typical settings

Service Web Project: JCAWeb

EAR Project: JCAModule

Hide Advanced <<

Resource Reference: MartAccountCFRef

JNDI lookup name: eis/MartAccountCF

Figure 12-19 Web Service Creation

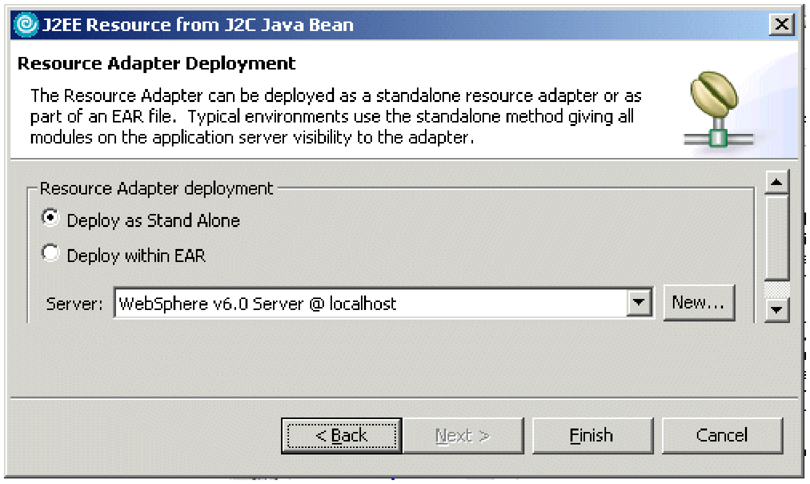
– Provide the Web project name (`JCAWeb`) and EAR project name (`JCAModule`).

– Click **Advanced** to view or modify the resource reference and JNDI lookup names. We will take the defaults.

Note: The resource reference name will update the Impl code, replacing the JNDI name with this name. It also create a resource reference in the deployment descriptor and will add the JNDI lookup name as the JNDI name to this reference name.

Click **Next**.

6. In the next screen (Figure 12-20) you can select the server to deploy to for testing and the deployment option.



The Resource Adapter can be deployed as a standalone resource adapter or as part of an EAR file. Typical environments use the standalone method giving all modules on the application server visibility to the adapter.

Resource Adapter deployment
- [✓ Deploy as Stand Alone]
- Deploy within EAR

Server: WebSphere v6.0 Server @ localhost

Figure 12-20 Resource adapter deployment options

– Select **Deploy as Stand Alone**.

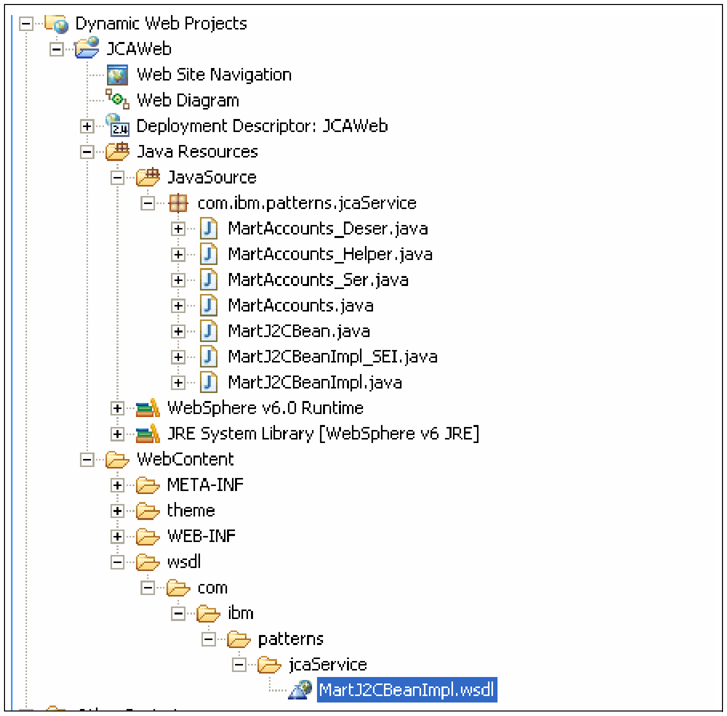
– We only have one unit test environment server defined so that server is selected by default.

7. Click **Finish**.

At the completion of this step, the J2C Java bean will be deployed as a Web service on the selected application server. The wizard will generate MartJ2CBeanImpl.wsdl in the WebContent /wsdl folder. In addition, the following files are generated:

- MartJ2CBeanImpl_SEI.java
- MartJ2CBean.java
- MartAccounts_Ser.java
- MartAccounts_Helper.java
- MartAccounts_Deser.java

All these files are in Project Explorer shown in Figure 12-21 on page 423.



```
Dynamic Web Projects
  JCAWeb
    Web Site Navigation
    Web Diagram
    Deployment Descriptor: JCAWeb
    Java Resources
      JavaSource
        com.ibm.patterns.jcaService
          MartAccounts_Deser.java
          MartAccounts_Helper.java
          MartAccounts_Ser.java
          MartAccounts.java
          MartJ2CBean.java
          MartJ2CBeanImpl_SEI.java
          MartJ2CBeanImpl.java
      WebSphere v6.0 Runtime
      JRE System Library [WebSphere v6 JRE]
    WebContent
      META-INF
      theme
      WEB-INF
      wsdl
        com
          ibm
            patterns
              jcaService
                MartJ2CBeanImpl.wsdl
```

Figure 12-21 Generated files for the Web service

You can browse the WSDL file by double-clicking on the file in the **Project Explorer** view. Rational Software Architect provides an editor especially suited to working with WSDL files.

Figure 12-22 on page 424 shows the relationship between service, bindings and port types defined in the WSDL for the service.

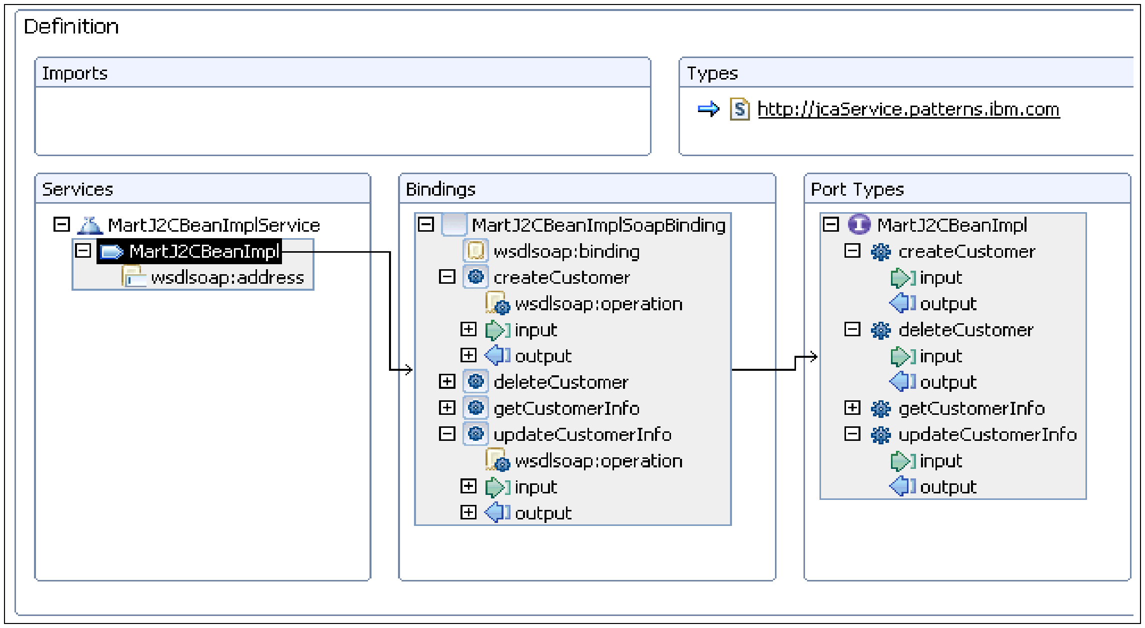


Figure 12-22 MartJ2CBeanImpl WSDL graph (generated)

Figure 12-23 shows the relationship between the port types and messages defined.

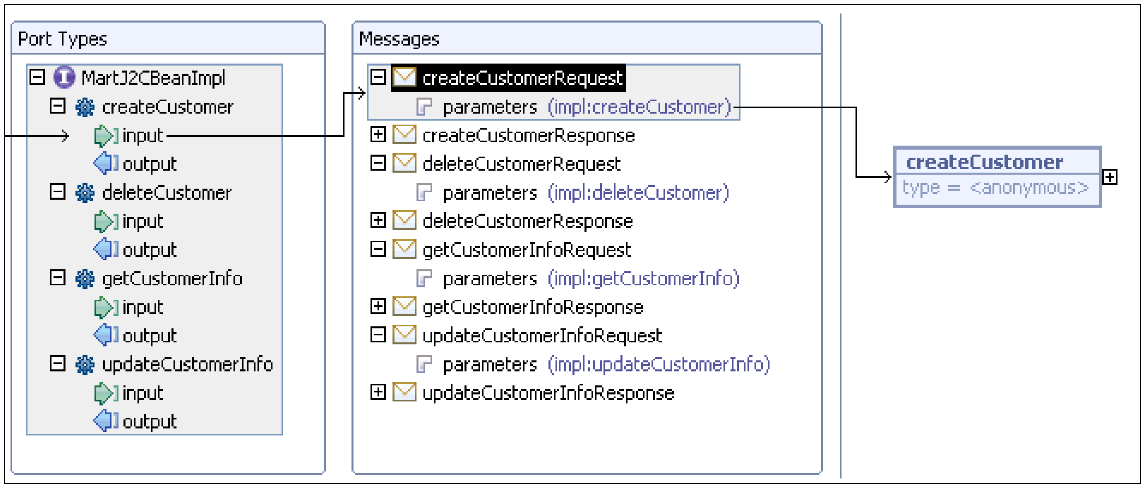


Figure 12-23 MartJ2CBeanImpl WSDL graph (generated)

Figure 12-24 shows the MartAccounts data binding message parts that are passed as arguments to each method.

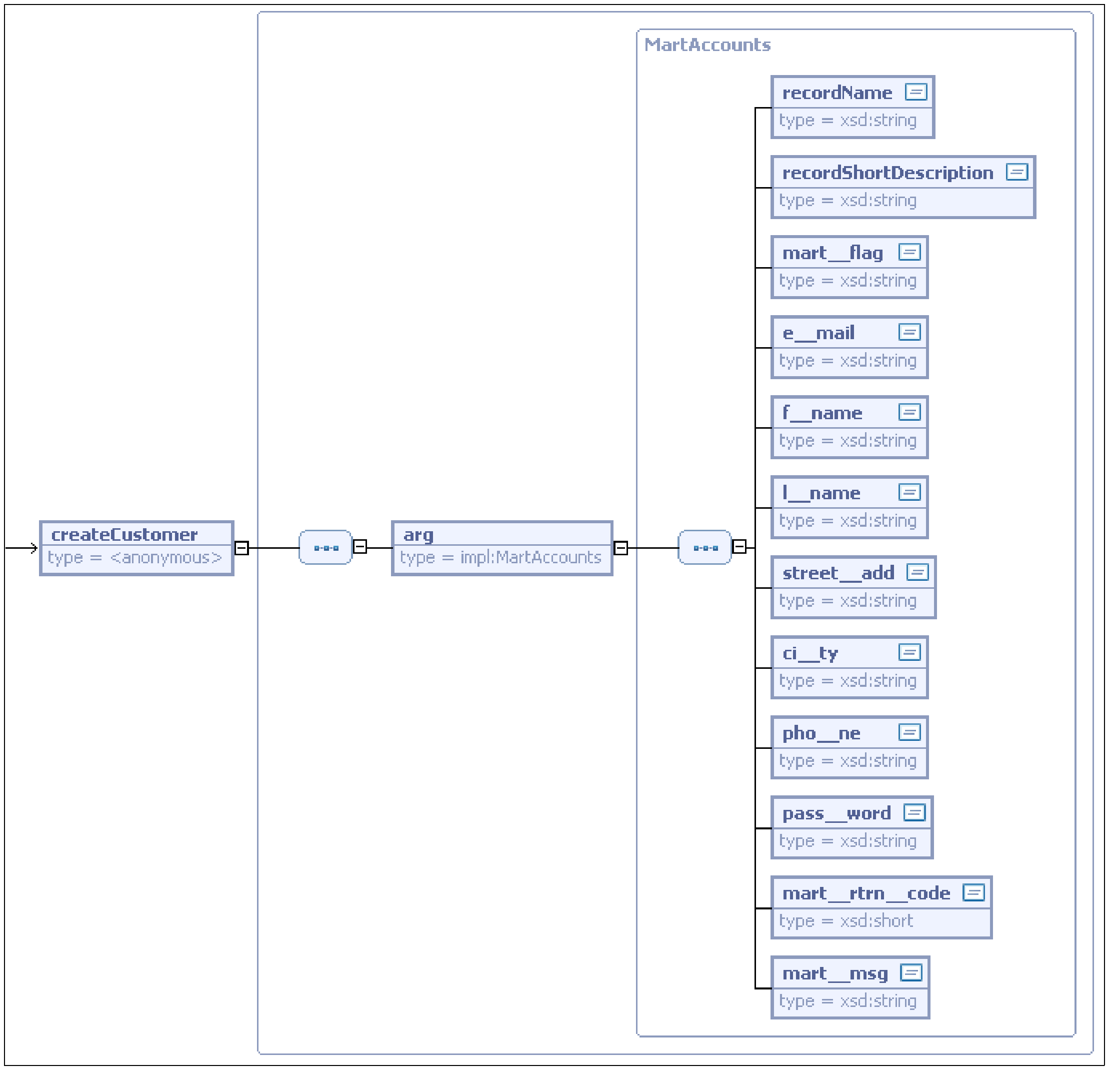


Figure 12-24 MartJ2CBeanImpl WSDL graph (generated)

This completes the process of creating the JCA application and the Web service.

12.4.3 Create the EJB Web service client

Note for users of the service integration bus: The Web service client is built using WSDL files that describe the Web service. This section assumes you are using the WSDL generated by the Web Service wizard in the previous section. However, in our runtime configuration, we plan to use the WebSphere Application Server service integration bus as an intermediate destination point for the Web service. During the bus configuration, new WSDL for this Web service will be created, pointing to the bus as the endpoint. This process is outlined in 12.5.3, “Configure the bus for the Web service” on page 438. If you are using the service integration bus, you should deploy the Web service, define it to the bus, and generate the new WSDL before generating the Web service client.

The process of generating a Web service client is detailed in , “Creating an EJB Web service client” on page 317. This section does not go into details on those steps already covered, but tell yous how the Web service client was generated for the JCA portion of the ITSOMart application.

To do this we completed the following tasks:

1. Create an EJB project
2. “Create a stateless session bean” on page 427
3. “Import the WSDL files” on page 427
4. “Create a namespace mapping file” on page 427
5. “Generate Web service client” on page 428

Create an EJB project

1. In the workbench, select **File** → **New** → **Other** → **EJB Project** in the tool bar.
2. Provide the name of the project, **JCAClient**. Click **Show Advanced**.
   – Make sure **EJB version is 2.1** is selected and the target server is WebSphere Application Server v6.0.
   – Check **Add module to an EAR project** and select the Processor EAR project.
   – Uncheck **Create an EJB Client JAR project to hold the client interfaces and classes**.
3. Click **Finish**.

Create a stateless session bean

1. In the workbench, select **File** → **New** → **Other** → **EJB** → **Enterprise Bean** in the tool bar.
2. Click **Next**.
   - Select **Session bean**.
   - EJB project: **JCAClient**
   - Bean name: **JCAWSClient**
   - Default Package: **com.ibm.patterns.jcaService**
3. Click **Next** and select **Local client view**.
4. Click **Finish**.

Import the WSDL files

The WSDL files need to be imported into the workspace. If you are using the service integration bus for Web services, the WSDL must be exported from the application server first. See “Export the new WSDL from the bus to generate the client” on page 440.

Before starting this process, create a new folder called `wsdl` in **JCAClient/ejbModule/META-INF**.

To import the MartJ2CBeanImplInboundService WSDL do the following:

1. In toolbar select **File** → **Import.**
2. In the dialog box, select **Zip file** and click **Next**.
3. In the next screen:
   a. Specify the location of the zip file containing the WSDL.
   b. Make sure the WSDL files are selected.
   c. Select the project to import the WSDL files into. We want to import the files to **JCAClient/ejbModule/META-INF/wsdl.**
   d. Click **OK**.
   e. Click **Finish**.

Create a namespace mapping file

Refer to “Create the namespace mapping files” on page 345 for information about creating namespace mapping files.

1. Under the META-INF folder, create a folder called **namespace mapping**, Create a new file **nsmappings.properties**, and click **OK.**
2. Determine the name spaces defined by the Web service. Open the service WSDL file, **ITSOMartESB.MartJ2CBeanImplInboundServiceService.wsdl**,

and examine the namespace definitions defined in the `<definitions>` tag, as in Example 12-13.

Example 12-13 Definitions attribute of a service WSDL file

```
<wsdl:definitions xmlns:impl="http://jcaService.patterns.ibm.com"
xmlns:intf="http://jcaService.patterns.ibm.com"
xmlns:sibusbinding="http://www.ibm.com/websphere/sib/webservices/IBM-PKHOODNode03Cell/ITSOMartE
SB/MartJ2CBeanImplInboundService/Binding" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:wsi="http://ws-i.org/profiles/basic/1.1/xsd" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.ibm.com/websphere/sib/webservices/IBM-PKHOODNode03Cell/ITSOMartESB/
Service">
```

3. From this file we can determine the value of the targetNamespace is:

   `http://www.ibm.com/websphere/sib/webservices/`*`IBM-PKHOODNode03Cell`*`/`*`ITSOMartESB`*`/Service`

   We can also determine the value of the sibusbinding namespace is:

   `http://www.ibm.com/websphere/sib/webservices/`*`IBM-PKHOODNode03Cell`*`/`*`ITSOMartESB`*`/MartJ2CBeanImplInboundService/Binding`

   Notice that these name spaces incorporate the cell and name of the service integration bus used.

4. Create a namespace mapping file to map these two name spaces to the Java package `com.ibm.patterns.jcaService`. Perform the following:

   We used the mappings shown in Example 12-14.

Example 12-14 Namespace mapping file

```
http\://www.ibm.com/websphere/sib/webservices/IBM-PKHOODNode03Cell/ITSOMartESB/Service=com.ibm.
patterns.jcaService
http\://www.ibm.com/websphere/sib/webservices/IBM-PKHOODNode03Cell/ITSOMartESB/MartJ2CBeanImplI
nboundService/Binding=com.ibm.patterns.jcaService
```

5. Save the changes to `nsmapping.properties`.

Generate Web service client

We are now ready to generate a Web service client for the JCA service. The input for this is the imported WSDL file. More information about generating Web service clients can be found in 10.5.2, “Creating a Web service from a session bean” on page 305.

1. Select **File** → **New** → **Other**. Then select **Web Services** → **Web Service Client** and click **Next**.

2. We want to create a Java proxy client, so ensure Client proxy type is set to **Java proxy** then click **Next**.
3. Select the WSDL file:

   /JCAClient/ejbModule/META-INF/wsdl/ITSOMartESB.MartJ2CBeanImplInboundServiceBindings.wsdl.

   Click **Next**.
4. In the next panel use the following values:
   - Client type: **EJB**
   - Client project: **JCAClient**
   - EAR project: **Processor**

   Click **Next**.

5.The next page allows security information to be specified. It also has a check box labelled **Define custom mappings for namespace to package**. Select this check box then click **Next**.

6. To import the namespace mappings file, click **Import**, expand n**amespace mapping**, highlight **nsmappings.properties**, and click **OK**.
7. Click **Finish** and the Web service client will be generated. Acknowledge any warning messages you receive during the Web service client generation.
8. You can confirm the Web service client has been generated by examining the [Service]Locator.java file, which in this case is **MartJ2CBeanImplInboundServiceLocator.java**. You will find this in the com.ibm.patterns.jcaService. Open this file and look for the following line of code which indicates the address of the ITSOMartService points to the service integration bus.

   ```
   private final java.lang.String JCAPort_address =
   "http://localhost:9080/wsgwsoaphttp1/soaphttpengine/ITSOMartESB/MartJ2CBean
   ImplInboundService/JCAPort";
   ```

We now have to implement the methods in the JCAWSClientBean.

1. Open the **JCAClient** project.
2. Navigate to the **/ejbModule/com.ibm.patterns.jcaService** package.
3. Open **JCAWSClientBean.java** and insert the following code in Example 12-15 on page 430 at the bottom of the bean. This method calls the proxy, which in turn finds the JCA Web service.
4. **Save** your changes.

Example 12-15 Add the following methods to JCAWSClientBean

```
public MartAccounts createMartAccount_CICS(MartAccounts input) throws
Exception{
      MartAccounts respOutput_forCreate = null;
      try{
         MartJ2CBeanImplProxy proxy = new MartJ2CBeanImplProxy();
         respOutput_forCreate = proxy.createCustomer(input);
      }catch (Exception e){
                  System.out.println("JCAWSClientBean -
createMartAccount_CICS failed" + e.toString());
               throw e;
      }
   return respOutput_forCreate;
   }



public MartAccounts getMartAccount_CICS(MartAccounts input) throws
Exception{
      MartAccounts respOutput_forGet = null;
      try{
         MartJ2CBeanImplProxy proxy = new MartJ2CBeanImplProxy();
         respOutput_forGet = proxy.getCustomerInfo(input);
      }catch (Exception e){

         System.out.println("JCAWSClientBean - getMartAccount_CICS failed"
+ e.toString());
               throw e;
      }
   return respOutput_forGet;
   }


public MartAccounts updateMartAccount_CICS(MartAccounts input) throws
Exception{
      MartAccounts respOutput_forUpdate = null;
      try{
         MartJ2CBeanImplProxy proxy = new MartJ2CBeanImplProxy();
         respOutput_forUpdate = proxy.updateCustomerInfo(input);
      }catch (Exception e){
                  System.out.println("JCAWSClientBean -
updateMartAccount_CICS failed" + e.toString());
               throw e;
      }
   return respOutput_forUpdate;
   }
```

```
public MartAccounts deleteMartAccount_CICS(MartAccounts input) throws
Exception{
      MartAccounts respOutput_forDelete = null;
      try{
         MartJ2CBeanImplProxy proxy = new MartJ2CBeanImplProxy();
         respOutput_forDelete = proxy.deleteCustomer(input);
      }catch (Exception e){
                  System.out.println("JCAWSClientBean -
deleteMartAccount_CICS failed" + e.toString());
               throw e;
      }
   return respOutput_forDelete;
   }
```

Attention: During this project we ran into a few problems using the early version of Rational Software Architect. We anticipate the fixes for these problems to be included in a near future Fix Pack. In the meantime, here are the problems that we ran into and the work-arounds we used.

- Problem 1: Change the componentNameLink value in bnd file.
  a. Open **JCAClient/ejbModule/META-INF/ibm-webservicesclient-bnd.xmi**.
  b. Replace the value ___SET_THIS_TO_ejb-name___ of componentNameLink to `JCAWSClient`.
- Problem 2: Change the componentNameLink value in ext file.
  a. Open **JCAClient/ejbModule/META-INF/ibm-webservicesclient-ext.xmi**.
  b. Replace the value ___SET_THIS_TO_ejb-name___ of componentNameLink to `JCAWSClient`.
- Problem 3: Add serializable statement in MartAccounts.java.
  a. Open **JCAClient/ejbModule/com.ibm.patterns.jcaService/MartAccounts.java**.
  b. Add `implements java.io.Serializable` to public class MartAccounts:

```
public class MartAccounts implements java.io.Serializable {
```

12.4.4 Integrate the JCA service client with Processor

Now we are ready to integrate the two components.

Processor.java calls the JCA service client.

- Processor.java can be found in **ProcessorEJB** EJB Project
- Package **com.ibm.patterns.serialProcess.**

Example 12-16 shows the snippet of code in Processor.java that invokes the Web service for the JCA call.

Example 12-16 Call to JCA in Processor.java

```
   private void createCustomer() {
      //Call to CreditRating application
      String creditRating = new ProcessorWebService().getCreditRating(customerDetails);
      if (isCreditSatisfactory(creditRating)) {
         //Credit is Satisfactory
         //Create customer in CRM and send Acceptance Mail

         ProcessorJCA.storeCustomerDetails_CICS(customerDetails);

         new MailService().sendActivationMail(customerDetails);
         System.out.println("Customer created successfully....");
      }
      else {
         //Credit rating is not satisfactory
         //Delete customer from Db2 and send rejection Email
         deleteCustomerfromDB();
         new MailService().sendRejectionEmail(customerDetails);
         System.out.println("The customer does not have satisfactory credit");
      }
   }

   private void deleteCustomer() {
      ProcessorJCA.deleteCustomerDetails_CICS(pkey);
      deleteCustomerfromDB();
      new MailService().sendDeletionEmail(pkey);
      System.out.println("Customer deleted successfully...");

   }

   private void updateCustomer() {

      ProcessorJCA.updateCustomerDetails_CICS(pkey);

      new MailService().sendUpdationEmail(customerDetails);
      System.out.println("Customer details updated successfully...");

   }
}
```

12.5 Runtime guidelines for JCA applications

In this section we discuss some of the more pertinent points about the runtime configuration of WebSphere Application Server V6, CICS Transaction Gateway 6.0. and IBM CICS required for the ITSOMart application.

Configuring the runtime requires the following actions:

- Install and configure the local CICS Transaction Gateway.
- Install and configure the CICS ECI resource adapter on WebSphere Application Server.
- Configure WebSphere Application Server and the service integration bus.

This process assumes that the CICS Transaction Server and CICS application are up and running.

12.5.1 CICS Transaction Gateway

There are three typical CICS Transaction Gateway runtime configurations for the ITSOMart application:

- Local CICS Transaction Gateway
- Remote CICS Transaction Gateway
- Remote CICS Transaction Gateway on z/OS

The ITSOMart scenario uses a remote CICS Transaction Gateway running on a distributed platform. It connects to the CICS server using TCP.

Local CICS Transaction Gateway

When designing a solution that includes CICS Transaction Gateway and WebSphere Application Server, there are several topologies to consider. Figure 12-25 on page 434 illustrates the use of a local connection between the application running in WebSphere Application Server and CICS Transaction Gateway. The CICS Transaction Gateway resides on the same machine as WebSphere Application Server. The application is using the CICSECI resource adapter.

In local mode the CICS Transaction Gateway Java code is loaded into the application server JVM. The client daemon is invoked through the CICS Transaction Gateway JNI layer. The client daemon then invokes an enterprise-tier CICS program using ECI calls. The underlying protocols that the ECI call uses to invoke CICS applications are TCP/IP, APPC, or TCP62 (APPC over TCP/IP protocol). The ECI call can be routed to any CICS server.

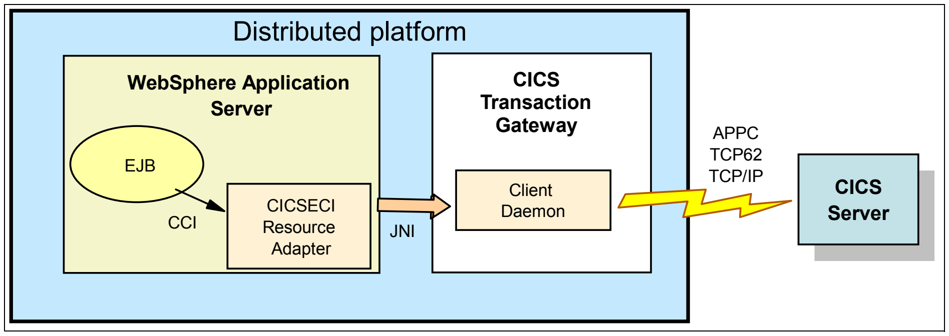


Figure 12-25 Local CICS Transaction Gateway

Remote CICS Transaction Gateway

In a remote topology (Figure 12-26), WebSphere Application Server and CICS Transaction Gateway reside on separate machines and use TCP/IP for communication. A gateway daemon on the CICS Transaction Gateway machine provides the interface to the WebSphere Application Server and the client daemon.

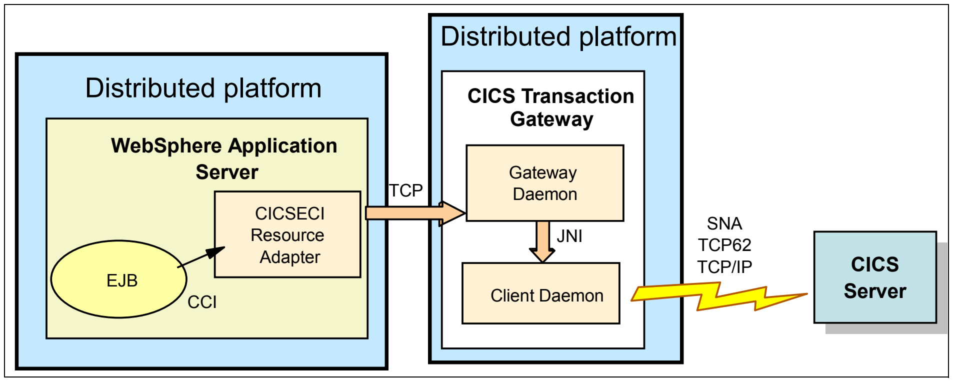


Figure 12-26 Remote CICS Transaction Gateway

This is the topology used in the sample ITSOMart application.

Remote CICS Transaction Gateway on z/OS

Another common topology, shown in Figure 12-27 on page 435, features a remote CICS Transaction Gateway on z/OS. A transaction is invoked from a distributed environment remotely using the TCP protocol. The protocol between an application server and a CICS Transaction Gateway can be either TCP or SSL. The CICS Transaction Gateway invokes a CICS enterprise-tier program using the External CICS Interface (EXCI). EXCI is an interface that allows a z/OS address space, such as a native Java process running under the z/OS UNIX service, to invoke a CICS transaction. The CICS program is invoked using a mirror transaction that resides in the same CICS region as the CICS application. EXCI calls can be routed to any CICS region within a sysplex, which means that the CICS application can reside in a different z/OS system from the CICS Transaction Gateway.

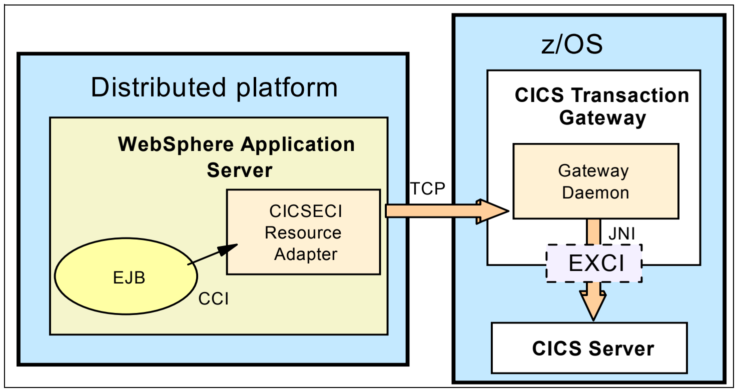


Figure 12-27 Remote CICS Transaction Gateway on z/OS

12.5.2 WebSphere Application Server V6 configuration

This section describes the runtime setup for the application in WebSphere Application Server V6. It consists of the following steps:

- Installing a CICS ECI resource adapter
- “Configuring a connection factory” on page 437
- Configure the bus

Installing a CICS ECI resource adapter

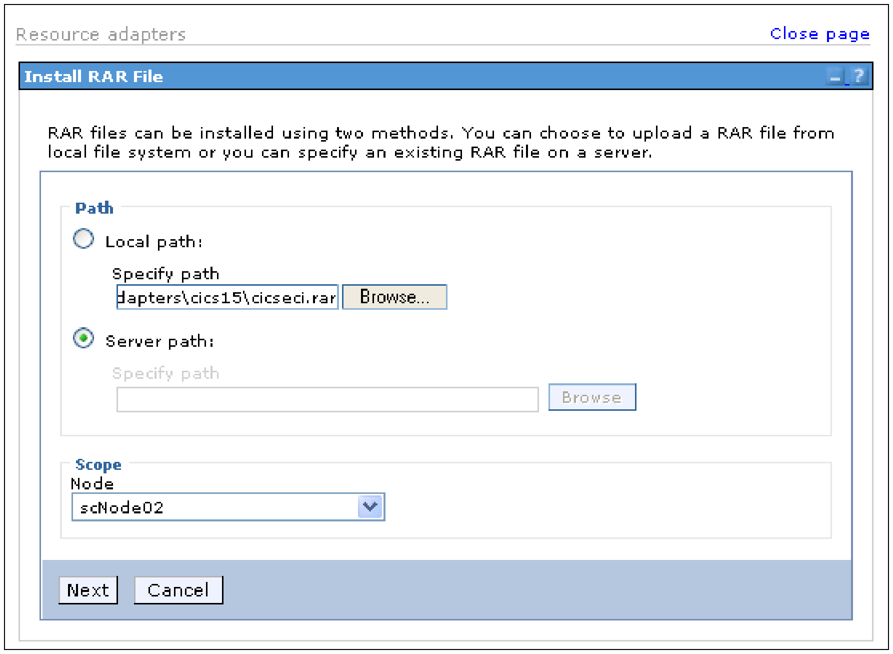
To install the CICS ECI resource adapter in WebSphere Application Server:

1. Locate the **CICS ECI J2C resource adapter archive (RAR)** file.

   The CICS ECI adapter we used is available with the Rational Application Developer and Rational Software Architect in the following location:

   ```
   <Rational_Install>\Resource Adapters\cics15\cicseci.rar
   ```

2. Open the WebSphere administrative console and select **Resources** → **Resource Adapters** in the navigation tree.
3. Click the **Install RAR** button to install the .rar file into WebSphere.
4. Browse to the cicseci.rar file and click **Next**. See Figure 12-28 on page 436.



RAR files can be installed using two methods. You can choose to upload a RAR file from local file system or you can specify an existing RAR file on a server.

dapters\cics15\cicseci.rar

scNode02

Figure 12-28 Installing the RAR file

5. In the next panel, leave all the fields blank so that WebSphere uses the values from the RAR file deployment descriptor. Click **OK**. This installs the resource adapter file. You will see an entry in the resource adapter list called ECIResourceAdapter. Next we add the resource adapter definition.
6. Click **New**.
7. In the Resource Adapters Configuration form:
   - Set the Name to **cicseci**.
   - Select the following archive path:

     ```
     ${CONNECTOR_INSTALL_ROOT}/cicseci.rar
     ```

   Click **OK.**

Create a J2C authentication data entry

The authentication data entry will be used to access the CICS system.

To add a J2C authentication data entry, do the following tasks:

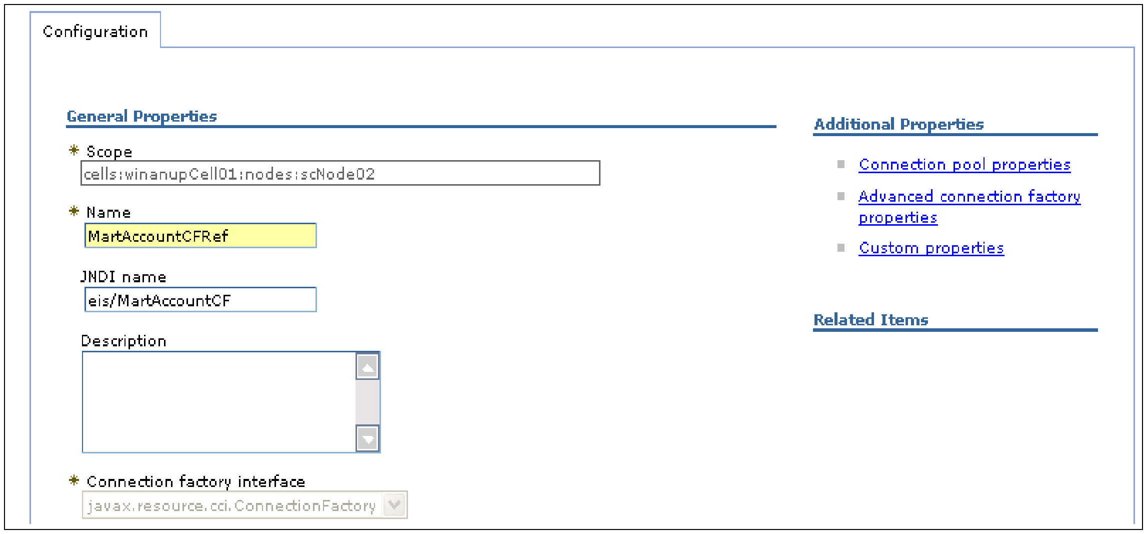
1. Select **Security** → **Global Security.**
2. Under the Authentication section expand **JAAS configuration** and select **J2C Authentication data/**

3. Click **New.**
4. Provide an alias for use when defining this to a resource, and supply a user ID and password with the authority to access CICS.

Configuring a connection factory

To add a connection factory for the application, perform the following tasks:

1. Select **Resources** → **Resource Adapters** in the navigation tree.
2. Click the new **cicseci** in the resource adapter list.
3. Under Additional Properties click **J2C Connection factories**.
4. In the J2C Connection factories form click **New**.
5. In the J2C Connection factories General Properties form, shown in Figure 12-29, provide the following information:



* Scope: cells:winanupCell01:nodes:scNode02

* Name: MartAccountCFRef

JNDI name: eis/MartAccountCF

Description:

* Connection factory interface: javax.resource.cci.ConnectionFactory

Figure 12-29 J2C Connection Factories General Properties

   – Name: `MartAccountCFRef`
   – JNDI name: `eis/MartAccountCF`
   – Select the authentication data entry you created in the drop-down list for the component-managed authentication alias.

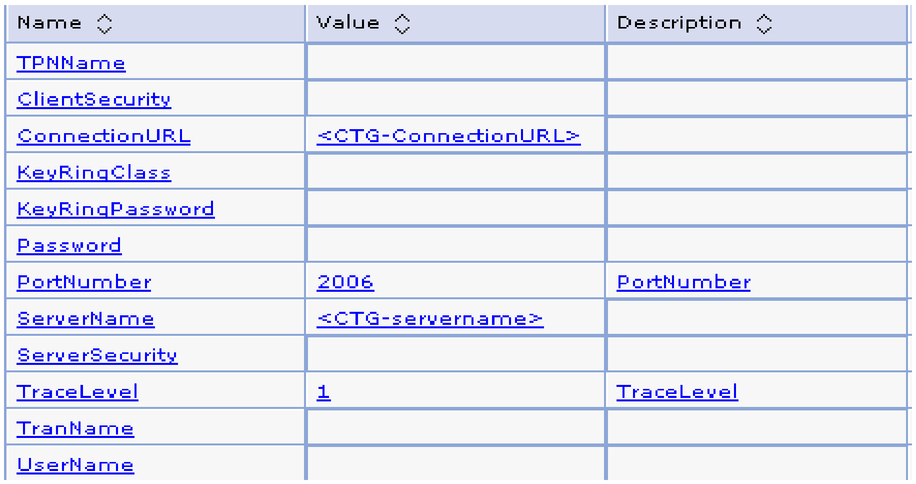
   Click **OK**.

6. Click the new **MartAccountCFRef** in the J2C Connection Factories list.
7. Under Additional Properties click **Custom Properties** and complete the required connection properties for CICS:

– Set the ServerName to the CICS TG server name.

– Set the fully qualified host name (ConnectionURL) to the machine CICS TG is installed.

– Set the PortNumber for CICS TG. The default is 2006.



Name	Value	Description
TPNName		
ClientSecurity		
ConnectionURL	<CTG-ConnectionURL>	
KeyRingClass		
KeyRingPassword		
Password		
PortNumber	2006	PortNumber
ServerName	<CTG-servername>	
ServerSecurity		
TraceLevel	1	TraceLevel
TranName		
UserName		

Figure 12-30 Connection factory Properties

8. Save your changes.

12.5.3 Configure the bus for the Web service

Note: Our scenario assumes that you have done the following:

- Created a bus called ITSOMartBus and added the application server as a member.

  For information about creating the bus and adding the members, see "Create a service integration bus" on page 463.

- Installed the Web services support for the bus. For information about how to do this, see "Install Web services support for the bus" on page 464.

This section is very much a repeat of the information found in 10.7, "Runtime guidelines for Web services" on page 331 so we will not go into detail, but just briefly touch on what needs to be done for the ITSOMart application.

Create an endpoint listener

Create an endpoint listener for the application server using the following settings. The instructions for this are in “Create an endpoint listener” on page 335.

- Name: `SOAPHTTPChannel2`.
- URL root:

  `http://localhost:9080/ws2soaphttp`
- WSDL serving HTTP URL root:

  `http://localhost:9080/JCAWeb/wsdl`
- Add a connection property and select **ITSOMartBus** as the bus.

Create an outbound service

Use the instructions in 10.7.4, “Create the outbound services” on page 337 to create an outbound service with the following settings:

- WSDL location type: **URL**
- WSDL location:

  ```
  http://localhost:9080/JCAWeb/wsdl/com/ibm/patterns/jcaService/MartJ2CBeanIm
  pl.wsdl
  ```
- Service: **MartJ2CBeanImplService**
- Port: **MartJ2CBeanImpl**

Create the inbound service

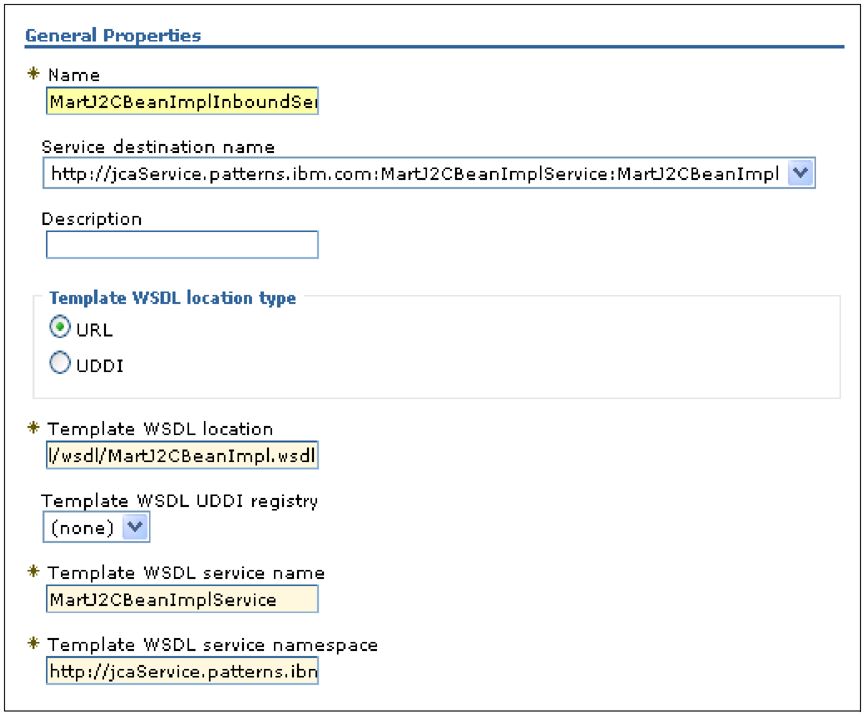
Use the instructions in 10.7.5, “Create the inbound services” on page 340 to create an inbound service with the following settings:

- Service destination name:

  `http://jcaService.patterns.ibm.com:MartJ2CBeanImplService:MartJ2CBeanImpl`
- Template type: **URL**
- Template WSDL location:

  ```
  http://localhost:9080/JCAWeb/services/MartJ2CBeanImpl/wsdl/MartJ2CBeanImpl.
  wsdl
  ```
- Inbound service name: **MartJ2CBeanImplInboundService**
- Endpoint listener: **SOAPHTTPChannel2**
- Port name: **JCAPort**

The results can be seen in Figure 12-31 on page 440.


General Properties

* Name
MartJ2CBeanImplInboundSe

Service destination name
http://jcaService.patterns.ibm.com:MartJ2CBeanImplService:MartJ2CBeanImpl

Description

Template WSDL location type
(●) URL
( ) UDDI

* Template WSDL location
l/wsdl/MartJ2CBeanImpl.wsdl

Template WSDL UDDI registry
(none)

* Template WSDL service name
MartJ2CBeanImplService

* Template WSDL service namespace
http://jcaService.patterns.ibn

Figure 12-31 Inbound service for JCA

Export the new WSDL from the bus to generate the client

Use the instructions in 10.7.6, “Generate and export new WSDL for the services” on page 343 to export the WSDL for use with Rational Software Architect. You can export the files by opening **MartJ2CBeanImplInboundService** in the inbound services list for the bus.

Use the instructions found in “Create the EJB Web service client” on page 426 to generate the Web client.

12.5.4 Setting up the CICS application

In this scenario, WebSphere Application Server connects to the CICS server using the CICS Transaction Gateway. The application developer and deployer only see this interface. The actual CICS server could be located on a mainframe z/OS environment or on an Encina-based TxSeries distributed environment.

To set up the ITSOMart CICS application, perform the following tasks:

1. Create a new region called MARTREGN on the CICS server.

2. Compile the MARTACCT.ccp file and add MARTACCT.ibmcob file in the bin directory for the CICS region.
3. Start the region.
4. Test the program using the Local CICSTERM program.

12.6 System management guidelines for JCA

This section provides system management guidelines for J2EE Connector-enabled applications and the underlying CICS environment. We look at the following topics:

- 12.6.1, “Logging and tracing” on page 441
- 12.6.2, “Performance monitoring and tuning” on page 442
- 12.6.3, “Scalability and availability considerations” on page 444
- 12.6.4, “Security considerations” on page 448

12.6.1 Logging and tracing

It is often helpful to examine log and trace files when your application experiences J2EE Connector errors or problems.

Application logging

It is always important for applications to record their activity to a logging facility. When you write a log to the standard output file or standard error file, the application server will record it to the corresponding log files.

Connection factory trace

Connection factory classes can be traced using the WebSphere Application Server trace service. The trace level can also be set as a connection factory property in WebSphere administrative console.

Connection factory tracing is often not particularly helpful when debugging the CICS interaction. CICS TG tracing is usually the better option.

CICS TG trace

CICS Transaction Gateway (CICS TG) trace records detail activities of the CICS TG gateway daemon, such as the processing of ECI requests from clients. The four levels of tracing are:

- stack tracing
- standard tracing
- debug tracing
- JNI tracing

JNI tracing is usually the most useful of all traces. It is the JNI level between the Java CICS TG and the native EXCI or CICS client.

The application can enable tracing programmatically. It can also be enabled dynamically in the Gateway daemon using the TCPAdmin protocol handler, or statically as a start option. CICS TG trace is recorded in the standard output file or standard error file.

External CICS Interface trace

The External CICS Interface (EXCI) provides a programming interface for the non-CICS address space to invoke CICS programs. CICS TG utilizes EXCI to communicate with the CICS program. The CICS Transaction Gateway writes trace entries to the EXCI trace when it issues an EXCI request. The trace entries in a dump can be printed using standard z/OS utilities (GTF).

CICS trace

CICS Transaction Server provides a facility for recording CICS activity. In CICS for MVS™, there are three destinations for trace entries: Internal trace, auxiliary trace, and generalized trace facility (GTF).

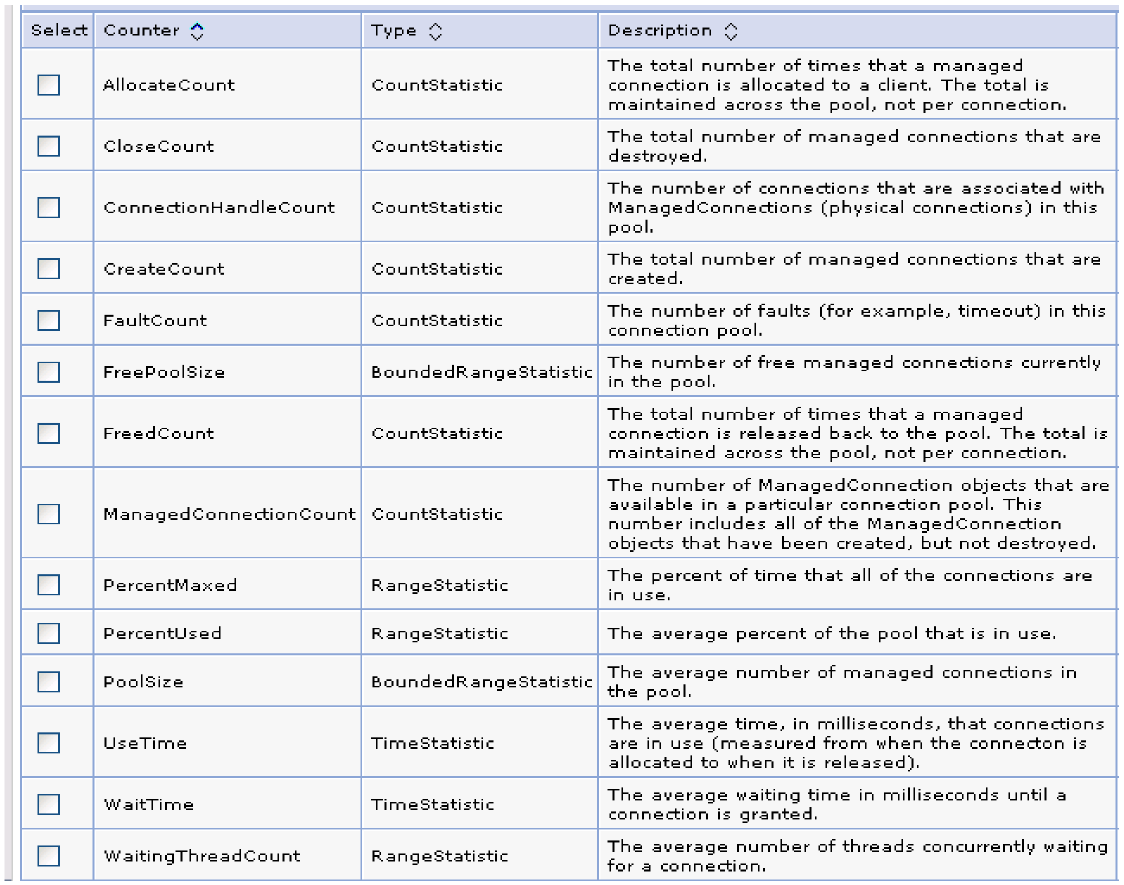
12.6.2 Performance monitoring and tuning

In this section we briefly look at performance monitoring and tuning for J2EE Connectors in WebSphere Application Server V6.0.

Using Tivoli Performance Viewer

The Tivoli Performance Viewer is a graphical performance monitor for WebSphere Application Server V6. This can be accessed through the WebSphere administrative console by selecting **Monitoring and Tuning** → **Performance Viewer**. You can use the Performance Viewer to retrieve performance data from application servers. Data is collected continuously by application servers and retrieved as needed from within the Viewer.

You can regulate the impact of data collection by modifying the PMI settings. These settings can be found viewed and modified by selecting **Monitoring and Tuning** → **Performance Monitoring Infrastructure (PMI)**. The counters in the JCA Connection Pools category provides information about J2EE connectors, such as the number of managed connections (ManagedConnections) and the number of connection handles (connections). Figure 12-32 shows the PMI options available under JCA connection pools.



Select	Counter	Type	Description
☐	AllocateCount	CountStatistic	The total number of times that a managed connection is allocated to a client. The total is maintained across the pool, not per connection.
☐	CloseCount	CountStatistic	The total number of managed connections that are destroyed.
☐	ConnectionHandleCount	CountStatistic	The number of connections that are associated with ManagedConnections (physical connections) in this pool.
☐	CreateCount	CountStatistic	The total number of managed connections that are created.
☐	FaultCount	CountStatistic	The number of faults (for example, timeout) in this connection pool.
☐	FreePoolSize	BoundedRangeStatistic	The number of free managed connections currently in the pool.
☐	FreedCount	CountStatistic	The total number of times that a managed connection is released back to the pool. The total is maintained across the pool, not per connection.
☐	ManagedConnectionCount	CountStatistic	The number of ManagedConnection objects that are available in a particular connection pool. This number includes all of the ManagedConnection objects that have been created, but not destroyed.
☐	PercentMaxed	RangeStatistic	The percent of time that all of the connections are in use.
☐	PercentUsed	RangeStatistic	The average percent of the pool that is in use.
☐	PoolSize	BoundedRangeStatistic	The average number of managed connections in the pool.
☐	UseTime	TimeStatistic	The average time, in milliseconds, that connections are in use (measured from when the connecton is allocated to when it is released).
☐	WaitTime	TimeStatistic	The average waiting time in milliseconds until a connection is granted.
☐	WaitingThreadCount	RangeStatistic	The average number of threads concurrently waiting for a connection.

Figure 12-32 Tivoli Performance Viewer JCA connection pool options

Tuning connection pooling properties

There are several parameters you can set to optimize connection pooling properties using the WebSphere administrative console:

- Connection timeout

  This is the number of milliseconds after which a connection request is determined to have timed out and a ResourceAllocationException is thrown. The wait might be necessary if the maximum value of connections has been reached (MaxConnections). This value has no meaning if the maximum connections property has not been set.

If the connection timeout is set to a very small number such as 1, the ResourceAllocationException is thrown almost immediately after the pool manager determines that the maximum number of connections has been used. If the connection timeout is set to 0, the pool manager waits until a connection can be allocated. In other words, it waits until the number of connections falls below the maximum connections.

- Maximum connections

  This is the maximum number of managed connections that can be created by a particular ManagedConnectionFactory. After this number is reached, no new connections are created, and either the requester waits or the ResourceAllocationException is thrown. If maximum connections is set to 0, the number of connections can grow indefinitely. Maximum connections must be larger than minimum connections.

- Minimum connections

  The minimum number of managed connections to maintain. If this number is reached, the garbage collector will not discard any managed connections. Note that if the actual number of connections is lower than the value specified by the minimum connections settings, no attempt will be made to increase the number of connections to the minimum. Minimum connections must be less than or equal to maximum connections.

- Reap time

  This is the number of seconds between runs of the garbage collector. The garbage collector discards all connections that have been unused for the value specified by the unused timeout.

  To disable the garbage collector, set the reap time to 0. Another way to disable the garbage collector is to set the unused timeout to 0.

- Unused timeout

  Number of milliseconds after which an unused connection is discarded. Setting this value to 0 disables the garbage collector.

12.6.3 Scalability and availability considerations

As shown in Figure 12-33 on page 445, there are several scalability and availability options when using J2EE connectors to access CICS enterprise applications:

- EJB workload management provided by WebSphere Application Server.
- Inbound CICS TG requests (TCP or HTTP) can be workload managed using various methods.

- CICS requests (ECI or EPI) can be workload managed using CICS scalability technologies.

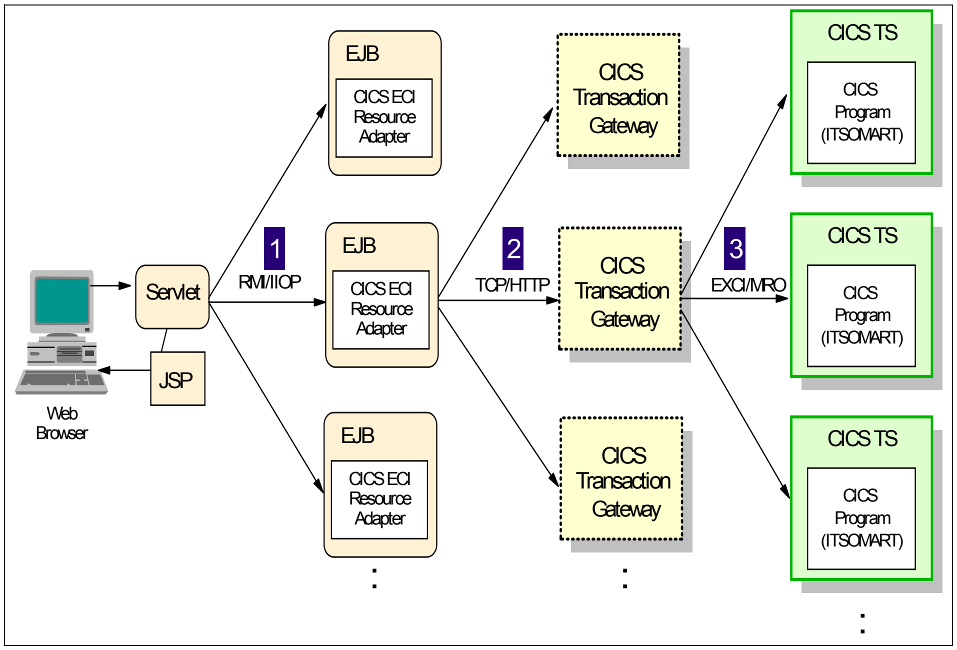


Figure 12-33 Scalability options

EJB workload management

EJBs deployed in WebSphere Application Server V6 can take advantage of the workload management (WLM) facility for EJBs. In WebSphere Application Server, workload management for EJBs is enabled automatically when clusters are created in a cell. There is no need for a special configuration to enable it.

Workload management of CICS TG requests

This section describes scalability and availability options between EJBs in a distributed environment (Windows or UNIX) and the CICS TG in a zSeries® environment. An ECI request from a J2EE connector resource adapter goes to the CICS TG on zSeries as an HTTP or TCP request.

WLM across LPARs

Options for implementing workload managing CICS TG for z/OS requests across LPARs include:

- IBM Load Balancer

  Load Balancer is an IBM solution that provides an advanced IP-level workload-balancing mechanism. The function is provided as a component of IBM WebSphere Edge Server. It can provide workload balancing for any TCP or UDP protocol, including HTTP requests to a Web server or TCP packets to an application such as the CICS Transaction Gateway. Load Balancer can be used to perform load balancing of inbound requests to CICS TG for z/OS.

- DNS connection optimization

  DNS connection optimization allows workloads to be distributed across multiple z/OS images. DNS connection optimization balances IP connections in a z/OS sysplex IP domain, by dynamically updating the z/OS DNS server database based on feedback from MVS WLM about the health of the registered applications. This is sometimes referred to as dynamic DNS, although this feature merely refers to the dynamic update function of the z/OS DNS server.

- Sysplex distributor

  Sysplex distributor is implemented in z/OS V2.10 and offers major enhancements to TCP/IP workload management in a sysplex. Balancing is enabled by using a single-cluster IP address, which routes packets onto multiple nodes. Sysplex distributor provides for close integration with the MVS WLM policy agent and service level agreements (SLAs) in making the routing decisions. This is different from Load Balancer because it has to poll the WLM advisor on z/OS to update its routing information. The cluster IP address is actually a VIPA (virtual IP address) and so can be dynamically routed to another z/OS LPAR in the sysplex. This allows for failover of the cluster address.

WLM within an LPAR

TCP/IP port sharing provides a simple way of workload balancing HTTP requests across a group of cloned address spaces running in the same z/OS image. For our purposes these could be cloned CICS regions, CICS TG Java gateway applications, or Web servers. To enable port sharing, the address spaces are configured to listen on the same TCP/IP port number, and the SHAREPORT parameter is specified in the TCP/IP profile. As incoming client connections arrive for this port, TCP/IP will distribute them across the address spaces that are listening on the shared port. TCP/IP will select the address space with the least number of connections (both active and in the backlog) at the time that the incoming client connection request is received. This allows you to do workload balancing for incoming HTTP requests across several cloned address spaces.

The workload balancing is based entirely on the number of IP connections, and so does not take into account the individual health or capacity of any given CICS region. However, it does provide a very simple means of providing failover and workload balancing across multiple regions within an LPAR. The TCP/IP Port Sharing function is provided by the Communication Server for z/OS.

Workload management of CICS requests

This section describes scalability options within a CICS world. The following functions provide workload management of requests from the CICS TG to a zSeries CICS region.

WLM from CICS Client daemon

When the CICS TG resides in a distributed environment, the CICS Client daemon, also sitting in the distributed environment, will kick off the CICS transaction residing on zSeries. The CICS Client daemon provides a workload management function for load balancing ECI requests. Options are provided that allow you to balance work across CICS regions using either a round-robin technique or a weighted distribution.

The CICS Client daemon also provides the ability to detect failed regions, and provides a configurable timeout period to check the status of regions that have previously failed. It does not, however, provide any form of performance agent for feedback on the status of the CICS regions. Thus, it is best viewed as a means of removing a single-point-of-failure in a listener region. Note that the CICS Client Daemon Workload Manager is not available with the CICS Client daemon on AIX® and Solaris™, but the exit on which it is based is provided, enabling you to implement your own customized workload manager.

WLM from CICS TG for z/OS

Workload management functions that are applicable when CICS Transaction Gateway resides in zSeries include:

- External CICS Interface (EXCI)

  EXCI provides a programming interface for non-CICS address space programs to invoke CICS programs. CICS TG utilizes EXCI to communicate with the CICS program. CICS TG provides a user-replaceable module called DFHXCURM to perform basic load balancing. It allows the destination CICS APPLID on and EXCI call to be altered and various retryable errors to be handled. This allows for basic workload balancing of EXCI calls, based on a simple availability check to be performed in this exit before the EXCI call is sent to the CICS system.

- CICS multi-region operation

  CICS multi-region operation (MRO) enables CICS systems that are running in the same MVS image, or in the same MVS sysplex, to communicate with each other. MRO does not support communication between a CICS system and a non-CICS system such as IMS. MRO is a widely used technique that is a central part of CICS scalability.

- CICS distributed program link

  CICS distributed program link (DPL) enables CICS application programs to run programs residing in other CICS regions by shipping program-control LINK requests. An application can be written without regard for the location of the requested programs. It simply uses program-control LINK commands in the usual way. Entries in the CICS program definition tables allow the system programmer to specify that the named program is not in the local region (s the *client* region), but in a remote region (*server* region).

  An ECI request from CICS TG in either zSeries or a distributed environment comes to an enterprise CICS program as a DPL request. The request can be routed to a CICS program that resides in any CICS region in a sysplex.

12.6.4 Security considerations

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

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

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

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

- Container-managed sign-on

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

- Component-managed sign-on

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

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

WebSphere Application Server V6.0 supports component-managed sign-on (Option C in the J2EE Connector Architecture Specification) which requires the component to pass user ID and password credentials through the ConnectionSpec to CICS. If the credentials in the ConnectionSpec are not set, then the credentials in the ManagedConnectionFactory are used to authenticate to CICS. WebSphere Application Server V6.0 also supports container-managed sign-on with the use of a user ID and password credential (Option A in the J2EE Connector Architecture Specification).

This section provides some security guidelines for J2EE Connector-enabled applications and the underlying CICS environment. We look at the following topics:

- Signing on to the enterprise tier
- SSL encryption support
- CICS security

Signing on to the enterprise tier

The J2EE Connector Architecture provides security contacts for an application component. A user ID and password can be specified by the application component, or it can be specified in the deployment descriptor when the security contract is managed by the EJB container.

Authentication can be performed against the Resource Access Control Facility (RACF®) using user ID and password authentication in the CICS TG for z/OS, by setting the variable `AUTH_USERID_PASSWORD=YES` in the ctgstart script.

If the CICS TG runs on a distributed platform it is possible to use the ESIRequest to verify user IDs and passwords with the destination CICS region. On all platforms, it is also possible to use SSL client certificates to identify the authenticity of the CICS TG.

The authorization attributed to the server program running in CICS is always based on a user ID. The Java client program should obtain this user ID (and password) from the Web user, and flow it with every ECI or EPI call. The ATTACHSEC setting on the CONNECTION between CICS and the CICS TG will determine how CICS will use the user ID flowed on the client's ECI and EPI calls.

As shown in Figure 12-34, steps for a typical sign-on scenario using CICS TG for z/OS are:

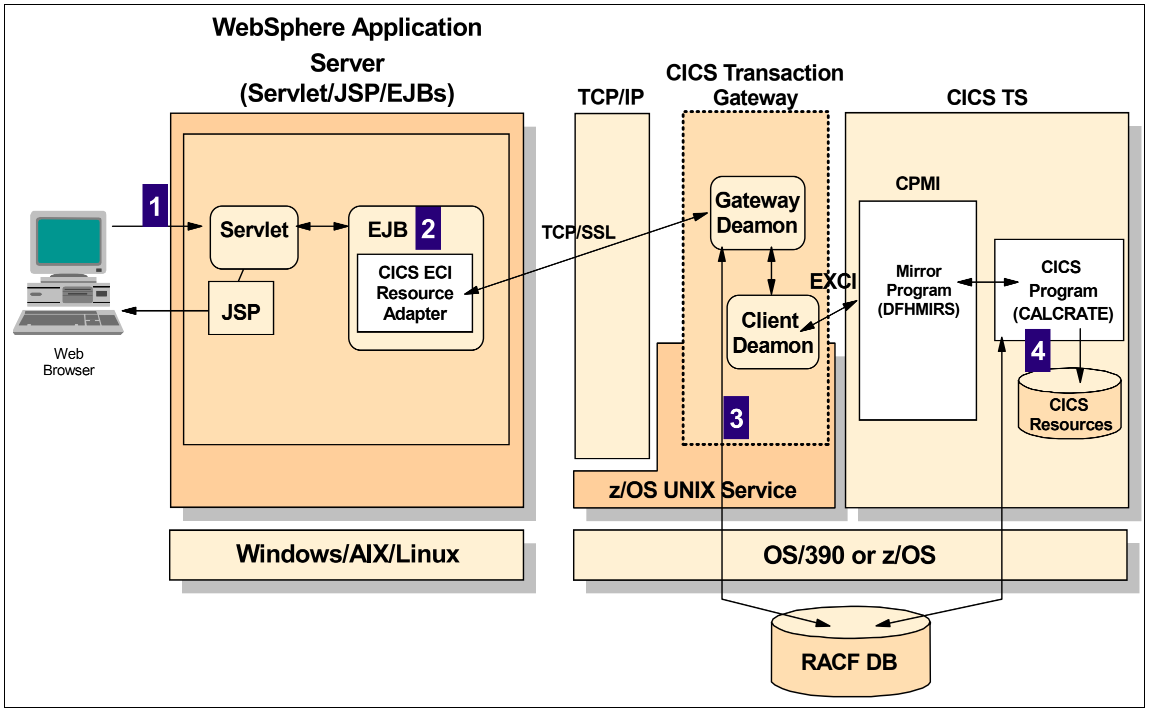


Figure 12-34 Sign-on scenario using CICS TG on a mainframe

1. An end user enters a user ID and password from a browser.
2. The user ID and password are set in the ConnectionSpec by the application component running in the application server.
3. The CICS TG for z/OS performs an authentication using RACF DB.

4. At the enterprise CICS region, the user ID is checked using RACF DB again, and authorization is done based on the CICS resource definition.

SSL encryption support

The client application connects to the gateway daemon using TCP/IP sockets (TCP). A secure version using SSL is also available. As an alternative to SSL, the CICS TG security exit can be used to support your own encryption/decryption procedure between the Java application and the CICS TG server.

CICS security

CICS uses the z/OS System Authorization Facility (SAF) to route authorization requests to an external security manager (ESM) to perform all its security checks. Any suitable ESM could be used, but because the IBM Resource Access Control Facility (RACF) product is the most commonly used ESM, we refer to RACF when discussing CICS external security. For complete information about CICS security, refer to the *CICS RACF Security Guide*, SC33-1701.

To support any CICS security checking, the appropriate security profiles must first be defined in RACF for all the users, groups, and resources you want to protect. Security is then enabled within a CICS region using the SEC parameter in the System Initialization Table (SIT). If `SEC=YES` is specified, CICS external security is enabled, and CICS then uses the SIT parameters XAPPC, XCMD, XDB2, XDCT, XFCT, XJCT, XLT, XPCT, XPPT, XPSB, XTRAN, XTST, and XUSER to further control security. If `SEC=NO` is specified, no security checking is performed, and users have unrestricted access.

For each CICS region that uses security, you are required to have two special-purpose user IDs, the *default user ID* and the *region user ID*. Because CICS tasks always run under a user ID, the default user ID is used when users do not explicitly sign on (and in a few other special instances). Thus, the default user ID should be given very low authorization levels. The default user ID is specified on the `DFLTUSER SIT` parameter. The region user ID is the user ID under which the CICS job itself runs. The region user IDt is by definition a powerful user ID and is also used in determining if connected CICS systems are equivalent.

Authentication of CICS users is the responsibility of RACF. Users can either be authenticated with the traditional mechanism of user ID and passwords (CICS sign-on), or, more recently, by the use of SSL client certificates when using CICS Web support or CICS CORBA client support. Having authenticated the user, CICS can apply two levels of authorization to a transaction. The first is *transaction security*, sometimes referred to as attach-time or transaction-attach security. The next level of security is *resource security* and applies to CICS-controlled resources such as programs, files, or queues used by the transaction.

Within an application, further authorization can be controlled by CICS. The CICS system programming command can be protected by the facilities of *command security*, and the issuing of requests on an interconnected CICS system can be controlled using intercommunication security. Lastly, *surrogate user security* can also be used to authorize an authenticated user to perform actions on behalf of a different user.

For more details on CICS security, refer to the publication *Securing Web Access to CICS*, SG24-5756.

12.7 For more information

For more information, see:

- *WebSphere Application Server Version 6.0 Information Center*, available at:

  `http://publib.boulder.ibm.com/infocenter/ws60help/index.jsp`

- J2EE Connector Architecture Specification Version 1.5

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

- *WebSphere Application Server V6 System Management and Configuration Handbook,* SG24-6451
- *Patterns: Self-Service Application Solutions Using WebSphere V5.0,* SG24-6591

A

Sample application install summary

This appendix contains the instructions needed to import the ITSOMart sample application into Rational Software Architect or Rational Application Developer and to configure the WebSphere Application Server V6 unit test environment to run the sample. The recommended release of either Rational product is 6.0.0.1 with the J2EE Connector Tools Feature installed.

Description of application files

The application contains the following EAR files:

- OrderSystemV6.ear
- Processor.ear
- CreditCheck.ear
- JCAModule.ear
- MailService.ear
- XlateToXML.ear
- DeliverySystem.ear
- RoutingMediationsEJBEAR.ear
- AggregatorEJBEAR.ear

Figure A-1 shows an overview of the ITSOMart sample application.

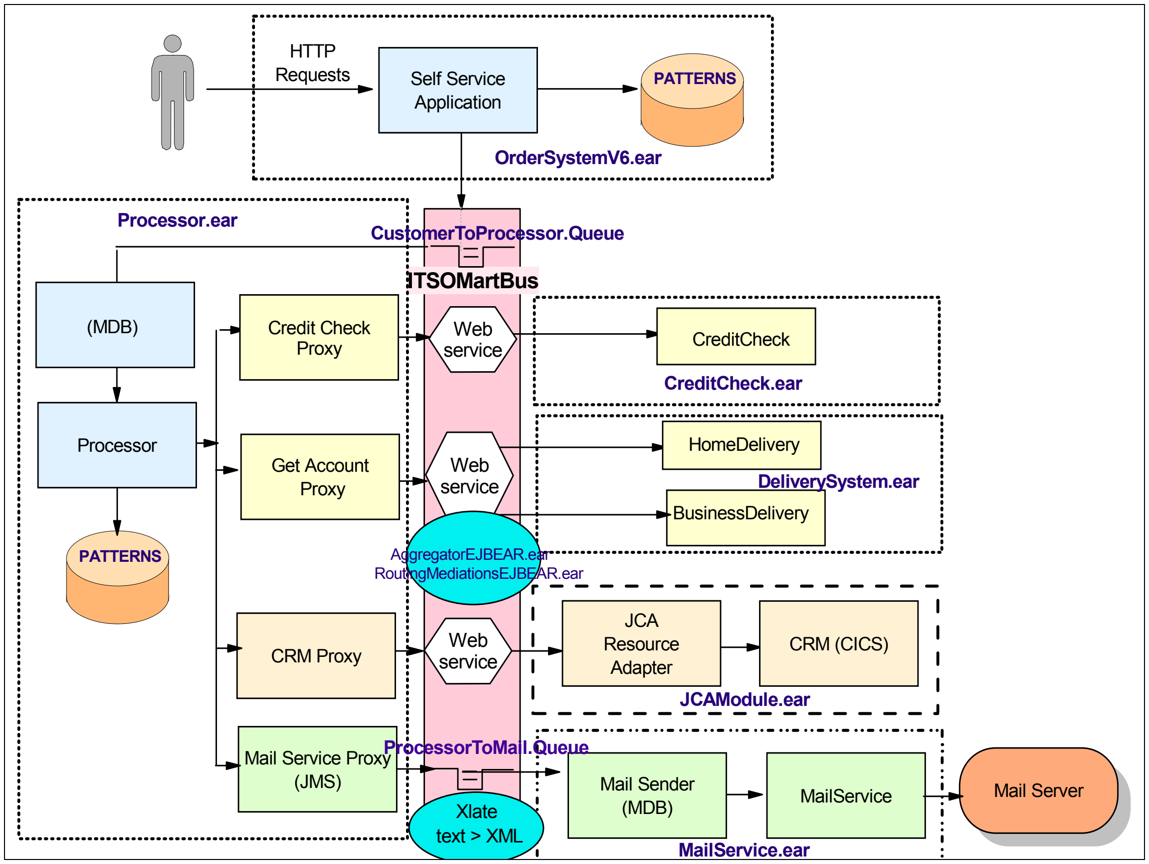


Figure A-1 ITSOMart runtime configuration

Tip: Installation on Windows-based systems might run into problems with reaching the maximum length of 256 characters for a directory path (URI)

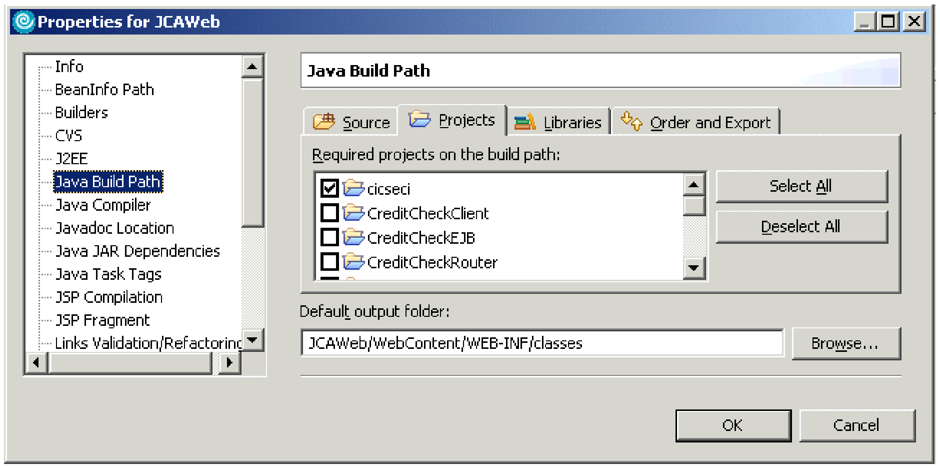
Work around: Create another profile under `C:/profiles/` and ensure that you install the Web services support using the newly created profile. Refer to *Rational Application Developer V6 Programming Guide,* SG24-6449 for details.

Import the source files to the workbench

In order to prepare the application for runtime, you need to have access to the application files from Rational Application Developer or Rational Software Architect.

Make sure you have the J2EE Connector Tools optional feature installed and that your service level is 6.0.0.1.

1. Unzip SG246680.zip into a temporary directory.
2. Open the development tool and switch to the J2EE perspective.
3. Import each EAR file included by doing the following:
   a. Select **File** → **Import**.
   b. Select **EAR file** and click **Next**.
   c. Browse to the first .ear file in the temporary directory and click **Open**.
   d. Click **Finish**.
4. Repeat step 3 for each of the EAR files.
5. In the Project Explorer, under Dynamic Web Projects, find **JCAWeb**. Right-click it and select **Properties**.
6. Click **Java Build Path** in the left side of the window, and in the right. Click the **Projects** tab. See Figure A-2 on page 456.
7. Check the box to the left of **cicseci** and click **OK**.



Required projects on the build path:

- [✓] cicseci
- [ ] CreditCheckClient
- [ ] CreditCheckEJB
- [ ] CreditCheckRouter

Default output folder: JCAWeb/WebContent/WEB-INF/classes

Figure A-2 JCAWeb Java build path settings

Note that you still have an error in the Problems view that says `CreditCheckProxy can not be resolved or is not a type`. This problem will be resolved later in "CreditCheck application" on page 472 when the WSDL file is copied to the project and a new client is generated. You can also see a few warning messages that you can ignore.

Runtime preparation

The sample application can be deployed to a WebSphere Application Server V6 application server, or to a WebSphere Application Server V6 test environment in Rational Application Developer or Rational Software Architect.

In both instances, the runtime environment is configured using the WebSphere administrative console. The difference will be in the method of deployment and in the location of the runtime files for WebSphere.

This assumes a standalone application server. If you are using Network Deployment, you might see some slight differences in the administrative console menu and in the range of options on the configuration pages.

In these instructions, we assume the following. You might need to make adjustments based on your installation.

- That you have a running standalone application server named `server1` using port `9080` for access to the Web container.

In the unit test environment, make sure the server has the type **Base, Express, or unmanaged Network Deployment server**. You can see this by double-clicking the server in the Servers view. The server configuration opens and you see the Server type box under the Network Deployment section.

The only time this matters with the sample is when you are installing the Web services support. In a distributed environment, you must use DB2 for the SDO repository. This is not difficult, but for this sample, it is best not to complicate matters.

- That you have opened the administrative console.

> **Test environment:** To get to the administrative console, do the following:
>
> 1. Switch to the J2EE perspective and select the **Servers** view.
> 2. Right-click the WebSphere Application Server v6.0 entry, and select **Start**. The Console view opens and you can see the messages from the start up of the server.
> 3. When you see the `Open for e-business` message, switch back to the Servers view. Right-click the server and select **Run administrative console.**

- That you are using a Windows system. If you are not using Windows most of these instructions will still work as documented but you may need to refer to the WebSphere InfoCenter for platform-specific information such as command locations or extensions.
- File locations will be denoted with <*was_install*>. For example, if you have WebSphere Application Server installed, <*was_install*> will look similar to `C:\WebSphere\AppServer`. In the Rational test environment, it might look similar to `C:\Rational\RSA\runtimes\base_v6`.

When dealing with relatively new WebSphere features such as the bus, we go into detail on how to perform these tasks. For other tasks that have remained much the same from previous versions of WebSphere such as creating data sources, we simply give you the information you need to populate those items.

Configuring the data source and creating the database

The application uses a customer database called PATTERNS. This database holds the customer registration information entered by the customer. The Patterns database used by the application can be implemented uding a DB2 database or a Cloudscape™ database.

Using a DB2 database

The advantage of using DB2 for the sample is the availability of the DB2 administrative tools. In particular, the Control Center is useful for displaying the database tables and querying the table entries. This allows you to easily see if the application is storing the customer information in the table. It also makes it easy to delete entries.

Create the database

Use these steps to create the database:

1. Copy the PATTERNS.sql file to a temporary location, for example `C:\temp`. This file has been included in the zip file you download. It is also located in located in `<workspace>\OrderSystemEJB\Scripts\Data`.
2. Open DB2 command window and enter the information in Example A-1.

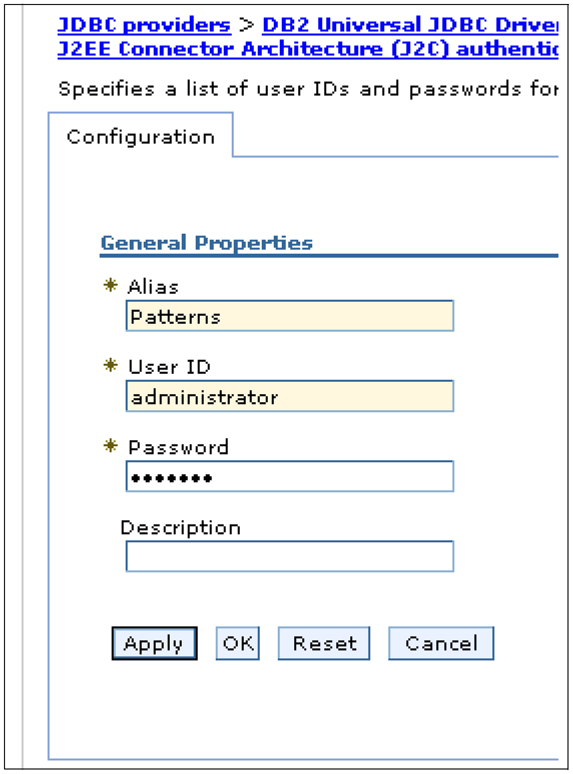
Example: A-1 Creating the database

```
cd C:\temp

db2cmd

db2 create db patterns
db2 connect to patterns
db2 -tvf PATTERNS.sql
db2 connect reset
```

Create the data source

Use the administrative console to create the following items.

1. Create a J2C authentication entry with the user ID / password authorized to access the database. See Figure A-3 on page 459.


JDBC providers > DB2 Universal JDBC Driver
J2EE Connector Architecture (J2C) authentic

Specifies a list of user IDs and passwords for

Configuration

General Properties

* Alias
Patterns

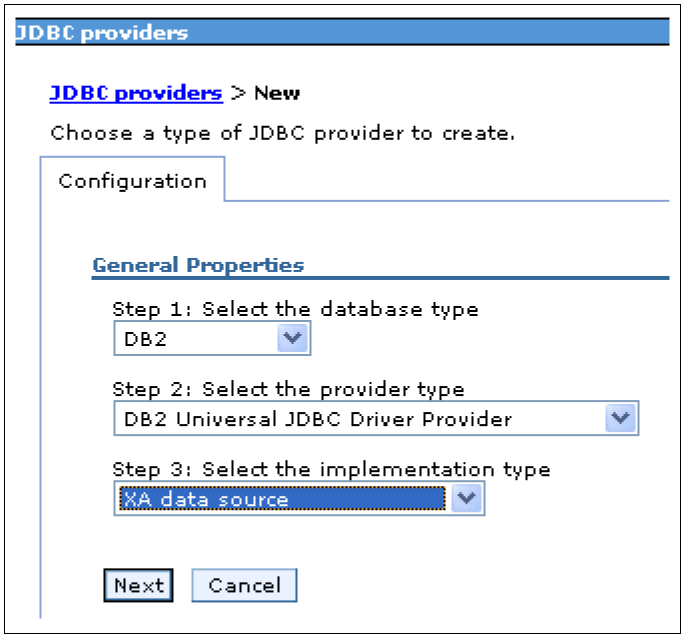
* User ID
administrator

* Password
•••••••

Description

Figure A-3 J2C authentication entry for the Patters database

2. Create a JDBC provider with the following properties, as in Figure A-4 on page 460.


JDBC providers

JDBC providers > New

Choose a type of JDBC provider to create.

Configuration

General Properties

Step 1: Select the database type

DB2

Step 2: Select the provider type

DB2 Universal JDBC Driver Provider

Step 3: Select the implementation type

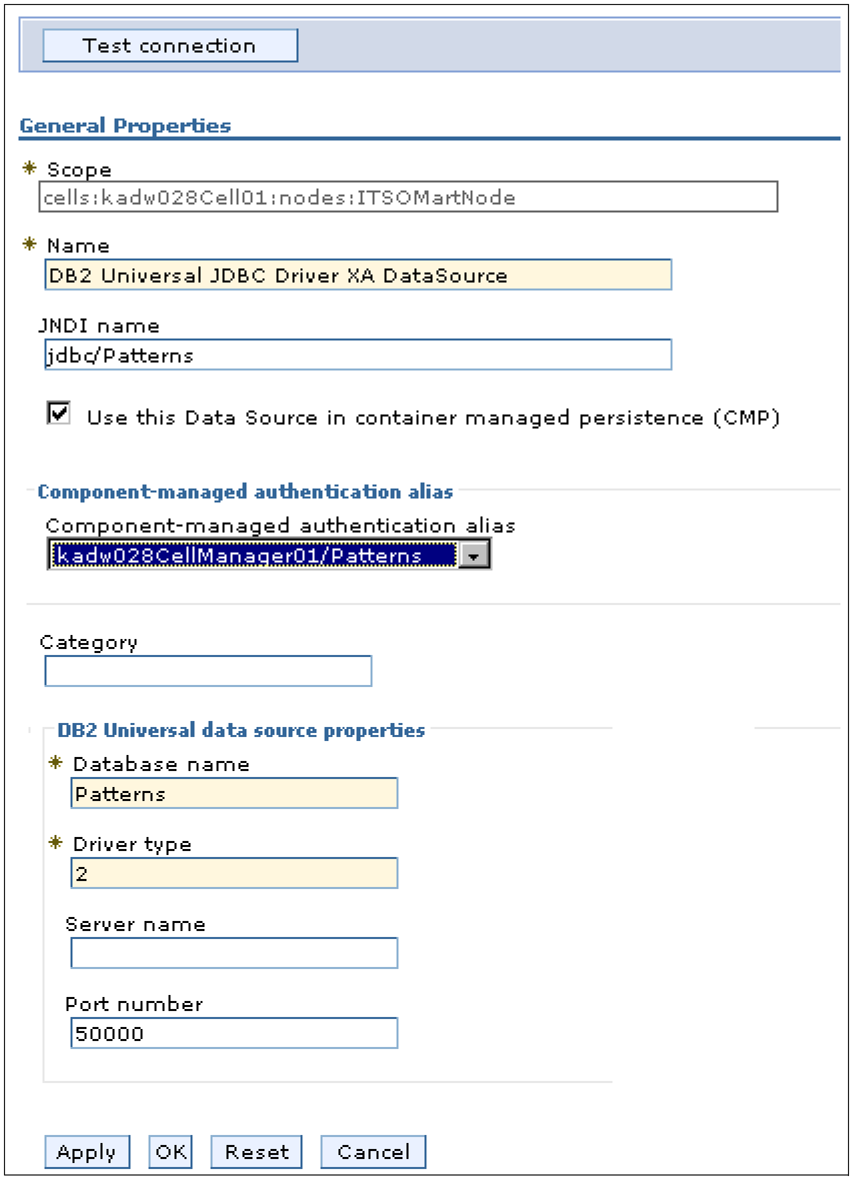
XA data source

Next Cancel

Figure A-4 JDBC driver for the Patterns database

> **Note:** The default configuration for this DB2 driver uses environment variables to specify the location of the various driver files. Be sure that you have set the following environment variable (select **Environment** → **WebSphere variables**)
>
> - DB2UNIVERSAL_JDBC_DRIVER_PATH to the proper value, for example, `c:\sqllib\java`.

3. Create a data source with the following properties, as in Figure A-5 on page 461.



General Properties

* Scope
cells:kadw028Cell01:nodes:ITSOMartNode

* Name
DB2 Universal JDBC Driver XA DataSource

JNDI name
jdbc/Patterns

☑ Use this Data Source in container managed persistence (CMP)

Component-managed authentication alias

Component-managed authentication alias
kadw028CellManager01/Patterns

Category

DB2 Universal data source properties

* Database name
Patterns

* Driver type
2

Server name

Port number
50000

Figure A-5 Data source for the Patterns Database

4. Test the connection to make sure the access is working.

Using a Cloudscape database

Using a Cloudscape database eliminates the need for access to a DB2 server. The sample includes a ready-to-use Cloudscape database that you can simply copy into your WebSphere Application Server files. Cloudscape databases can be viewed and manipulated using the cview.bat tool in *<was_install>*\cloudscape\bin\embedded directory.

Create the database

The Cloudscape database, PATTERNS.zip is included in the sample zip file.

1. Create a new folder in the *<was_install>*\cloudscape directory called PATTERNS.
2. Unzip this file into the *<was_install>*\cloudscape\PATTERNS directory. Figure A-6 shows the directory structure when unzipped into the directory structure for Rational Software Architect test server environment.

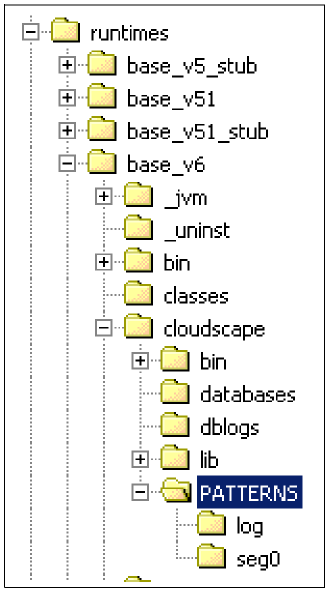
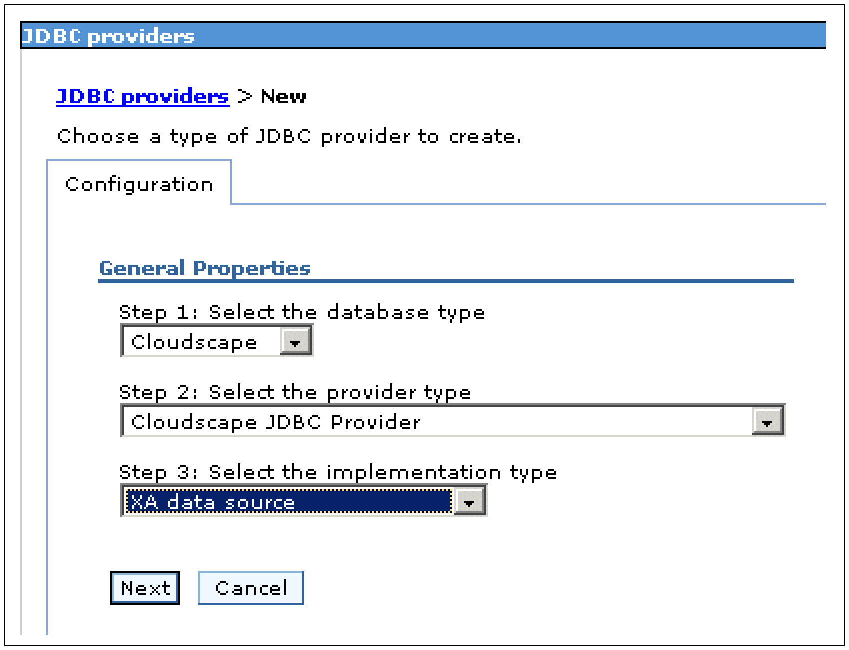


Figure A-6 Cloudscape database

Create the data source

Use the administrative console to create the following items.

1. Create a JDBC provider with the following properties, as in Figure A-7 on page 463.


JDBC providers

JDBC providers > New

Choose a type of JDBC provider to create.

Configuration

General Properties

Step 1: Select the database type: Cloudscape

Step 2: Select the provider type: Cloudscape JDBC Provider

Step 3: Select the implementation type: XA data source

Figure A-7 JDBC driver for the Patterns database

2. Create a data source. Use the defaults along with the following properties:
   – JNDI name: jdbc/Patterns
   – Select Use this Data Source in container managed persistence
   – Database name: PATTERNS
3. Test the connection to make sure the access is working.

Create a service integration bus

The application sample uses a service integration bus as the default JMS provider and for Web service destinations. You need to create the bus and add the application server where the application will run as a member of the bus.

1. Create a bus by selecting **Service integration** → **Buses.**
2. Click **New**.
3. Enter `ITSOMartBus` for the name and click **Apply**.
4. Click the **Bus members** link under Additional Properties.
5. Click **Add**.

6. Click **Server**, and select the server on which you will run the application. If you only have a standalone server, the defaults are correct. Click **Next**.
7. Click **Finish**.
8. Restart the server.

Install Web services support for the bus

The service integration bus Web services support is not installed by default when you install WebSphere Application Server, so this needs to be performed manually.

Installing this support requires you to do the following:

- Install the SDO Repository application.
- Install the resource adapter.
- Install the SIBWS application.
- Install the SOAP over HTTP endpoint listener application.

Scripts are provided to assist you with installing these features. All commands should be run from a command prompt in the `<was_install>/bin` directory.

Install the SDO Repository application

The SDO Repository application must be installed to manage the repository. In a standalone server environment, you can use Cloudscape for the repository. This step will create it for you.

Note: These instructions assume you have a standalone application server. Managed servers in a Network Deployment environment must use DB2 for the SDO repository. Refer to the InfoCenter for instructions on created the DB2 database and installing the SDO repository application, if this is the case.

1. Run the following commands:

   ```
   cd <was_install>\bin

   wsadmin.ext -f installSdoRepository.jacl -createDb
   ```

   Where `ext` is the file extension, `bat` for a Microsoft Windows system and `sh` for a UNIX system.
2. Look for the message that confirms that the SDO repository installation completed successfully.

The `-createDb` flag instructs the script to install IBM Cloudscape as the underlying database for the SDO repository.

Tip: If changes have to be made to the SDO installation, users should first run the script to uninstall the SDO repository before reinstalling:

```
wsadmin.ext -f uninstallSdoRepository.jacl -removeDb
```

This removes the application and JDBC configuration information. To remove the actual database itself, you will have to delete the database from the

`<was_install>/profiles/your_profile/databases/SdoRepDb directory.`

Install the resource adapter

The resource adapter is required to enable outbound services to be correctly configured within the bus. It should be installed before the core application and endpoint listeners. To install the resource adapter, perform the following steps:

1. Run the following command in Example A-2. The clusterName is optional and should only be specified in a clustered environment:

Example: A-2 Installing the resource adapter

```
cd <was_install>\bin

wsadmin -f <was_install>/util/sibwsInstall.jacl INSTALL_RA -installRoot
<was_install> -nodeName node-name -profileName profile_name [-clusterName
cluster_name]
```

For example:

```
./wsadmin.sh -f /opt/WebSphere/AppServer/util/sibwsInstall.jacl INSTALL_RA
-installRoot /opt/WebSphere/AppServer -nodeName myNode01 -profileName
new_profile_name
```

In our Rational test environment we entered:

```
wsadmin -f c:/Rational/RSA/runtimes/base_v6/util/sibwsInstall.jacl
INSTALL_RA -installRoot "c:/Rational/RSA/runtimes/base_v6" -nodeName Node01
```

Tip: The second *<was_install>* must have elements in the path separated by a forward slash (/) even on Windows system. So a path of `c:\WebSphere\AppServer` becomes `c:/WebSphere/AppServer`.

2. Look for the message that confirms that the SIB_RA installation completed successfully.

Tip: The concept of nodes is not apparent in a standalone application server. If you are not sure what your node name is, select **Servers** → **Application Servers** in the administrative console. The node name is listed next to the server in the application server list.

Install the SIBWS application

The core application is required tor making bus-generated WSDL available. To install the core application:

1. Run the following command in Example A-3.

Example: A-3 Installing the core application

```
cd <was_install>\bin

wsadmin.ext -f <was_install>/util/sibwsInstall.jacl INSTALL
   -installRoot <was_install> -serverName server_name
   -nodeName node_name
```

You can specify installation on either a server and node or a cluster. For example:

```
./wsadmin.sh -f /opt/WebSphere/AppServer/util/sibwsInstall.jacl INSTALL
-installRoot /opt/WebSphere/AppServer -serverName server1 -nodeName
myNode01 -profileName new_profile_name
```

If you are installing on a cluster, you should run the following command instead:

```
./wsadmin.sh -f /opt/WebSphere/AppServer/util/sibwsInstall.jacl INSTALL
-installRoot /opt/WebSphere/AppServer -clusterName myCluster01
```

In our Rational test environment we entered:

```
wsadmin -f c:/Rational/RSA/runtimes/base_v6/util/sibwsInstall.jacl INSTALL
-installRoot "c:/Rational/RSA/runtimes/base_v6" -nodeName Node01
-serverName server1
```

2. Look for the message that confirms that the installation completed successfully and that the application has been started.

Install the SOAP over HTTP endpoint listener application

The endpoint listener applications are required when users want to expose services using the bus to Web services clients wanting to connect using SOAP/HTTP or SOAP/JMS.

To install the HTTP endpoint listeners, run the following command:

```
wsadmin.ext -f install_root/util/sibwsInstall.jacl INSTALL_HTTP
   -installRoot install_root_using_forward_slashes -serverName server_name
   -nodeName node_name -profileName profile_name
```

For example:

```
./wsadmin.sh -f /opt/WebSphere/AppServer/util/sibwsInstall.jacl INSTALL_HTTP
-installRoot /opt/WebSphere/AppServer -serverName server1 -nodeName myNode01
-profileName AppSrv01
```

In our Rational test environment we entered:

```
wsadmin -f c:/Rational/RSA/runtimes/base_v6/util/sibwsInstall.jacl INSTALL_HTTP
-installRoot c:/Rational/RSA/runtimes/base_v6 -nodeName Node01 -serverName
server1
```

Look for the message that confirms that the installation completed successfully and that both applications have been started.

Note: If you want to make services available over SOAP/JMS, you have to install the SOAP/JMS endpoint listener applications. Before doing this, however, you *must* configure the necessary JMS resources as specified in the *WebSphere Application Server Information Center,* otherwise the applications will not start.

In our sample, we are only using the SOAP over HTTP listener so it is not necessary to install the SOAP over JMS listener. If you decide to use this listener, refer to the *WebSphere Application Server InfoCenter* for installation instructions.

Configure the bus for JMS messaging

The sample application also uses the bus as the default messaging provider. This support is built in to the bus and doesn't need to be installed, but the following application-specific configuration does need to be performed:

- Create the queue destinations
- Configure the queue mediation
- Configure the JMS connection factories
- Create the JMS queues
- Create the JMS activation specifications

Create the queue destinations

The OrderSystemV6 and Processor applications use JMS to trigger work to occur asynchronously. This requires two destination queues to be created. These next steps describe how to create these destinations:

1. From the bus details page for ITSOMartBus under **Additional Properties** click **Destinations**.
2. Click **New**.
3. For the destination type, accept the default of **Queue** and click **Next**.
4. The first page of the wizard, asks for an identifier and description to be entered. The identifier is the queue name. Enter `CustomerToProcessor.Queue` and click **Next**.
5. The next page allows you to specify to which bus member to assign the destination. There is only one bus member in our scenario so accept the default and click **Next**.
6. The final page is just a summary, click **Finish** and the destination will be created.
7. Create a second queue type destination by repeating steps 2 through 6. The identifier for this queue is **ProcessorToMail.Queue**.

Configure the JMS connection factories

The next step is to create the JMS connection factories that allow the application to send a JMS message.

1. From the admin console expand **Resources** → **JMS Providers** and click **Default messaging**.
2. Under Connection Factories click **JMS connection factory.**
3. Click **New**.

4. The next page allows you to specify the properties for the JMS connection factory. Take the defaults for everything but the following:
   - Name

     Enter a value of `CustomerToProcessor`.
   - JNDI Name

     This is where the application resource reference will be bound to. Enter a value of `jms/SelfService/CustomerCF`.
   - Bus name

     Select **ITSOMartBus** in the pull-down menu.
5. Click **OK** and save the changes.
6. Create a second connection factory using the values shown in Table A-1.

Table A-1 Connection factory values

Field	Value
Name	ProcessortoMail
JNDI Name	jms/SelfService/ProcessorCF
BusName	ITSOMartBus

Create the JMS queues

Now we need to create some JMS queues, one for each of the service integration bus queue type destinations we defined.

1. From the administrative console expand **Resources** → **JMS Providers** and click **Default messaging**.
2. Under **Destinations** click **JMS queue**.
3. Click **New**.
4. The next page allows you to specify the values for the queue.
   - Name

     Enter a value of `CustomerToProcessor`
   - JNDI Name

     This is where the application's message reference will be bound to. Enter a value of `jms/SelfService/CustomerToProcessorQ`
   - Bus name

     Select the value of **ITSOMartBus**. This will cause the page to be reloaded with the Queue names list filled populated.

- Queue name

  This field specifies the service integration bus queue type destination that will be used to store the messages sent to this JMS queue. Select the value of **CustomerToProcessor.Queue**

5. Click **OK**.
6. Repeat the process to create a second JMS queue using the values shown in Table A-2.

Table A-2 JMS queue values

Feild	Value
Name	ProcessorToMail
JNDI Name	jms/SelfService/ProcessorToMailQ
BusName	ITSOMartBus
Queue Name	ProcessorToMail.Queue

7. Save the changes.

Create the JMS activation specifications

Now we need to create some activation specifications. We will need one for each JMS queue we created earlier.

1. From the admin console expand **Resources** → **JMS Providers,** and click **Default messaging**.
2. Under **Activation Specifications** click **JMS activation specification**.
3. Click **New**.
4. The next page allows you to specify the values for the activation specification.

   Most of the values can keep their default values. Described below are the ones of most interest.

   - Name

     An administrative name used for locating the JMS activation specification. Enter a value of `Processor MDB Activation Spec`.

   - JNDI name

     This is where the application's message-driven beans will be bound to for message delivery. Enter a value of `esb/SelfService/Processor`.

   - Destination type

The type of the JMS destination that will be used to deliver messages to the message-driven bean. Accept the default of **Queue**.

- Destination JNDI name

  The location in JNDI of the JMS destination that should be used to receive messages from. Enter a value of `jms/SelfService/CustomerToProcessorQ`

- Bus name

  The name of the bus the JMS destination will receive messages from. This is not required, but for consistency select the value of **ITSOMartBus.**

5. Click **OK**.
6. Repeat steps 3 through 5 to create a second activation specification for the Mail application using the values in Table A-3.

Table A-3 Mail activation specification

Name	**Value**
Name	Mail MDB Activation Spec
JNDI Name	esb/SelfService/Mail
Destination Type	Queue
Destination JNDI Name	jms/SelfService/ProcessorToMailQ
Bus Name	ITSOMartBus

7. Save the changes.

CreditCheck application

Installing applications: To install an application to the Rational Software Architect unit test environment, perform the following steps:

1. Select the server in the Servers view, right-click, and select **Add and remove projects ...**
2. Select the application in the list of available projects and **click Add >**.
3. Click **Finish**.

To install an application to WebSphere Application Server:

1. Export the EAR file from Rational Software Architect to the `<`*`was_install`*`>installableApps` directory.
2. In the administrative console, select **Applications** → **Enterprise Applications**.
3. Click **Install** and follow the wizard.

1. Install the CreditCheck.ear file on the application server.
2. Create an endpoint listener for your application server using the process outlined in 10.7.3, “Create an endpoint listener” on page 335, and the following values:
   - Name: `SOAPHTTPChannel1`
   - URL root:

     `http://localhost:9080/wsgwsoaphttp1`
   - WSDL serving HTTP URL root:

     `http://localhost:9080/sibws/wsdl`
   - Connection properties bus name: `ITSOMartBus`
3. Create an outbound service definition for the CreditCheck Web service using the process outlined in 10.7.4, “Create the outbound services” on page 337 using the values below. Make sure the application server is started.
   - WSDL location:

     ```
     http://localhost:9080/CreditCheckRouter/wsdl/com/ibm/patterns/
     creditCheck/CreditCheck.wsdl
     ```
   - For all other selections use the defaults. When selecting the service and port, take the only option available.

4. Create an inbound service definition for the CreditCheck Web service using the process outlined in 10.7.5, “Create the inbound services” on page 340 and use the following values:
   - Service destination name:

     `http://creditCheck.patterns.ibm.com:CreditCheckService:CreditCheck`
   - Template WSDL location:

     ```
     http://localhost:9080/CreditCheckRouter/wsdl/com/ibm/patterns/
     creditCheck/CreditCheck.wsdl
     ```
   - Inbound Service name: `CreditCheckInboundService`.
   - For all other selections use the defaults. When selecting the service and port, take the only option available.

   Change the inbound port name to `CreditCheckInboundPort`.
5. Generate the new WSDL files for the Web service and export them using the process outlined in 10.7.6, “Generate and export new WSDL for the services” on page 343.
6. Follow the process outlined in 10.7.7, “Update the Web service clients to use the bus” on page 344 to import the new WSDL and update the Web service client to use the new WSDL.
   a. Use the process in “Import the generated WSDL” on page 345 to import the WSDL files to CreditCheckClient/ejbModule/META-INF/wsdl.
   b. Use the process in “Create the namespace mapping files” on page 345 to create a namespace mapping file:
      - Located in CreditCheckClient/ejbModule/META-INF/namespace mappings.
      - With the entries in Example A-4, substitute your cell name and bus name where appropriate. Each entry starting with http\ should be entered on one line.

Example: A-4 Name space mappings

```
http\://www.ibm.com/websphere/sib/webservices/cell/bus/Service=com.ibm.patterns.creditCheck
http\://www.ibm.com/websphere/sib/webservices/cell/bus/CreditCheckInboundService/Binding=com.ib
m.patterns.creditCheck
```

7. Use the process outlined in “Generate the Web service clients” on page 348 to regenerate the client to use the bus.
8. Enable the tool, by selecting **Windows** → **Preferences** → **Workbench** → **Capabilities**. Enable all the features listed under Web Services Developer.

9. Create a simple project and a simple folder under it. Unzip the WSDL zip file, and import it into the folder. Right-click the service wsdl, and select **Web Services** → **Test with Web Services Explorer**.

JCAModule CRM application

The following project files are included with the zip files so you can import and browse the source in Rational Software Architect:

- JCAClient
- JCAModule
- JCAWeb

However, the call to it in processor.java has been commented out. Running this piece would require that you:

- Have a CICS server installed with an application that updates a customer database.
- Install the CICS ECI resource adapter.

For this reason, we do not expect you to attempt to deploy this portion of the application. The process of creating the runtime configuration for this portion can be seen in 12.5, “Runtime guidelines for JCA applications” on page 433.

The code that calls this portion of the application has been commented out in Processor.java. See “Processor application” on page 480.

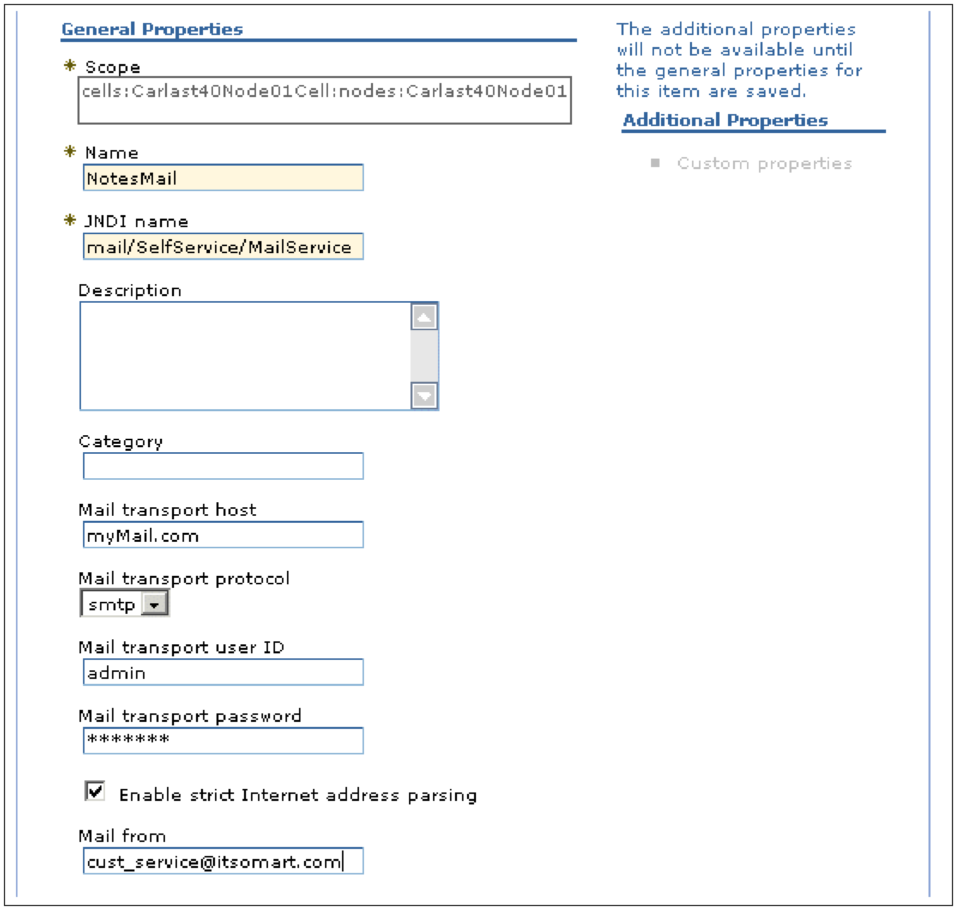
Mail service application

To use the mail portion of the application, you need access to a mail service. In our testing, we used a Notes mail server.

Configure a Mail provider

To configure the mail provider, perform the following tasks:

1. Select **Resources** → **Mail Providers.** Click on **Built-in Mail Provider.**
2. Under **Additional Properties** select **Mail Sessions.**
3. Click **New.** See Figure A-8 on page 475.


General Properties

* Scope
cells:Carlast40Node01Cell:nodes:Carlast40Node01

* Name
NotesMail

* JNDI name
mail/SelfService/MailService

Description

Category

Mail transport host
myMail.com

Mail transport protocol
smtp

Mail transport user ID
admin

Mail transport password
****

[✓] Enable strict Internet address parsing

Mail from
cust_service@itsomart.com

The additional properties will not be available until the general properties for this item are saved.

Additional Properties

- Custom properties

Figure A-8 Create a new mail provider

Complete the fields that define your mail provider. For the sample, the JNDI name should be `mail/SelfService/MailService`.

Click **OK**.

4. Save your changes.

Install the applications

The following applications need to be installed for the mail service portion of the application to work:

- MailService.ear
- XlateToXML.ear

- OrderSystemV6.ear. During the installation, make sure that the correct backend-ID (Cloudscape vs. DB2) for the database is chosen.

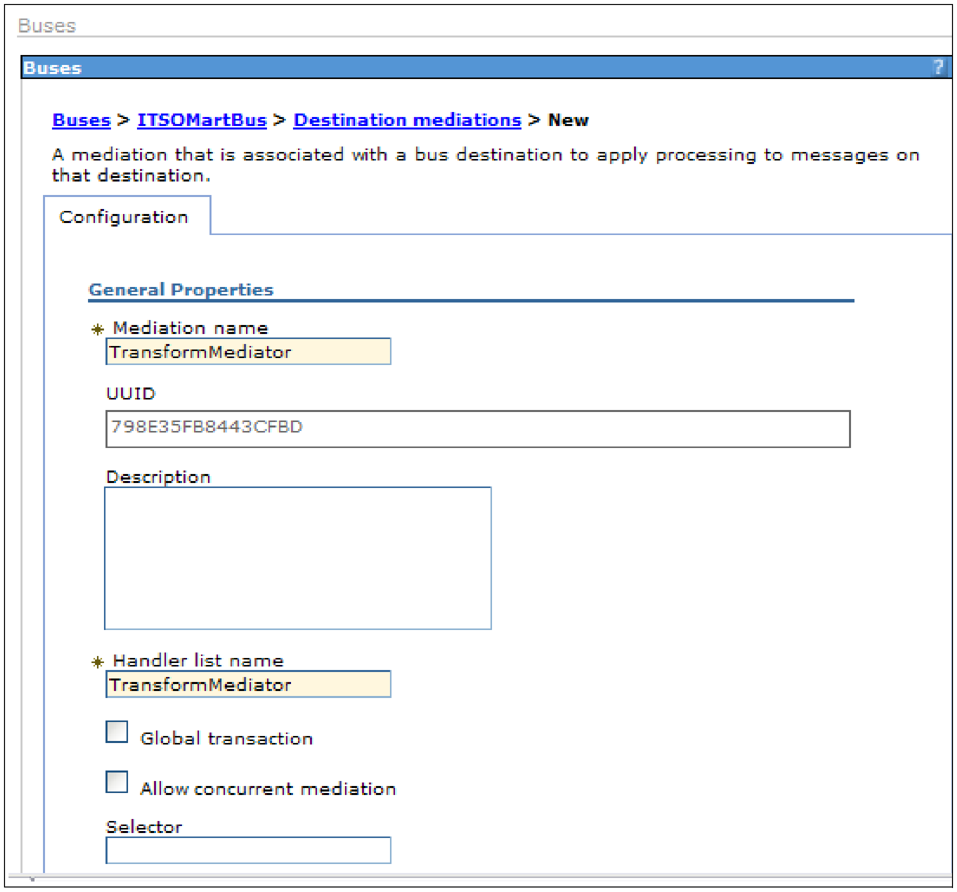
During the installation of the applications, you can ignore the following warning:

```
ADMA8019E: The resources that are assigned to the application are beyond the
deployment target scope. Resources are within the deployment target scope if
they are defined at the cell, node, server, or application level when the
deployment target is a server, or at the cell, cluster, or application level
when the deployment target is a cluster. Assign resources that are within the
deployment target scope of the application or confirm that these resources
assignments are correct as specified.
```

Configure the mediation

The following steps take you through the configuration process for the mediation application.

1. Install the mediation application, XlateToXML.
2. Locate the bus definition by selecting **Service integration** → **Buses**. Click the bus name (**ITSOMartBus**) to open it.
3. Under **Additional Properties** click **Mediations**.
4. Click **New**. See Figure A-9 on page 477.


Buses

Buses

Buses > ITSOMartBus > Destination mediations > New

A mediation that is associated with a bus destination to apply processing to messages on that destination.

Configuration

General Properties

* Mediation name: TransformMediator

UUID: 798E35FB8443CFBD

Description:

* Handler list name: TransformMediator

☐ Global transaction

☐ Allow concurrent mediation

Selector:

Figure A-9   Define a mediation

Enter the following values:

– Mediation name: `TransformMediator`
– Handler list name: `TransformMediator`

Click **OK**.

Mediate the destination

To mediate the destinations, perform the following steps:

1. Locate the bus definition by selecting **Service integration** → **Buses**. Click the bus name (**ITSOMartBus**) to open it.
2. Under **Additional Properties** click **Destinations**.
3. In the list of destinations, check the box to the left the queue name. In this case, the queue is **ProcessorToMail.queue**.

   Click the **Mediate** button.
4. In the next screen, select **TransformMediator** as the mediation to apply to the destination.

   Click **Next**.
5. In the next screen, select the bus and click **Next**.
6. Click **Finish**.

Save your changes.

Note: Be sure to uncomment the appropriate lines in Processor.java to activate this portion of the application.

DeliverySystem application

To install the DeliverySystem application, follow these steps

1. Install the DeliverySystem.ear file on the application server.
2. Create an outbound service definition for the HomeDelivery Web service using the process outlined in 10.7.4, “Create the outbound services” on page 337 using the values below. Make sure the application server is started.
   – WSDL location:

     ```
     http://localhost:9080/HomeDeliveryRouter/wsdl/com/ibm/patterns/
     delivery/HomeDelivery.wsdl
     ```
   – For all other selections use the defaults. When selecting the service and port, take the only option available.

3. Create another outbound service definition for the BusinessDelivery Web service using the process outlined in 10.7.4, “Create the outbound services” on page 337 using the values below. Make sure the application server is started.

   – WSDL location:

     `http://localhost:9080/BusinessDeliveryRouter/wsdl/com/ibm/patterns/delivery/BusinessDelivery.wsdl`

   – For all other selections use the defaults. When selecting the service and port, take the only option available.

4. Create an inbound service definition for the HomeDelivery Web service using the process outlined in 10.7.5, “Create the inbound services” on page 340 and use the following values:

   – Service destination name:

     `http://delivery.patterns.ibm.com:HomeDeliveryService:HomeDelivery`

   – Template WSDL location:

     `http://localhost:9080/HomeDeliveryRouter/wsdl/com/ibm/patterns/delivery/HomeDelivery.wsdl`

   – Inbound Service name: `HomeDeliveryInboundService`.

   – Select SOAPHTTPChannel1 for the endpoint listener.

   – For all other selections use the defaults. When selecting the service and port, take the only option available.

   Change the inbound port name to `HomeDeliveryInboundPort`.

5. Generate the new WSDL files for the Web service and export them using the process outlined in 10.7.6, “Generate and export new WSDL for the services” on page 343.

6. Follow the process outlined in 10.7.7, “Update the Web service clients to use the bus” on page 344 to import the new WSDL and update the Web service client to use the new WSDL.

   a. Use the process in “Import the generated WSDL” on page 345 to import the WSDL files for HomeDelivery to **ClientDelivery/ejbModule/META-INF/wsdl**.

   b. Use the process in “Create the namespace mapping files” on page 345 to create a namespace mapping file:

      - Located in ClientDelivery/ejbModule/META-INF/namespace mappings.
      - With the entries in Example A-5 on page 480, substitute your cell name and bus name where appropriate. Each entry starting with http\ should be entered on one line.

Example: A-5 Name space mappings

```
http\://www.ibm.com/websphere/sib/webservices/cell/bus/Service=com.ibm.patterns.delivery
http\://www.ibm.com/websphere/sib/webservices/cell/bus/HomeDeliveryInboundService/Binding=com.i
bm.patterns.delivery
```

   c. Use the process outlined in “Generate the Web service clients” on page 348 to regenerate the client to use the bus.

7. To test the service, follow these steps:
   a. Enable the Web Services Explorer tool, by selecting **Windows** → **Preferences** → **Workbench** → **Capabilities**. Enable all the features listed under Web Services Developer.
   b. Create a simple project and a simple folder under it. Unzip the WSDL zip file, and import it into the folder. Right-click the service WSDL and select **Web Services** → **Test with Web Services Explorer**.
8. Install and configure the router mediation application, for the incoming request for HomeDelivery using the process outlined in 10.7.8, “Configure the router mediation” on page 350. In this step you will:
   a. Install the RoutingMediationsEJBEAR application
   b. Define the DeliveryRequestMediation mediation on the bus
   c. Mediate the port for the HomeDelivery destination.
9. Install and configure the aggregator mediation application, for the incoming responses to the for HomeDelivery and BusinessDelivery services using the process outlined in 10.7.9, “Configure the aggregator mediation” on page 351. In this step you will:
   a. Create two queue destinations on the bus, DeliveryResponseDestination and TempDestination
   b. Install the AggregatorEJBEAR mediation application.
   c. Define the DeliveryResponseMediation mediation on the bus.
   d. Mediate the DeliveryResponseDestination queue.
10. Save the configuration.

Processor application

If you are going to run with the MailService or JCAModule applications, you will need to modify Processor.Java to uncomment the lines that activate these calls. You will find Processor.java in the ProcessorEJB module at com.ibm.patterns.serialProcess.Processor.java class.

The lines of code to be uncommented are found in the following three methods in Example A-6:

Example: A-6 Uncommenting MailService and JCAModule

```
   private void createCustomer() {
      //Call to CreditRating application
      String creditRating = new
ProcessorWebService().getCreditRating(customerDetails);
      if (isCreditSatisfactory(creditRating)) {
         //Credit is Satisfactory
         //Create customer in CRM and send Acceptance Mail

         //Uncomment if you have the JCA service configured
         //ProcessorJCA.storeCustomerDetails_CICS(customerDetails);

         //Uncomment if you have the Mail service configured
         //new MailService().sendActivationMail(customerDetails);
         System.out.println("Customer created successfully....");
      }
      else {
         //Credit rating is not satisfactory
         //Delete customer from Db2 and send rejection E-mail
         deleteCustomerfromDB();
         //Uncomment if you have the Mail service configured
         //new MailService().sendRejectionEmail(customerDetails);
         System.out.println("The customer does not have satisfactory credit
and will not be created...");
      }
   }

   private void deleteCustomer() {
      //Uncomment if you have the JCA service configured
      //ProcessorJCA.deleteCustomerDetails_CICS(pkey);
      deleteCustomerfromDB();
      //Uncomment if you have the Mail service configured
      //new MailService().sendDeletionEmail(pkey);
      System.out.println("Customer deleted successfully...");

   }

   private void updateCustomer() {

      //Uncomment if you have the JCA service configured
      //ProcessorJCA.updateCustomerDetails_CICS(pkey);

      //Uncomment if you have the Mail service configured
      //new MailService().sendUpdationEmail(customerDetails);
      System.out.println("Customer details updated successfully...");
}
```

Install Processor.ear

The last step is to install Processor.ear into the server and start it.

Access the application

The application can be accessed from a Web browser using the following URL:

```
http://localhost:9080/OrderSystemWeb/index.jsp
```

Common errors:

Following are some common errors and how to correct them.

- **Error:**

  ```
  [5/4/05 22:50:46:953 EDT] 00000036 Helpers W NMSV0605W: A Reference object
  looked up from the context "java:" with the name
  "comp/env/jms/SelfService/ProcessorToMailQ" was sent to the JNDI Naming
  Manager and an exception resulted.
  ```

  Tip: Ensure the JMS queue with the name jms/SelfService/MailService/MailSenderQueue is defined on the server.

- **Error:**

  ```
  [5/4/05 22:50:47:172 EDT] 00000036 RALifeCycleMa E   J2CA0052E: The lookup
  of the ActivationSpec with JNDI Name  esb/SelfService/Processor failed due
  to exception javax.naming.NameNotFoundException: Context:
  sandygNode01Cell/nodes/ITSONode/servers/server1, name:
  esb/SelfService/Processor: First component in name
  esb/SelfService/Processor not found.  Root exception is
  org.omg.CosNaming.NamingContextPackage.NotFound:
  IDL:omg.org/CosNaming/NamingContext/NotFound:1.0
  ```

  Tips: Ensure the ActivationSpec with the name esb/SelfService/Processor is defined on the server.

- **Error:**

  ```
  [5/4/05 22:50:47:516 EDT] 00000036 EJBContainerI E   WSVR0062E: Unable to
  start EJB, Processor#ProcessorEJB.jar#ProcessorMDB:
  javax.resource.ResourceException: Failed to lookup
  ActivationSpec.esb/SelfService/Processor
  ```

Tip: Ensure the ActivationSpec esb/SelfService/Processor is defined on the server.

- **Error:**

```
[5/2/05 12:00:08:141 EDT] 00000041 J2CUtilityCla E   J2CA0036E: An
exception occurred while invoking method setDataSourceProperties on
com.ibm.ws.rsadapter.spi.WSManagedConnectionFactoryImpl used by resource
OrderSystem dataSource_CF : java.lang.NoClassDefFoundError:
com/ibm/db2/jcc/SQLJConnection
```

Tip: Make sure the DB2 driver jars are in the classpath of the server

- **Error:**

```
[5/5/05 8:45:59:188 EDT] 00000037 StaleConnecti A   CONM7007I: Mapping the
following SQLException, with ErrorCode -30082 and SQLState 08001, to a
StaleConnectionException: java.sql.SQLException: SQL30082N  Attempt to
establish connection failed with security reason "24" ("USERNAME AND/OR
PASSWORD INVALID").  SQLSTATE=08001
DSRA0010E: SQL State = 08001, Error Code = -30,082
```

Tip: Check the user name and password in the authentication alias defined for the data source

- **Error:**

```
[5/27/05 14:33:09:979 EDT] 0000003d SystemErr     R
com.ibm.patterns.order.exception.CustomerCreationException
```

Tip: Make sure the e-mail address entered for the customer is unique.

B

Additional material

This redbook refers to additional material that can be downloaded from the Internet as described below.

Locating the Web material

The Web material associated with this redbook is available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser to:

```
ftp://www.redbooks.ibm.com/redbooks/SG246680
```

Alternatively, you can go to the IBM Redbooks Web site at:

```
ibm.com/redbooks
```

Select the **Additional materials** and open the directory that corresponds with the redbook form number, SG246680.

Using the Web material

The additional Web material that accompanies this redbook includes the following files:

File name	*Description*
SG246680.zip	Zipped sample application EAR files

System requirements for downloading the Web material

The following system configuration is recommended:

Hard disk space:	6 MB
Operating System:	Windows

How to use the Web material

Create a subdirectory (folder) on your workstation, and unzip the contents of the Web material zip file into this folder. Instructions for using the sample application are in Appendix A, “Sample application install summary” on page 453.

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

IBM Redbooks

For information on ordering these publications, see “How to get IBM Redbooks” on page 491. Note that some of the documents referenced here might be available in softcopy only.

- *WebSphere Studio 5.1.2, JavaServer Faces and Service Data Objects,* SG24-6361
- *WebSphere Application Server V6 System Management and Configuration Handbook,* SG24-6451
- *WebSphere Version 6 Web Services Handbook Development and Deployment,* SG24-6461
- *Patterns: Self-Service Application Solutions Using WebSphere V5.0,* SG24-6591
- *Rational Application Developer V6 Programming Guide,* SG24-6449
- *Legacy Modernization with WebSphere Studio Enterprise Developer*, SG24-6806
- *Self-Service Applications using IBM WebSphere V4.0 and IBM MQSeries Integrator*, SG24-6160
- *MQSeries Programming Patterns*, SG24-6506
- *EJB 2.0 Development with WebSphere Studio Application Developer*, SG24-6819
- *Patterns: Implementing an SOA Using an Enterprise Service Bus,* SG24-6346
- *Patterns: Implementing an SOA using an Enterprise Service Bus in WebSphere Application Server V6,* SG24-6494
- *Java Connectors for CICS: Featuring the J2EE Connector Architecture*, SG24-6401
- *Revealed! Architecting Web Access to CICS*, SG24-5466

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

Other publications

These publications are also relevant as further information sources:

- Adams, Jonathan, Srinivas Koushik, Guru Vasudeva, and George Galambos. *Patterns for e-business: A Strategy for Reuse* by Jonathan Adams. IBM Press, 2001. ISBN 1-931182-02-7.
- *CICS Transaction Server for OS/390: Application Programming Guide*, SC33-1687
- Flanagan, David, *JavaScript: The Definitive Guide*, Third Edition, O'Reilly & Associates, Inc., 1998. ISBN 1565923928.
- Flanagan, David, Jim Farley, William Crawford and Kris Magnusson, *Java Enterprise in a Nutshell*, O'Reilly & Associates, Inc., 1999. ISBN 1565924835.
- Gamma, Erich, Richard Helm, Ralph Johnson, John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995. ISBN 0-201-63361-2.
- Maruyama, Hiroshi, Kent Tamura and Naohiko Uramoto, *XML and Java: Developing Web Applications*, Addison-Wesley 1999. ISBN 0201485435.
- *WebSphere MQ Application Programming Guide*, SC34-6064
- *WebSphere MQ Using Java*, SC34-6066

Online resources

These Web sites and URLs are also relevant as further information sources:

- Apache Jakarta Project Struts Web site:

  `http://jakarta.apache.org/struts/`
- *An introduction to Model Driven Architecture Part 1: MDA and today's systems*

  `http://www-128.ibm.com/developerworks/rational/library/3100.html`
- *An introduction to Model-Driven Architecture (MDA): Part II: Lessons from the design and use of an MDA toolkit*

  `http://www-128.ibm.com/developerworks/rational/library/content/RationalEdge /apr05/brown/`

- ECMAScript language specification

  `http://www.ecma-international.org/publications/standards/ECMA-262.HTM`
- IBM CICS

  `http://www.ibm.com/software/ts/cics`
- *IBM WebSphere and MQSeries Integration Using Servlets and JavaServer Pages*

  `http://www7b.boulder.ibm.com/wsdd/library/techtip/pwd/wsmq_integration.html`
- IBM developerWorks

  http://www.ibm.com/developerWorks
- *Integrating IBM WebSphere Application Server and the WebSphere MQ Family*

  `http://www7b.boulder.ibm.com/wsdd/techjournal/0110_yusuf/yusuf.html`
- Java APIs and technology

  `http://java.sun.com/products`
- Java Message Service API documentation

  `http://java.sun.com/products/jms`
- JSF specification

  `http://java.sun.com/j2ee/javaserverfaces/`
- JCA specification:

  `http://java.sun.com/j2ee/connector/`
- Java 2 Platform Micro Edition (J2ME):

  `http://java.sun.com/j2me`
- Java Portlet V1.0 specification:

  `http://jcp.org/en/jsr/detail?id=168`
- Model Driven Architecture (MDA) standards:

  `http://www.omg.org/mda`
- OASIS WS-Security 1.0 and token profiles:

  `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss`
- Open source XML frameworks:

  `http://xml.apache.org/`
- Patterns for e-business:

  `http://www.ibm.com/developerWorks/patterns/`

- Reusable Asset Specification

  `http://www.flashline.com/ras/RAS_060604.pdf`

- Sun JavaServer Faces Technology Page

  http://java.sun.com/j2ee/javaserverfaces/index.jsp

- *Security in a Web Services World: A Proposed Architecture and Roadmap*, proposed by IBM and Microsoft.

  `http://www-106.ibm.com/developerworks/webservices/library/ws-secmap/`

- Service-oriented architecture and Web services

  `http://www.ibm.com/software/solutions/webservices/resources.html`

- *The role of private UDDI nodes in Web services, Part 1: Six species of UDDI*

  `http://www.ibm.com/developerworks/webservices/library/ws-rpu1.html`

- *The role of private UDDI nodes, Part 2: Private nodes and operator nodes*

  `http://www.ibm.com/developerworks/webservices/library/ws-rpu2.html`

- *IBM WebSphere Developer Technical Journal: Transactional integration of WebSphere Application Server and CICS with the J2EE Connector Architecture*:

  `http://www-128.ibm.com/developerworks/websphere/techjournal/0408_wakelin/0408_wakelin.html`

- *Understanding quality of service for Web services*:

  `http://www-106.ibm.com/developerworks/library/ws-quality.html`

- *Understand Enterprise Service Bus scenarios and solutions in Service-Oriented Architecture, Part 1*:

  `http://www.ibm.com/developerworks/webservices/library/ws-esbscen/`

- *Using J2EE Resource Adapters in a Non-managed Environment* at:

  `http://www7b.boulder.ibm.com/wsdd/library/techarticles/0109_kelle/0109_kelle.html`

- WebSphere Developer Domain:

  `http://www7b.software.ibm.com/wsdd/`

- *WebSphere Application Server Version 6.0 Information Center*, available at:

  `http://publib.boulder.ibm.com/infocenter/ws60help/index.jsp`

- W3C Web Services Architecture specification:

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

- W3C HTML Validation Service, are available at:

  `http://validator.w3.org/`

- W3C CSS Validation Service at:

  http://jigsaw.w3.org/css-validator/
- WS-I Basic Profile Version 1.0 specification:

  http://www.ws-i.org/Profiles/BasicProfile-1.0-2004-04-16.html
- WS-Coordination specification:

  http://www-106.ibm.com/developerworks/library/ws-coor/
- WS-Transaction specification:

  http://www-106.ibm.com/developerworks/library/ws-transpec/
- WS-Security specification 1.0:

  http://www.ibm.com/developerworks/library/ws-secure/
- Web Services Security: SOAP Message: Errata 1.0

  http://www.oasis-open.org/committees/download.php/9292/oasis-200401-wss-soap-message-security-1%200-errata-003.pdf
- Web Services Security: UsernameToken Profile: Errata 1.0

  http://www.oasis-open.org/committees/download.php/9290/oasis-200401-wss-username-token-profile-1.0-errata-003.pdf
- Web Services Security: X.509 Token Profile: Errata 1.0

  http://www.oasis-open.org/committees/download.php/9287/oasis-200401-x509-token-profile-1.0-errata-003.pdf
- World Wide Web Consortium (W3C) site

  http://www.w3.org/
- XML Signature workgroup home page:

  http://www.w3.org/Signature/
- XML Encryption workgroup home page:

  http://www.w3.org/Encryption/

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

IBM

Redbooks

Patterns: Implementing Self-Service in an SOA Environment

(1.0” spine)
0.875”<->1.498”
460 <-> 788 pages

Patterns: Implementing Self-Service in an SOA Environment

Redbooks

Integrate Web applications with the enterprise tier

Explore Web services, J2EE Connectors, and JMS solutions

Use SOA and ESB technology

The Patterns for e-business are a group of proven, reusable assets that can be used to increase the speed of developing and deploying Web applications. This IBM Redbook focuses on the use of service-oriented architecture and the enterprise service bus to build solutions that help organizations achieve rapid, flexible integration of IT systems.

It includes the Self-Service::Directly Integrated Single Channel pattern for implementing point-to-point connections with back-end applications, the Self-Service::Router pattern for implementing intelligent routing among multiple back-end applications, and the Self-Service::Decomposition pattern for decomposing a request into multiple requests and recomposing the results into a single response.

This IBM Redbook teaches you by example how to design and build sample solutions using WebSphere Application Server V6 with Web services, J2EE Connectors and IBM CICS, and JMS using the WebSphere Application Server default messaging provider. WebSphere Application Server service integration technology is used to implement enterprise service bus functionality.

SG24-6680-01 ISBN 073849626X