IBM

IBM WebSphere Application Server V7.0 Web Services Guide

Explore new technology

Develop Web services by example

Find leading practices

Henry Cui
Raymond Josef Edward A. Lara
Rosaline Makar
Nicky Moelholm
Felipe Pittella Rodrigues

Redbooks

IBM International Technical Support Organization

IBM WebSphere Application Server V7.0 Web Services Guide

August 2009

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

First Edition (August 2009)

This edition applies to WebSphere Application Server V7.

Contents

Notices

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

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

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

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

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

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

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

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

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

COPYRIGHT LICENSE:

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

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at http://www.ibm.com/legal/copytrade.shtml

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

DB2®	Rational®	Tivoli®
developerWorks®	Redbooks®	WebSphere®
IBM®	Redbooks (logo) ®	z/OS®

The following terms are trademarks of other companies:

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

Interchange, JBoss, and the Shadowman logo are trademarks or registered trademarks of Red Hat, Inc. in the U.S. and other countries.

EJB, Enterprise JavaBeans, J2EE, Java, Java runtime environment, JavaBeans, JavaServer, JDBC, JDK, JRE, JSP, JVM, Sun, Sun Enterprise, and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

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

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

Preface

This IBM® Redbooks® publication describes how to implement Web services in IBM WebSphere® Application Server V7. It starts by describing the concepts of the major building blocks on which Web services rely and leading practices for Web services applications. It then illustrates how to use Rational® Application Developer and the WebSphere tools to build and deploy a Web services application.

In addition to the fundamentals of Web services development, this book provides information about advanced topics, including WS-Policy, WS-MetadataExchange, Web services transactions, WS-Notification, and WS-SecureConversation.

The team that wrote this book

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

Henry Cui is a Software Developer working at the IBM Toronto lab. Henry has been on the IBM Rational Application Developer service and support team for six years. He has helped many customers resolve design, development, and migration issues with Web services development. He is the subject matter expert (SME) for Web services on his team. His areas of expertise include developing Java™ EE applications using Rational tools, configuring WebSphere Application Servers, Enterprise JavaBeans™ (EJBs), application security, Web services, and service-oriented architecture (SOA). Henry is a frequent contributor of developerWorks® articles. He also co-authored three IBM Redbooks publications related to Web services. Henry holds a degree in Computer Science from York University.

Raymond Josef Edward A. Lara is an IT Specialist for IBM Software Lab Services in the Association of Southeast Asian Nations (ASEAN) region, covering Singapore, Malaysia, Thailand, Vietnam, Indonesia, and the Philippines, where he is based. He is a certified instructor for WebSphere Education and conducts software education classes for the WebSphere portfolio. He has spent the last six years with IBM as a WebSphere Specialist and covers a wide range of products including WebSphere Application Server,

WebSphere Process Server, WebSphere MQ, and WebSphere Portal. Raymond has 14 years of industry experience as a technical consultant. He specializes in application design and development using IBM and open-source technology.

Rosaline Makar is a Software Engineer in IBM Egypt, Cairo Technology Development Center (C-TDC). She earned a Bachelor of Science in Computer Engineering and a Master of Science in Computer Science. Rosaline's areas of expertise include Web services, WebSphere Integration Developer, WebSphere Process Server, WebSphere Enterprise Service Bus, WebSphere Message Broker, and WebSphere Service Registry and Repository.

Nicky Moelholm is an IT Specialist working for IBM Software Services WebSphere in Denmark. His primary focus is on Java EE development using IBM WebSphere products. He holds a Master's degree in Information Technology from the IT University of Copenhagen, has multiple Java certifications, and is a certified WebSphere Application Server Administrator. In total Nicky has 8.5 years of experience developing enterprise Java applications.

Felipe Pittella Rodrigues is an IT Specialist and IT Architect who has been working for IBM Brazil and Global Services since 2005. He has six years of professional IT experience in many areas covering financial and Java EE enterprise projects. Felipe holds certifications as a Sun™ Certified Java Programmer (SCJP), Sun Certified Web Component Developer (SCWCD), Sun Certified Developer For Java Web Services (SCDJWS), Sun Certified Business Component Developer (SCBCD), and Sun Enterprise™ Architect. He is also a Sun Authorized Instructor certified for Java 5 Platform. Felipe graduated in IT Information Systems from the University Technological and Federal of Paraná (UTFPR/Brazil). Felipe is expert in WebSphere Application Server and Web services applications. SOA architecture, patterns, RFID, and autonomic computing implementations are among his areas of expertise.

Jacek Laskowski
IBM Poland

Charles Levay
IBM US

Greg Truty
IBM US

Thanks to the authors of the following books:

► Authors of *Web Services Handbook for WebSphere Application Server 6.1*, SG24-7257, published in October 2006, were:

Ueli Wahli, Owen Burroughs, Owen Cline, Alec Go, Larry Tung

► Authors of *Web Services Feature Pack for WebSphere Application Server V6.1*, SG24-7618, published in August 2008, were:

Peter Swithinbank, Russell Butek, Henry Cui, Andrew Das, David Illsley, Mark Lewis

Become a published author

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

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

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

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

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

► Use the online **Contact us** review IBM Redbooks publications form found at:

   ibm.com/redbooks

► Send your comments in an e-mail to:

   redbooks@us.ibm.com

► Mail your comments to:

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

Introduction to Web services technology and programming model

1

Introduction

This chapter introduces several key features that enable you to develop and expose Web services applications by using WebSphere Application Server V7. By doing so, you can provide an even more flexible and reliable foundation for your service-oriented architecture (SOA) Web services environment than you have today.

This chapter includes the following topics:

1.1 WebSphere, Web services, and SOA

Businesses face two fundamental concerns:

► The ability to change quickly
► The need to reduce costs

To remain competitive, businesses must adapt quickly to internal factors, such as acquisitions and restructuring, or external factors, such as competitive forces and customer requirements.

IBM WebSphere Application Server V7 is a major release that offers dramatic runtime improvements, in addition to simpler and easier ways to develop and deploy SOA applications. An SOA consists of a set of business-aligned services that collectively fulfill an organization's business process goals and objectives. These services can be choreographed into composite applications and can be invoked through Internet-based open standards and protocols.

Web services are self-contained, modular applications that you can describe, publish, locate, and invoke over a network to perform encapsulated business functions. These functions range from a simple request-reply interaction to full business process interactions using Internet standards and protocols.

A cost-effective, flexible IT infrastructure is required to support these business needs. IBM WebSphere, along with SOA and Web services, can address these needs in a powerful manner.

1.1.1 Service-oriented architecture

Companies want to integrate existing systems to implement IT support for business processes that cover the entire business value chain. A variety of patterns are used to make their IT systems available to internal departments or external customers. However, such interactions are not flexible and do not provide standardized architecture.

Because of this increasing demand for technologies that support connecting and sharing resources and data, a need exists for a flexible and standardized architecture. SOA is a flexible architecture that unifies business processes by structuring large applications into building blocks, or small modular functional units or services, for various groups of people to use inside and outside the company.

In an SOA, applications are made up from loosely coupled services, which interact to provide all the functionality that is needed by the application. To efficiently use an SOA, you must follow these requirements:

► Interoperability between multiple systems and programming languages

The most important basis for a simple integration between applications on various platforms is to provide a communication protocol that is available for most systems and programming languages.

► Clear and unambiguous description language

To use a service offered by a service provider, it is not only necessary to access the provider system, but the syntax of the service interface must also be clearly defined in a platform-independent manner.

► Retrieval of the service

To support a convenient integration at design time or even system run time, a search mechanism is required to retrieve services. Classify these services according to their category and how they can be discovered.

More information: For additional information about service-oriented architecture, see "Service-oriented architecture" at the following address:

```
http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/index.jsp?topic
=/com.ibm.websphere.soafep.multiplatform.doc/info/ae/ae/cwbs_soa.html
```

SOA architecture and benefits

SOA offers the following benefits to help organizations succeed in a dynamic environment:

► Leverages existing assets

SOA provides a layer of abstraction that enables an organization to continue using its IT investment by wrapping these existing assets as services that consequently provide business functions. Organizations can potentially continue getting value from existing resources instead of rebuilding applications from scratch.

► Is easier to integrate and manage complexity

The integration point in an SOA is the service specification and not the implementation. This service specification integration point provides implementation transparency and minimizes the impact when infrastructure and implementation changes occur. By providing a service specification in front of existing resources and assets that are built on disparate systems, integration becomes more manageable, because complexities are isolated. This ease of integration becomes even more important as more businesses work together to provide the value chain.

- ► Is more responsive and offers a faster time-to-market

  The ability to compose new services from existing services provides a distinct advantage to an organization that must be agile to respond to demanding business requirements. Usage of existing components and services reduces the time needed to go through the software development life cycle of gathering requirements and performing design, development, and testing. This shorter cycle leads to the rapid development of new business services and allows an organization to respond quickly to changes and reduce the time-to-market.

- ► Reduces cost and increases reuse

  With core business services exposed in a loosely coupled manner, they can be more easily used and combined based on business needs, which means less duplication of resources and more potential for reuse and, therefore, lower costs.

- ► Helps businesses prepare for what lies ahead

  SOA allows businesses be ready for the future. Business processes that are comprised of a series of business services can be more easily created, changed, and managed to meet the needs of the time. SOA provides the flexibility and responsiveness that is critical to businesses to survive and thrive.

SOA is by no means a magical solution, and migration to SOA is not an easy task. Rather than migrating the entire enterprise to an SOA overnight, migrate an appropriate subset of business functions as the business need arises or is anticipated.

> **More information:** For more information about SOA architecture and benefits see *Patterns: Service-Oriented Architecture and Web Services*, SG24-6303.

IBM SOA Foundation

The IBM SOA Foundation is an integrated, open standard-based set of IBM software, best practices, and patterns. It is designed to provide what you need to get started with SOA from an architectural perspective. A key element of the SOA Foundation is the SOA Foundation scenarios.

The SOA Foundation scenarios (or simply, SOA scenarios) represent common scenarios regarding the use of IBM products and solutions for SOA engagements. The SOA scenarios communicate the business value, architecture, and IBM scenarios that can be then used as reference materials to accelerate an SOA implementation based on your requirements.

> **More information:** For more information about IBM SOA Foundation
> scenarios, see *Best Practices for SOA Management*, REDP-4233.

1.1.2 WebSphere and SOA

IBM WebSphere Application Server V7 delivers an agile, solid foundation with
high performance for SOA applications by aligning innovation in both IT and
business. The WebSphere Platform provides simplification for developers by
enabling the reuse and creation of applications and services that promote
business agility, both anticipating and adjusting to the critical issues that help
businesses win in the marketplace.

WebSphere has the following major goals regarding the SOA environment for
Web services:

► Achieve maximum flexibility and high productivity for SOA initiatives.
► Protect your critical SOA applications with strong security management.
► Increase the effectiveness of SOA application infrastructure management.

1.1.3 Web services approach to an SOA

Web services provides a technology foundation for implementing an SOA. A
major focus of Web services is to make functional building blocks accessible over
standard Internet protocols that are independent from platforms and
programming languages. These services can be new applications or just new
adapters wrapped around existing systems to make them network enabled.

Web services are self-contained, modular applications that you can describe,
publish, locate, and invoke over a network. They perform encapsulated business
functions ranging from a simple request-reply interaction to a full business
process interaction by using Internet standards and protocols.

Web services technology is an ideal technology choice for implementing an SOA,
for the following reasons:

► Web services are standards based. Interoperability is a key business
  advantage within the enterprise and in business-to-business scenarios.

► Web services are widely supported across the industry. Most major vendors
  recognize and provide support for Web services.

► Web services provide a migration path to gradually enable existing business
  functions as Web services are needed.

Conversely, many Web services implementations are not SOAs. For example, the use of Web services to connect two heterogeneous systems directly together is not an SOA. This use of Web services solves real problems and provides significant value on its own, and it might form the starting point of an SOA.

In general, an SOA environment must be implemented at an enterprise or organizational level to provide many of the benefits.

> **More information**: For additional information about the Web services approach for SOA, see *Web services approach to a service-oriented architecture* at:
>
> http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/index.jsp?topic=/com.ibm.websphere.soafep.multiplatform.doc/info/ae/ae/cwbs_soawbs.html

Web services business models supported

The characteristics and benefits of using an SOA environment, such as Web services, is well suited for binding small modules that perform independent tasks within a highly heterogeneous e-business model. Web services can be easily wrapped around existing applications in your business model and plugged into various business processes.

Web services support the following business models:

► Business information

Share information with consumers or other businesses. You can use Web services to expand the reach of the business through services, such as news streams, local weather reports, integrated travel planning, and intelligent agents.

► Business integration

Provide transactional, fee-based services for customers. You can easily create a global network of suppliers. You can implement Web services in auctions, e-marketplaces, and reservation systems.

► Business process externalization

You can use Web services to model value chains by dynamically integrating processes to create a new solution within an organizational unit or with those processes of other e-businesses. You can achieve this modeling by dynamically linking internal applications to new partners and suppliers to offer their services in order to complement internal services.

1.1.4 WebSphere Application Server V7 and Web services

WebSphere Application Server V7 has powerful new features and enhancements
to help you achieve heightened productivity, stronger security, integration, and
interoperability with Web services-based applications. It builds upon the stable
core of previous releases and provides key improvements in the area of systems
software development to support the latest specifications and programming
models. Together, these features further expand on WebSphere Application
Server platform coverage, runtime management capabilities, and application
deployment options to help you decrease costs and grow your business.

In addition, WebSphere Application Server V7 provides the following features:

► Quality of service (QoS)

   A set of Web services standards that supports the creation and administration
   of reliable, securable, and transactionable Web services applications.

► Interoperability

   Extensive Web services support, which makes it easier to integrate
   applications inside the enterprise and externally with customers, partners,
   and suppliers.

► SOA

   A secure and scalable SOA run time, which provides resource-efficient
   features and a faster run time with a new high-performance Web services
   engine.

► Security

   Web services security model enhancements, including WS-I Reliable Secure
   Profile support and policy sets for simple QoS definition.

► Programming model

   Fast new Java API for XML Web services (JAX-WS) engine with improved
   administration for Web services applications.

1.2 Web services roadmap

WebSphere Application Server V6.1 added support for the Web services Feature Pack that introduced the third engine for Web services in the WebSphere family. In many respects, the three generations of Web services engines reflected the last three stages in the history of Web services:

► Web Services Description Language (WSDL) and SOAP specifications
► Web Services Standards
► Web Services Interoperability (WS-I) Organization profiles

The feature pack included an implementation of JAX-WS 2.0, SOAP 1.1, Reliable Asynchronous Messaging Profile (RAMP), and Basic Profile 1.0 functionalities as part of its core distribution.

> **More information:** For additional information about the Web services Feature Pack, see *Web Services Feature Pack for WebSphere Application Server V6.1*, SG24-7618.

WebSphere Application Server V7 supports Web services for Java Platform JSR-109 Version 1.2, which means that it fully supports the new edition of Web services for Java EE, which includes Java Platform, Standard Edition (SE) 6. This support provides even more flexibility than before with regard to the new programming models and qualities of service, reflecting the progress in the world of Web services in recent years.

WebSphere V7 supports the following standards and specifications:

► Transport, SOAP, and XML description (Figure 1-1)

Technology or Specification	WebSphere Application Server 7.0
Transports	
HTTP/HTTPS	• V1.0 and V1.1
JMS	• Supported for JAX-RPC EJB Web services • Supported for JAX-WS EJB Web services
SOAP specification	
SOAP specification	• V1.1 supported for all Web services • V1.2 support limited to JAX-WS Web services
SOAP attachments	• SAAJ 1.2 and 1.3
SOAP MTOM	• V1.0 support limited to JAX-WS Web services
Description	
UDDI	• Unit Test UDDI wizard creates V3.0 registries • Web Services Explorer works with V2.0 and 3.0 registries
WSDL	• V1.1
WSIL	• V1.0

Figure 1-1   WebSphere Application Server V7 support

► Other core standards, such as JSRs, JAX-WS, and JAX Binding (JAXB) (Figure 1-2)

Technology or Specification	WebSphere Application Server 7.0
Other Standards (Java Specification Requests)	
JSR 109 and JSR 921	• JSR 109 1.0 – J2EE 1.3 • JSR 921 1.0 – J2EE 1.4 • JSR 109 1.1 – Java EE 5 • JSR 109 1.2 – Java EE 5
Other Standards (Web services engines)	
JAX-RPC	• V1.0 for J2EE 1.3 • V1.1 for J2EE 1.4
JAX-WS	• V2.0, 2.1
Other related Standards (APIs)	
JAXB/JSR-222	•V2.0, V2.1
JSR-181 - Web Services Metadata (Annotations)	•V2.0 for JAX-WS Web services •Not supported for JAX-RPC Web services
JSR-175 - Metadata Facility for the Java Programming Language	•V2.1 for JAX-WS Web services •Not supported for JAX-RPC Web services

Figure 1-2   WebSphere Application Server V7 support

1.2.1 JSR-109 Web services for Java EE Version 1.2

The goal of the maintenance release for JSR-109 Web services for Java EE Version 1.2 is to align JSR-109 to the latest Web service specification and to fix programming errors and inconsistencies in the previous version of the specification. Since the last release, further progress has been made with several relevant standards and profiles. Transport protocols, descriptors, messaging, security, and interoperability are many of the important updates.

This maintenance release includes the following major updates:

► New for developers:

– WS-I Basic Profile 1.2

– JAX-WS 2.1 - JSR-224

– JAXB 2.1 - JSR-222

– Web services Metadata 2.0 (JSR-181) and a Metadata Facility for the Java Programming Language (JSR-175)

– SOAP protocol 1.2

- SOAP Message Transmission Optimization Mechanism (MTOM)
- Web services Security (WS-Security) 1.1 enhancements
- Web services ReliableMessaging (WS-ReliableMessaging) 1.1
- Web services SecureConversation (WS-SecConv) 1.3
- WS-MEX
- Web services Policy (WS-Policy) and policy sets
- New for security specialists:
  - Configuring the Kerberos token profile 1.1 for WS-Security
  - General JAX-WS default bindings for WS-Security
  - Multiple security domains
- New for administrators:
  - Adding assured delivery to Web services through WS-ReliableMessaging with WS-I Secure Reliable Profile (WS-I SRP) 1.0
  - Configuring transaction properties for an application server
  - Web services Transaction (WS-Transaction) policy types for WS-AtomicTransaction (WS-AT) and WS-BusinessActivity (WS-BA) protocols
  - Using SOAP over Java Message Service (JMS) to transport Web services

WebSphere Application Server V7 supports JSR-109 1.2 and, therefore, also supports all of the features mentioned here.

> **More information:** You can obtain the JSR-109 V1.2 specification at the following address:
>
> http://jcp.org/aboutJava/communityprocess/maintenance/jsr109/jsr-109 -changelog-1_2-fcs.html

1.2.2 Web Services Interoperability

The WS-I Organization is an open industry organization that is designed to promote Web service interoperability across platforms, operating systems, and programming languages. It was founded specifically with the intent of facilitating interoperability of Web services between various vendor products and to clarify where gaps or ambiguities exist between the various standards.

There are various standards organizations, such as the World Wide Web Consortium (W3C), Organization for the Advancement of Structured Information

Standards (OASIS), and Internet Engineering Task Force (IETF). All of these organizations work to publish Web services standards. Each standard has been developed to address a specific Web services problem set. Developers who are in charge of building a Web services solution are required to discover, interpret, and apply the rules of multiple Web services standards. Even within an enterprise, multiple development teams are likely to interpret and apply the rules for the group of standards differently. WS-I has formulated the concept of profiles to solve this problem and to reduce complexity.

All organizations that are interested in promoting interoperability among Web services are encouraged to become members of the WS-I Organization.

WS-I Basic Profile updates

The WS-I Basic Profile is a set of non-proprietary Web services specifications that promote interoperability. The WS-I Basic Profile is governed by a consortium of industry-leading corporations, including IBM, under the direction of the WS-I Organization.

WS-I Basic Profile is an outline of requirements to which WSDL and Web service protocol (SOAP/HTTP) traffic must comply to claim WS-I conformance. It consists of a set of principles and existing specifications that relate to defining open standards for Web services technology and introducing the restrictions necessary to improve interoperability.

Several technology components are used in the composition and implementation of Web services, including messaging, description, discovery, and security. Each component is supported by specifications and standards. The WS-I Basic Profile specifies how these technology components are used together to achieve interoperability and mandates the specific use of each technology when appropriate.

WebSphere Application Server V7 conforms to WS-I Basic Profile Version 1.1, WS-I Basic Profile Version 1.2, and WS-I Basic Profile Version 2.0.

WS-I Basic Profile Version 1.1

The WS-I Basic Profile Version 1.1 requires support for the following specifications:

► XML 1.0

► HTTP 1.1

► SOAP 1.1 (over HTTP 1.1)

► WSDL 1.1

- Universal Description, Discovery, and Integration (UDDI) 2.03
- HTTP over Transport Layer Security (TLS) or Secure Sockets Layer (SSL) 3.0 (not mandatory)

> **More information:** WS-I Basic Profile Version 1.1 can be found at the following address:
>
> http://www.ws-i.org/Profiles/BasicProfile-1.1.html

WS-I Basic Profile Version 1.2

WS-I Basic Profile V1.2 builds on WS-I Basic Profile V1.0 and 1.1 and adds the following support:

- SOAP 1.1
- RFC2616: HTTP/1.1
- RFC2965: HTTP State Management Mechanism
- WS-Addressing 1.0 - Core
- WS-Addressing 1.0 - SOAP Binding
- WS-Addressing 1.0 - WSDL Binding
- SOAP 1.1 Request Optional Response HTTP Binding
- SOAP Message Transmission Optimization Mechanism
- XML-Binary Optimized Packaging
- SOAP 1.1 Binding for MTOM 1.0

At the time this book was written, Version 1.2 had not been finalized. The draft is currently available on the WS-I official Web site:

http://www.ws-i.org/Profiles/BasicProfile-1.2.html

WS-I Basic Profile Version 2.0

The WS-I Basic Profile Version 2.0 is a follow-on version of the Basic Profile Version 1.2 with the addition of the following support:

- SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)
- SOAP Version 1.2 Part 2: Adjuncts (Second Edition)
- RFC2616: HTTP/1.1
- RFC2965: HTTP State Management Mechanism
- WS-Addressing 1.0 - Core
- WS-Addressing 1.0 - SOAP Binding
- WS-Addressing 1.0 - Metadata
- SOAP MTOM
- XML-Binary Optimized Packaging
- UDDI 3

At the time this book was written, this specification was still a working group draft version.

> **More information:** WS-I Basic Profile Version 2.0 can be found at the following address:
>
> http://www.ws-i.org/Profiles/BasicProfile-2_0(WGD).html

WS-I Reliable Secure Profile 1.0

The RAMP profile comprises the WS-I Basic Profile 1.1 and WS-I Basic Security Profile 1.0 and adds the following specifications:

- ► WS-Addressing
- ► WS-ReliableMessaging
- ► WS-SecureConversation

The major difference in the proposed work is that, for the new Reliable Secure Profile 1.0 (RSP), WS-I includes WS-Addressing in an amended version of the WS-I Basic Profile 1.1 that is called *Basic Profile 1.2*. Additionally, to address the interoperability of attachments support, support for MTOM/XOP in a SOAP1.1 context is considered. Also, when the rechartered Basic Profile workgroup completes its work on Basic Profile 1.2, it then begins work on Basic Profile 2.0 that is based on SOAP1.2 and MTOM/XOP.

Figure 1-3 illustrates the relationship between the WS-I Basic Profile specification and WS* standards.
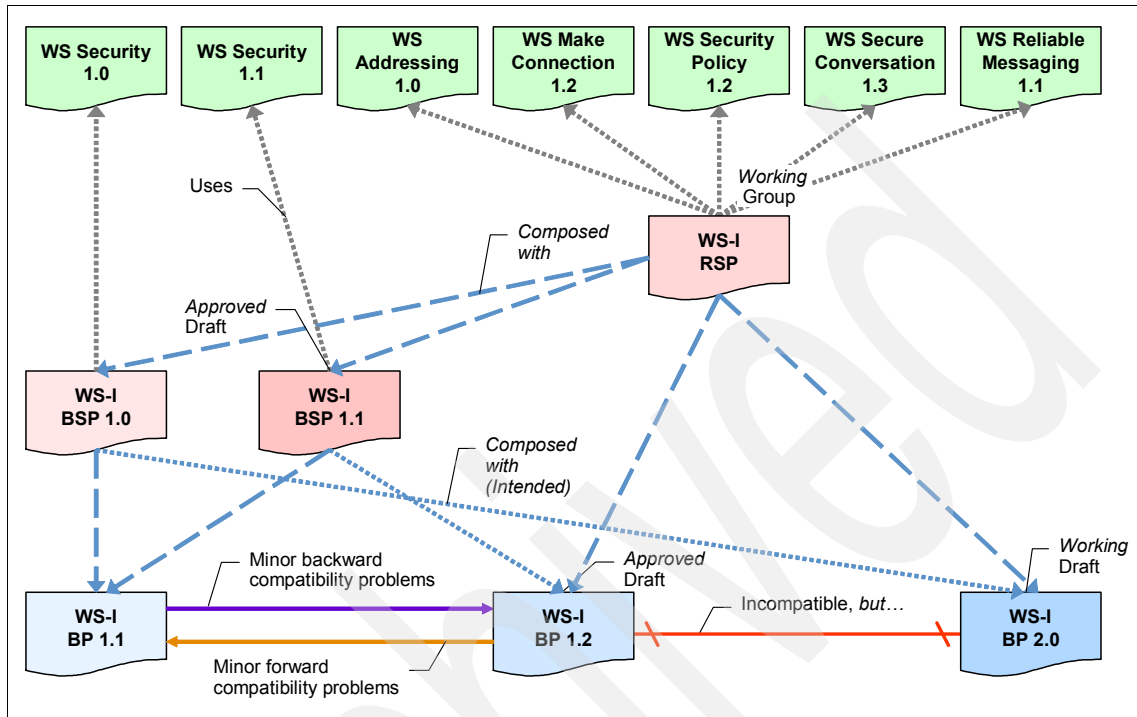
Figure 1-3   WS-I Basic Profiles specification and WS-Standards relationship

Figure 1-4 shows that the proposed Secure Reliable Profile 1.0 comprises the remaining specifications in the RAMP profile after moving WS-Addressing to the basic profile. It is designed so that it comprises versions 1.2 and 2.0 of the basic profile. The WS-ReliableMessaging and WS-SecureConversation specifications provide bindings to both SOAP 1.1 and SOAP 1.2. Figure 1-4 shows a complete table with WS-Profiles for WebSphere Application Server V7.

Technology or Specification	WebSphere Application Server 7.0
Interoperability	
WS-I Basic Profile	• 1.1.2, 1.2, 2.0
WS-I Simple SOAP Binding Profile	• 1.0.3, 1.1
WS-I Attachments Profile	• 1.0
WS-I Basic Security Profile	• 1.0
WS-I Reliable Secure Profile	• 1.0

Figure 1-4   WS-Profiles for WebSphere Application Server V7

1.3 Web services core technologies overview

In this section we discuss the following standards as part of the core technologies for Web services:

► SOAP
► JAX-WS
► JAXB
► Web Services Invocation Framework (WSIF)

We introduce each technology and explain what it contributes to the new JSR-109 V1.2.

1.3.1 SOAP 1.2

SOAP 1.2 provides a more specific definition of the SOAP processing model. It removes many of the ambiguities that sometimes lead to interoperability problems in the absence of the WS-I profiles.

Support for SOAP 1.2 has been added to JAX-WS 2.1, which supports SOAP Versions 1.1 and 1.2. This allows you to send binary attachments, such as images or files, along with Web services requests, which adds support for the optimized transmission of data as specified by MTOM. For more information about MTOM, see 1.3, "Web services core technologies overview" on page 18.

With regard to the SOAP 1.2 specification, there are a few practical differences between SOAP 1.1 and SOAP 1.2:

► SOAP 1.2 has been rewritten in terms of XML information sets (Infoset).

► SOAPAction is optional.

► SOAP 1.2 adds a few new attributes and more crisply defines several existing attributes.

► SOAP 1.2 adds a few new fault codes.

► The SOAP encoding and remote procedure call (RPC) definitions have been cleaned up. For example, the styleEncoding attribute is no longer supported in SOAP 1.2.

Figure 1-5 shows the relationship between SOAP standards and WS-I Basic Profiles.
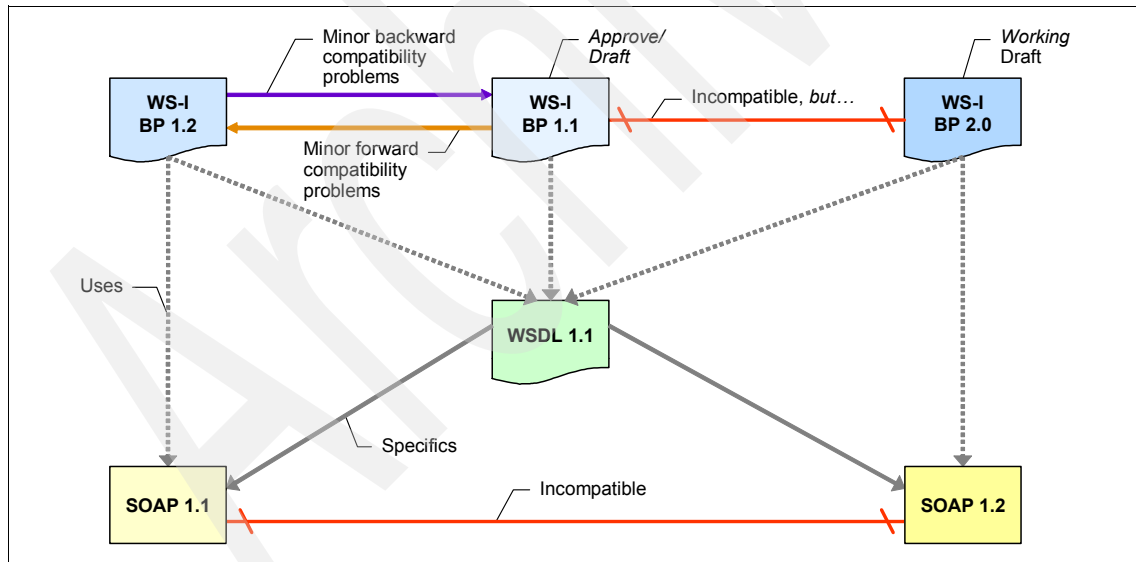
Figure 1-5   SOAP standards and WS-I Basic Profiles relationship

In addition, it illustrates the dependency between the WS profile standards and the SOAP specifications, thereby dictating how to build an application that is fully compliant with WS-I Organization Basic Profiles specifications.

SOAP 1.2 is not incorporated into version 1 of the WS-I Organization Basic Profiles. SOAP 1.2 is planned for Basic Profile 2.0.

> **More information:** For more information see the official SOAP 1.2 specification Web site at the following address:
>
> http://www.w3.org/TR/2003/REC-soap12-part0-20030624

1.3.2 JAX-WS 2.1

JAX-WS, previously known as JSR-224, is a required part of Java SE 6 that delivers a new programming model for developing Web services. JAX-WS simplifies Java applications with more platform independence by using new features, such as dynamic clients, asynchronous invocations, and dependency injection with Java 5 annotations.

JAX-WS is the centerpiece of a newly re-architected API stack for Web services, called the *integrated stack*, that strategically aligns with the current industry trend toward a more document-centric messaging model. The integrated stack supersedes the foundation that was provided by the RPC programming style that was previously defined by JAX-RPC API. This new foundation represents a logical re-architecture of Web services functionality in the open-source Java EE 5-compliant application server. JAX-WS JSR-224 is designed to take the place of JAX-RPC in Web services and Web applications.

While the JAX-RPC programming model and applications are still supported by WebSphere Application Server V7, JAX-RPC has limitations and does not support various complex document-centric services. Figure 1-6 outlines the main features for JAX-WS 2.1.
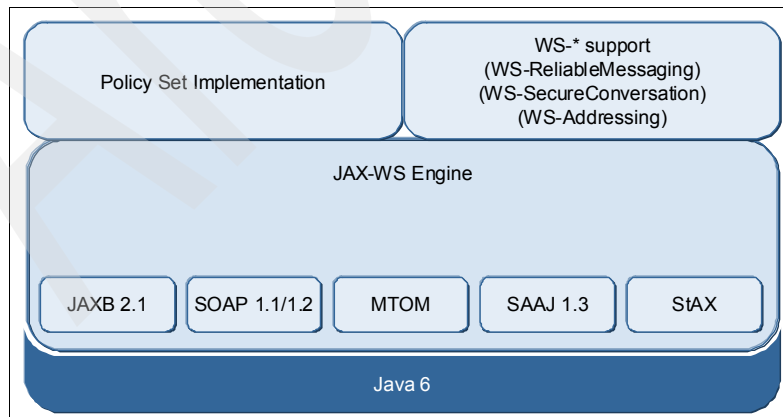
Figure 1-6   JAX-WS 2.1

The implementation of the JAX-WS 2.1 programming model provides many enhancements for developing Web services applications. The following sections provide insight into JAX-WS features and enhancements to previous technologies.

Features

JAX-WS 2.1 introduces the concept of features as a way to programmatically control specific functions and behaviors. Three standard features are specified as follows:

► jaax.xml.ws.soap.AddressingFeature for WS-Addressing reliable messaging

► jaax.xml.ws.soap.MTOMFeature when optimizing the transmission of binary attachments

► jaax.xml.ws.soap.RespectBindingFeature for wsdl:binding extensions

Better platform independence for Java applications

By using JAX-WS APIs, the development of Web services and clients is simplified with better platform independence. JAX-WS takes advantage of the dynamic proxy mechanism to provide a formal delegation model with a pluggable provider, which is an enhancement over JAX-RPC, which relies on the generation of vendor-specific stubs for invocation.

Extended support for WS-Addressing in an API

With the new WS-Addressing standard API, you can create, transmit, and use endpoint references to target a specific Web service endpoint. You can also explicitly specify the action URIs associated with the WSDL operations of your Web service.

Support for annotations

JAX-WS introduces support for annotating Java classes with metadata to indicate that the Java class is a Web service. JAX-WS supports the use of annotations that are based on the Metadata Facility for the Java Programming Language (JSR-175) specification and the Web services Metadata for the Java Platform (JSR-181) specification.

The use of annotations within the Java source simplifies the development of Web services. You use annotations to define information that is typically specified in deployment descriptor files, such as WSDL, or by mapping metadata from XML and WSDL files into the source artifacts.

Example 1-1 shows a Java bean that contains the @javax.jws.WebService annotation and then exposes the bean as a Web service (service implementation class (service endpoint interface (SEI)).

Example 1-1   Java bean containing the @javax.jws.WebService annotation

```
@javax.jws.WebService
public class WeatherBean implements IWeatherForecast{
    public Weather getDayForecast(Calendar theDate) {
        ...
    }
}
```

The use of annotations also improves the development of Web services within a team structure, because you do not need to define every Web service in a single or common deployment descriptor as was required with JAX-RPC Web services. Taking advantage of annotations with JAX-WS Web services enables parallel development of the services and metadata information.

> **The webservices.xml deployment descriptor**: Using the `webservices.xml` deployment descriptor is now optional for JAX-WS services, because you can use annotations to specify all the information that is within the deployment descriptor file. You can use the deployment descriptor file to augment or override existing JAX-WS annotations. Any information that you define in the `webservices.xml` deployment descriptor overrides any corresponding information that is specified by annotations.

Support for resource injection

JAX-WS provides support for a subset of annotations that are defined in the JSR-250 specification for resource injection and application life cycle to further simplify the development of Web services.

The @Resource and @WebServiceRef annotations are part of the Common Annotations specification that is delivered by the JSR-250 specification and included in Java EE 5. JAX-WS uses these key features of Java EE 5 to shift the burden of creating and initializing common resources in a Java runtime environment™ (JRE™) from your Web service application to the application container environment.

The @Resource annotation is used in the wsContext field to obtain an instance of a WebServiceContext object and then to invoke the getMessageContext() method and work with the MessageContext object. Example 1-2 shows that by placing the @Resource annotation on a variable of the type javax.xml.ws.WebServiceContext within a service endpoint implementation class you can request a resource injection and collect the javax.xml.ws.WebServiceContext interface that is related to that particular endpoint invocation.

Example 1-2   @Resource annotation

```
@javax.jws.WebService
public class ProxyProvider implement Provider {
   @Resource WebServiceContext wsContext;

   public SOAPMessage invoke(SOAPMessage input) {
        MessageContext mc = wsContext.getMessageContext();
        //…
   }
}
```

In Example 1-3, the application server, through JAX-WS support, also accepts the use of the @WebServiceRef annotation to request injection of JAX-WS services and ports.

Example 1-3   @WebServiceRef annotation

```
@javax.jws.WebService
public class ProxyProvider implement Provider {
   //WebServiceRef using the generated service interface type
   @WebServiceRef
   public StockQuoteService service;

   //WebServiceRef using the SEI type
   @WebServiceRef(StockQuoteProvider.class)
   private StockQuoteProvider provider;

   //......
}
```

In conclusion, either one of these annotations can be used on a field or method and result in injection of a JAX-WS service or port instance. The usage of these annotations also results in the type that is specified by the annotation being bound into the Java Naming and Directory Interface (JNDI) namespace.

> **Important:** Usage of the @Resource and @WebServiceRef annotations must be applied to JAX-WS-managed clients. If you apply these annotations to a non-managed Web service client, it will not work.

For more information see Chapter 2, "Web services programming model" on page 59.

You can also see 5.2.1 and 5.3 of the JAX-WS specification for further information about resource injection and the JSR-250 specification.

Dynamic and static clients

The static client programming model is called proxy client and is conceptually implemented by the JAX-RPC applications. The proxy client uses the javax.xml.rpc.Call object to invoke a Web service based on an SEI that must be provided at compile time. The Web services application does *not* need to get access to the WSDL artifact at run time, and the binding is performed statically.

The dynamic client API for JAX-WS, which is a XML messaging-oriented client model, uses the javax.xml.ws.Dispatch object implementation to provide support for operating at either of the following levels of interaction:

► Hiding the details of converting between the Java method invocation and the corresponding XML messages

► Operating at the XML message level to obtain more control over the message

The dynamic client does not work at compile time and requires access to the service implementation at run time, thereby discovering the service and binding dynamically.

The Dispatch implementation also supports two usage modes, which are identified by the constants javax.xml.ws.Service.Mode.MESSAGE and javax.ws.xml.Service.Mode.PAYLOD.

In addition, the dynamic client also supports asynchronous invocations by using a callback or polling mechanism. JAX-WS does not add additional information to the message.

> **More information:** For further information about dispatch objects and dynamic invocation, see Chapter 2, "Web services programming model" on page 59.

Implementation models

With JAX-WS, Web services are called both synchronously and asynchronously. JAX-WS adds support for both a *polling* mechanism and a *callback* mechanism when calling Web services asynchronously. By using a polling model, a Web service client can issue a request and get a response object back. The response object is polled to determine whether the server has responded. When the server responds, the actual response is retrieved. On the other hand, by using the callback model, the client provides a callback handler implementation to accept and process the inbound response object.

Both polling and callback mechanisms enable the Web service clients to continue to process work without waiting for a response to be returned, thereby providing a more dynamic and efficient model to invoke Web services.

> **More information**: For further details about asynchronous messages, see Chapter 2, "Web services programming model" on page 59.

MTOM/XOP

The Message Transmission Optimization Mechanism is a mechanism for sending binary data, which is often called an *attachment*, along with Web services requests. Prior to MTOM, there was no universally accepted interoperable way to transmit attachments, although SOAP with Attachments (SwA) came close.

JAX-WS 2.1 adds support for the optimized transmission of binary data as specified by MTOM and dictates that a compliant Web service engine must support MTOM and SwA.

MTOM uses the XML-binary Optimized Packaging (XOP) in the context of SOAP and Multipurpose Internet Mail Extensions (MIME) over HTTP to define a serialization mechanism for the XML Infoset with binary content that is applicable to SOAP and MIME packaging, as well as any XML Infoset and packaging mechanism.

WebSphere Application Server V7.0 supports MTOM and SwA Versions 1.2 and 1.3.

> **More information:** For further details about asynchronous messages see Chapter 2, "Web services programming model" on page 59.

Command-line tools to generate portable artifacts

JAX-WS provides the `wsgen` and `wsimport` command-line tools to generate portable artifacts for JAX-WS Web services. When creating JAX-WS Web services, you can start with either a WSDL file or an implementation bean class.

If you start with an *implementation bean class*, use the `wsgen` command-line tool to generate all the Web services provider artifacts, including a WSDL file if requested.

If you start with a *WSDL file*, use the `wsimport` command-line tool to generate all the Web services artifacts for either the server or the client. The `wsimport` command-line tool processes the WSDL file with schema definitions to generate the portable artifacts, which include the service class, the service endpoint interface class, and the JAXB 2.1 classes for the corresponding XML schema.

More information: For any further information about JAX-WS Version 2.1, see Chapter 2, "Web services programming model" on page 59.

Extensions to Web services clients

WebSphere Application Server provides extensions to Web services clients using the JAX-WS programming model. You can customize Web services by using the following extensions to the JAX-WS client programming model:

► SOAP headers

  Set the JAXWS_OUTBOUND_SOAP_HEADERS and JAXWS_INBOUND_SOAP_HEADERS properties on the request context of the dispatch or proxy object to enable a JAX-WS Web services client to send or retrieve implicit SOAP headers.

► HTTP headers

  Set the REQUEST_TRANSPORT_PROPERTIES and RESPONSE_TRANSPORT_PROPERTIES properties to enable a Web services client to send or retrieve transport headers.

More information: For further information see "Implementing extensions to JAX-WS Web services clients" in the WebSphere Application Server - Express Information Center at the following address:

`http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/index.jsp?topic=`
`/com.ibm.websphere.express.iseries.doc/info/iseriesexp/ae/twbs_extend`
`pmjaxws.html`

Transport neutral

JAX-WS 2.1 provides support for HTTP/HTTPS. Other transports are planned for future releases.

Application handlers

JAX-WS helps you to insert and retrieve data from a message as it moves through the Web services engine. *Handlers* are simple Java beans that implement a handler contract and can be associated with Web services endpoints and Web services clients, allowing the interception of a message at various points in its transmission. JAX-WS provides two levels of handlers:

- ► *Logical handlers* deal with the payload level of the message.
- ► *Protocol handlers* deal with protocol information, such as SOAP headers.

More information: For more information about handlers, see Chapter 2, "Web services programming model" on page 59.

1.3.3 JAXB 2.1

For historical reasons, there was a considerable overlap of the data binding functionality between the JAX-RPC 1.*x* and JAXB 1.*x* APIs in the previous Web services stack. JAX-RPC 1.*x* originally included basic data binding functionality. When JAXB 1.*x* emerged after JAX-RPC and when data binding functionality became more comprehensive with enhanced standards, such as XML Schema, the need for separating the Web services definition and the data binding components became more obvious.

Java Architecture for XML Binding (JAXB 2.1) API, which is specified by JSR-222, replaces the data binding that is described by the JAX-RPC specification. JAXB 2.1 provides enhancements, such as improved compilation and annotation support, to leverage the flexibility of platform-neutral XML data in Java applications to bind XML schema to Java applications without requiring an extensive knowledge of XML programming.

JAX-WS relies on XML binding technology, which consists of a runtime API and accompanying tools that simplify access to XML documents, as the primary technology for the default two-way data binding mappings between Java objects and XML documents that both conform to and validate to the XML schema.

The result is an easier-to-understand architecture for Web services development to build Web applications and Web services, as shown in Figure 1-7.
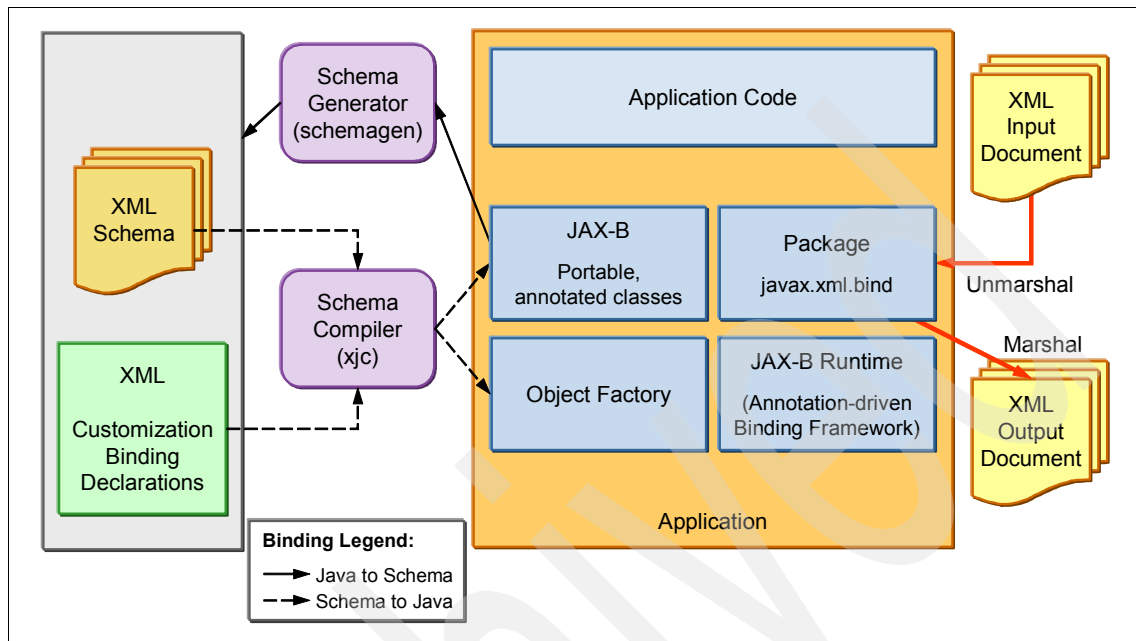
Figure 1-7   XML binding technology

JAXB annotated classes and artifacts contain all the information that is needed by the JAXB runtime API to process XML instance documents. The JAXB runtime API supports marshaling of JAXB objects to XML and unmarshalling the XML document back to JAXB class instances. Optionally, you can use JAXB to provide XML validation to enforce both incoming and outgoing XML documents to conform to the XML constraints that are defined within the XML schema.

New for JAXB 2.1

The new features are:

► Tools

With the improved compilation support, we now have the flexibility to control whether a new schema file is generated when using the `schemagen` schema generator. Also, you can configure the `xjc` schema compiler so that it does not automatically generate new classes for a particular schema.

► Annotations

You can also use the @XMLSeeAlso annotation to know about all classes that are potentially involved in marshaling or unmarshalling, because it is not

always possible or practical to list all of the subclasses of a given Java class. JAX-WS 2.1 also supports the use of the @XMLSeeAlso annotation on an SEI or on a service implementation bean to ensure that all of the classes that are referenced by the annotation are passed to JAXB for processing.

> **Note:** You can obtain detailed information about the JAXB 2.1 features in Chapter 2, "Web services programming model" on page 59.

1.3.4 Web Services Invocation Framework

WSIF aims to extend the flexibility provided by SOAP services into a general model for invoking Web services, irrespective of the underlying binding or access protocols.

SOAP bindings for Web services are part of the WSDL specification, which for extensibility enables points that describe alternate ways of invoking a Web service. Therefore, when you think of using a Web service, you probably think of assembling a SOAP message and sending it across the network to a service endpoint, using a SOAP client API.

While this process works for SOAP, it is limited in its use as a general model for invoking Web services for the following reasons:

► Web services are more than just SOAP services.
► Tying client code to a particular protocol implementation is restricting.
► Incorporating new bindings into client code is difficult.
► Multiple bindings can be used in flexible ways.
► A freer Web services environment enables intermediaries.

Therefore, the goals of the WSIF are:

► To define a binding-independent mechanism for Web service invocation

► To free client code from the complexities of any particular protocol that is used to access a Web service

► To enable dynamic selection between multiple bindings to a Web service

► To help the development of Web service intermediaries

Depending on the type of Web service that is created, you might want your Web service to comply with the WS-I profiles. For example, the default level of compliance is to generate a warning if a non-WS-I Simple SOAP Binding Profile 1.0 (WS-I SSBP)-complaint Web service option is selected and to ignore any non-WS-I Attachments Profile 1.0 (WS-I AP)-compliant selections. However, you can set the level of WS-I compliance at the workspace or project level. The Web services wizards, the WebSphere runtime environments, the WSDL editor, and

other Web services tools provide support and encourage the development of WS-I-compliant services.

WSIF clients

A WSIF client can make use of non-SOAP descriptions to invoke a service in a more efficient way. For example, a Web service provider might offer a SOAP binding for the service and a local Java binding, which enables you to treat the local service implementation (a Java class) as a Web service. If the client is deployed in the same environment as the service, the local Java binding for the service can be used, which provides more efficient communication with the service by making direct Java calls rather than sending and receiving SOAP messages through the network.

To deploy a Web service as a WSIF-enabled service, you first develop and deploy the Web service, and then you develop the WSIF client based on the WSDL document for that Web service.

> **More information:** For more information see the following Web sites:
>
> ► For more information about WS-I, see the Web Services Interoperability Organization Web site at the following address:
>
>   http://www.ws-i.org/
>
>   This site contains resources, such as an overview of Web services interoperability, usage scenarios, and specifications.
>
> ► *Learning about the Web services Invocation Framework (WSIF)*
>
>   http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/topic/com.ibm .websphere.nd.doc/info/ae/ae/twsf_learning.html

1.4 WS-* standards

A variety of specifications are associated with Web services. These specifications vary in their degree of maturity and are maintained or supported by various standards bodies and entities. Specifications can complement, overlap, and compete with each other. Web service specifications are occasionally collectively referred as *WS-**. The reference term *WS-** is more of a generalization, because many specifications use *WS-* as their prefix. This section includes many of the specifications that are considered the *core* part of WS-* for Web services JSR-109 V1.2 and WebSphere 7.

WS-* standards are classified by non-functional requirements such as the QoS, for example, messaging transmission, security, transaction, and management.

Figure 1-8 shows the core WS-* standards for Web services development supported by WebSphere Application Server V7.

Technology or Specification	WebSphere Application Server 7.0
Messaging	
WS-ReliableMessaging	• OASIS Standard 1.1
WS-Addressing	• W3C 1.0
WS-SecureConversation	• OASIS Standard 1.3
WS-Notification	• OASIS Standard 1.3
WS-Resource	• OASIS Standard 1.2
Security	
WS-Security	• OASIS Standard 1.1
WS-Policy	• W3C 1.5
WS-Metadata Exchange	• W3C 1.1
Security Kerberos Token profile	• 1.1
WS-SecurityPolicy	• 1.2
WS-Trust	• OASIS Standard 1.3
Transaction	
WS-Transaction	• OASIS Standard 1.1
WS-BusinessActivity	• OASIS Standard 1.1
WS-Coordination	• OASIS Standard 1.1

Figure 1-8   Core WS- standards used to develop the JSR-109 V1.2 support*

1.4.1 WS-ReliableMessaging

The WS-ReliableMessaging specification describes a protocol that allows messages to be delivered reliably between distributed applications. The WS-ReliableMessaging protocol ensures that error conditions are detectable and facilitates the successful transmission of messages that are sent synchronously or asynchronously from a source to a destination.

Figure 1-9 illustrates the core environment for WS-ReliableMessaging and the relevant standards and APIs to which it relates.
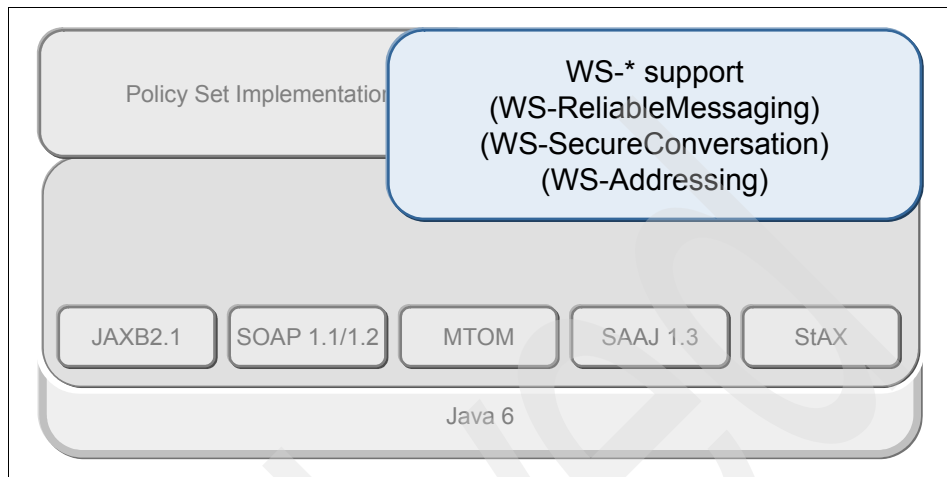
Figure 1-9   WS-ReliableMessaging

Figure 1-10 shows a sequence diagram that shows how the WS-ReliableMessaging flow works.
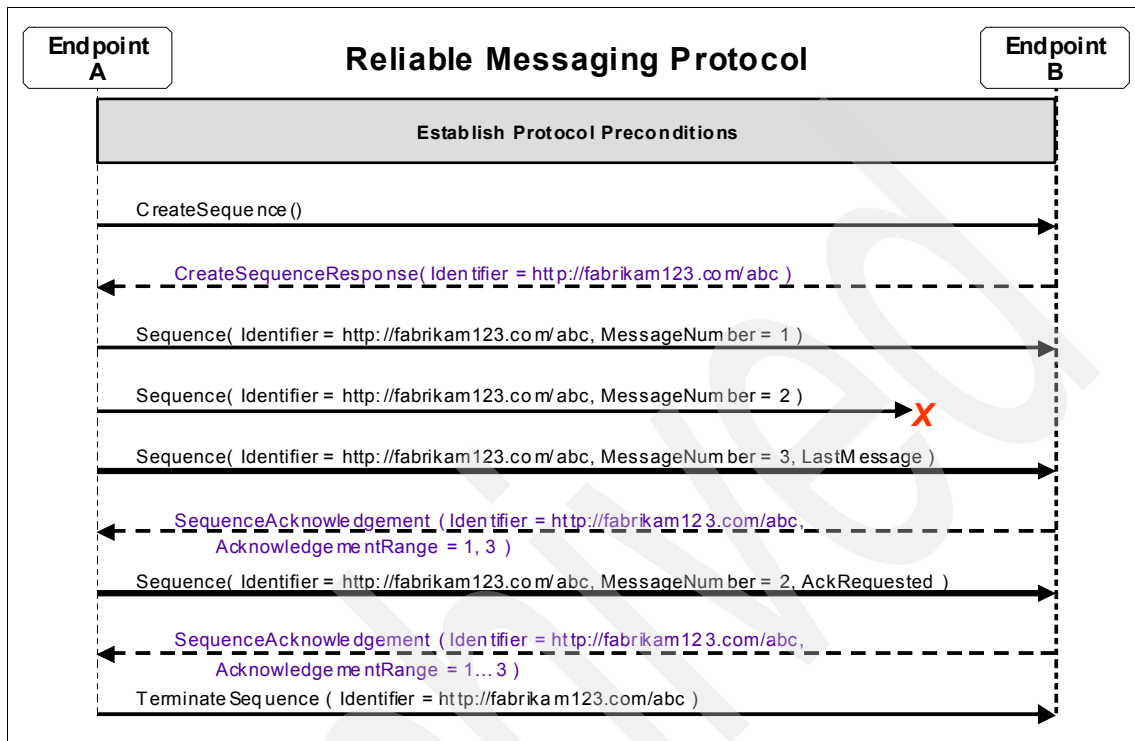
Figure 1-10   WS-ReliableMessaging flow

When a Web service client issues a request synchronously, WS-ReliableMessaging information is sent in the headers of the normal request and response messages. However, when you break the response from the request in asynchronous messaging, endpoint B needs to know the address of endpoint A so that it can send responses back to A. Likewise, in reliable asynchronous messaging, which is shown in Figure 1-10, the acknowledge message might be sent from B to A outside of the normal response messages, and B again needs A's address. The mechanism for exchanging this address information is defined by the WS-Addressing specification. To support interoperable Web services, a SOAP binding is defined. The protocol is transport-independent, which allows it to be implemented using various network technologies.

WS-ReliableMessaging is, therefore, a requirement of many enterprise systems. Because large companies are adopting architectural systems based on Web

services, a new specification was written to address these communication issues and to help then to meet the industry requirements.

> **Note:** WS-ReliableMessaging is *not* a messaging and queuing system, such as WebSphere MQ, nor is it a message service, such as JMS. The most obvious distinction is that the Web services reliable messaging mechanism provides *qualities of service* that are applied to application services, as compared to JMS and WebSphere MQ, which provide distributed messaging and queuing software layers on top of which applications are built. The goal of WS-ReliableMessaging is to make the service request traverse the Internet reliably and securely. WebSphere MQ and JMS provide a platform for building reliable, loosely coupled, message-driven, distributed applications that can be deployed to multiple networks, such as the Internet.

New for WS-ReliableMessaging

Support for the WS-ReliableMessaging standard was first introduced as part of the WebSphere Application Server 6.1 Feature Pack for Web services. At that time, the RAMP Version 1.0 specification, which was used to ensure the reliable delivery of messages, was included as the default policy set.

With WebSphere Application Server V7, you can migrate from WebSphere Application Server V6.1 Feature Pack WS-ReliableMessaging configurations, which use RAMP 1.0-based policy sets, to WS-I RSP 1.0 policy sets.

As a result of this transition, the WS-ReliableMessaging 1.1 specification supports WS-I RSP policy sets and integrates with WS-SecureConversation 1.3, WS-Addressing, and many other Web services standards, such as WS-Security and WS-Policy, to secure messages. Combined, these Web services standards lead to a wide range of reliable and secure messaging options.

More information: For more information see the following Web sites:

- WS-ReliableMessaging specification

  http://docs.oasis-open.org/ws-rx/wsrm/200702

- WS-ReliableMessaging artifacts (schema, WSDL)

  http://www.ibm.com/developerworks/webservices/library/specification
  /ws-rm/?S_TACT=105AGX04&S_CMP=LP

- Learning about WS-ReliableMessaging

  http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/topic/com.ibm.
  websphere.express.iseries.doc/info/iseriesexp/ae/twbs_wsrm_learning.
  html

- WS-I Reliable Secure Profile specification

  http://www.ws-i.org/deliverables/workinggroup.aspx?wg=reliablesecure

- Web services Reliable Messaging Policy Assertion

  http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-rm
  /ws-rmpolicy200502.pdf

1.4.2 WS-Addressing

In a moderately complex system, a receiver of a message must know information about the sender. With an asynchronous request-response implementation model, the receiver must know at least the sender's address to send back responses that are disconnected from the request channel that was previously used by the Web services client to issue the request.

The purpose of the WS-Addressing specification is to provide an interoperable way of communicating between senders and receivers by providing a transport-neutral mechanism to address Web services and messages. The WS-Addressing specification defines XML elements to identify the Web services endpoints and to secure end-to-end endpoint identification in messages. This specification enables messaging systems to support message transmission through networks that include processing nodes, such as endpoint managers, firewalls, and gateways.

WS-Addressing is a World Wide Web Consortium (W3C) specification that aids interoperability between Web services by defining a standard way to address Web services and to provide addressing information in messages. The WS-Addressing specification introduces two primary concepts:

► Endpoint references

Endpoint references provide a standard mechanism to encapsulate information about specific endpoints. Endpoint references can be propagated to other parties and then used to target the Web service endpoint that they represent.

► Message addressing properties

Message addressing properties (MAPs) are a set of well-defined WS-Addressing properties that can be represented as elements in SOAP headers and provide a standard way of conveying information, such as the endpoint to which to a direct message replies, or information about the relationship that the message has with other messages.

New for WS-Addressing

For JAX-WS applications, you can enable WS-Addressing support in several ways, such as configuring policy sets or using annotations in code. You can now use JAX-WS 2.1 annotations and feature classes to do the following tasks:

► Enable WS-Addressing from either the server or the client.

► Have more control over the behavior of WS-Addressing when using policy sets.

► Specify whether WS-Addressing is enabled and whether to use synchronous, asynchronous, or both messaging patterns.

► Specify actions to be associated with a Web service operation or fault response.

For Web service clients, WS-Addressing support is disabled by default. For Web service providers, WS-Addressing support is enabled by default. Therefore, you do not have to enable this support. However, you can use the enabling mechanisms to modify other WS-Addressing behavior for the service, such as whether WS-Addressing information is required and what is included in the generated WSDL document.

The following additional features are related to the JAX-WS enhancements:

► Java representations of WS-Addressing endpoint references are available.

► You can create Java endpoint reference instances for the application endpoint, or other endpoints in the same application, at run time. You do not have to specify the URI of the endpoint reference.

- You can create Java endpoint reference instances for endpoints in other applications by specifying the URI of the endpoint reference.
- On services, you can use annotations to specify whether WS-Addressing support is enabled and whether it is required.
- On clients, you can use features to specify whether WS-Addressing support is enabled and whether it is required.
- You can configure client proxy or dispatch objects by using endpoint references.
- Java support for endpoint references that represent Web services Resource (WS-Resource) instances is available.
- You can associate reference parameters with an endpoint reference at the time of its creation to correlate it with a particular resource instance.
- In targeted Web services, you can extract the reference parameters of an incoming message so that the Web service can route the message to the appropriate WS-Resource instance.

For complete information about setting up provider policy settings, see Table 1 in the "Enabling Web services Addressing support for JAX-WS applications" topic at the following Web site:

http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/index.jsp?topic=/
com.ibm.websphere.base.doc/info/aes/ae/twbs_wsa_dep_jaxws.html

Features of the IBM proprietary WS-Addressing support

WebSphere Application Server V7 provides an IBM proprietary implementation of the WS-Addressing specification that you can use with JAX-WS applications to undertake more advanced functions, such as creating endpoint references that represent highly available objects, or directly setting message addressing properties in the SOAP header. Use these APIs if you want to create JAX-RPC applications that use addressing or if you want to undertake more advanced functions that are not possible with the JAX-WS 2.1 APIs.

The IBM proprietary API provides the following features:

- You can easily create Java endpoint reference instances to represent any endpoint in the server, based on the deployment environment of the application. You do not have to specify the URI of the endpoint reference. Additionally, endpoint references can represent highly available or workload-managed objects.
- You can configure client JAX-WS BindingProvider request context objects, or JAX-RPC Stub or Call objects, with a WS-Addressing endpoint reference. Future invocations to these objects are targeted at the endpoint that is represented by the endpoint reference. The invocations also automatically

conform to the WS-Addressing specification (namespace) that is associated with that endpoint reference.

More information: For more information see the following Web sites:

► WS-Addressing specification

http://www.w3.org/TR/ws-addr-core/

► Web services Addressing support

http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/topic/com.ibm.websphere.nd.doc/info/ae/ae/cwbs_wsa.html

1.4.3 WS-SecureConversation

WS-Security focuses on the message authentication model rather than the security context, which leads to several forms of security attacks. The Web services Secure Conversation Language (WS-SecureConversation or WS-SecConv) specification defines mechanisms to provide a secured session for long-running message exchanges and to use the symmetric cryptographic algorithm.

WS-SecureConversation provides session-based security. Session-based security optimizes and secures a sequence of message exchanges by using symmetric cryptography that can be used to sign and encrypt the messages. Typically, the symmetric cryptographic algorithm is less CPU-intensive than the asymmetric cryptography. Therefore, symmetric cryptographic algorithms provide better performance and throughput when compared to the asymmetric cryptographic algorithms. The symmetric cryptographic algorithm also provides a means to secure other session-based protocol and exchange patterns, such as WS-ReliableMessaging.

WS-SecureConversation, however, does not provide a complete security solution by itself. WS-SecureConversation is a building block that is used in conjunction with other standards and application-specific protocols, such as WS-Trust and WS-Security, to accommodate a wide variety of security models and technologies. WS-SecureConversation defines extensions to allow security context establishment and sharing, and session key derivation, which allows contexts to be established and, potentially, more efficient keys or new key material to be exchanged.

New for WS-SecureConversation

Figure 1-11 shows a comparison of the OASIS specifications between WebSphere Application Server versions. WS-SecureConversation 1.3 and WS-Trust 1.3 are now supported by WebSphere Application Server V7, which previously supported the Version 1.1 draft for both standards.

WebSphere Application Server 6.1 Feature Pack	WebSphere Application Server 7.0
WS-SecureConversation	
OASIS Standard 1.1 draft	• OASIS Standard 1.3

Figure 1-11   WebSphere Application Server support for WS-SecureConversation

WS-SecureConversation support

Several key functions are supported in WebSphere Application Server 7:

► A security context token (SCT) that is established between the initiating party and the receiving party is supported.

► The WS-SecureConversation operations, such as issue token, renew token, and cancel token, are supported on the SCT. The Validate token is supported by using the WS-Trust protocol.

► A derived key (explicit and implied) is supported.

> **More information**: For more information see the following sources:
>
> ► Chapter 10, "WS-SecureConversation" on page 471
>
> ► WS-SecureConversation specification
>
>   http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512/ws-secureconversation-1.3-os.html
>
> ► Web services secure conversation
>
>   http://www.ibm.com/developerworks/library/specification/ws-secon/
>
>   http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/topic/com.ibm.websphere.nd.doc/info/ae/ae/cwbs_wssecureconv.html

1.4.4 Web Services Resource Framework

The WSRF defines a system for creating stateful resources between Web services in terms of an implied resource pattern. The motivation for this new specification is that, while Web service implementations typically do not maintain state information during their interactions, their interfaces must frequently allow

for the manipulation of state, that is, data values that persist across and evolve as a result of Web service interactions.

The WSRF defines a family of specifications for accessing stateful resources by using Web services. It includes the following specifications:

► *WS-Resource Properties* define how the data that is associated with a stateful resource can be queried and changed by using Web services technologies. This capability allows a standard means by which data that is associated with a WS-Resource can be accessed by clients. The declaration of the WS-Resource's properties represents a projection, or view, of the WS-Resource's state. This projection represents an implied resource type, which serves to define a basis for access to the resource properties through Web service interfaces.

► *WS-Resource Lifetime* defines two ways of destroying a WS-Resource:

– Immediate
– Scheduled

This capability allows designers the flexibility to design how their Web services applications can clean up resources that are no longer needed.

► *WS-BaseFaults* defines an XML schema type for a base fault, along with rules for how this fault type is used by Web services. A designer of a Web services application often uses interfaces that were defined by other designers. Managing faults in this type of application is more difficult when each interface uses a separate convention for representing common information in fault messages. Support for problem determination and fault management can be enhanced by specifying Web services fault messages in a common way. When the information that is available in faults from various interfaces is consistent, it is easier for requestors to understand the faults. It is also more likely that common tooling can be created to assist in handling faults.

► *WS-ServiceGroup* defines a means by which Web services and WS-Resources can be aggregated or grouped together for a domain-specific purpose. In order for requestors to form meaningful queries against the contents of the ServiceGroup, membership in the group must be constrained. The constraints for membership are expressed by intension using a classification mechanism. Furthermore, the members of each intension must share a common set of information over which queries can be expressed.

WSRF programming model

The WSRF specifications define only the protocol messages and the semantic behavior that is expected of a WS-Resource when it processes these messages. The specifications do not prescribe the means to implement WS-Resource objects. WSRF is primarily an application-level protocol, and the tools for implementing WS-Resources are the same tools that are used for implementing

any other type of Web service. WSRF uses WS-Addressing endpoint references. The application programming model for WS-Resources is similar to the model for any Web service that uses WS-Addressing.

WSRF extends the WebSphere Application Server WS-Addressing programming model in two ways, which differentiate a WS-Resource from a generic resource that is accessed through a Web service by using WS-Addressing:

► WSRF requires the ResourceProperties attribute on the wsdlPortType element.

This attribute declares that the portType element is implemented by a WS-Resource rather than a generic Web service. The WS-Resource must declare which WSRF operations it supports by copying those operations into the portType element of its WSDL definition. The WS-Resource is free to choose any implementation strategy to represent the stateful resource and to process the WSRF messages. You can implement a resource by using a simple Java class, a stateless session enterprise bean, an entity bean backed by a relational database, a Service Data Object (SDO), and so on.

► WSRF defines a hierarchy of Java BaseFault types.

WS-Resource property and life-cycle operations
WSRF contains specifications that describe the operations that a WS-Resource can implement to get, set, or query the state of the resource by operating on the resource properties document.

For a complete description of all the standard property and lifetime operations that are defined by the WSRF, see "Web services Resource Framework resource property and life-cycle operations" at the following Web site:

http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/index.jsp?topic=/
com.ibm.websphere.nd.doc/info/ae/ae/rwbs_wsrf_ops.html

WS-Resource base faults
WSRF provides a recommended basic fault message element type from which you can derive all service-specific faults. The advantage of a common basic type is that all faults can, by default, contain common information. This behavior is useful in complex systems where faults might be systematically logged or forwarded through several layers of software before being analyzed.

To check the list of base faults, see "Web services Resource Framework base faults" at the following address:

http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/index.jsp?topic=/
com.ibm.websphere.nd.iseries.doc/info/iseriesnd/ae/cwbs_wsrf_basefault.
html

1.4.5 WS-Security

The WS-Security specification and its associated token profiles define a way to send security tokens and provide message integrity and confidentiality. The WS-SecureConversation specification establishes a secure context, based on shared keys, for the client and server to use for a series of messages. This standard provides a framework across organizations that defines how to secure the entire conversation. Use the WS-Security policy to define how the SOAP messages are secured. It has following options, among others:

► The option to define which message parts are signed and encrypted
► The option to specify the types of tokens to be included
► The option to choose whether to use symmetric or asymmetric cryptography

You can also use the WS-Security policies to define the bootstrap policy that is used to acquire security context tokens. Security context tokens are used by the WS-SecureConversation specification.

New for WS-Security

In WebSphere Application Server V7, there are many security enhancements for Web services. The enhancements include supporting sections of the WS-Security specifications and providing architectural support for plugging in and extending the capabilities of security tokens.

In WebSphere Application Server, the WS-Security for SOAP Message Version 1.1 specification is flexible and accommodates the requirements of Web services. For example, the specification does not have a mandatory security token definition. Instead, the specification defines a generic mechanism to associate the security token with a SOAP message.

The use of security tokens is defined in the various Version 1.0 and Version 1.1 security token profiles, such as the following examples:

► The Username Token Profile
► The X.509 Token Profile
► The Kerberos Token Profile

The WS-Security for SOAP Message Version 1.1 updates the WS-Security for SOAP Message core specifications and the various security token profiles. For this release, WebSphere Application Server V7 implements the Username Token Profile 1.1 and the X.509 Token Profile 1.1, which includes support for the Thumbprint type of security token reference. In addition, it supports the signature confirmation and encrypted header portions of the WS-Security Version 1.1 standard.

In addition, WebSphere Application Server V7 includes the following key enhancements:

► Support for the Lightweight Third Party Authentication (LTPA) Version 2 token

► Support for the configuration of multiple callers and an order attribute on the caller to determine which caller is used for the WebSphere credential

► Support for the published WS-SecurityPolicy Version 1.2 specification embedded in WSDL

► Support for the WS-SecureConversation Version 1.3 specification and the WS-Trust Version 1.3 specification (used by WS-SecureConversation)

► Support for Kerberos token profile for JAX-WS (and JAX-RPC) as defined in the WS-Kerberos Token Profile Version 1.1 specification:

  – For JAX-RPC applications, the Kerberos token can only be used as an authentication token.

  – For JAX-WS applications, the Kerberos token can be used for both authentication and message protection (signing and encrypting).

> **More information**: See the following resources:
> - ► Chapter 10, "WS-SecureConversation" on page 471
> - ► WS-Security specification
>
>   http://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf
> - ► OASIS Standards for security Web services
>
>   http://www.oasis-open.org/specs/#wssv1.0
> - ► What is new for security
>
>   http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/topic/com.ibm.websphere.base.doc/info/aes/ae/cwbs_welcwebsvcsec.html

1.4.6 WS-Policy

The Web Services Policy Framework (WS-Policy) is an interoperability standard that is used to describe and communicate the policies of a Web service so that service providers can export policy requirements in a standard format. Clients can combine the service provider requirements with their own capabilities to establish the policies required for a specific interaction.

A policy represents the capabilities and requirements of a Web service, for example, whether a message is secure and how to secure it, and whether a message is delivered reliably and how this is achieved. In addition, you can communicate the policy configuration to any other client, service registry, or service that supports the WS-Policy specification, including non-WebSphere Application Server products in a heterogeneous environment.

For a service provider, the policy configuration can be shared in a published Web Services Description Language (WSDL) file. The WSDL file is then obtained by a client using an HTTP Get request or by using the Web Services Metadata Exchange (WS-MEX) protocol. The WSDL is in the standard WS-PolicyAttachments format. The client can use this information to establish a configuration that is acceptable to both the client and the service provider. In other words, the client can be configured dynamically, based on the policies supported by its service provider. The provider policy can be attached at the application or service level.

The WS-Policy assertion specifications that are supported in this version of WebSphere Application Server 7 are:

- WS-Addressing policy settings
- WS-ReliableMessaging settings
- WS-Security policy settings
- WS-Transaction policy settings

More information: For more information about WS-Policy see:

- Chapter 7, "WS-Policy and WS-MetadataExchange" on page 327.
- Learning about WS-Policy

  http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/topic/com.ibm
  .websphere.nd.doc/info/ae/ae/twbs_wsp_learning.html

- Web services Policy Working Group

  http://www.w3.org/2002/ws/policy/

- Web services Policy 1.5 - Framework

  http://www.w3.org/TR/ws-policy/

- WS-PolicyAttachment

  http://www.ibm.com/developerworks/library/specification/ws-polatt/

- WS-PolicyAssertions

  http://www.ibm.com/developerworks/webservices/library/specification
  /ws-polas/

1.4.7 WS-MetadataExchange

WS-MEX is a Web services protocol specification that is part of the WS-Federation roadmap. It is designed to work in conjunction with WS-Addressing, WSDL, and WS-Policy to allow the retrieval of metadata about a Web services endpoint.

Web services use metadata to describe what other endpoints must know to interact with the service. For example, WS-Policy describes the capabilities, requirements, and general characteristics of a Web service. WSDL describes abstract messages (operations), concrete network protocols (bindings), and endpoint addresses (SEI) that are used by the Web service. XML schema describes the structure and content of XML-based messages that are received and sent by a Web service.

To bootstrap communication with a Web service, the WS-MEX specification defines how an endpoint can request the various types of metadata that it might

need to effectively communicate with the Web service. In response to the request, this specification defines an encapsulation that contains the three ways in which the metadata can be returned:

► The metadata itself can be included in the response.

► A URI can be returned, to which an HTTP GET can then be sent to retrieve the metadata from that location.

► A WS-Addressing Endpoint Reference of a WS-Transfer Metadata Resource can be returned, to which a WS-Transfer Get can be issued to retrieve the metadata. This specification also defines how a WS-Addressing Endpoint Reference can be modified to include this encapsulation.

> **More information**: For more information see:
>
> ► Chapter 7, "WS-Policy and WS-MetadataExchange" on page 327
>
> ► WS-MEX specification
>
>   http://www.w3.org/TR/2009/WD-ws-metadata-exchange-20090317/

1.4.8 Policy sets

An important new systems management capability in WebSphere Application Server V7.0 is in the area of Web services. *Policy sets* are assertions about how services are defined. It provides a mechanism for centrally defining the QoS configuration for Web services.

Policy sets combine configuration settings, including those configuration settings for transport and message-level configuration, such as WS-Addressing, WS-ReliableMessaging, WS-Security, and WS-AtomicTransaction.

There are two major types of policy sets:

► *Application policy sets* are used for business-related assertions. These assertions are related to the business operations that are defined in the WSDL file.

► *System policy sets* are used for non-business-related system messages. These messages are not related to the business operations that are defined in the WSDL. Instead, they refer to messages that are defined in other specifications that apply to QoS. Consider the following examples:

   – Security token (RST) messages that are defined in WS-Trust

   – The creation of sequence messages that are defined in WS-Reliable Messaging metadata exchange messages of the WS-MEX

An instance of a policy set consists of a collection of policies. For example, the WS-I RSP default policy set consists of instances of the WS-Security, WS-Addressing, and WS-ReliableMessaging policy types. A policy set is identified by a unique name that is unique across the cell. An *empty policy set* is a policy set with no policies defined.

Policy sets can be applied to all Web service applications to which they are applicable instead of defining individual policies and applying them to each Web service. This approach ensures a uniform QoS for a specific type of Web service.

The following default policy sets are provided:

► WS-I RSP default
► LTPA WS-I RSP default
► Username WS-I RSP default
► SecureConversation
► LTPA SecureConversation
► Username SecureConversation
► WSAddressing default
► WSHTTPS default
► Kerberos V5 HTTPS default
► TrustServiceKerberosDefault
► WSReliableMessaging default
► WSReliableMessaging persistent
► WSReliableMessaging 1_0
► WSSecurity default
► LTPA WSSecurity default
► Username WSSecurity default
► WS-Transaction
► SSL WS-Transaction

For more information about each of these default policy sets, go to:

`http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/index.jsp?topic=/com`
`.ibm.websphere.web20fep.multiplatform.doc/info/ae/ae/cwbs_wsspsps.html`

More information: For more information see:

► Chapter 6, "Policy sets" on page 261
► Web Services Policy Working Group

   `http://www.w3.org/2002/ws/policy/`

► Web Services Policy 1.5 - Framework

   `http://www.w3.org/TR/ws-policy/`

1.4.9 WS-Security Policy Language

A key benefit of the emerging Web services architecture is the ability to deliver integrated and interoperable solutions. This capability makes it critical to ensure the integrity, confidentiality, and overall security of these services.

The recently updated WS-Security Policy Language (WS-SecurityPolicy) specification defines a set of security policy assertions that applies to WS-Security:

► SOAP Message Security
► WS-Trust
► WS-SecureConversation

The intent of WS-SecurityPolicy is to provide enough information for compatibility and interoperability to be determined by Web services participants, along with all of the information that is necessary to actually enable a participant to engage in a secure exchange of messages.

> **More information**: For more information see:
>
> ► Chapter 7, "WS-Policy and WS-MetadataExchange" on page 327
> ► WS-SecurityPolicy 1.2 specification
>
>   `http://docs.oasis-open.org/ws-sx/ws-securitypolicy/v1.2/ws-security` `policy.html`
> ► Web services Security Policy Language
>
>   `https://www.ibm.com/developerworks/webservices/library/specification/` `ws-secpol/`

1.4.10 WS-SecurityKerberos

Kerberos Version 5 is a mature, open standard that provides a secure third-party authentication mechanism. IBM WebSphere Application Server V7 provides Kerberos token support for WS-Security at the message level. The support is based on the OASIS WS-Security Kerberos Token Profile Version 1.1.

The Kerberos specification references the Kerberos token in the SOAP message. Web services applications can use the Kerberos token to send identities and protect messages more securely. Overall, Kerberos support involves Kerberos support in Java EE security and the Kerberos token support in WS-Security.

The Kerberos token is a binary security token for Web services message-level security. WS-Security provides SOAP message-level security, such as security

token propagation, message signature, and message encryption. The Kerberos token is used for message security, specifically with the SOAP message security specification for Web services, and is another supported token, such as the username token and the secure conversation token.

Security Kerberos Token Profile features

There are two new functions for the Security Kerberos Token Profile:

► Kerberos token for JAX-WS applications
► SPNEGO Web authentication

Kerberos and JAX-WS

The support for Kerberos with WS-Security in WebSphere Application Server V7.0 is based on the OASIS WS-Security Kerberos Token Profile 1.1 specification. The Kerberos token for JAX-WS applications is configured by using policy sets and bindings. The JAX-WS application is attached with a custom policy, and the Kerberos token is configured as a message protection token or an authentication token.

The implemented Kerberos functionality for Web services security also uses existing tools and frameworks for the Kerberos token profile configuration for authentication and message protection.

SPNEGO Web authentication

WebSphere Application Server Version 6.1 introduced the trust association interceptor (TAI). TAI uses the Simple and Protected GSS-API Negotiation Mechanism (SPNEGO) to securely negotiate and authenticate HTTP requests for secured resources. In WebSphere Application Server V7.0, the TAI function is deprecated. SPNEGO Web authentication has taken its place to provide the dynamic reload of the SPNEGO filters and to enable fallback to the application login method.

> **More information**: For more information see:
>
> ► Security Kerberos Token Profile 1.1 specification
>
>   http://www.oasis-open.org/committees/download.php/16788/wss-v1.1-spec-os-KerberosTokenProfile.pdf
>
> ► Kerberos Token support
>
>   http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/topic/com.ibm.websphere.express.iseries.doc/info/iseriesexp/ae/cwbs_kerberos.html

1.4.11 WS-Trust Language

The WS-Trust Language (WS-Trust) uses the secure messaging mechanisms of WS-Security to define additional primitives and extensions for security token exchange to enable the issuance and dissemination of credentials within separate trust domains. To secure communication between two parties, the parties must exchange security credentials (directly or indirectly). However, each party must determine whether it can *trust* the asserted credentials of the other party.

This specification defines extensions to WS-Security for issuing and exchanging security tokens and for ways to establish and access the presence of trust relationships. By using such extensions, applications can engage in secure communication that is designed to work with the general Web services framework, including WSDL service descriptions, UDDI businessServices, binding templates, and SOAP messages.

New for WS-Trust

WebSphere Application Server V7 provides support for the WS-Trust 1.3 specification (Figure 1-12).

WebSphere Application Server 6.1 Feature Pack	WebSphere Application Server 7.0
WS-Trust	
OASIS Standard 1.1 draft	• OASIS Standard 1.3

Figure 1-12   WS-Trust support for WebSphere Application Server V7

More information: For more information see:

► Chapter 10, "WS-SecureConversation" on page 471

► WS-Trust V1.3 specification

  http://docs.oasis-open.org/ws-sx/ws-trust/200512/ws-trust-1.3-os.html

► WS-Trust IBM developerWorks article, "Web services Trust Language"

  http://www.ibm.com/developerworks/library/specification/ws-trust/

► Trust service

  http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/topic/com.ibm
  .websphere.nd.doc/info/ae/ae/cwbs_wstrust.html

1.4.12 WS-AtomicTransaction

The WS-AT support in the WebSphere Application Server provides transactional QoS to the Web services environment. Distributed Web services applications, and the resources that they use, can take part in distributed global transactions.

The WS-AT is an interoperable standard that provides the definition of the atomic transaction coordination type to be used with the extensible WS-Coordination framework. It defines three specific agreement coordination protocols for the atomic transaction coordination type:

► Completion
► Volatile two-phase commit
► Durable two-phase commit

WS-AT is a two-phase commit transaction protocol and is suitable for short-duration transactions only. Therefore, it is most appropriate for distributing transaction context across Web services that are deployed in a single enterprise. However, developers can use any or all of these protocols when building applications that require consistent agreement on the outcome of short-lived distributed activities that have the all-or-nothing property.

The WS-AT support introduces no new programming interfaces for transactional support. Global transaction demarcation is provided by the standard enterprise application use of the Java Transaction API (JTA) UserTransaction interface.

Note: When an application component that is running under a global transaction makes a Web services request, a WS-AT CoordinationContext is implicitly propagated to the target Web service. This happens only if the appropriate application deployment descriptors are set. For more information see "Configuring transactional deployment attributes" at:

http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/index.jsp?topic=/com.ibm.websphere.soafep.multiplatform.doc/info/ae/ae/tjta_entra2.html

New for WS-Transaction

WebSphere Application Server V7 supports both the WS-Transaction 1.1 and the WS-Transaction 1.0 specifications. You can configure the default WS-Transaction specification level for use for outbound requests if the specification level that the server requires cannot be determined from the provider policy. This level applies to outbound requests that include a WS-AT or WS-BA coordination context.

For JAX-WS applications, enable WS-AT support by creating a policy set, adding the WS-Transaction policy type to the policy set, optionally configuring the policy type, and attaching the policy set to the application or client that will be involved

in the WS-AT communication. The JAX-WS run time supports WS-AT 1.0, WS-AT 1.1, and the WS-Policy assertion for WS-AT.

More information: For more information see:

► Chapter 8, "Web services transaction specifications" on page 361

► WS-Transaction specification

http://docs.oasis-open.org/ws-tx/wstx-wsat-1.1-spec-os/wstx-wsat-1.1-spec-os.html

► Learning about WS-Transaction

http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/topic/com.ibm.websphere.soafep.multiplatform.doc/info/ae/ae/twbs_wstx_learning.html

1.4.13 WS-Coordination

The WS-Transaction specifications define mechanisms for transactional interoperability between Web services domains. They provide a means to compose transactional qualities of service into Web services applications.

The WS-Coordination (WS-COORD) specification describes an extensible framework for providing protocols that coordinate the actions of distributed applications. Such coordination protocols are used to support several applications, including those applications that must reach consistent agreement on the outcome of distributed activities. The framework enables an application service to create a context that is needed to propagate an activity to other services and to register for coordination protocols. It also enables existing transaction processing, workflow, and other systems for coordination to hide their proprietary protocols and to operate in a heterogeneous environment.

Additionally, the WS-Coordination specification describes a definition of the structure of the context and the requirements for propagating the context between cooperating services.

However, this specification is not enough to coordinate transactions among Web services. It describes an extensible coordination framework (WS-Coordination) and specific coordination types for the following transactions:

► Short duration and atomicity, consistency, isolation, durability (ACID) transactions (WS-AT)

► Longer running business transactions (WS-BA)

> **More information**: For more information see:
>
> ► WS-Coordination specification
>
>   http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-tx/WS-Coordination.pdf
>
> ► WS-Coordination
>
>   http://www.ibm.com/developerworks/library/specification/ws-tx/

1.4.14 WS-BusinessActivity

With WS-BA support in WebSphere Application Server, Web services on separate systems can coordinate activities that are more loosely coupled than atomic transactions. These activities can be difficult or impossible to roll back atomically. Therefore, they require a compensation process if an error occurs.

Web services protocols are defined by the OASIS group and provide standard ways of defining Web services applications. They allow the applications to operate independently of the product, platform, or programming language that is used. The WS-BA support is an implementation specification in the application server. These specifications define a set of protocols that enables Web services applications to participate in loosely coupled business processes that are distributed across the heterogeneous Web services environment, with the ability to compensate actions if an error occurs.

The WS-BA specification provides the definition of the business activity coordination type that is to be used with the extensible WS-Coordination framework. It defines two specific agreement coordination protocols for the business activity coordination type:

► BusinessAgreementWithParticipantCompletion
► BusinessAgreementWithCoordinatorCompletion

Developers can use either or both of these protocols when building applications that require consistent agreement on the outcome of long-running distributed activities.

1.5 Web services for Java EE

One of the most notable supported standards in WebSphere Application Server V7 is the Java EE 5 Platform powered by JSR-109 V1.2. In the following sections we describe the major changes for Java EE regarding Web services.

1.5.1 EJB 3.0 for WebSphere Application Server Version V7

The Web services feature pack for WebSphere Application Server V6.1 does not support the @Web service or @Web Method annotations on Enterprise JavaBeans (EJB™) 3.0 stateless session beans (identifying the stateless session bean as a JAX-WS implementation). Nor does it support injection of Web services references. Alternatively, you can invoke EJB 3.0 beans indirectly by defining a servlet as the JAX-WS implementation and placing code in the servlet that invokes the target EJB 3.0 bean. You can do this by declaring a link between the desired endpoint name in the Web service deployment descriptor of the EJB module. During deployment and installation of the bean into the application server environment, the bean is linked to the specified Web service endpoint.

WebSphere Application Server V7 (and JAX-WS 2.1) provides complete support for the Java EE 5 specification by enabling you to expose an EJB stateless session bean as a Web service.

1.5.2 Web services for EJB 3.0

The improvements to the Java EE 5 programming model are not limited to EJB development. Web services development is also greatly simplified with generics, annotations, and dependency injection features.

Future

A *Future* represents the result of an asynchronous invocation. It is a placeholder for a result that does not exist at the time of creation but will exist at a point in the future. Therefore, the definition of the asynchronous APIs of JAX-WS includes the Java Future type (Example 1-4).

Example 1-4   Java Future type

```
CreditRatingService svc = ...;

Future<?> invocation = svc.getCreditScoreAsync(customerTom, new
AsyncHandler<Score>() {
   public void handleResponse (Response<Score> response) {
      score = response.get();
      // process the request...
   }
   });
```

Generics

Java *generics* allow the definition of a class that contains data, but the type of that data is immaterial to the logic of the class. JAX-WS uses generics. The Java 5 Future class is a generic class that can be declared as a specific type that, in the case of JAX-WS, is the response type.

The polling version of an asynchronous operation invocation returns a response object. The response is defined with a specific response type in the generated interface, as shown in Example 1-5.

Example 1-5   Response type

```
Response<EchoOperationResponse> echoResponse = ...;
```

Handlers can be defined as generics. For example, Example 1-6 contains a MessageContext.

Example 1-6   MessageContext

```
javax.xml.ws.handler.Handler handler;
```

However, Example 1-7 contains a LogicalMessageContext, which is a subclass of MessageContext.

Example 1-7   LogicalMessageContext

```
javax.xml.ws.handler.Handler<LogicalMessageContext> logicalHandler;
```

Annotations

Both JAX-WS (JSR-224) and WS-Metadata (JSR-181) specifications introduce the WS-Metadata Facility (JSR-175) annotation programming model. This model aims to make it easier to create and modify a Web service. The annotations embed metadata in the compiled `.class` file of a Web service implementation and SEIs and can be processed at run time similar to deployment descriptors.

Dependency injection

Developer productivity is further enhanced when it is time to override the defaults. Overrides can be accomplished quickly and simply by using annotations rather than writing code. Annotations are used in conjunction with a programming pattern known as *dependency injection*, or Inversion of Control (IoC). In this patter, the application code declares variables and annotates them to indicate what is needed. Then the container *injects* the specified object or resource references.

Example 1-8 shows a Java EE 1.4 JAX-RPC application.

Example 1-8   JAX-RPC 1.1 EJB code

```
public interface IWeatherForecast extends Remote {
    public Weather getDayForecast(Calendar theDate) throws
RemoteException;
}
public class WeatherEJBBean implements IWeatherForecast{
public Weather getDayForecast(Calendar theDate) throws RemoteException
{ ... }
}
```

The JAX-RPC code in Example 1-8 is still tied to the java.io.Remote interface to handle the external communication. Besides, the WeatherBean service implementation is not annotated.

The corresponding Java EE 5 JAX-WS application is shown in Example 1-9. It shows the code with annotations where there is no need to include the java.io.Remote exception any longer.

Example 1-9   JAX-WS 2.1 EJB code

```
@javax.jws.WebService
@Remote
public interface IWeatherForecast  {
    public Weather getDayForecast(Calendar theDate) throws Exception;
}
@javax.jws.WebService
@Stateless
```

```
public class WeatherEJBBean implements IWeatherForecast{

@WebMethod
public Weather getDayForecast(@WebParam(name="date")Calendar theDate)
{
    ...
}


}
```

> **More information:** For more information see 4.4, "EJB Web services" on page 197.

Top-down and bottom-up methods for JAX-WS Web services EJB

You can create Web services by using two approaches:

► Top-down EJB Web services

Top-down EJB Web services development involves creating a Web service from a WSDL file. When creating a Web service by using a top-down approach, first you design the implementation of the Web service by creating a WSDL file. You can use the WSDL Editor. Then you can use the Web services wizard to create the Web service and skeleton Java classes to which you can add the required code.

► Bottom-up EJB Web services

Bottom-up EJB Web services development involves creating a Web service from an existing Java bean or enterprise bean and then using the Web services wizard to create the WSDL file and the Web service.

> **Note:** You can use either IBM Rational Application Developer 7.5 or the command-line tools (`wsgen` and `wsimport`), which are provided by JAX-WS support, to work with the top-down and bottom-up approaches.

Although bottom-up EJB Web service development might be faster and easier, especially if you are new to Web services, the top-down approach is the recommended way to create a Web service. By creating the WSDL file first, you will ultimately have more control over the Web service. In addition, you can eliminate interoperability issues that might arise when creating a Web service by using the bottom-up method.

2

Web services programming model

This chapter introduces the Web services programming model that is supported by WebSphere Application Server V7. It introduces the programming model by using many simple, but complete, Java code examples. The following key topics are discussed in this chapter:

► "Web service development with JAX-WS 2.1" on page 60
► "Working with SOAP using SAAJ 1.3" on page 117
► "Working with XML using JAXB 2.1" on page 123
► "Web services for Java EE" on page 131

Although this chapter does not introduce any Web services tools, you can find more information about such tools in Chapter 4, "Developing Web services applications" on page 161.

2.1 Web service development with JAX-WS 2.1

The Java API for XML-based Web services (JSR-224), commonly abbreviated JAX-WS, is the next generation Web services programming model. It effectively replaces the earlier JAX-RPC technology. JAX-WS defines a model that uses Java annotations to develop Web service providers and Web service clients. In addition, compared to JAX-RPC, JAX-WS Web service clients and Web service providers are more portable because no vendor-specific artifacts are necessary.

Although the Java EE 5 platform requires compatible application servers to support JAX-WS 2.0, WebSphere Application Server V7 has already taken the next step and supports JAX-WS 2.1. As an example, with this support, you can take advantage of the new Web services Addressing (WS-Addressing) functionality.

In the next section we explain how to develop a simple Web service and client.

2.1.1 Creating a Web service and client

The example provided in this section illustrates how to program a simple SOAP-based JAX-WS Web service and client. As shown in "Provider-based Web services" on page 85, JAX-WS offers an alternative Web services approach that is not based on the standardized SOAP, but rather on your own custom protocol.

The provider

Using JAX-WS, developing a Web service can be as simple as adding a single annotation. Example 2-1 shows a Web service with one method that greets the caller.

Example 2-1   The HelloMessenger Web service

```
package itso.hello;

import javax.jws.WebService;

@WebService
public class HelloMessenger {

    public String sayHello(String name) {
        return String.format("Hello %s", name);
    }
}
```

The Java bean in Example 2-1 on page 60 provides enough information to the JAX-WS runtime environment to let it be deployed as a real SOAP-based Web service. The *WebService* annotation specifies that, upon deployment, the bean and all of its public methods should be exposed as a Web service. The default conventions governed by the JAX-WS specification mean that the bean will be exposed as a SOAP 1.1-based document/literal Web service. In fact, the same rules allow the JAX-WS engine to expose a Web Services Description Language (WSDL) 1.1-compliant document, which can then be used by any client, Java or not, to use the service.

Granted, this Web service is almost as simple as it can get. By using JAX-WS annotations, you can specify much more information. The reasons for specifying further Web service metadata include the ability to affect the WSDL document that is exposed to clients, enabling a specific feature such as Message Transmission Optimization Mechanism (MTOM).

Also in Example 2-1 on page 60, the bean class represents both the service endpoint interface (SEI) and the service implementation class. In a production grade Web service, you typically create a separate SEI that specifies the Web service contract to avoid mixing it with the actual business logic.

Taking it for a test run

For the sake of simplicity, this section demonstrates how you can expose the Web service from a simple Java application. In a production grade application, you are advised to deploy your Web services to WebSphere Application Server V7.

Example 2-2 shows how you can use the JAX-WS javax.xml.ws.Endpoint class to expose the HelloMessenger Web service.

Example 2-2   Publishing the Web service endpoint

```
package itso.hello;

import javax.xml.ws.Endpoint;

public class HelloServer {

   public static void main(String... args) {

      String address = "http://localhost:9999/Hello";
      HelloMessenger endpointImpl = new HelloMessenger();
```

```
        Endpoint.publish(address, endpointImpl);
    }
}
```

The HelloServer application exposes the Web service endpoint in the URL
`http://localhost:9999/Hello`. The main method consists of three lines of code.
In the first line, we configure the address variable so that it contains the URL at
which the Web service endpoint is found. In the second line, we instantiate the
Web service that we created in "The provider" on page 60. In the last line of
code, we use the endpoint class to expose the HelloMessenger Java bean as a
Web service.

> **JDK™ to run the HelloServer application:** To run the HelloServer
> application, you must have either of the following Java Development Kits
> (JDKs):
>
> ► A Java Platform, Standard Edition (Java SE), 6 compliant JDK
>
> ► A Java SE 5-compliant JDK and the WebSphere Application Server V7
>   Web service thin client library
>
> WebSphere Application Server V7 comes with its own Java SE 6-compliant
> JDK, which is in the `WAS_HOME`/java directory.
>
> Before you can publish the HelloServer application, you must generate
> auxiliary JAX-WS classes for the HelloMessenger Web service endpoint. To
> generate these classes, you can use the Java SE 6 **wsgen** tool, which we
> describe in Chapter 4, "Developing Web services applications" on page 161.

When you run the HelloServer application, the HelloMessenger Web service is
published and ready for requests. You can now access the WSDL document,
which according to the JAX-WS specification, is accessible at the endpoint URL
followed by the WSDL query string `?wsdl` or `?WSDL`. Example 2-3 shows the
WSDL document that is generated by the JAX-WS run time.

Example 2-3   HelloMessenger WSDL document

```
<?xml version="1.0" encoding="UTF-8"?>

<definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:tns="http://hello.itso/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns="http://schemas.xmlsoap.org/wsdl/"
targetNamespace="http://hello.itso/"
   name="HelloMessengerService">
   <types>
```

```
    <xsd:schema>
        <xsd:import namespace="http://hello.itso/"
schemaLocation="http://localhost:9999/Hello?xsd=1"></xsd:import>
    </xsd:schema>
</types>
<message name="sayHello">
    <part name="parameters" element="tns:sayHello"></part>
</message>
<message name="sayHelloResponse">
    <part name="parameters" element="tns:sayHelloResponse"></part>
</message>
<portType name="HelloMessenger">
    <operation name="sayHello">
        <input message="tns:sayHello"></input>
        <output message="tns:sayHelloResponse"></output>
    </operation>
</portType>
<binding name="HelloMessengerPortBinding" type="tns:HelloMessenger">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
        style="document"></soap:binding>
    <operation name="sayHello">
        <soap:operation soapAction=""></soap:operation>
        <input>
            <soap:body use="literal"></soap:body>
        </input>
        <output>
            <soap:body use="literal"></soap:body>
        </output>
    </operation>
</binding>
<service name="HelloMessengerService">
    <port name="HelloMessengerPort"
binding="tns:HelloMessengerPortBinding">
        <soap:address
location="http://localhost:9999/Hello"></soap:address>
    </port>
</service>
</definitions>
```

The WSDL document in Example 2-3 on page 62 can be used by applications to generate client-side code that can interact with the Web service. In 2.1.2, "Relation of WSDL and Java types" on page 65, we explain how the Java code is mapped to the WSDL document. We provide a brief description of the WSDL parts as follows:

- ► At the top of the WSDL document, you find the types section, which in this example imports an XML schema document. (The schema can also be in-line inside the types section.) The XML schema document describes all complex types that are used in the Web service input and output.

- ► After the types section, you find one or more message sections. These messages are referenced later and provide the *bridge* from the WSDL document constructs to the schema document types.

- ► The message sections are followed by one or more portType sections. A *portType* is the WSDL equivalent of the SEI, in that it describes the abstract service interface in terms of service operations.

- ► The portType section is followed by one or more binding sections. A *binding* associates a portType with an actual protocol. In this example, it specifies that the HelloMessenger portType is exposed as a SOAP 1.1 service (indicated by the transport attribute with the value "http://schemas.xmlsoap.org/soap/http").

- ► At the end of the document, you find the service section, which summarizes the location of the applicable bindings.

In the next section we explain how a client application can use JAX-WS generated code to interact with the HelloMessenger Web service.

The client

Writing a Web service by using JAX-WS is almost as simple as writing the Web service. Example 2-4 shows a simple Java application client that uses the HelloMessenger Web service.

Example 2-4   The HelloMessenger Web service client

```
import itso.hello.HelloMessenger;
import itso.hello.HelloMessengerService;


public class HelloClient {

    public static void main(String... args) throws Exception {

        HelloMessengerService service = new HelloMessengerService();

        HelloMessenger port = service.getHelloMessengerPort();
```

```
        String message = port.sayHello("Milo");

        System.out.println(message);
    }
}
```

HelloClient invokes the sayHello method on the HelloMessenger Web service and prints the result to the standard output stream. The application uses two generated JAX-WS types:

HelloMessengerService   A subclass of the JAX-WS javax.xml.ws.Service that acts as a factory of dynamic proxies

HelloMessenger   The SEI that represents the remote Web service port

In the source code in Example 2-4 on page 64, when the client obtains HelloMessenger from the HelloMessengerService, a dynamic proxy is returned that implements the SEI. The dynamic proxy is the actual implementation code that knows how to issue the outbound request and return the response. The underlying request and response messages that are exchanged between the dynamic proxy and the Web service endpoint are based on the SOAP XML message format and are typically transmitted over the HTTP protocol. See 2.1.3, "Web service providers" on page 78, for an example of the SOAP messages that are exchanged between the Web service and client.

> **Important:** Before you can run the HelloClient application, you must generate auxiliary JAX-WS classes for the HelloMessenger Web service endpoint. To generate these classes, use the Java SE 6 `wsimport` tool, which is described in Chapter 4, "Developing Web services applications" on page 161.

2.1.2 Relation of WSDL and Java types

The JAX-WS 2.1 specification defines how Web service run times should map between Java and a Web services Interoperability (WS-I) Basic Profile 1.0-compliant WSDL 1.1 document. In this section we introduce these mapping rules.

Whether you develop Web services endpoints using a bottom-up approach or a top-down approach, the end result is the same: a collection of Java types (interfaces and classes) that comprise the implementation. These Java types typically are richly annotated as defined by the JAX-WS 2.1 specification (JSR-224) and the Web services Metadata Facility for the Java Platform (JSR-181).

The point of view presented in this section is how annotated Java Web services code is mapped to the automatically generated WSDL 1.1 document. In most cases, the mapping rules that apply when generating Java code from a WSDL 1.1 document are the same. However, this is not always the case. Therefore, consult the JAX-WS 2.1 specification in situations where you must know the exact mapping rules.

The main difference that you might encounter when generating Java code from an existing WSDL document (either on the client side or as an initial Web service implementation skeleton) is that the source code contains several JAX-WS annotations. These annotations occur because the JAX-WS specification in general requires tools to insert these annotations. For example, JAX-WS requires that generated exceptions be annotated with WebFault annotations and all method parameters be annotated with WebParam annotations.

> **Annotation reference:** For an overview of the JAX-WS annotations and how they apply to SEIs, see "Annotation reference" on page 80.

Mapping of Java packages

In JAX-WS, the Web service Java package maps to a WSDL namespace. In the HelloMessenger example, the itso.hello package was mapped to the `http://hello.itso/` XML target namespace. Example 2-5 shows the relevant part of the WSDL.

Example 2-5   HelloMessenger WSDL document: target namespace

```
<?xml version="1.0" encoding="UTF-8"?>

<definitions
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://hello.itso/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    targetNamespace="http://hello.itso/"
    name="HelloMessengerService">

<!-- abbreviated for clarity...-->

</definitions>
```

As you can see in Example 2-5, the namespace is the inverse of the Java package name. To be more exact, the JAX-WS mapping algorithm states that the package name is tokenized by using the period (.) delimiter, the order of tokens is

reversed, and the target namespace value is prepended with "http://" and appended with "/".

By using JAX-WS, you can explicitly configure the WSDL target namespace by using the targetNamespace attribute applicable to the WebService annotation:

```
@WebService(targetNamespace="http://my.specific.namespace/")
```

Mapping Java SEIs

The HelloMessenger Web service shown so far has an implicit SEI where the bean implementation itself contains all the Web service metadata. JAX-WS also allows you to define an SEI explicitly. To define the SEI, you define an ordinary Java interface that specifies the Web service metadata and a Web service implementation bean that implements the business logic. Example 2-6 shows an explicit HelloMessenger SEI.

Example 2-6   HelloMessenger explicit SEI

```
package itso.hello;

import javax.jws.WebService;

@WebService()
public interface HelloMessenger {
    public String sayHello(String name);
}
```

Example 2-7 shows the corresponding endpoint implementation bean.

Example 2-7   HelloMessenger endpoint implementation bean

```
package itso.hello;

import javax.jws.WebService;

@WebService(endpointInterface="itso.hello.HelloMessenger")
public class HelloMessengerImpl implements HelloMessenger {

    public String sayHello(String name) {
        return String.format("Hello %s", name);
    }
}
```

The difference is highlighted in bold. As you can see, the bean declares the endpointInterface attribute on the WebService annotation. This attribute tells the JAX-WS runtime environment where to find the explicit SEI.

The SEI, explicit or not, is mapped by JAX-WS to the WSDL portType element. Example 2-8 shows how the HelloMessenger SEI maps to the WSDL portType element.

Example 2-8   The SEI mapping to the WSDL portType element

```
<?xml version="1.0" encoding="UTF-8"?>

<definitions ...>
   <!-- abbreviated for clarity...-->

   <portType name="HelloMessenger">
      <operation name="sayHello">
         <input message="tns:sayHello"></input>
         <output message="tns:sayHelloResponse"></output>
      </operation>
   </portType>

   <!-- abbreviated for clarity...-->
</definitions>
```

The portType element in a WSDL document represents an interface. The name of the interface is specified by using the WSDL name. By default, JAX-WS generates the same name as the original SEI name, hence HelloMessenger. Just as Java interfaces define methods, the WSDL portType also defines operations. In this example, the portType contains a single operation corresponding to the Java SEI method. In the next section, we explain how the Java methods and WSDL operations relate.

With JAX-WS, you can customize the portType name by using the name attribute, which you can apply on the WebService annotation. For example, to change the WSDL portType name to MyHelloMessenger, change the WebService annotation as follows:

```
@WebService( name="MyHelloMessenger" )
```

By using the WebService annotation, you can customize other aspects of the WSDL document. For a complete list of applicable attributes, see the WebService annotation JavaDoc.

Mapping Java methods

All public methods on the SEI are automatically exposed as Web service methods. If the SEI inherits public methods from another interface, then all of these methods are exposed, too. Given that you have an explicit SEI, public

methods declared in a service endpoint implementation bean are only exposed as part of the Web service if they are also on the SEI.

Each Java method exposed on the SEI is mapped to a corresponding WSDL operation element in the portType, as shown in Example 2-9.

Example 2-9   WSDL operation name

```
<?xml version="1.0" encoding="UTF-8"?>

<definitions ...>
   <!-- abbreviated for clarity...-->

   <portType name="HelloMessenger">
      <operation name="sayHello">
         <input message="tns:sayHello"></input>
         <output message="tns:sayHelloResponse"></output>
      </operation>
   </portType>

   <!-- abbreviated for clarity...-->
</definitions>
```

The operation is highlighted in bold. The name is identical to that of the Java SEI.

You can customize the WSDL operation name by using the operationName attribute, which is applicable to the @WebMethod annotation. Example 2-10 shows how to change the WSDL operation name to "MySayHello".

Example 2-10   Customizing the WSDL operation name

```
package itso.hello;

import javax.jws.WebMethod;
import javax.jws.WebService;

@WebService
public class HelloMessenger {

   @WebMethod(operationName="MySayHello")
   public String sayHello(String name) {
      return String.format("Hello %s", name);
   }
}
```

Example 2-11 shows the corresponding WSDL.

Example 2-11   WSDL with custom operation name

```
<?xml version="1.0" encoding="UTF-8"?>

<definitions ...>
   <!-- abbreviated for clarity...-->

   <portType name="HelloMessenger">
      <operation name="MySayHello">
         <input message="tns:MySayHello"></input>
         <output message="tns:MySayHelloResponse"></output>
      </operation>
   </portType>

   <!-- abbreviated for clarity...-->
</definitions>
```

The JAX-WS specification specifies that, in the absence of the name attribute of the WebMethod annotation, the WSDL operation name must be the same as the Java method name.

With JAX-WS, you can exclude certain public methods by using the exclude attribute (Example 2-12). The exclude attribute is part of the WebMethod annotation that you can add to your Java methods.

Example 2-12   Excluding methods from the SEI

```
package itso.hello;

import javax.jws.WebMethod;
import javax.jws.WebService;

@WebService()
public class HelloMessenger {

   @WebMethod()
   public void includeMe() {
      // This method is a Web method
   }
```

```
    @WebMethod(exclude = true)
    public void excludeMe() {
        // This method is NOT a Web method
    }
}
```

If you deploy this Web service, the HelloMessenger WSDL portType only
contains the includeMe operation.

Mapping Java parameters and a return type

The HelloMessenger Web service contains a two-way operation. That is, it has
exactly one input message and one output message:

► `<input message="tns:sayHello"></message>`
► `<output message="tns:sayHelloResponse"></message>`

By using the message attributes, these messages refer to XML schema types
that are defined in the types block of the WSDL documents. In turn, the types
section imports an externally generated XML schema document. Example 2-13
shows the contents of this schema.

Example 2-13   HelloMessenger schema

```
<?xml version="1.0" encoding="UTF-8"?>

<xs:schema xmlns:tns="http://hello.itso/"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
    version="1.0" targetNamespace="http://hello.itso/">

    <xs:element name="sayHello" type="tns:sayHello"></xs:element>
    <xs:element name="sayHelloResponse"
type="tns:sayHelloResponse"></xs:element>

    <xs:complexType name="sayHello">
        <xs:sequence>
            <xs:element name="name" type="xs:string"
minOccurs="0"></xs:element>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="sayHelloResponse">
        <xs:sequence>
            <xs:element name="return" type="xs:string"
minOccurs="0"></xs:element>
        </xs:sequence>
```

```
        </xs:complexType>
</xs:schema>
```

The schema shown in Example 2-13 on page 71 is generated according to the JAXB binding rules. The schema contains a complexType for the method input and one for the method output. For further details see 2.3, "Working with XML using JAXB 2.1" on page 123.

You can use the JAX-WS WebParam annotation to affect the element names that are displayed inside the generated schema. In Example 2-13 on page 71, the input parameter shows the name *name* inside the generated schema under the complexType called *sayHello*. Example 2-14 shows how to change the schema name to myName.

Example 2-14   Changing the schema name

```
...
   @WebMethod
   public String sayHello(@WebParam(name = "myName") String name) {
       return String.format("Hello %s", name);

...
```

The WebParam configuration instructs the JAX-WS run time to generate the schema shown in Example 2-15.

Example 2-15   The changed schema name

```
   <xs:complexType name="sayHello">
      <xs:sequence>
         <xs:element name="myName" type="xs:string"
minOccurs="0"></xs:element>
      </xs:sequence>
   </xs:complexType>
```

The WebParam annotation also has a *mode* attribute. The mode attribute specifies whether the parameter in question is an input parameter, output parameter, or both. The default mode is IN, which means that the parameter is an ordinary input parameter. Because the Java programing language does not have real output parameters or input/output parameters, JAX-WS defines special javax.xml.ws.Holder classes instead.

Example 2-16 shows an output parameter mapping.

Example 2-16 An output parameter

```
package itso.hello;

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebService;
import javax.jws.WebParam.Mode;
import javax.xml.ws.Holder;

@WebService
public class HelloMessenger {

    @WebMethod
    public String sayHello(
        @WebParam(mode = Mode.IN) String name,
        @WebParam(mode = Mode.OUT) Holder<NameOutputResult>
nameOutputResult) {

        // This is an example of specifying an out parameter
        nameOutputResult.value = new NameOutputResult();
        nameOutputResult.value.setName(String.format("Hello %s", name));

        // This is the usual Java return statement
        return String.format("Hello %s", name);
    }
}
```

Notice how the second method parameter declares a javax.xml.ws.Holder
instance that encapsulates an output object. The NameOutputResult type is a
simple Java bean with a name property.

The use of javax.xml.ws.Holder to work with output parameters and input/output
parameters can be avoided if you design the Web service contract yourself. You
use a single Java object that contains all the return values. However, you might
find it necessary to work with javax.xml.ws.Holder objects if you are required to
adhere to an existing WSDL contract.

Mapping Java exceptions

The HelloMessenger Web service presented so far does not use any Java
exceptions. However, there might be situations where your business must model
service specific exceptions.

The JAX-WS specification indicates how and when Java exceptions are mapped to WSDL fault elements. To be exposed as a Web service fault, the Java exception must have the following characteristics:

- ► It must be a checked exception, and therefore, not extend RuntimeException.
- ► It must not extend the checked java.rmi.RemoteException.

To demonstrate how JAX-WS maps Java exceptions to WSDL 1.1 fault elements, we provide an updated version of the HelloMessenger Web service. Example 2-17 shows a new version of the Web service that throws an itso.hello.UnknownCallerException if the caller is not named "milo".

Example 2-17   HelloMessenger with Exception logic

```
package itso.hello;

import javax.jws.WebService;

@WebService
public class HelloMessenger {

    public String sayHello(String name) throws UnknownCallerException {
        if ("milo".equalsIgnoreCase(name)) {
            return String.format("Hello %s", name);
        } else {
            throw new UnknownCallerException(name);
        }
    }
}
```

For completeness, Example 2-18 shows the exception it its entirety.

Example 2-18   UnknownCallerException thrown by HelloMessenger

```
package itso.hello;

public class UnknownCallerException extends Exception {

    public UnknownCallerException(String caller) {
        super(String.format("%s is unknown!", caller));
    }

}
```

The exception is an ordinary checked Java exception. The constructor prepares the exception message and sends it to the java.lang.Exception super class.

Upon deployment of this Web service, the JAX-WS runtime environment generates the WSDL fault information, as shown in Example 2-19.

Example 2-19   WSDL with fault element

```xml
<?xml version="1.0" encoding="UTF-8"?>

<definitions ...>

    <!-- abbreviated for clarity...-->

    <message name="UnknownCallerException">
        <part name="fault" element="tns:UnknownCallerException"></part>
    </message>

    <portType name="HelloMessenger">
        <operation name="sayHello">
            <input message="tns:sayHello"></input>
            <output message="tns:sayHelloResponse"></output>
            <fault message="tns:UnknownCallerException"
name="UnknownCallerException"></fault>
        </operation>
    </portType>

    <!-- abbreviated for clarity...-->

</definitions>
```

The portType now has a new element, called the *WSDL fault element*. This element references the UnknownCallerException message, which in turn references an element in the generated XML schema. Example 2-20 shows the referenced schema.

Example 2-20   The XML schema type representing the exception

```xml
<?xml version="1.0" encoding="UTF-8"?>

<xs:schema ...>

    <!-- abbreviated for clarity...-->

    <xs:element name="UnknownCallerException"
type="tns:UnknownCallerException"></xs:element>

    <xs:complexType name="UnknownCallerException">
        <xs:sequence>
```

```
        <xs:element name="message" type="xs:string"
minOccurs="0"></xs:element>
      </xs:sequence>
    </xs:complexType>

    <!-- abbreviated for clarity...-->

</xs:schema>
```

The element in the schema references a complexType, which defines an XML schema type that is capable of containing the Java exception information.

With JAX-WS, you can customize the WSDL fault by annotating your exception with the *WebFault annotation*. As previously explained, JAX-WS requires that exceptions generated from WSDL faults are annotated with the WebFault annotation. In fact, any generated exception simply wraps a Java bean that encapsulates the actual Web service fault detail. See the JAX-WS 2.1 specification for further details about the mapping between Java exceptions and WSDL faults.

WSDL styles

The JAX-WS 2.1 specification supports Web services by using different SOAP binding styles. A SOAP binding style is specified in the binding section of a WSDL document. The following styles are supported:

- ▶ RPC style
- ▶ Document style

From a developer's perspective, the choice of style does not matter. In most cases, the SEI code is the same, with the exception of the style attribute value in the SEI javax.jws.sopa.SOAPBinding annotation. For example, to expose the HelloMessenger endpoint by using the RPC style, add the following line to the endpoint class definition:

```
@SOAPBinding(style=Style.RPC)
```

If you are generating a client SEI for an RPC style Web service, then the JAX-WS run time inserts the annotation.

However, the choice of SOAP binding style has an impact on the WSDL document itself and most likely on the format of the SOAP messages that are exchanged. Although JAX-WS 2.1 supports both the remote procedure call (RPC) and document style, it defaults to the document style. We recommend the default document style because it enforces strictly typed payloads. The entire payload content, using the document style, adheres to the XML schema types

declared in the WSDL document's types section (either inlined schema or externally referenced schema).

In addition to having different SOAP binding styles, a binding style can also have a special use, which can be either encoded or literal. JAX-WS only supports literal. Therefore, the default WSDL style in JAX-WS is typically referred to as the *document/literal* style. Further description of the WSDL styles and use modes is outside the scope of this book and we do not discuss it further.

> **More information about WSDL styles**: For a good article about WSDL styles, see "Which style of WSDL should I use?" from IBM developerWorks at:
>
> `http://www.ibm.com/developerworks/webservices/library/ws-whichwsdl`
>
> The article provides detailed examples of how the WSDL style and use combinations affect the SOAP messages and SEI code. It also provides a good discussion about the strengths and weaknesses of each combination.

In the next section we introduce a common pattern known as *message wrapping*, which relates to the document/literal WSDL style.

The document/literal wrapped pattern

As explained in the previous section, the default WSDL style or use mode employed by JAX-WS 2.1 is the document/literal style. In the RPC style, the immediate element in the SOAP message body names the operation to be invoked. In the document style, SOAP messages are not required to specify the operation that is being invoked. Therefore, the payload does not need to reveal which method to invoke on the endpoint implementation. A problem with this is that it makes it a bit harder for developers to make the connection between the SOAP message and the endpoint method that is being used.

The primary purpose of the document/literal wrapped pattern is to make document/literal-generated SEI code look like RPC style code. The pattern ensures that the SOAP messages always contain a top-level element that wraps the actual input/output. For request messages, the wrapped pattern ensures that the top-level element has the same name as the operation that is being invoked. For response messages, the wrapper element is the operation name appended with "Response". However, in contrast to the RPC style, the element is part of the well-defined XML schema that is declared in the WSDL documents types section (either inline or external document).

If you are generating Java code from an existing WSDL document, the JAX-WS specification specifies in detail the conditions that must apply before the Java code can be generated by using the wrapped pattern. That is, you might encounter situations where the generated code is *not wrapped*, which is also

known as *bare*. An SEI specifies whether it uses wrapped or bare code by using the javax.jws.soap.SOAPBinding annotation as follows:

```
@SOAPBinding( parameterStyle = SOAPBinding.ParameterStyle.WRAPPED)
```

If you omit the SOAPBinding customization on the SEI, JAX-WS 2.1 run times default to use the document/literal style combined with the wrapped pattern. In addition, by using both IBM Rational Application Developer 7.5 and the JDK 1.6 JAX-WS `wsimport` tool, you can specify whether message wrapping should be enabled when the tools generate SEI code from a WSDL document.

2.1.3 Web service providers

By using JAX-WS, you can develop server-side Web service endpoints by using two different approaches:

- ► SEI-based Web services
- ► Provider-based Web services

The *SEI-based Web services model* is the one that you have already seen in the HelloMessenger example. When developing SEI-based Web services, the developer works on a high level using ordinary Java types. At this level, the underlying SOAP message and transport protocol details are hidden. In addition, JAX-WS implementations provide powerful tools that ease SEI-based Web service development.

Provider-based Web services is an alternative approach with JAX-WS. It is a lower-level approach in which the developer is directly aware of the Web service requests and responds. In addition, provider-based Web service messages can be based on the SOAP message protocol, but they do not have to be. Provider-based Web services are flexible enough that you can use them to implement Representational State Transfer (REST) type Web services.

SEI-based Web services

The HelloMessenger Web service is an example of a SEI-based Web service. As described in 2.1.1, "Creating a Web service and client" on page 60, and 2.1.2, "Relation of WSDL and Java types" on page 65, JAX-WS has well-defined annotations that you use in developing SEI-based service endpoints.

In this section we briefly introduce the SOAP messages that are exchanged on the wire. In addition, you will find an annotation reference that explains which annotations are applicable to SEI-based endpoints and to which level they apply (class/interface, method, or parameter).

The underlying SOAP messages

In cooperation with JAXB 2.1 binding rules, JAX-WS is capable of generating the underlying request and response messages. Example 2-21 shows an example of the SOAP 1.1 request message that is generated by the HelloMessenger Web service client.

Example 2-21   SOAP 1.1 request

```
<?xml version="1.0" encoding="UTF-8"?>

<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:q0="http://hello.itso/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <soapenv:Body>
      <q0:sayHello>
         <arg0>Milo</arg0>
      </q0:sayHello>
   </soapenv:Body>
</soapenv:Envelope>
```

A SOAP envelope encapsulates the entire request. However, a SOAP envelope can initially have an optional header section that can be used to transfer message metadata such as security authentication credentials (not shown in this example). The SOAP envelope also contains a mandatory body section that is used to transfer the actual message payload. Because the default style is document/literal wrapped, the root element inside the SOAP envelope body identifies the Web service operation, which is sayHello, that is being invoked. Within that element, you find the operation's input data. The XML payload adheres to the XML schema rules that are defined in the WSDL document's types section.

Example 2-22 shows an example of the corresponding SOAP 1.1 response message that is generated by the HelloMessenger Web service endpoint implementation.

Example 2-22   SOAP 1.1 response

```
<?xml version="1.0" encoding="UTF-8"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Body>
      <ns2:sayHelloResponse xmlns:ns2="http://hello.itso/">
         <return>Hello Milo</return>
      </ns2:sayHelloResponse>
```

```
    </S:Body>
</S:Envelope>
```

As is the case with the request envelope, the XML payload adheres to the XML schema rules that are defined in the WSDL document's types section.

The marshalling rules for the request and response SOAP payload are defined by JAXB 2.1. See 2.3, "Working with XML using JAXB 2.1" on page 123, for further information about JAXB.

Annotation reference

In this section we provide an overview of the JAX-WS 2.1 annotations that apply to SEI-based Web service providers. The annotations presented here can be either generated automatically or added manually:

► In top-down development, the annotations are automatically generated by JAX-WS tools, such as the Web service wizards in Rational Application Developer 7.5 or the JDK 1.6 `wsimport` tool.

► In bottom-up development, the annotations are manually added by the Java developer.

Most of the annotations that apply to SEI-based providers also apply to the generated SEIs for proxy clients.

Example 2-23 shows the annotations that apply to JAX-WS 2.1 SEI-based Web services.

Example 2-23   JAX-WS 2.1 annotations and their applicability to the SEI

```
package itso.hello;

//JSR 181 (Web services meta data specification):
import javax.jws.HandlerChain;
import javax.jws.Oneway;
import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebResult;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;

//JSR 224 (JAX-WS 2.1 specification):
import javax.xml.ws.BindingType;
import javax.xml.ws.RequestWrapper;
import javax.xml.ws.RespectBinding;
import javax.xml.ws.ResponseWrapper;
import javax.xml.ws.soap.Addressing;
```

```
import javax.xml.ws.soap.MTOM;

@WebService()
@SOAPBinding()
@HandlerChain(file = "")
@BindingType()
@RespectBinding()
@Addressing()
@MTOM()
public class HelloMessenger {

    @WebMethod()
    @Oneway()
    @WebResult()
    @SOAPBinding()
    @RequestWrapper()
    @ResponseWrapper()
    public String sayHello(@WebParam() String name) {
      return String.format("Hello %s", name);
    }
}
```

Example 2-23 on page 80 shows the annotations and their applicability at the class, method, and parameter level. The comments in the import part indicate to which JSR specification the annotations belong. A short description of each of these annotations follows. Annotations have been described in the context of their applicability (classes, methods, variables, and method parameters) and with a short description of their attributes. Unless otherwise noted, the annotation attribute types are of type java.lang.String.

Example 2-23 on page 80 has the following class-level annotations:

► javax.jws.WebService

This annotation marks a Java class as implementing a Web service or marks a Java interface as defining a WS-I. This annotation includes the following applicable attributes:

name   Specifies the portType name in the WSDL document. Defaults to the simple name of the class.

targetNamespace   Specifies the wsdl:portType elements namespace.

serviceName   Specifies the service name in the service in the WSDL document. Defaults to the simple name of the class appended with "Service".

portName	Used as the name of the wsdl:port when mapped to WSDL.	
wsdlLocation	The location of a predefined WSDL that describes the service. The wsdlLocation is a URL (relative or absolute) that refers to a pre-existing WSDL file. The presence of a wsdlLocation value indicates that the service implementation bean is implementing a predefined WSDL contract.	
endpointInterface	Can be used to specify the explicit full class name of an SEI.	

► javax.jws.soap.SOAPBinding

This annotation specifies the mapping of the Web service onto the SOAP binding protocol. When the SOAPBinding.style is DOCUMENT, this annotation can also be used on methods. This annotation includes the following applicable attributes:

style	Specifies the WSDL encoding style. The valid values are DOCUMENT and RPC. The default is DOCUMENT.	
use	Specifies the formatting style. The valid values are LITERAL or ENCODED. The default is LITERAL.	
parameterStyle	Specifies the parameter mapping strategy. The WRAPPED style ensures that parameters are wrapped in an element named after the method. The valid values are BARE or WRAPPED. The default is WRAPPED.	

► javax.jws.HandlerChain

This annotation associates a Web service endpoint with a handler chain.

This annotation includes the *file* attribute. This mandatory attribute specifies the relative or absolute URL location of the handler chain XML file.

► javax.xml.ws.BindingType

This annotation specifies the binding to use when publishing the endpoint.

This annotation includes the *value* attribute. This attribute specifies the binding ID, which essentially is a URI. The default is SOAP 1.1/HTTP, which is represented by the constant SOAP11HTTP_BINDING on the javax.xml.ws.soap.SOAPBinding annotation.

- javax.xml.ws.RespectBinding

  This annotation controls whether the JAX-WS implementation must honor the contents of the endpoint's corresponding wsdl:binding section in the WSDL document.

  This annotation includes the *enabled* attribute. This attribute is a boolean that indicates whether the binding must be honored. The default is Boolean.TRUE.

- javax.xml.ws.soap.Addressing

  This annotation controls the use of WS-Addressing. The annotation includes the following applicable attributes:

enabled	A boolean that specifies whether WS-Addressing is enabled. The default is Boolean.TRUE.
required	A boolean that specifies whether WS-Addressing headers must be present on incoming messages. The default is Boolean.FALSE.

- javax.xml.ws.soap.MTOM

  This annotation controls the use of MTOM. The annotation includes the following applicable attributes:

enabled	A boolean that specifies whether MTOM is enabled. The default is Boolean.TRUE.
threshold	An integer that specifies how many bytes binary data should be before it is sent as an attachment. The default is 0 bytes. (All binary types are sent as attachments.)

Example 2-23 on page 80 has the following method-level annotations:

- javax.jws.WebMethod

  This annotation customizes a Java method that is exposed as a Web service operation. The annotation includes the following applicable attributes:

operationName	Specifies the name of the wsdl:operation in the WSDL document that matches the method.
action	Specifies the action for the method (SOAP action for SOAP bindings).
exclude	A boolean that specifies whether the method is exposed as a Web service operation. The default is Boolean.False.

- ► javax.jws.Oneway

  This annotation can be used on Java methods that do not have a return value (void methods) and denotes the method as a Web service one-way operation that only has an input message and no output message. A one-way method typically returns to the calling application prior to executing the actual business method. This annotation has no applicable attributes.

- ► javax.jws.WebResult

  This annotation customizes the mapping of a Java return value to the WSDL document's Web service message part and XML element. This annotation includes the following applicable attributes:

name	Specifies the local name of the XML element that represents the return value.
partName	Specifies the name of the wsdl:part element that represents the return value.
targetNamespace	Specifies the XML namespace of the return value.
header	A boolean value that specifies whether the return valued is stored in a message header rather than the message body. The default is Boolean.FALSE.

- ► javax.xml.ws.RequestWrapper

  This annotation specifies the SOAP request JAXB wrapper bean and the XML target namespace. The annotation includes the following applicable attributes:

localName	Specifies the local element name. Defaults to the operationName property of the WebMethod annotation.
targetNamespace	Specifies the request element target namespace. Defaults to the target namespace of the SEI.
className	Specifies the JAXB wrapper bean class.

- ► javax.xml.ws.ResponseWrapper

  This annotation specifies the SOAP response JAXB wrapper bean and the XML target namespace. The annotation includes the following applicable attributes:

localName	Specifies the local element name. The default is the operationName property of the WebMethod annotation appended with "Response".
targetNamespace	Specifies the response element target namespace. The default is to the target namespace of the SEI.
className	Specifies the JAXB wrapper bean class.

Example 2-23 on page 80 has the following parameter-level annotation:

► javax.jws.WebParam

This annotation customizes the mapping of a Java method parameter to the WSDL document's Web service message part and XML element. The annotation includes the following applicable attributes:

name
Specifies the local name of the XML element that represents the parameter.

partName
Specifies the name of the wsdl:part element that represents the parameter.

targetNamespace
Specifies the XML namespace of the parameter.

mode
Specifies whether the parameter is an IN parameter, OUT parameter, or INOUT parameter. The default is IN for non-holder types and INOUT for holder types.

header
A boolean value that specifies whether the parameter is pulled from a message header rather than the message body. The default is Boolean.FALSE.

More information: For more details about the JAX-WS 2.1 annotations and their properties, see the WebSphere Application Server 7 Information Center at the following address:

http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/index.jsp?topic=
/com.ibm.websphere.express.doc/info/exp/ae/rwbs_jaxwsannotations.html

Alternatively, you can consult the following specifications, which are available as PDF documents from the Java Community Process home page at the following address:

http://www.jcp.org

► Web services Metadata for the Java Platform (JSR-181)
► The Java API for XML-Based Web Services 2.1 (JSR-224)
► Web services for Java EE Version 1.2 (JSR-109)

Provider-based Web services

With provider-based Web services, you work directly with the Web services request and response messages. The messages can be based on the SOAP message format or your own custom message format.

By implementing the provider interface on your endpoint implementation bean and annotating it with the WebServiceProvider annotation, you signal to the JAX-WS runtime environment that you are implementing a provider-based Web service endpoint. Example 2-24 shows a simple custom Web service provider.

Example 2-24   A simple custom provider

```
package itso.hello;

import java.io.StringReader;

import javax.xml.transform.Source;
import javax.xml.transform.stream.StreamSource;
import javax.xml.ws.Provider;
import javax.xml.ws.Service;
import javax.xml.ws.ServiceMode;
import javax.xml.ws.WebServiceProvider;

@WebServiceProvider
@ServiceMode(value=Service.Mode.PAYLOAD)
public class HelloMessengerProvider implements Provider<Source> {

    public Source invoke(Source request) {

        // The fixed response
        String responseXML = "<h1>Hello Milo</h1>";

        // Return the XML reply
        return new StreamSource(new StringReader(responseXML));
    }
}
```

The Web service in Example 2-24 illustrates how to implement a javax.xml.ws.Provider, which works with javax.xml.transform.Source objects. This example ignores any data from the incoming request and always returns a response that says "Hello Milo". In fact, when published, you can use a browser and go to the endpoint URL to receive that message. This method works because the provider in this particular example does not expect any well-defined XML request message and, therefore, returns the message upon any POST or GET HTTP request that is sent to the endpoint.

> **Publishing the endpoint by using the JAX-WS endpoint publisher**: To
> publish Example 2-24, by using the JAX-WS endpoint publisher, you must
> explicitly create the endpoint with an HTTP binding:
>
> 1. Instantiate the provider-based endpoint:
>
>    ```
>    HelloMessengerProvider e = new HelloMessengerProvider();
>    ```
>
> 2. Create the endpoint so that it is exposed with the HTTP binding:
>
>    ```
>    Endpoint endpoint = Endpoint.create(HTTPBinding.HTTP_BINDING, e)
>    ```
>
> 3. Publish the endpoint:
>
>    ```
>    endpoint.publish(address);
>    ```

When you work with provider-based endpoints, note the following points:

► The Java bean must implement a *typed provider*. The provider type, which is
  specified in angle brackets, has three possible values:

  – javax.xml.transform.Source

    By using this object, the endpoint can work directly with the XML data,
    including both request XML data and response XML data. You can work
    with the source XML stream in different ways. One way is to use the JAXB
    2.1 API to convert the XML data stream into a Java object. For more
    information about the JAXB API, see 2.3, "Working with XML using JAXB
    2.1" on page 123.

  – javax.xml.soap.SOAPMessage

    This object is part of the SAAJ 1.3 API. By using this object, you can
    consume and produce SOAP messages by using a typed SOAP message
    API. For more information about the SAAJ API, see 2.2, "Working with
    SOAP using SAAJ 1.3" on page 117.

  – javax.activation.DataSource

    With this object, the endpoint can work with MIME-typed messages.

► The endpoint must be annotated with the WebServiceProvider annotation, as
  opposed to the WebService annotation specified by the SEI-based endpoints.

► The endpoint can specify a service mode and has the following possible
  values:

  – javax.xml.ws.Service.Mode.PAYLOAD

    This is the default mode, which indicates that the endpoint only works on
    the request payload. When working with SOAP by using the SAAJ API,
    you gain access to the SOAP body only.

– javax.xml.ws.Service.Mode.MESSAGE

This mode indicates that the endpoint works on the entire protocol message. When working with SOAP using the SAAJ API, you gain access to the entire SOAP envelope.

Accessing the context

JAX-WS allows Web service endpoints to gain access to transport protocol information by allowing resource injection of a special context-aware object, the WebServiceContext object. By using this object, the endpoint can, for example, obtain information about the HTTP headers and determine the actual HTTP method used to invoke the endpoint (such as GET, PUT, POST, DELETE).

Example 2-25 shows a modified version of the provider endpoint example that accesses the resource injected WebServiceContext.

Example 2-25   Accessing the context

```
package itso.hello;

import java.io.StringReader;

import javax.annotation.Resource;
import javax.xml.transform.Source;
import javax.xml.transform.stream.StreamSource;
import javax.xml.ws.Provider;
import javax.xml.ws.Service;
import javax.xml.ws.ServiceMode;
import javax.xml.ws.WebServiceContext;
import javax.xml.ws.WebServiceProvider;
import javax.xml.ws.handler.MessageContext;

@WebServiceProvider
@ServiceMode(value=Service.Mode.MESSAGE)
public class HelloMessengerProvider implements Provider<Source> {

    @Resource
    private WebServiceContext context;

    public Source invoke(Source request) {

        // Determine the HTTP method
        MessageContext messageContext = context.getMessageContext();
        String httpMethod =
messageContext.get(MessageContext.HTTP_REQUEST_METHOD).toString();
```

```
        // Include HTTP method information in the response
        String responseXML = String.format("<h1>Hello Milo (HTTP %s
method)</h1>", httpMethod);

        // Return the XML reply
        return new StreamSource(new StringReader(responseXML));
    }
}
```

The relevant code is highlighted in bold. The endpoint gains access to the
context by annotating a WebServiceContext member variable with a resource
annotation. Inside the invoke method, the endpoint uses the WebServiceContext
to access the MessageContext, which contains information about the currently
active HTTP request method. The MessageContext object contains other useful
information also, such as the HTTP request headers sent by the client.
Descriptive constants are available on the MessageContext class that you can
use to retrieve the data.

2.1.4 Web service clients

JAX-WS defines two service usage models:

► Proxy clients
► Dispatch clients

The *proxy-based client model* was demonstrated in 2.1.1, "Creating a Web
service and client" on page 60. In this model, your applications work on local
proxy objects that implement the SEI that is being exposed by the Web service
endpoint.

The *dispatch-client model* offered by JAX-WS is a lower-level model that requires
you to supply the necessary XML request yourself. This model can be used in
the situations where you want to dynamically build up the SOAP request itself or
where you must use a non-SOAP-based Web service endpoint.

Collectively, both client types are also known as *BindingProviders* because both
clients realize the JAX-WS javax.xml.ws.BindingProvider interface. The
BindingProvider interface allows for a common configuration model, which is
described in "Configuring the client BindingProviders" on page 102.

Proxy clients
By using the proxy client model, you can write SOAP 1.1 and SOAP 1.2 Web
service clients without having an intimate knowledge of the underlying SOAP

protocol. The client applications use a dynamic proxy object that implements a statically generated SEI.

Keep in mind the following points:

► Since Java is a statically typed language, the proxy client model requires you to use a tool, such as the **wsimport** tool or Rational Application Developer V7.5, to generate the necessary Web service client code that you will use in your applications.

► The JAX-WS run time accesses the WSDL document at run time to generate the dynamic proxy implementation. The implication is that the stub cannot be instantiated unless the WSDL document is available.

In the following sections, we describe the synchronous and asynchronous programming models for working with the JAX-WS generated proxy clients.

Synchronous clients

By using the synchronous model, you can develop SOAP-based Web service client code without worrying about the underlying protocol details. The HelloMessenger client shown in Example 2-4 on page 64 uses a generated JAX-WS-compliant subclass of javax.xml.ws.Service. Example 2-26 shows this class in its entirety, although comments have been removed for brevity.

Example 2-26   JAX-WS generated HelloMessengerService

```
package itso.hello;

import java.net.MalformedURLException;
import java.net.URL;
import javax.xml.namespace.QName;
import javax.xml.ws.Service;
import javax.xml.ws.WebEndpoint;
import javax.xml.ws.WebServiceClient;
import javax.xml.ws.WebServiceFeature;

@WebServiceClient(
   name = "HelloMessengerService",
   targetNamespace = "http://hello.itso/",
   wsdlLocation = "http://localhost:9999/Hello?wsdl")
public class HelloMessengerService extends Service {

   private final static URL HELLOMESSENGERSERVICE_WSDL_LOCATION;

   static {
      URL url = null;
      try {
```

```
            url = new URL("http://localhost:9999/Hello?wsdl");
        } catch (MalformedURLException e) {
            e.printStackTrace();
        }
        HELLOMESSENGERSERVICE_WSDL_LOCATION = url;
    }

    public HelloMessengerService(URL wsdlLocation, QName serviceName) {
        super(wsdlLocation, serviceName);
    }

    public HelloMessengerService() {
        super(HELLOMESSENGERSERVICE_WSDL_LOCATION, new QName(
                "http://hello.itso/", "HelloMessengerService"));
    }

    @WebEndpoint(name = "HelloMessengerPort")
    public HelloMessenger getHelloMessengerPort() {
        return (HelloMessenger) super.getPort(new
QName("http://hello.itso/",
                "HelloMessengerPort"), HelloMessenger.class);
    }

    @WebEndpoint(name = "HelloMessengerPort")
    public HelloMessenger getHelloMessengerPort(WebServiceFeature...
features) {
        return (HelloMessenger) super.getPort(new
QName("http://hello.itso/",
                "HelloMessengerPort"), HelloMessenger.class, features);
    }

}
```

You can tell that Example 2-26 on page 90 is a JAX-WS-generated client
because of its class-level WebServiceClient annotation. The class has two
constructors:

▶ The first constructor is the default constructor. It configures the service so that
  any dynamic proxies created from it are produced by using the WSDL
  document that was used to generate the client code.

  In the HelloMessenger example, the tool was not instructed to create a local
  copy of the WSDL document. This is why there is an absolute reference to the
  actual URL at which the Endpoint publisher makes the WSDL document
  available. Because we do not recommend this, make sure that you generate

the Web service client code so that it is copied to the client. See Chapter 4, "Developing Web services applications" on page 161, for more information.

One of the implications of not having a local WSDL document is that the constructor throws an exception in cases where the JAX-WS run time cannot connect to the server that is exposing the document.

► The second constructor initializes the service by using a specified WSDL document.

In addition to these two constructors, the generated client has a couple of getHelloMessenger methods with which you can get a dynamic proxy that binds to the specified Web service endpoint. The HelloMessenger client that we presented in Example 2-4 on page 64 uses the default constructor to connect to instantiate the Web service:

```
HelloMessengerService service = new HelloMessengerService();
```

This approach can present a problem when you want the client application to switch from the test environment's Web service endpoint to the production environment's Web service endpoint. There are a couple of ways to override this endpoint location from your client code:

► Use the overloaded constructor of the generated javax.xml.ws.Service subclass that takes another WSDL document location. This supplied WSDL document can, in turn, specify the service endpoint location of interest.

► Use the default constructor, but specify the endpoint location on the dynamic proxy returned by the service.

After the HelloClient application has obtained a HelloMessengerService instance, it uses that instance to obtain a dynamic stub, which binds to the actual Web service endpoint. Example 2-27 shows the generated Web service client type that is implemented by the stub.

Example 2-27 The generated Web service client interface

```
package itso.hello;

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebResult;
import javax.jws.WebService;
import javax.xml.bind.annotation.XmlSeeAlso;
import javax.xml.ws.RequestWrapper;
import javax.xml.ws.ResponseWrapper;

@WebService(name = "HelloMessenger", targetNamespace =
"http://hello.itso/")
```

```
@XmlSeeAlso({
    ObjectFactory.class
})
public interface HelloMessenger {

    @WebMethod
    @WebResult(targetNamespace = "")
    @RequestWrapper(localName = "sayHello", targetNamespace =
"http://hello.itso/", className = "itso.hello.SayHello")
    @ResponseWrapper(localName = "sayHelloResponse", targetNamespace =
"http://hello.itso/", className = "itso.hello.SayHelloResponse")
    public String sayHello(
        @WebParam(name = "arg0", targetNamespace = "")
        String arg0);

}
```

The HelloMessenger type is an ordinary Java interface with some
JAX-WS-specific annotations. Although the return type of the
HelloMessengerService is of this interface type, in reality what is returned is a
dynamic stub that implements this interface.

Example 2-28 shows a new HelloMessenger client application that explicitly
specifies, on the dynamic stub, a new Web service endpoint location.

Example 2-28   Configuring the Web service endpoint location

```
import itso.hello.HelloMessenger;
import itso.hello.HelloMessengerService;

import javax.xml.ws.BindingProvider;


public class HelloClientCustomEndpoint {

    public static void main(String... args) throws Exception {

        HelloMessengerService service = new HelloMessengerService();

        HelloMessenger port = service.getHelloMessengerPort();


((BindingProvider)port).getRequestContext().put(BindingProvider.ENDPOIN
T_ADDRESS_PROPERTY, "http://itso.ibm.com:69693/Hello");
```

```
        String message = port.sayHello("Thilde");

        System.out.println(message);
    }
}
```

The client-relevant code is highlighted in bold. The application casts the dynamic
Web service port proxy to a javax.xml.ws.BindingProvider. The BindingProvider
is implemented by the dynamic client proxies and gives you access to the
request and the response contexts. The application specifies the endpoint
address on the *request* context using the
BindingProvider.ENDPOINT_ADDRESS_PROPERTY property.

Asynchronous clients

If you are familiar with JAX-RPC, you might already know that it does not offer an
asynchronous model. The asynchronous client programming model in JAX-WS
is merely a convenient functionality for developing Web service clients. It does
not refer to real asynchronous message exchanges. You can create
asynchronous clients by configuring the tool that you use to generate JAX-WS
Web service client code.

JAX-WS offers two asynchronous programming models:

► Polling clients
► Callback clients

These approaches merely differentiate, in the Java method, signatures that are
generated on the client-side Web service port interface. When you enable
asynchronous clients in your tool, JAX-WS generates three methods for every
operation that is defined in the Web service portType:

► Asynchronous method
► An asynchronous polling method
► An asynchronous callback method

Example 2-29 shows the HelloMessenger client-side endpoint interface that is
generated when asynchronous method generation is activated by the JAX-WS
tool.

Example 2-29   HelloMessenger asynchronous client interface

```
package itso.hello;

import java.util.concurrent.Future;
import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebResult;
```

```java
import javax.jws.WebService;
import javax.xml.bind.annotation.XmlSeeAlso;
import javax.xml.ws.AsyncHandler;
import javax.xml.ws.RequestWrapper;
import javax.xml.ws.Response;
import javax.xml.ws.ResponseWrapper;

@WebService(name = "HelloMessenger", targetNamespace =
"http://hello.itso/")
@XmlSeeAlso({
    ObjectFactory.class
})
public interface HelloMessenger {

    @WebMethod(operationName = "sayHello")
    @RequestWrapper(localName = "sayHello", targetNamespace =
"http://hello.itso/", className = "itso.hello.SayHello")
    @ResponseWrapper(localName = "sayHelloResponse", targetNamespace =
"http://hello.itso/", className = "itso.hello.SayHelloResponse")
    public Response<SayHelloResponse> sayHelloAsync(
        @WebParam(name = "arg0", targetNamespace = "")
        String arg0);


    @WebMethod(operationName = "sayHello")
    @RequestWrapper(localName = "sayHello", targetNamespace =
"http://hello.itso/", className = "itso.hello.SayHello")
    @ResponseWrapper(localName = "sayHelloResponse", targetNamespace =
"http://hello.itso/", className = "itso.hello.SayHelloResponse")
    public Future<?> sayHelloAsync(
        @WebParam(name = "arg0", targetNamespace = "")
        String arg0,
        @WebParam(name = "asyncHandler", targetNamespace = "")
        AsyncHandler<SayHelloResponse> asyncHandler);


    @WebMethod
    @WebResult(targetNamespace = "")
    @RequestWrapper(localName = "sayHello", targetNamespace =
"http://hello.itso/", className = "itso.hello.SayHello")
    @ResponseWrapper(localName = "sayHelloResponse", targetNamespace =
"http://hello.itso/", className = "itso.hello.SayHelloResponse")
    public String sayHello(
```

```
          @WebParam(name = "arg0", targetNamespace = "")
          String arg0);

}
```

The method signatures are highlighted in bold. The first signature is used to create asynchronous polling clients. The second signature is used to create asynchronous callback clients. The third signature is used to create synchronous clients and is the signature that is used by the HelloClient application.

In the following sections we explain you how to use the asynchronous methods.

Polling clients

The polling client programming model refers to the usage of the asynchronous method that returns a typed javax.xml.ws.Response. Example 2-30 shows an asynchronous HelloMessenger Web service client application.

Example 2-30   HelloAsyncPollingClient

```
import javax.xml.ws.Response;

import itso.hello.HelloMessenger;
import itso.hello.HelloMessengerService;
import itso.hello.SayHelloResponse;


public class HelloAsyncPollingClient {

    public static void main(String... args) throws Exception {

        HelloMessengerService service = new HelloMessengerService();

        HelloMessenger port = service.getHelloMessengerPort();

        Response<SayHelloResponse> sayHelloAsync =
port.sayHelloAsync("Thilde");

        while ( ! sayHelloAsync.isDone() ) {
           // Do something useful for now
        }

        // Web service endpoint has now responded:
        SayHelloResponse sayHelloResponse = sayHelloAsync.get();
        String message = sayHelloResponse.getReturn();
```

```
            System.out.println(message);
    }
}
```

The relevant code is highlighted in bold. The client application invokes the sayHelloAsync method, which returns a response object. This object provides methods to query for response arrival, cancel a response, and get the actual response. The application, in this case, performs a busy wait, looping until the Response.isDone() method returns true, which indicates that the response has been received. The application then fetches the response by using the get() method. This method returns the response wrapper element that contains the actual method return value, which in this case is a simple java.lang.String object. If an endpoint throws a service exception, the get() method can throw a java.util.concurrent.ExecutionException, which can then be queried for the cause.

Callback clients

The callback client programming model refers to the usage of the asynchronous method that accepts an input parameter of a typed javax.xml.ws.AsyncHandler. Example 2-31 shows an asynchronous callback HelloMessenger Web service client application.

Example 2-31   HelloAsyncCallbackClient

```
import itso.hello.HelloMessenger;
import itso.hello.HelloMessengerService;
import itso.hello.SayHelloResponse;

import javax.xml.ws.AsyncHandler;
import javax.xml.ws.Response;


public class HelloAsyncCallbackClient {

    public static void main(String... args) throws Exception {

        HelloMessengerService service = new HelloMessengerService();

        HelloMessenger port = service.getHelloMessengerPort();

        port.sayHelloAsync("Teresa", new AsyncHandler<SayHelloResponse>()
{

            public void handleResponse(Response<SayHelloResponse> res) {
                try {
                    SayHelloResponse response = res.get();
```

```
                    String message = response.getReturn();
                    System.out.println(message);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }

        });
    }
}
```

The application passes in an anonymous inner class of type AsyncHandler. This class has a method called *handleResponse* that is invoked by the JAX-WS run time when the message is received. The argument to this method is of the same type that is used for the polling method, a typed response object.

Example 2-31 on page 97 does not show that you can save the return value of the asynchronous method invocation into a java.util.concurrent.Future:

```
Future<?> future = port.sayHelloAsync("Teresa", ...)
```

As is the case with the polling response object, you can query this object to obtain status and possibly cancel the operation.

Asynchrony on the wire

The asynchronous examples presented so far show you how to write clients that call a Web service asynchronously. However, that is only a statement about the client program. It does not mean that the SOAP messages that flow between the client and server over HTTP are themselves asynchronous. In fact, they are not. The SOAP/HTTP protocol is no different in the asynchronous case from the synchronous one. The client Web services run time opens a connection, sends the request, and receives the response back along the same connection.

What are the implications of this? From a client application, functional point of view, it does not matter. The client programming model is defined by the WSDL definitions of messages and port types. As long as that does not change, whether the pair of request and response messages are sent in the same TCP/IP session or in different sessions does not affect the client program. However, from an architectural and resource perspective, it matters a great deal. Tightly coupling the resources in the connection layer to the behavior of the client and server in the application layer has a lot of implications, especially for reliability, availability, resource use, and performance.

WebSphere Application Server V7 provides an easy-to-use feature that provides *real* wire-level asynchronous message exchange to your client applications. You simply configure the client-side proxy with a special property as follows:

```
// Get the request context from the SEI

Map<String, Object> requestContext =
((BindingProvider)port).getRequestContext():

// Configure the client for wire-level asynchronous message exchange

requestContext.put("com.ibm.websphere.webservices.use.async.mep",
true);
```

When enabling wire-level asynchronous message exchange like this, the client listens on a separate channel to receive the response messages from a service-initiated channel. Upon the request the client uses WS-Addressing to provide the ReplyTo endpoint reference (EPR) value to the service. After the request message is sent, the connection is effectively closed. When the service finishes the appropriate processing, it initiates a connection to the ReplyTo EPR to send a response.

> **Important:** This feature is specific to WebSphere Application Server V7. It is non-portable and not defined in the JAX-WS 2.1 specification.

To learn more about this feature, consult the WebSphere Application Server V7 Information Center at the following address:

http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/topic/com.ibm.
websphere.nd.multiplatform.doc/info/ae/ae/twbs_jaxwsclientasynch.html

Dispatch clients

With JAX-WS, you can work at the XML level by using a typed low-level interface called *javax.xml.ws.Dispatch*. By using the dispatch interface, you can work on the following objects:

► javax.xml.transform.Source objects

   You can use the usual Java XML-based approaches to work with source request and response objects.

► JAXB objects

   By specifying a JAXBContext, you can send and receive JAXB 2.1 objects.

- javax.xml.soap.SOAPMessage objects

  As previously explained, this is part of the SAAJ 1.3 API that allows you to work with a SOAP envelope in a typesafe manner. Example 2-46 on page 119 illustrates how to develop a dispatch client that uses the SAAJ API.

- javax.activation.DataSource objects

  By using these objects, you can work with MIME-typed messages.

Although the dispatch client programming model is particularly useful when developing clients for non-SOAP XML-based Web services, you are by no means limited to that. Example 2-32 illustrates how the HelloMessenger SOAP-based Web service can be used from a dispatch client.

Example 2-32   HelloDispatchClient

```
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;

import javax.xml.namespace.QName;
import javax.xml.transform.Source;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.stream.StreamResult;
import javax.xml.transform.stream.StreamSource;
import javax.xml.ws.Dispatch;
import javax.xml.ws.Service;
import javax.xml.ws.http.HTTPBinding;

public class HelloDispatchClient {

    private static final String TNS = "http://hello.itso/";

    public static void main(String... args) throws Exception {

        // Define the service name, port name, and endpoint address
        QName serviceName = new QName(TNS, "HelloMessengerService");
        QName portName = new QName(TNS, "HelloMessenger");
        String endpointAddress = "http://localhost:9999/Hello";

        // Create a service that can bind to the HelloMessenger port
        Service service = Service.create(serviceName);
        service.addPort(portName, HTTPBinding.HTTP_BINDING,
endpointAddress);

        // Create a Dynamic Dispatch client
```

```
        Dispatch<Source> dispatch = service.createDispatch(portName,
                Source.class, Service.Mode.MESSAGE);

        // Create a SOAP request String
        String request =
                "<?xml version='1.0' encoding='UTF-8'?>"
            + "<soap:Envelope "
            +
"xmlns:soap='http://schemas.xmlsoap.org/soap/envelope/'"
            +     "xmlns:q0='http://hello.itso/'>"
            +  "<soap:Body>"
            +     "<q0:sayHello>"
            +        "<arg0>Milo</arg0>"
            +     "</q0:sayHello>"
            +  "</soap:Body>"
            + "</soap:Envelope>";

        // Invoke the HelloMessenger web service
        Source soapRequest = new StreamSource(new
ByteArrayInputStream(request.getBytes()));
        Source soapResponse = dispatch.invoke( soapRequest );

        // Convert the response to a String
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        Transformer transformer =
TransformerFactory.newInstance().newTransformer();
        transformer.transform(soapResponse, new StreamResult(baos));
        String response = baos.toString();

        // Print the SOAP response String
        System.out.println(response);
    }
}
```

The application in Example 2-32 on page 100 first configures a JAX-WS Service
class that can bind to the HelloMessenger endpoint. Then it creates a Dispatch
object typed for working with javax.xml.transform.Source objects by using the
createDispatch method on the JAX-WS Service object. The createDispatch
method takes, as its last argument, a constant that indicates whether you intend
to work on the entire message (for instance, an entire SOAP envelope) or on the
payload only (for instance, a SOAP body). In this application, we use the
Service.Mode.MESSAGE constant to indicate that we are supplying the entire
SOAP message.

The application then creates a request variable, which is an ordinary string that contains a SOAP envelope with the request message payload in the body.

The last line that is highlighted in bold shows the Web service invocation that results in SOAP messages being sent between the client and the endpoint.

Finally, by using the Java XML transformation API, the application produces a string that contains the SOAP response envelope and prints it to the standard outputstream.

Configuring the client BindingProviders

As previously explained, both proxy-based clients and dispatch-based clients share a common configuration model. The configuration is performed programmatically in the context of the Java Web service client code. By using this configuration, you can specify an explicit endpoint location, HTTP protocol session behavior, HTTP authentication credentials, and more.

The actual programmatic configuration is performed on the javax.xml.ws.BindingProvider client-side object. The dynamic proxies that are generated by the JAX-WS run time implement the javax.xml.ws.BindingProvider interface, where the Dispatch interface extends it. Therefore, dispatch objects produced by the JAX-WS Service class, by contract, also implement the BindingProvider behavior.

To configure the client BindingProvider, you add information to the request context, which is an ordinary java.util.Map<String, Object> that contains the actual configuration. The keys are *strings* and the values are *objects*. The map is available by using the BindingProvider.getRequestContext() method.

JAX-WS 2.1 specifies the following standard properties that can be used to configure the request context:

► `javax.xml.ws.service.endpoint.adress` (value type: String)

Specifies the Web service endpoint address

► `javax.xml.ws.security.auth.username` (value type: String)

Specifies the user name in a set of HTTP basic authentication credentials

► `javax.xml.ws.security.auth.password` (value type: String)

Specifies the password in a set of HTTP basic authentication credentials

► `javax.xml.ws.session.maintain` (value type: Boolean, default: false)

Specifies whether the client is willing to participate in a server-initiated session

Example 2-28 on page 93 illustrates how a client application casts the dynamic proxy object to a BindingProvider and configures that object with an explicit endpoint location. Example 2-33 shows how to allow the Web service client to participate in Web service endpoint initiated sessions.

Example 2-33   HelloClientHttpSession

```
import java.util.Map;

import itso.hello.HelloMessenger;
import itso.hello.HelloMessengerService;

import javax.xml.ws.BindingProvider;

public class HelloClientHttpSession {

    public static void main(String... args) throws Exception {

        // Obtain the dynamic stub from the Service:
        HelloMessengerService service = new HelloMessengerService();
        HelloMessenger proxy = service.getHelloMessengerPort();

        // Cast the proxy to a BindingProvider:
        BindingProvider bindingProvider = (BindingProvider) proxy;

        // Get the request context
        Map<String, Object> requestContext = bindingProvider
                .getRequestContext();

        // Configure session preference
        requestContext.put(BindingProvider.SESSION_MAINTAIN_PROPERTY,
true);

        // Perform Web service method invocation and print the result
        String message = proxy.sayHello("Bass");
        System.out.println(message);
    }
}
```

Knowing that all dynamic proxies also are BindingProviders, the application explicitly casts the proxy. It then grabs the request context map and specifies to the JAX-WS run time that it is willing to participate in server-side initiated sessions.

> **Transport level options and a response context map:** This section described the request context map that is available on the BindingProvider implementation. After examining the map, you might notice that it contains additional transport-level configuration options. These options are specific to the protocol binding that you use. If the application communicates by using SOAP/HTTP, for example, you have the option to configure HTTP headers in the request.
>
> The BindingProvider implementation also exposes a response context map. Similarly to the request context, this object gives you access to transport level data. Again, by using SOAP/HTTP, you are most likely to find the HTTP response code inside the response context.

2.1.5 Handlers

The handler framework allows interception of a message at various points in its transmission. Handlers are simple Java bean classes that implement a handler contract and can be associated with Web service endpoints and Web service clients. With outgoing messages, handlers are invoked before a message is sent to the wire. With incoming messages, handlers are invoked before the receiving application receives the message. The same handler implementation is used for both incoming and outgoing messages. Handler classes are organized into *handler chains*. There are detailed rules about the order of handler invocation, particularly during fault handling. See the JAX-WS 2.1 specification for details.

JAX-WS provides two levels of handlers:

► *Logical handler*s deal with the payload level of the message. Logical handlers can be used for building non-functional behavior, such as logging and caching, that is common across protocols.

► *Protocol handlers* deal with protocol information, such as SOAP headers.

Logical handlers

Example 2-34 shows a logical handler skeleton. The main method is the handleMessage method. The close method is to clean up any resources that handler invocation might have consumed. The handleFault method is invoked if an error condition occurs, for example, if a response message contains a fault.

Example 2-34   HelloMessengerLogicalHandler skeleton

```
package itso.hello;

import javax.xml.ws.handler.LogicalHandler;
import javax.xml.ws.handler.LogicalMessageContext;
```

```
import javax.xml.ws.handler.MessageContext;

public class HelloMessengerLogicalHandler implements
LogicalHandler<LogicalMessageContext> {

   public void close(MessageContext ctx) {
   }

   public boolean handleFault(LogicalMessageContext ctx) {
      return false;
   }

   public boolean handleMessage(LogicalMessageContext ctx) {
      return false;
   }
}
```

The handleMessage method (and handleFault) return a boolean. Returning *true*
from the handleMessage method tells the JAX-WS run time that processing
should move to the next handler in the chain. Returning *false* tells the JAX-WS
run time that processing of the handler chain should end.

The parameter to handleMessage is a LogicalMessageContext. It is an extension
of java.util.Map and contains <key, value> pairs of context properties. There are
properties for items such as WSDL element names and attachment information
(if any). Since handlers are invoked for both incoming and outgoing messages, a
useful property is the MESSAGE_OUTBOUND_PROPERTY defined on the
MessageContext interface, which gives you the direction of the message:

```
Boolean outbound = ctx.get(MessageContext.MESSAGE_OUTBOUND_PROPERTY);
```

Another useful property is the message itself:

```
LogicalMessage message = ctx.getMessage();
```

You can get the payload as XML data by using the javax.xml.transform.Source:

```
Source payload = message.getPayload();
```

Alternatively, you can get the payload as JAXB objects:

```
Object jaxbPayload = message.getPayload(jaxbContext);
```

In either case, you get the payload. In the case of SOAP, the payload is the
contents of the soap:body, either in XML form (javax.xml.transform.Source) or in
JAXB form (java.lang.Object).

In 2.3.3, "Developing a JAX-WS logical handler that uses JAXB" on page 130, we show how you can use the JAXB 2.1 API in combination with a logical handler.

Protocol handlers

The only protocol handler that is currently supported by WebSphere Application Server V7 is the SOAP handler. The primary reason for writing a SOAP handler is to manipulate SOAP headers. Example 2-35 shows an example of a SOAP handler skeleton.

Example 2-35   HelloMessengerProtocolHandler skeleton

```
package itso.hello;

import java.util.Set;

import javax.xml.namespace.QName;
import javax.xml.ws.handler.MessageContext;
import javax.xml.ws.handler.soap.SOAPHandler;
import javax.xml.ws.handler.soap.SOAPMessageContext;

public class HelloMessengerProtocolHandler implements
      SOAPHandler<SOAPMessageContext> {

   public Set<QName> getHeaders() {
      return null;
   }

   public void close(MessageContext ctx) {
   }

   public boolean handleFault(SOAPMessageContext ctx) {
      return false;
   }

   public boolean handleMessage(SOAPMessageContext ctx) {
      return false;
   }
}
```

The SOAP handler skeleton in Example 2-35 has the familiar methods of close, handleFault, and handleMessage. The parameter to handleFault and handleMessage, however, is now SOAPMessageContext instead of LogicalMessageContext. In addition, the *getHeaders* method is a new method to implement.

The getHeaders method returns the set of the header names that the handler understands. The JAX-WS run time calls this method to determine whether the handler can process SOAP headers that *must* be understood (as indicated by the SOAP mustUnderstand attribute). It does not call this method to filter handler invocation. All handlers in a chain are called for all messages.

The SOAPMessageContext class adds a few properties to the context map that are SOAP-specific, such as roles. The getMessage method returns a SOAPMessage, which is an SAAJ class. By using the SOAPMessage, you can programmatically examine or modify SOAP message headers.

In 2.2.3, "Developing a JAX-WS protocol handler" on page 121, we show how you can use the SAAJ 1.3 API in combination with a protocol handler.

> **Using JAX-WS properties to send and receive SOAP headers:**
> WebSphere Application Server V7 provides extensions to the JAX-WS client programming model in which special properties are used to send and receive SOAP headers. For performance reasons, use this extension to process SOAP headers instead of using the SAAJ API in handlers. For more information see the following address:
>
> http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/index.jsp?topic=/com.ibm.websphere.nd.multiplatform.doc/info/ae/ae/rwbs_impsoapheadexmpjaxws.html

Enabling handlers in Web services and clients

In this section we explain how to enable JAX-WS handlers for Web service endpoints and Web service clients.

Enabling handlers in Web services

With JAX-WS, you can declaratively specify a Web service handler chain by using the javax.jws.HandlerChain annotation. Example 2-36 shows how to apply a handler chain to the HelloMessenger Web service.

Example 2-36   Enabling handlers for HelloMessenger

```
package itso.hello;

import javax.jws.HandlerChain;
import javax.jws.WebService;

@WebService
@HandlerChain(file = "handler-chain.xml")
public class HelloMessenger {
   public String sayHello(String name) {
```

```
        return String.format("Hello %s", name);
    }
}
```

The modified HelloMessenger Web service contains a class-level HandlerChain annotation, which specifies that the handler configuration should be loaded from the `handler-chain.xml` file that is available in the class path. Example 2-37 shows the `handler-chain.xml` file in its entirety.

Example 2-37   The handler-chain.xml file

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<javaee:handler-chains xmlns:javaee="http://java.sun.com/xml/ns/javaee"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <javaee:handler-chain>
        <javaee:handler>
            <javaee:handler-class>itso.hello.HelloMessengerProtocolHandler
            </javaee:handler-class>
        </javaee:handler>
    </javaee:handler-chain>

</javaee:handler-chains>
```

The highlighted line tells the JAX-WS run time to use the HelloMessengerProtocolHandler to handle incoming and outgoing messages.

Enabling handlers in Web service clients

JAX-WS specifies two approaches to enabling handlers in Web service clients:

► Declarative configuration
► Programmatic configuration

The declarative configuration works for proxy clients only and works in the same way as for the endpoint example. That is, you simply add an annotation to the client-side generated SEI, which enables the handlers for all proxies and dispatch clients that are generated by using any ports on the SEI.

The programmatic configuration works for all JAX-WS services, generated service interfaces, and the generic service. Handlers are added to the client-side service by using the following method:

`Service.setHandlerResolver(HandlerResolver handlerResolver)`

Example 2-38 shows a modified version of the HelloClient that programmatically adds the HelloMessengerProtocolHandler to the client-side SEI.

Example 2-38   HelloClientWithHandler

```
import itso.hello.HelloMessenger;
import itso.hello.HelloMessengerService;

import java.util.Collections;
import java.util.List;

import javax.xml.ws.handler.HandlerResolver;
import javax.xml.ws.handler.PortInfo;

public class HelloClientWithHandler {

    public static void main(String... args) throws Exception {

        HelloMessengerService service = new HelloMessengerService();

        service.setHandlerResolver(new HandlerResolver() {

            public List getHandlerChain(PortInfo inf) {
                return Collections.singletonList(new
HelloMessengerProtocolHandler());
            }
        });

        HelloMessenger port = service.getHelloMessengerPort();

        String message = port.sayHello("Bastian");

        System.out.println(message);
    }
}
```

The highlighted code shows the relevant handler configuration code. The application creates a new anonymous inner HandlerResolver that returns a singleton list that contains an instance of the HelloMessengerProtocolHandler.

An advantage of the programmatic approach is that you are not required to change generated code.

2.1.6 Handling binary content

By using JAX-WS 2.1, you can send binary data in SOAP-based Web service applications. You can choose between the following two approaches to send the binary data:

► Sending the encoded binary data in the SOAP requests payload
► Sending the binary data as an attachment to the SOAP payload

The first approach is easy, mostly interoperable, and supported over any transport protocol (SOAP,/HTTP, SOAP/JMS, and so on). However, use of this method means that the data being sent inside the SOAP body is encoded by using the Base64 algorithm by JAXB 2.1 and transferred in the SOAP body payload by using the xs:base64Binary XML schema data type. The Base64 encoding scheme can result in large SOAP messages being sent over the wire.

In the second approach, the binary data is attached to the SOAP envelope in a transport protocol-specific manner. In general, this results in much smaller SOAP messages simply because the payload is not encoded. In JAX-WS 2.1, you use the MTOM functionality to handle the attachments automatically. However, JAX-WS 2.1 only requires run times to support MTOM with SOAP 1.1/HTTP and SOAP 1.2/HTTP.

The following sections provide a simple example of these two approaches.

SOAP messages with attachments versus MTOM: In addition to MTOM, WebSphere Application Server V7 allows the sending of attachments by using SOAP Messages with Attachments. Both approaches send attachments as MIME parts. However, there are several reasons for using MTOM instead:

► MTOM has good interoperability. Since MTOM is a W3C recommendation that is endorsed by most of the major players (Microsoft®, IBM, Oracle®, and so on), there is a good chance for better interoperability than SOAP Messages with Attachments solutions offer.

► MTOM attachments can be processed by the Web services functionalities as needed. As part of the serialization process, the SOAP engine is invoked with a temporary Base64 representation of the attachments. This allows the SOAP engine to use them for generating a message signature, performing encryption, and so on.

► MTOM has no impact on development. Contrary to SOAP Messages with Attachments, there is no special API for handling the SOAP attachments.

Binary content in the payload

JAX-WS 2.1 allows for different binary Java types as determined by the JAXB 2.1 binding rules. Examples of Java types that are treated as binary types include java.awt.Image, javax.activation.DataHandler, javax.xml.transform.Source, and byte arrays. These types are all transmitted by using Base64 encoding (with the xs:base64Binary schema type) in the SOAP payload.

Example 2-39 shows a modified version of the HelloMessenger Web service that allows for binary data transfer.

Example 2-39   HelloBinaryMessenger

```
package itso.hello;

import javax.jws.WebService;

@WebService
public class HelloBinaryMessenger {

    public byte[] sayHello(byte[] nameAsBytes) {
        return String.format("Hello %s", new
String(nameAsBytes)).getBytes();
    }

}
```

The endpoint naively assumes that the byte array received in the method input argument contains valid numeric char values. The result of invoking the endpoint method sayHello is the usual `Hello <name>` name message, but in the form of a byte array. Example 2-40 shows the corresponding binary SEI client generated by JAX-WS tools.

Example 2-40   HelloBinaryClient

```
import itso.hello.HelloBinaryMessenger;
import itso.hello.HelloBinaryMessengerService;

import javax.xml.ws.BindingProvider;


public class HelloBinaryClient {

    public static void main(String... args) throws Exception {

        HelloBinaryMessengerService service = new
HelloBinaryMessengerService();
```

```
      HelloBinaryMessenger port =
service.getHelloBinaryMessengerPort();

      byte[] message = port.sayHello( "Milo".getBytes() );

      System.out.println( new String(message) );
   }
}
```

The highlighted code illustrates that the generated port signature is identical to that of the service endpoint implementation.

> **Note:** The WSDL interface that is exposed by the example Web service endpoint uses the xs:base64Binary schema for representation of the binary types. The implication of this is that Web service client generation tools, such as the **wsimport** tool, by default, generate a client-side SEI that uses the byte[] signatures. Even though the original Web service endpoint has a java.awt.Image type in its method signature, the generated client code might result in a byte[] signature.
>
> A simple solution is to annotate the schema types that are used by the WSDL document with a metadata attribute. By using this method, you can tell the JAXB run time which content type is really expected. With this method, you can affect which binary types are used in the client-side SEI. See the JAXB 2.1 specification for more information.

Example 2-41 shows the SOAP messages that are exchanged as a result of running this example.

Example 2-41   Binary data in the SOAP payload messages

```
<!-- SOAP Request Envelope -->

<?xml version="1.0" encoding="UTF-8"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Body>
      <ns2:sayHello xmlns:ns2="http://hello.itso/">
         <arg0>TWlsbw==</arg0>
      </ns2:sayHello>
   </S:Body>
</S:Envelope>
```

```
<!-- SOAP Response Envelope -->

<?xml version="1.0" encoding="UTF-8"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
    <S:Body>
        <ns2:sayHelloResponse xmlns:ns2="http://hello.itso/">
            <return>SGVsbG8gTWlsbw==</return>
        </ns2:sayHelloResponse>
    </S:Body>
</S:Envelope>
```

As expected, the SOAP messages reveal that the default behavior of sending binary data is to send it in the SOAP envelope payload by using a Base64 encoding.

Binary content as attachments using MTOM

With JAX-WS 2.1, you can send binary content as attachments by using the MTOM. The MTOM describes a mechanism for optimizing the transmission of a SOAP message by selectively re-encoding portions of the message while still presenting an XML information set (Infoset) to the SOAP application.

MTOM uses *XML-binary Optimized Packaging* (XOP) in the context of SOAP and MIME over HTTP. XOP defines a serialization mechanism for the XML Infoset with binary content that is applicable to SOAP and MIME packaging, as well as any XML Infoset and any packaging mechanism. It is an alternate serialization of XML that happens to look similar to a MIME multipart or related package, with XML documents as the root part.

That root part is similar to the normal XML serialization of the document, except that Base64 encoded data is replaced by a reference to one of the MIME parts, which is *not* Base64 encoded. This reference allows the JAX-WS run time to avoid the overhead in terms of size and processing that is associated with encoding.

To enable MTOM on a service endpoint, simply annotate it by using MTOM-specific metadata. Example 2-42 shows a modified version of the HelloBinaryMessenger endpoint.

Example 2-42 HelloBinaryMessenger using MTOM

```
package itso.hello;

import javax.jws.WebService;
import javax.xml.ws.BindingType;
import javax.xml.ws.soap.SOAPBinding;
```

```
@WebService
@BindingType(value=SOAPBinding.SOAP11HTTP_MTOM_BINDING)
public class HelloBinaryMessenger {

   public byte[] sayHello(byte[] nameAsBytes) {
      return String.format("Hello %s", new
String(nameAsBytes)).getBytes();
   }

}
```

The endpoint specifies that its binding should be SOAP 1.1/HTTP with MTOM support. Now the endpoint can use MTOM on its *outbound* SOAP responses. Running HelloBinaryClient against this MTOM-enabled endpoint does not mean that the SOAP request message will be MTOM encoded. That requires configuration on the client-side BindingProvider (proxy client or dispatch client).

Example 2-43 shows the complete response message including HTTP protocol information.

Example 2-43   HelloBinaryMessenger response

```
HTTP/1.1 200 OK
Content-Type: multipart/related;
boundary=MIMEBoundaryurn_uuid_A235ABD296C37F53271239995376835;
type="application/xop+xml";
start="<0.urn:uuid:A235ABD296C37F53271239995376836@apache.org>";
start-info="text/xml"
Content-Language: en-US
Content-Length: 881
Date: Fri, 17 Apr 2009 19:09:34 GMT
Server: WebSphere Application Server/7.0

--MIMEBoundaryurn_uuid_A235ABD296C37F53271239995376835
Content-Type: application/xop+xml; charset=UTF-8; type="text/xml"
Content-Transfer-Encoding: binary
Content-ID: <0.urn:uuid:A235ABD296C37F53271239995376836@apache.org>

<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
   <soapenv:Body>
      <ns2:sayHelloResponse xmlns:ns2="http://hello.itso/">
```

```
            <return>
                <xop:Include
xmlns:xop="http://www.w3.org/2004/08/xop/include"
href="cid:urn:uuid:A235ABD296C37F53271239995303364@apache.org"/>
            </return>
        </ns2:sayHelloResponse>
    </soapenv:Body>
</soapenv:Envelope>
--MIMEBoundaryurn_uuid_A235ABD296C37F53271239995376835
Content-Type: application/octet-stream
Content-Transfer-Encoding: binary
Content-ID: <urn:uuid:A235ABD296C37F53271239995376837@apache.org>

Hello Milo
--MIMEBoundaryurn_uuid_A235ABD296C37F53271239995376835--
```

In the first block highlighted in bold, the SOAP payload uses the include element
from the XOP namespace to refer to the actual attachment part. At the bottom
you see the attachment. Notice that the binary data has not been encoded.

For a client BindingProvider to use MTOM when sending binary data, you must
configure it programmatically. Example 2-44 shows a modified version of the
HelloBinaryClient that configures the Binding for MTOM.

Example 2-44   HelloBinaryClient with MTOM enabled

```
import itso.hello.HelloBinaryMessenger;
import itso.hello.HelloBinaryMessengerService;

import javax.xml.ws.BindingProvider;
import javax.xml.ws.soap.SOAPBinding;


public class HelloBinaryClient {

    public static void main(String... args) throws Exception {

        HelloBinaryMessengerService service = new
HelloBinaryMessengerService();

        HelloBinaryMessenger port =
service.getHelloBinaryMessengerPort();

        BindingProvider bindingProvider = (BindingProvider) port;
```

```
        SOAPBinding soapBinding = (SOAPBinding)
bindingProvider.getBinding();
        soapBinding.setMTOMEnabled(true);

        byte[] message = port.sayHello( "Milo".getBytes() );

        System.out.println( new String(message) );
    }
}
```

The MTOM-related change is highlighted in bold. Because MTOM is specific to the SOAP binding, the application obtains the SOAPBinding object from the BindingProvider class. The SOAPBinding interface contains a method, setMTOMEnabled, which is used by the application to enable MTOM for outgoing SOAP messages. The request SOAP message is identical in structure to the response message and is therefore not described further.

2.1.7 Enabling SOAP 1.2

JAX-WS 2.1 uses SOAP 1.1 as the default binding protocol. In order for the JAX-WS run time to expose a Web service endpoint by using the SOAP 1.2 binding protocol, the endpoint must explicitly provide the metadata.

Example 2-45 shows a modified HelloMessenger endpoint that binds to SOAP 1.2/HTTP.

Example 2-45   SOAP 1.2/HTTP enabled HelloMessenger endpoint

```
package itso.hello;

import javax.jws.WebService;
import javax.xml.ws.BindingType;
import javax.xml.ws.soap.SOAPBinding;

@WebService
@BindingType(value=SOAPBinding.SOAP12HTTP_BINDING)
public class HelloMessenger {
    public String sayHello(String name) {
        return String.format("Hello %s", name);
    }
}
```

The highlighted line shows how the endpoint configures the JAX-WS run time for the SOAP 1.2/HTTP binding. The endpoint WSDL document reveals that SOAP

1.2 is enabled because of its SOAP 1.2 namespace declaration on the WSDL definitions element:

```
xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
```

> **SOAP 1.2 Web service clients**: With regard to JAX-WS 2.1 Web service clients, there is no need to manually specify the SOAP 1.2/HTTP protocol binding information. Rather, at run time, the client-side JAX-WS implementation uses binding information from the WSDL document of the Web service endpoint to produce an appropriate dynamic proxy implementation.

2.2 Working with SOAP using SAAJ 1.3

Until now, we have primarily demonstrated how to work on a higher level with SEI-based Web services. However, there might be times where it is necessary for you to work on the actual SOAP envelope details. An example of where low-level access is necessary is when you develop JAX-WS handlers that process SOAP header metadata such as security information, trace information, and so on. JAX-WS itself does not contain a low-level SOAP API. Rather, it delegates it to another matured API called *SOAP with Attachments API for Java*.

WebSphere Application Server V7 comes with SAAJ 1.3, which supports both SOAP 1.1 and 1.2 messages. Additionally, by using SAAJ 1.3, you can write WS-I BP 1.1-compliant messages.

2.2.1 SAAJ overview

Although the SAAJ API can be used from JAX-WS applications, it can also be used alone. In fact, the SAAJ API contains all the classes that you need to send and receive SOAP messages. All the SAAJ classes belong to the javax.xml.soap package namespace.

As shown in Figure 2-1, the core SAAJ API exposes classes that closely mimic the actual SOAP messages structure.
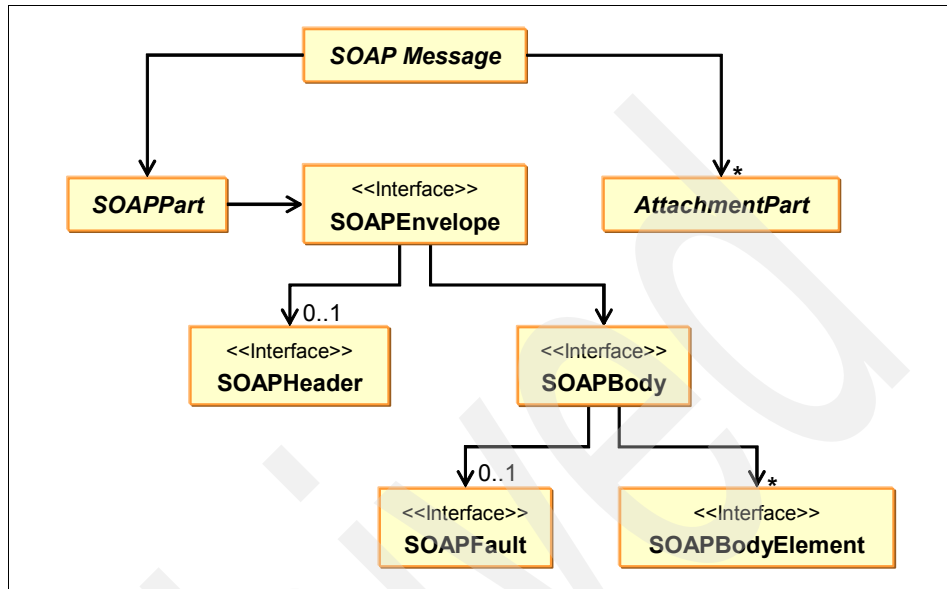
Figure 2-1   Core SAAJ 1.3 API

The core SAAJ API includes the following classes:

► The SOAPMessage object represents the entire SOAP message. It has a single SOAPPart and possibly one or more AttachmentParts.

► The SOAPPart object contains a SOAPEnvelope message. The SOAPEnvelope represents the actual SOAPEnvelope.

► The SOAPEnvelope has an optional SOAPHeader and a mandatory SOAPBody.

► The SOAPHeader represents the SOAP header block in a SOAP message and is allowed to be empty as is the case with the header section in a SOAP 1.1 or 1.2 message.

► The SOAPBody element can contain either a SOAPFault object or the actual SOAP payload XML content (SOAPBodyElement).

► The SOAPFault object represents a SOAP fault message.

The SAAJ types and the types from the org.w3c.dom Java XML package are closely related. In fact, many of the classes in SAAJ extend or implement behavior from classes in the org.w3c.dom package namespace. An example of this is the SOAPPart object that implements the org.w3c.dom.Document interface.

In the next section we illustrate one way of putting the API into use, which is by developing a dispatch client that uses SAAJ.

2.2.2 Developing a dispatch client that uses SAAJ

You can use SAAJ, among other tasks, to write SOAP-based clients. Example 2-46 shows a dispatch client application that uses SAAJ to construct the core SOAP envelope.

Example 2-46   SimpleSaajClient

```
import javax.xml.namespace.QName;
import javax.xml.soap.MessageFactory;
import javax.xml.soap.SOAPBody;
import javax.xml.soap.SOAPBodyElement;
import javax.xml.soap.SOAPMessage;
import javax.xml.ws.BindingProvider;
import javax.xml.ws.Dispatch;
import javax.xml.ws.Service;
import javax.xml.ws.soap.SOAPBinding;

public class HelloSaajClient {

    private static final String TNS = "http://hello.itso/";

    public static void main(String... args) throws Exception {

        // Define the service name, port name, and endpoint address
        QName serviceName = new QName(TNS, "HelloMessengerService");
        QName portName = new QName(TNS, "HelloMessenger");
        String endpoint = "http://localhost:80/Hello";

        // Create a service that can bind to the HelloMessenger port
        Service service = Service.create(serviceName);
        service.addPort(portName, SOAPBinding.SOAP11HTTP_BINDING,
endpoint);

        // Create a Dynamic Dispatch client
        Dispatch<SOAPMessage> dispatch = service.createDispatch(portName,
                SOAPMessage.class, Service.Mode.MESSAGE);

        // Grab the SOAPBinding which has a SAAJ MessageFactory
        BindingProvider bindingProvider = (BindingProvider) dispatch;
        SOAPBinding binding = (SOAPBinding) bindingProvider.getBinding();
```

```
      // Use the SAAJ API to create the request
      MessageFactory factory = binding.getMessageFactory();
      SOAPMessage requestMessage = factory.createMessage();
      SOAPBody soapBody = requestMessage.getSOAPBody();
      QName payloadRootElem = new QName(TNS, "sayHello", "h");
      SOAPBodyElement bodyElement =
soapBody.addBodyElement(payloadRootElem);
      bodyElement.addChildElement("arg0").addTextNode("Milo");

      // Invoke the HelloMessenger Web service
      SOAPMessage responseMessage = dispatch.invoke(requestMessage);

      // Convert the response message
      String response = responseMessage.getSOAPBody().getTextContent();

      // Print the response
      System.out.println(response);
   }
}
```

The application creates a dispatch client typed for usage with the SOAPMessage element, which is part of SAAJ.

The third argument in the createDispatch method (value Service.Mode.MESSAGE) indicates that the application is responsible for building the entire SOAP envelope. It then gets a SAAJ MessageFactory from the SOAPBinding object and uses that object to create a SAAJ SOAPMessage object (variable requestMessage).

By using the SOAPBody and SOAPBodyElement SAAJ API objects, the application adds the sayHello request wrapper element (sayHello) together with the argument (arg0) to the SOAP message payload. Having built the request message, the application uses the invoke method on the dispatch object to send the message and subsequently gets the response SOAPMessage object.

Finally, the application gets the result message ("Hello Milo") by extracting all text content from the SOAPBody payload element and prints it to the standard outputstream.

2.2.3 Developing a JAX-WS protocol handler

As explained in "Protocol handlers" on page 106, with JAX-WS you can write SOAP protocol handlers by using the SAAJ API. Example 2-47 shows a protocol handler that adds a custom SOAP header.

Example 2-47   HeaderMessageHandler

```
package itso.hello;

import java.util.Set;

import javax.xml.namespace.QName;
import javax.xml.soap.SOAPEnvelope;
import javax.xml.soap.SOAPException;
import javax.xml.soap.SOAPHeader;
import javax.xml.soap.SOAPHeaderElement;
import javax.xml.soap.SOAPMessage;
import javax.xml.soap.SOAPPart;
import javax.xml.ws.handler.MessageContext;
import javax.xml.ws.handler.soap.SOAPHandler;
import javax.xml.ws.handler.soap.SOAPMessageContext;

public class HeaderMessageHandler implements
SOAPHandler<SOAPMessageContext> {

    public boolean handleMessage(SOAPMessageContext ctx) {

        String outboundProp = MessageContext.MESSAGE_OUTBOUND_PROPERTY;
        boolean outbound = (Boolean) ctx.get(outboundProp);

        if (outbound) {

            try {
                SOAPMessage soapMessage = ctx.getMessage();
                SOAPPart soapPart = soapMessage.getSOAPPart();
                SOAPEnvelope soapEnvelope = soapPart.getEnvelope();

                QName headerQName = new QName("http://hello.itso/",
"header-message", "hm");
                String headerValue = "Hello SAAJ";

                SOAPHeader soapHeader = soapEnvelope.addHeader();
                SOAPHeaderElement soapHeaderElement =
soapHeader.addHeaderElement(headerQName);
```

```
                soapHeaderElement.addTextNode(headerValue);

        } catch (SOAPException e) {
            e.printStackTrace();
        }

    }

    return true /* continue chain */;
}

public Set<QName> getHeaders() {
    return null;
}

public void close(MessageContext ctx) {
}

public boolean handleFault(SOAPMessageContext ctx) {
    return false;
}
}
```

The HelloMessageHandler checks whether the message being processed is outbound. If it is, the handler adds a SOAP header by using the SAAJ SOAPHeader object. To the SOAPHeader element, it then adds a custom header-message header that contains the greeting Hello SAAJ.

Example 2-48 shows a SOAP response message that is generated from the HelloMessenger service when it uses the HeaderMessageHandler.

Example 2-48   SOAP response message with custom SOAP header

```
<?xml version="1.0" encoding="UTF-8"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Header>
      <hm:header-message xmlns:hm="http://hello.itso/">Hello
SAAJ</hm:header-message>
   </S:Header>
   <S:Body>
      <ns2:sayHelloResponse xmlns:ns2="http://hello.itso/">
         <return>Hello Milo</return>
      </ns2:sayHelloResponse>
```

```
        </S:Body>
    </S:Envelope>
```

2.3 Working with XML using JAXB 2.1

From a developer's perspective, Java Architecture for XML Binding (JAXB) provides an object-oriented approach to working with XML documents. The JAXB API contains classes with which you can serialize annotated Java objects to XML and back again.

In reality, JAXB provides a detailed description of the mapping between XML types and Java types. JAX-WS describes a mapping between WSDL and Java. Note the distinction. As described in 2.1.2, "Relation of WSDL and Java types" on page 65, JAX-WS maps WSDL messages, portTypes, bindings, and services. Anything that is displayed in the WSDL document's types section that is XML schema types is not mapped by JAX-WS. JAX-WS defers that mapping to JAXB. WebSphere Application Server V7 supports JAX-WS 2.1.

JAXB can be used in applications that are not related to Web services in any way. However, in the context of JAX-WS Web services development, knowledge of JAXB can be helpful in the following situations:

► You are developing JAX-WS handlers that need access to the message payload. With JAXB, you can avoid working with technologies such as XPath and DOM.

► You are developing a dispatch client that communicates with a non-SOAP Web service that has an XML schema that describes the message exchange.

► You are developing non-SOAP Web service endpoints (as described in "Provider-based Web services" on page 85) but want to work on XML schema-compliant request/response documents.

► You must customize generated SEI Java code. By providing certain JAXB binding hints, for example, you can change the way JAX-WS tools generate code.

While this list is not complete, the key point is that JAXB can, either directly or indirectly, be a core technology in your daily Java Web service development toolbox.

2.3.1 Overview of JAXB

The JAXB mapping rules between XML schema types and Java types are quite simple. As an example, primitives, such as the Java int, are mapped to a corresponding xsd:int schema primitive, and complex Java beans are mapped to xsd:complexType declarations. A binding compiler, such as the JDK tool xjc.sh, can use these rules to generate Java objects from an XML schema file. The Java objects that are generated can then, by means of the JAXB API classes, be marshalled into schema-compliant XML documents and vice versa.

Table 2-1 shows some of the default mapping rules that are defined by JAXB 2.1 for binding XML schema data types to Java.

Table 2-1   JAXB mapping rules

XML schema type	Java type
xsd:string	java.lang.String
xsd:int	int
xsd:long	long
xsd:decimal	java.math.BigDecimal
xsd:date	javax.xml.datatype.XMLGregorianCalendar
xsd:boolean	boolean
xsd:base64Binary	byte[]

Before you can develop and run the HelloClient application (Example 2-4 on page 64), you must first use a JAX-WS tool, such as the `wsimport` tool, to generate client-side support code. The generated code is produced according to the JAXB binding rules such as those in Table 2-1.

As is the case with JAX-WS WSDL-to-Java and Java-to-WSDL mapping rules, there are cases where JAXB rules from Java to XML schema and XML schema to Java might not always agree. For example, if you have a schema that has xsd:hexBinary type declarations, they will map to Java byte[]. However, going the other way can result in xsd:base64Binary schema declarations.

Although we initially claimed that the conversion rules are simple, JAXB 2.1 is quite an elaborate specification that thoroughly describes many elaborate mapping scenarios. For details about the exact mapping rules between XML schema and Java, consult the JAXB 2.1 specification.

Example JAXB mapping

JAXB is able to marshal and unmarshal Java objects that carry JAXB annotations. You can develop the Java beans yourself or generate them by using a JAXB compiler (either directly or indirectly through a JAX-WS tool).

In the HelloMessenger Web service endpoint, we describe how to access the WSDL document (that is, access the endpoint URL with an appended ?wsdl query string with a browser). The WSDL document contains a types section that defines the XML schema types that are being used by the endpoint. Although the actual schema type definitions might be inline in the WSDL types section, you will typically find that the WSDL document imports an external XML schema document, which you also can view from the browser. If you look at the XML schema, you can see the definition of the response wrapper element that is being used in the marshalling process when the HelloMessenger endpoint sends SOAP responses to the client.

Example 2-49 shows the sayHelloResponse complexType definition that is present in the XML schema.

Example 2-49   sayHelloResponse XML schema type

```
<xs:complexType name="sayHelloResponse">
    <xs:sequence>
        <xs:element name="return" type="xs:string"
minOccurs="0"></xs:element>
    </xs:sequence>
</xs:complexType>
```

The line highlighted in bold shows the definition of the actual operation's return value. From this schema type, a JAXB compiler can then generate a corresponding Java bean.

The Java bean in Example 2-50 shows the corresponding response wrapper Java bean that is generated by the JAXB rules when running the `wsimport` tool on the HelloMessenger WSDL document. Note that the JAXB-generated comments have been removed for brevity.

Example 2-50   SayHelloResponse

```
package itso.hello;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlType;
```

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "sayHelloResponse",
         namespace = "http://hello.itso/",
         propOrder = { "_return" })
public class SayHelloResponse {

   @XmlElement(name = "return")
   protected String _return;

   public String getReturn() {
      return _return;
   }

   public void setReturn(String value) {
      this._return = value;
   }

}
```

The actual mapping is quite simple. It only requires a few JAXB annotations from the javax.xml.bind.annotation package.

The core annotation that makes this bean serializable by using JAXB is the XmlType annotation. However, to make JAXB marshalled and unmarshalled instances of this bean into schema-compliant XML documents, it requires further metadata. The first letter in the Java bean class name is uppercase, but the schema type is not. Therefore, specifying the name attribute on the XmlType annotation specifically defines the serialized name. The XmlType namespace attribute ensures that the bean maps to the target namespace that is defined inside the schema.

The XmlType propOrder attribute ensures that the bean properties are mapped in the exact same order as defined by the XML schema sequence element.

The XmlAccessorType with the value XmlAccessType.FIELD ensures that the _return field (annotated with XmlElement) is used in the marshal or unmarshal process rather than the get and set methods.

As you might suspect, JAXB allows for further customization of the mapping. For more information, see the JAXB 2.1 specification.

Using the JAXB API
In addition to defining a set of binding rules, JAXB also defines an API that can be used to initiate the marshalling between XML documents and Java object trees.

JAXB includes the following central objects:

► javax.xml.bind.JAXBContext

  Acts as a factory and is used for creating marshaller and unmarshaller objects

► javax.xml.bind.Marshaller

  Can serialize a Java object graph into an XML stream

► javax.xml.bind.Unmarshaller

  Can deserialize an XML stream into a Java object graph

Example 2-51 shows the HelloJaxb application, which uses the JAXBContext and unmarshaller classes to unmarshal a chunk of raw XML that is identical to the payload that is sent by the HelloMessenger SOAP Web service.

Example 2-51   HelloJaxb

```
import itso.hello.SayHelloResponse;

import java.io.StringReader;

import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBElement;
import javax.xml.bind.Unmarshaller;

public class HelloJaxb {

    public static void main(String... args) throws Exception {

        StringReader xml = new StringReader(
                "<ns2:sayHelloResponse xmlns:ns2='http://hello.itso/'>" +
                    "<return>Hello Thilde</return>" +
                "</ns2:sayHelloResponse>");

        JAXBContext context = JAXBContext.newInstance("itso.hello");
        Unmarshaller unmarshaller = context.createUnmarshaller();
        JAXBElement<SayHelloResponse> jaxbElement =
                (JAXBElement<SayHelloResponse>)unmarshaller.unmarshal(xml);
        SayHelloResponse response = jaxbElement.getValue();

        System.out.println(response.getReturn());
    }

}
```

The lines that are highlighted in bold show how to use the JAXB API to deserialize a raw XML document into a Java object: the JAXB generated response wrapper Java bean.

The first object created is the JAXBContext object, which, in this example, is created by using a package name that contains the Java beans that can be marshalled or unmarshalled by using JAXB.

From the context, the application then creates the unmarshaller object, which in the subsequent line is used to unmarshal the XML document in the form of an ordinary java.io.Reader subclass. The result of this unmarshal process is the creation of a JAXBElement wrapper that contains the unmarshalled Java object.

Finally, the application prints the response value, which in this case is `Hello Thilde`.

You might wonder why the JAXBElement container element is involved in the previous example. The reason is that the SayHelloResponse object generated by the binding compiler omits the XmlRootElement annotation. You can avoid the JAXBElement container if you specify this annotation on the SayHelloResponse class and create the context with an SayHelloResponse.class argument instead. In short, the relevant lines change as follows:

```
JAXBContext context = JAXBContext.newInstance(SayHelloResponse.class);
Unmarshaller unmarshaller = context.createUnmarshaller();
SayHelloResponse response = (SayHelloResponse)
unmarshaller.unmarshal(xml);
```

2.3.2 Developing a dispatch client that uses JAXB

Example 2-52 demonstrates how JAX-WS allows you to work with JAXB objects from a dispatch client.

Example 2-52   HelloJaxbClient

```
import itso.hello.ObjectFactory;
import itso.hello.SayHello;
import itso.hello.SayHelloResponse;

import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBElement;
import javax.xml.namespace.QName;
import javax.xml.ws.Dispatch;
import javax.xml.ws.Service;
import javax.xml.ws.soap.SOAPBinding;
```

```
public class HelloJaxbClient {

    private static final String TNS = "http://hello.itso/";

    public static void main(String... args) throws Exception {

        // Define the service name, port name, and endpoint address
        QName serviceName = new QName(TNS, "HelloMessengerService");
        QName portName = new QName(TNS, "HelloMessenger");
        String endpoint = "http://localhost:80/Hello";

        // Create a service that can bind to the HelloMessenger port
        Service service = Service.create(serviceName);
        service.addPort(portName, SOAPBinding.SOAP11HTTP_BINDING,
endpoint);

        // Create a JAXB enabled Dynamic Dispatch client
        JAXBContext context = JAXBContext.newInstance("itso.hello");
        Dispatch<Object> dispatch = service.createDispatch(portName,
context,
                Service.Mode.PAYLOAD);

        // Create JAXB request object
        ObjectFactory objectFactory = new ObjectFactory();
        SayHello request = objectFactory.createSayHello();
        request.setArg0("Milo");
        JAXBElement<SayHello> requestMessage =
objectFactory.createSayHello(request);

        // Invoke the HelloMessenger Web service
        JAXBElement<SayHelloResponse> responseMessage =
            (JAXBElement<SayHelloResponse>)
dispatch.invoke(requestMessage);

        // Get the JAXB response
        SayHelloResponse response = responseMessage.getValue();
        String value = response.getReturn();

        // Print the response
        System.out.println(value);
    }
}
```

The lines in bold illustrate where JAXB functionality is being used.

The key point to note with regards to the dispatch client creation is that the Service.createDispatch method is invoked with a JAXBContext and a PAYLOAD service mode. The PAYLOAD argument specifies to the JAX-WS run time that it should take care of handling the SOAP envelope details. The JAXBContext argument specifies that we will let JAXB handle the marshalling and unmarshalling of the actualSOAP payload.

After creation of the dispatch client, the application demonstrates how the generated ObjectFactory class is used to produce the request objects that will be sent over the wire. Since the generated classes (SayHello and SayHelloResponse) in this example have not been annotated with a XmlRootElement by the JAXB binding compiler, JAXB requires that they be wrapped in a JAXBElement wrapper.

2.3.3 Developing a JAX-WS logical handler that uses JAXB

JAXB works well with logical handlers that are unaware of the actual transport protocol details. Example 2-53 demonstrates a logical JAX-WS handler that transforms outgoing payload text to uppercase.

Example 2-53   UppercaseMessageHandler

```
package itso.hello;

import itso.hello.jaxws.SayHelloResponse;

import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.ws.LogicalMessage;
import javax.xml.ws.handler.LogicalHandler;
import javax.xml.ws.handler.LogicalMessageContext;
import javax.xml.ws.handler.MessageContext;

public class UppercaseMessageHandler implements
      LogicalHandler<LogicalMessageContext> {

   public boolean handleMessage(LogicalMessageContext ctx) {

      String outboundProp = MessageContext.MESSAGE_OUTBOUND_PROPERTY;
      boolean outbound = (Boolean) ctx.get(outboundProp);

      if (outbound) {
         try {
```

```
            LogicalMessage message = ctx.getMessage();

            JAXBContext context = JAXBContext
                    .newInstance(SayHelloResponse.class);

            SayHelloResponse response = (SayHelloResponse) message
                    .getPayload(context);

            response.setReturn(response.getReturn().toUpperCase());

            message.setPayload(response, context);

        } catch (JAXBException e) {
            e.printStackTrace();
        }
    }

    return true /* continue chain */;
}

public void close(MessageContext ctx) {
}

public boolean handleFault(LogicalMessageContext ctx) {
    return false;
}
}
```

The lines that involve JAXB are highlighted in bold. The handler checks whether
the current message being handled is outbound or inbound. If the message is
outbound, it gets the JAXB payload object from the LogicalMessage, updates the
return value to uppercase, and overrides the entire payload. The
SayHelloResponse object in this example was generated with an
XmlRootElement annotation. Therefore, we did not need to be concerned with
the JAXBElement wrapper.

2.4 Web services for Java EE

In this section we introduce how the JAX-WS programming model fits into the
Java EE environment. The requirements are formally defined in JSR-109, "Web
services for Java EE specification (WSEE). Because compliance with

WSEE V1.2 is a defined requirement in the Java EE 5 specification, WebSphere Application Server V7 provides full support for it.

In short, WSEE defines the required architecture for Web services running inside the Java EE environment. WSEE standardizes the packaging, deployment, and programming model for Web services in a Java EE environment. WSEE-compliant services are portable and interoperable across different application server platforms.

2.4.1 Overview of WSEE

Prior to the appearance of the WSEE, there was no standard definition of how to deploy a Web service in a Java EE environment. Thus, the process to do so was mainly dependent on the destination run time. WSEE standardizes the process and makes it portable to every Java EE-compliant server platform. Specifically, WSEE defines the concepts, interfaces, file formats, and responsibilities to support the model for defining and deploying Web services.

The end result for developers is that WSEE brings the Web services programming models JAX-WS and JAX-RPC to the Java EE container. The JAX-WS programming model is the new standard Web services programming model for use in Java EE 5 containers. The JAX-RPC programming model was the standard Web services model used in Java 2 Platform, Enterprise Edition (J2EE™), 1.4 containers. Therefore, it is outside the scope of this book and is not described further.

WSEE defines two programming models:

► Server programming models
► Client programming model

For the server programming model, WSEE requires that Java EE 5-compliant containers must support the following two methods of implementing a Web service endpoint:

► Stateless session EJB in an EJB container
► Java classes running in a Web container

For the client programming model, WSEE describes how Java EE components, such as servlets and EJBs, should use the JAX-WS API.

The primary focus here is to describe these how JAX-WS fits into the two programming models. WSEE describes the usage of Java EE deployment descriptors in great detail. However, because Web-service-specific deployment descriptor information is optional in JAX-WS, we do not discuss this subject here. WSEE additionally provides details about the subjects of assembly and

deployment of both Web service client modules and endpoints modules. For an in-depth description of these matters, consult the WSEE specification.

2.4.2 Server programming model

The server programming model provides the server guidelines for standardizing the deployment of Web services in a Java EE server environment. Depending on the run time, two implementation models are described:

► Web container programming model

    An ordinary Java class hosted in the Web container

► EJB container programming model

    A stateless session EJB hosted in the EJB container

From a developer's perspective the implementations look similar. That is caused by the new lightweight annotation-based Java EE 5 programming model. However, the choice that you make can impact the quality of service provided to your module:

► The Web container programming model is simplistic and easily adapted by Web developers.

► The EJB container programming model automatically brings thread safety and endpoint local transactions to your Web services. In addition, with Java EE 5, development of Enterprise JavaBeans (EJB) 3.0 components have been dramatically simplified and, therefore, are a viable alternative to the Web container programming model.

In the following sections we explain how you can use JAX-WS with these two approaches.

> **Publishing endpoints using the Endpoint API not allowed**: The WSEE specification requires Java EE 5-compliant application server vendors to disallow publishing Web service endpoints by using the javax.xml.ws.Endpoint API. The reason is that the usage is considered non-portable in a managed environment. Vendors are instructed *not* to grant publishingEndpoint security permission to applications.

Web container programming model

The Web container programming model supports both SEI-based endpoints and provider-based endpoints. To deploy the HelloMessenger endpoint from the provider (see "The provider" on page 60) to WebSphere Application Server V7, place it in a Java EE 5-compliant WAR module and deploy it.

Figure 2-2 shows a WAR module made with Rational Application Developer V7.5 that contains the HelloMessenger Web service endpoint.
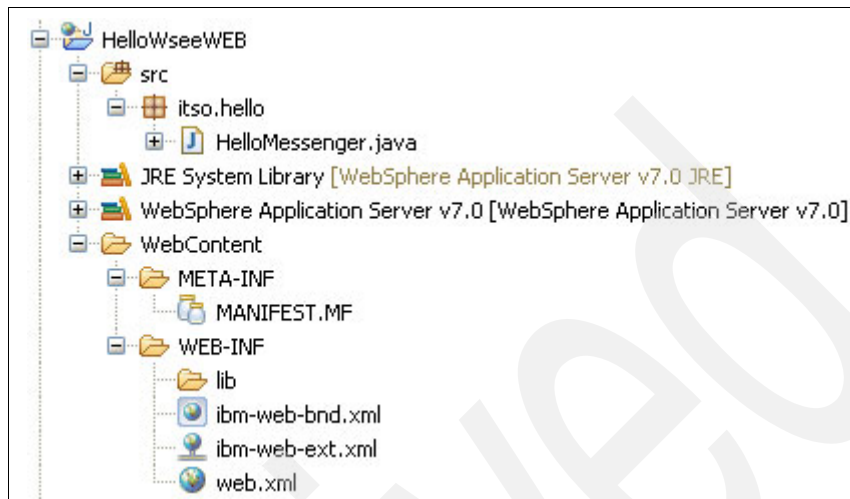
Figure 2-2   HelloWseeWEB WAR archive

This HelloWseeWEB project is a regular dynamic Web project, which with regards to Web services-related files, only contains the HelloMessenger endpoint. The WEB-INF folder contains the mandatory `Web.xml` deployment descriptor and the IBM-specific extension and binding file. Upon deployment of this module, WebSphere Application Server V7 exposes a dynamically generated WSDL document at the following URL:

`http://localhost:[port]/HelloWseeWEB/HelloMessengerService?wsdl`

In essence, besides wrapping Web services code in a WAR module, the programming model is basically mandated by JAX-WS. If you are used to the JAX-RPC programming model, you might notice that the module does not contain a `webservices.xml` file, the JAX-RPC XML mapping file, a WSDL document, and so on. By using JAX-WS 2.1, there is no longer a need for such files, unless you insist on developing JAX-RPC-based services under Java EE 5.

The `webservices.xml` file can still be used in JAX-WS applications. If you use this file, it can override the metadata that you specified by using the JAX-WS annotations. For more information about usage of the `webservices.xml` file, see the WSEE specification.

> **Thread safety in the Web container programming model**: A JAX-WS
> service endpoint in the Web container can be either single threaded or
> multi-threaded. WSEE specifies that endpoints that require single threaded
> access must implement the javax.servlet.SingleThreadModel interface. When
> this interface is implemented by the endpoint, the container is required to
> serialize all method requests to the service implementation bean.

Lifecycle events

The Web service endpoint life cycle is completely determined by the Web
container. However, a service implementation bean can use
javax.annotation.PostConstruct or javax.annotation.PreDestroy annotations on
methods for life-cycle event callbacks. Example 2-54 shows a modified version of
the HelloMessenger endpoint that uses these annotations.

Example 2-54   HelloMessenger using life-cycle annotations

```
package itso.hello;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import javax.jws.WebService;

@WebService
public class HelloMessenger {

   public String sayHello(String name) {
      return String.format("Hello %s", name);
   }

   @PostConstruct
   @WebMethod(exclude = true)
   public void init() {
      System.out.println("Initializing bean");
   }

   @PreDestroy
   @WebMethod(exclude = true)
   public void destroy() {
      System.out.println("Destroying bean");
   }
}
```

The method annotated with the PostConstruct annotation is invoked by the Web container after bean instantiation but before the container starts dispatching requests to the methods exposed as Web service operations of the bean.

The method annotated with the PreDestroy annotation is invoked to notify the bean of its intent to remove it. The method is not called during request processing, and the endpoint can safely assume that the container will not dispatch further Web service requests after the method is invoked.

In this example, the bean writes a string to the standard output stream. In a real program, you might use these methods to initialize and destroy resources that are used by the bean, for example, data sources, JMS resources, and so on.

EJB container programming model

The EJB container programming model supports both SEI-based endpoints and provider-based endpoints. To change the HelloMessenger endpoint into an EJB-based Web service and deploy it to WebSphere Application Server V7, the following steps are required:

1. Place the bean class into a Java EE 5-compliant EJB Jar module.

2. Turn the endpoint into a stateless session EJB:

   a. Add the javax.ejb.Stateless annotation to the bean class.

   b. Create an interface, annotate it with javax.ejb.Local, and add the sayHello method signature.

   c. Change the bean class so that it implements the interface.

3. Assemble the EJB Java archive (JAR) module into a Java EE 5-compliant enterprise archive (EAR) module and generate a Web router module for it.

4. Deploy the application.

Note that the concept of Web router modules is not defined by WSEE. In WebSphere Application Server V7, Web router modules are used to handle the SOAP/HTTP protocol specifics. WebSphere Application Server V7 also supports Java Message Service (JMS)-based router modules so that you can expose an EJB-based endpoint over the SOAP/JMS protocol. (Notice that SOAP/JMS is not WS-I compliant.) Tools such as Rational Application Developer V7.5 and the WebSphere Application Server V7 `endptEnabler` command-line tool can automatically generate the router module.

Figure 2-3 shows an EJB JAR module made with Rational Application Developer V7.5 that contains the HelloMessenger Web service endpoint.

Figure 2-3   HelloWseeEJB EJB JAR module

This HelloWseeEJB project is a regular EJB 3.0 project. The HelloMessenger endpoint has been renamed to *HelloMessengerBean* to imply that it is also a session bean. The HelloMessenger file is the interface that defines the local EJB view. The META-INF folder contains an IBM-specific binding file that indicates to WebSphere Application Server which Web archive is the router module. Example 2-55 shows the EJB 3.0 interface.

Example 2-55   HelloMessenger EJB 3.0 local view

```
package itso.hello;

import javax.ejb.Local;

@Local
public interface HelloMessenger {
    public String sayHello(String name);
}
```

The javax.ejb.Local annotation indicates that the EJB must be exposed with a local view only. Notice that there is no Web service information here. Example 2-56 shows the EJB 3.0-based Web service endpoint.

Example 2-56   HelloMessengerBean EJB 3.0 Web service endpoint

```
package itso.hello;

import javax.ejb.Stateless;
import javax.jws.WebService;
```

```
@WebService
@Stateless
public class HelloMessengerBean implements HelloMessenger {

    public String sayHello(String name) {
        return String.format("Hello %s", name);
    }
}
```

The only difference between this endpoint and the original HelloMessenger
endpoint is that it carries the javax.ejb.Stateless annotation and implements the
local EJB 3.0 view interface (changes are highlighted in bold). The stateless
annotation indicates that the bean is to be exposed as a stateless session bean.

Although the code might look the same, remember that the EJB container gives
you free services for immediate use. The HelloMessengerBean, for example, is
threadsafe. In addition, all business logic that occurs *within* the sayHello method
is part of a container-managed transaction. However, this Web service does *not*
participate in any global transaction that is established by the Web service client.

Note: The WSEE specification mandates that only stateless session EJBs are
eligible as EJB-based Web service endpoints.

2.4.3 Client programming model

The WSEE specification describes the model for implementing Web service
clients hosted in a Java EE environment. A Java EE Web service client can be a
Java EE application client, Web component (for instance a servlet), EJB
component, or another Web service. The client uses the WSEE run time to
access and invoke Web service methods.

With a typical JAX-WS dynamic proxy-based client, the run time can inject a
service object or a port object into a member variable annotated with
javax.xml.ws.WebServiceRef and use that variable to exchange messages with
the Web service. Alternatively, the client can look up the service object or port
object in the JNDI namespace. After the client obtains a reference to the client
proxy, it uses that reference as it might with any other Java client.

> **More information**: The "Developing deployment descriptors for a JAX-WS client" topic in the WebSphere Application Server 7 Information Center, at the following address, describes how managed clients can configure and access the JNDI to use Web services:
>
> `http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/topic/com.ibm.`
> `websphere.nd.multiplatform.doc/info/ae/ae/twbs_jaxwsclientdd.html`

Figure 2-4 shows an ordinary dynamic Web project made with Rational Application Developer V7.5. The project contains a servlet client that uses the HelloMessenger Web service.
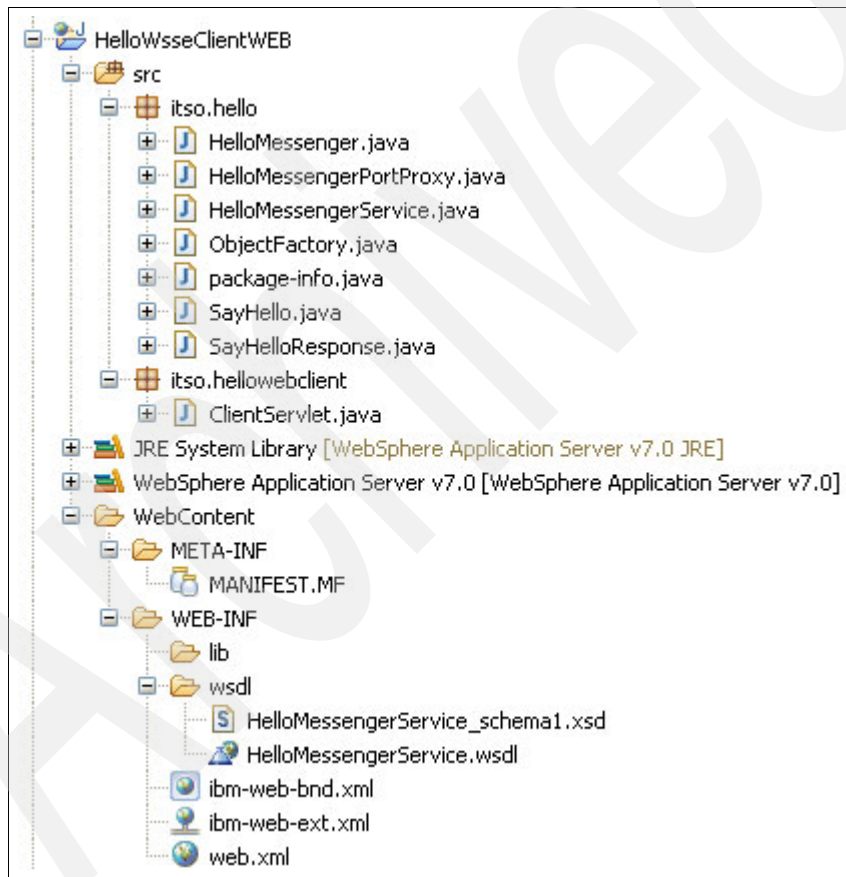
Figure 2-4   HelloWsseClientWEB project with Servlet Web service client

The src Java source code folder contains two packages:

- ► itso.hello
- ► itso.hellowebclient

The first package contains the generated JAX-WS classes. The second package contains the client servlet component, ClientServlet, which uses the generated client classes to communicate with the HelloMessenger Web service.

Under the WEB-INF folder is the usual web.xml deployment descriptor together with the IBM-specific binding file and extension file. In addition, the client was generated so that it has a local copy of the WSDL document that describes the Web service contract. WSEE specifies that WSDL documents by convention should be in WEB-INF/wsdl, but this is not required. Example 2-57 shows the ClientServlet class.

Example 2-57   HelloClientServlet

```
package itso.hellowebclient;

import itso.hello.HelloMessenger;
import itso.hello.HelloMessengerService;

import java.io.IOException;

import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.xml.ws.WebServiceRef;

public class ClientServlet extends HttpServlet {

    @WebServiceRef(HelloMessengerService.class)
    private HelloMessenger port;

    protected void doGet(HttpServletRequest req, HttpServletResponse
resp)
        throws IOException {

      String result = port.sayHello("Milo");

      String html = String.format(
            "<html><body>Message was: %s</body></html>", result);
```

```
        resp.getWriter().write(html);
    }

}
```

The ClientServlet class is an ordinary HTTP servlet that calls the
HelloMessenger endpoint's sayHello method and prints the result as a simplified
HTML string.

Notice the WebServiceRef declaration, which points, using the argument, to the
generated HelloMessengerService class. The HelloMessengerService class is
annotated with the javax.xml.ws.WebServiceClient annotation, which has a
wsdlLocation attribute. This attribute points to the actual WSDL document rooted
at `WEB-INF/wsdl/HelloMessengerService.wsdl`. By consulting this WSDL
document, the run time can find the endpoint address and the information
necessary to generate the dynamic proxy that implements the HelloMessenger
SEI interface.

In the previous example, the WebServiceRef annotation was used to inject a
dynamic proxy into the servlet. The annotation can also be used to inject the
generated Service class instead:

`@WebServiceRef private HelloMessengerService service;`

From the doGet method, the servlet then uses the service variable to obtain a
dynamic proxy object. See the JavaDoc for more information about the
WebServiceRef annotation.

Restrictions on the use of the asynchronous callback model: All JAX-WS client models are allowed by the WSSE specification. However, WSEE defines a few requirements for usage of the asynchronous callback approach. The requirements are specific to the client container in use (EJB container or Web container).

For the EJB container:

► An EJB instance cannot be passed as a callback handler instance. Place the handler in a separate class from the EJB bean class.

► The developer must not attempt to access the EJBContext from the handler. The behavior is undefined if it is accessed from the handler.

► The developer must not attempt to access the EJB bean instance from the handler. The behavior is undefined if it is accessed from the handler.

For the Web container:

► A servlet instance cannot be passed as a callback handler instance. Place the handler in a separate class from the servlet instance class.

► The developer must not attempt to access the servlet instance from the handler. The behavior is undefined if it is accessed from the handler.

► Developers must not cache the HttpSession and HttpRequest objects from the servlet in the handler.

Handlers

Web service clients in a Java EE 5-compliant container, such as WebSphere Application Server V7, can use the javax.jws.HandlerChain annotation in combination with the WebServiceRef annotation.

Example 2-58 shows the ClientServlet that is modified to apply a client-side handler chain to any message exchange with the HelloMessenger Web service.

Example 2-58   ClientServlet with HandlerChain annotation

```
package itso.hellowebclient;

import itso.hello.HelloMessenger;
import itso.hello.HelloMessengerService;

import java.io.IOException;

import javax.jws.HandlerChain;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
```

```
import javax.servlet.http.HttpServletResponse;
import javax.xml.ws.WebServiceRef;

public class ClientServlet extends HttpServlet {

    @WebServiceRef(HelloMessengerService.class)
    @HandlerChain(file="handler-chain.xml")
    private HelloMessenger port;

    protected void doGet(HttpServletRequest req, HttpServletResponse
resp)
            throws IOException {

        String result = port.sayHello("Milo");

        String html = String.format(
                "<html><body>Message was: %s</body></html>", result);

        resp.getWriter().write(html);
    }

}
```

Usage of the HandlerChain annotation is highlighted in bold. This HandlerChain
annotation is a convenient way to use client-side handlers with Java EE 5 Web
service clients. Compare this to the approach used in an unmanaged
environment. In an unmanaged environment, you must add the handler
programmatically or manually edit the generated service class to add the
HandlerChain annotation. For a description of the unmanaged approach, see
2.1.5, "Handlers" on page 104.

> **Handler configuration in Java EE module deployment descriptors**: An
> alternative to using the HandlerChain annotation in Java EE-managed clients
> is to declare handlers in the appropriate module's deployment descriptor
> (ejb-jar.xml, web.xml, and so on). When using this method, client-side
> handlers are added to the <service-ref> element by using the <handler> child
> elements.

Part 2

Developing and deploying Web services

145

3

The WeatherForecast
sample application

In this chapter we describe the base Java code of the WeatherForecast
application that is used to demonstrate Web services technology. The
WeatherForecast application is similar to the one that is used in *Web Services
Handbook for WebSphere Application Server 6.1*, SG24-7257. However, we
modified the application in this book to reflect some of the programming
language features of Java 5.

This chapter contains the following topics:

3.1 The WeatherForecast application components

The WeatherForecast application is used throughout this book to demonstrate how to create and use Web services. The WeatherForecast application simulates weather forecast predictions.

> **Downloadable materials**: The sample application used in this chapter is available in the downloadable materials for this book. For information about downloading the files, see Appendix A, "Additional material" on page 537.
>
> The Chapter3 folder contains the following compressed (.zip) files:
>
> ► The ch03_sample_app.zip file contains the ITSO package files that are required to create the WeatherForecast application. If you follow the instructions in this chapter, extract these files into a temporary directory for use in the example.
>
> ► The ch03_PIF_testapp.zip file contains a project interchange file with the completed application. The file can be imported into Rational Application Developer. If you want to simply reference the application discussed in this chapter rather than build it, import this file.
>
> "Importing project interchange files" on page 542 provides the steps to import a project interchange file
>
> The examples assume that a Derby database is available for use. For information about creating this database, see "Set up the WEATHER database (Derby)" on page 540.

3.1.1 The WeatherForecast application packages

The WeatherForecast application consists of four packages:

► itso.businessobjects
► itso.objects
► itso.dao
► itso.utils

> **Attention:** In later chapters you will see these classes packaged as the WeatherBase utility project for inclusion in sample applications.

Figure 3-1 shows the class diagram of the WeatherForecast application.
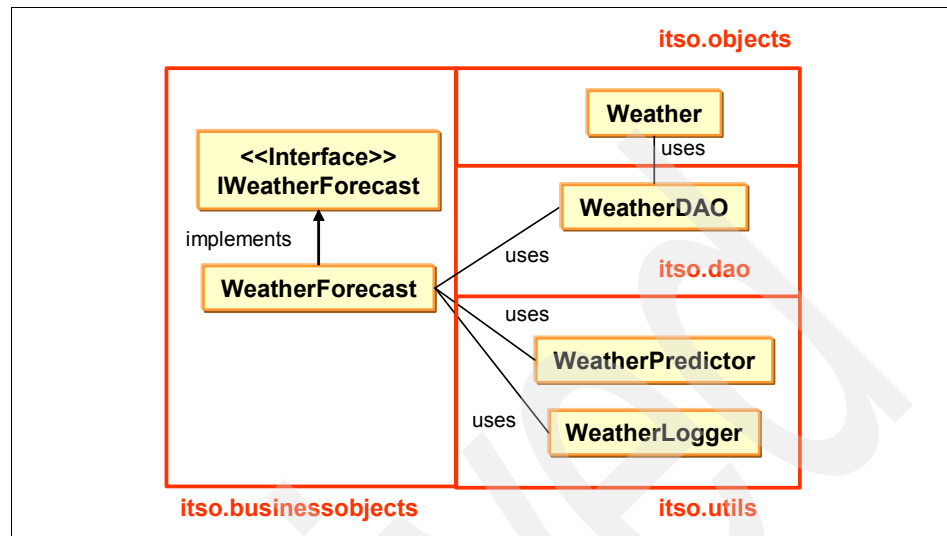
Figure 3-1   WeatherForecast application class diagram

The itso.businessobjects package

The itso.businessobjects package contains the implementations of the
WeatherForecast application that we turn into Web services:

- ▶ The *IWeatherForecast* interface is the service's abstraction.
- ▶ The *WeatherForecast* class provides the service's implementation.

The functionality offered by the WeatherForecast application is described by the
IWeatherForecast interface with the method signatures shown in Table 3-1.

Table 3-1   WeatherForecast application interface

Method summary	
`Weather getDayForecast(java.util.Calendar)`	Get weather information for a specific day.
`List<Weather> getForecast(java.util.Calendar, int)`	Get the forecast for a specific period of time.
`int[] getTemperatures(java.util.Calendar, int)`	Get temperatures for a specific period of time.
`void setWeather(Weather)`	Set a forecast for a specific day.

> **getForecast() method generics**: Observe that the getForecast() method uses generics, which is a new feature in the Java 5 programming language.

The itso.objects package

The itso.objects package contains one class, `Weather`, that represents the WeatherForecast application's encapsulated data (with corresponding getter and setter methods) in a Java bean:

- ► A String variable, *windDirection*, which represents the compass point from which the wind is blowing for a particular date

- ► An int variable, *windSpeed*, which represents the wind's speed in Km/h for a particular date

- ► An int variable, *temperatureCelcius*, which represents the temperature in degrees Celcius for a particular date

- ► A string variable, *condition*, which represents the general weather condition (sunny, partly cloudy, cloudy, rainy, stormy) for a particular date

- ► A calendar variable, *date*, which represents the date of the weather forecast

- ► An boolean variable, *dbflag*, which signifies whether the forecast was extracted from the database

The itso.dao package

The itso.dao package contains one class, WeatherDAO, which consists of all the functionality to store and retrieve weather information from the database.

The itso.utils package

The itso.utils package contains utility classes that are used by the WeatherForecast application:

- ► The WeatherLogger class sends a log message into the system console.

- ► The WeatherPredictor class randomly creates a weather forecast for the current date.

3.1.2 Information flow

Figure 3-2 shows the internal flow of the system information for the query methods.
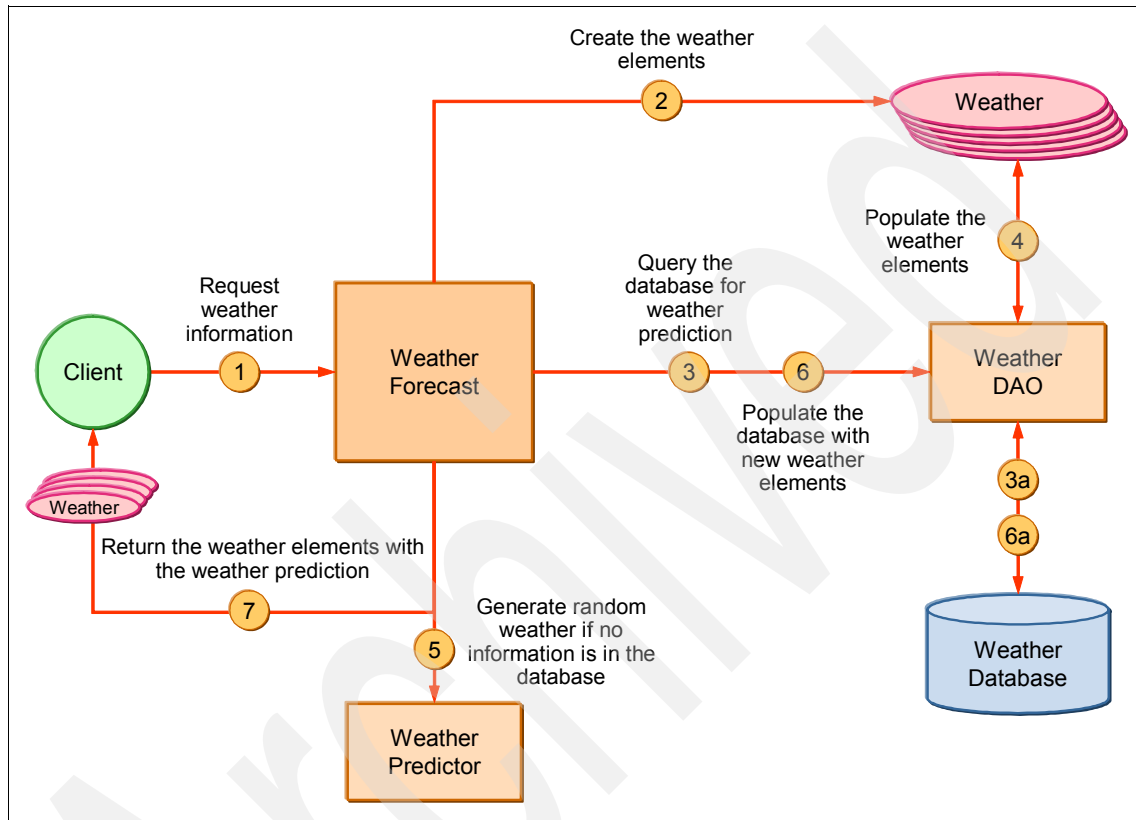
Figure 3-2   Weather forecast information flow

The internal flow follows this sequence of actions:

1. A client requests weather information from the WeatherForecast class.

2. The WeatherForecast class creates a Weather element (or elements) for the response to the client's weather request.

3. The WeatherForecast class queries the weather prediction from the weather database by using WeatherDAO.

4. The WeatherDAO class populates the weather element (or elements) based on the information present at that moment in the database.

5. The weather information that is not in the database is requested from WeatherPredictor.

6. The database is populated by the queries with the new weather element (or elements) generated by WeatherPredictor.

7. The WeatherForecast class returns the weather element (or elements) to the client.

> **WeatherPredictor class**: The WeatherPredictor class uses a random number algorithm to populate weather information, which makes our example simple. However, it enables us to concentrate on the important Web services aspects instead of trying to write a sophisticated back-end application.

Figure 3-3 shows the internal flow of the system information for the load method.
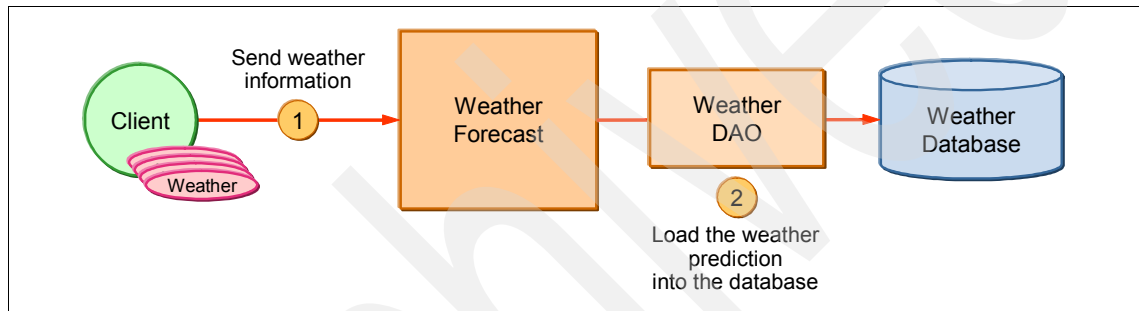
Figure 3-3   Weather forecast load flow

The load flow entails the following actions:

1. A client sends weather information to the WeatherForecast bean to load the database.

2. The WeatherForecast bean populates the database with the Weather element by using the WeatherDAO class.

3.2 The weather database

The WeatherForecast application uses a simple database to store data for weather forecasts. The database is implemented by using the default Derby database that is embedded in WebSphere Application Server.

Database implementation

The weather database contains one table, ITSO.SANJOSE, which has five columns:

WEATHERDATE DATE	Date of weather prediction, primary key
CONDITION VARCHAR (20)	Condition: sunny, partly cloudy, cloudy, rainy, stormy
WINDDIR VARCHAR (20)	Wind direction: N, NE, E, SE, S, SW, W, NW
WINDSPEED INTEGER	Wind speed (kilometers/hour)
TEMPERATURE INTEGER	Temperature (degree Celsius)

Data source

The WeatherDAO class uses a data source to connect to the weather database. All the modules that use the WeatherForecast packages must define a data source in the enhanced EAR or within the system resources of WebSphere Application Server. The WeatherDAO class uses the JNDI name `jdbc/weather` to look up the data source.

3.3 Testing the WeatherForecast application

In this section we create and test the WeatherForecast application by using a Java Platform Enterprise Edition (Java EE) 5 enterprise application client that was developed with the Rational Application Developer V7.5 integrated development environment.

The WeatherForecast enterprise application client

In the following sections we create an application client that runs the code for the WeatherForecast sample application.

Creating an application client project

To create an application client project in Rational Application Developer:

1. Switch to the **Java EE** perspective.

2. From the main menu, select **File** → **New** → **Application Client Project**.

3. In the New Application Client Project window:

    a. For the project name, type `WeatherForecast`.

    b. Under EAR Membership, ensure that the **Add project to an EAR** check box is selected.

    c. Click **New**.

d. In the New EAR Application Project window, for the project name, type `WeatherForecastEAR` and click **Finish**.

e. Back in the New Application Project window, click **Finish**.

Creating a dynamic Web project

Next create an empty dynamic Web project and include it in the enterprise application (EAR) file that you just created. By doing this, you can deploy the enterprise application into the WebSphere Application Server V7.0 test environment.

To create the dynamic Web project and include it in the EAR file:

1. Switch to the **Java EE** perspective if it is not already open.

2. From the main menu, select **File → New → Dynamic Web Project**.

3. In the New Dynamic Web Project window, enter the following:

   a. For Project name, type `WeatherForecastDummyWeb`.

   b. For EAR Project Name, type `WeatherForecastEAR`.

   > **Note:** Explicitly specify the enterprise application project that was made for the application client project. If other enterprise application projects are in the workspace, this might not be the default.

   c. Click **Finish**.

4. If asked to switch perspectives, click **No**.

Importing the application packages

The WeatherForecast application packages must be imported in the application client project. To import the WeatherForecast application packages:

1. Extract the `ch03_sample_app.zip` file from the download material into a temporary directory.

2. Right-click **appClientModule** on the application client project (WeatherForecast) and select **Import**.

3. Select **General → File System**. Click **Next**.

4. Browse into the directory where the download materials have been extracted. Select the **itso** folder and its contents and import them.

5. Click **Finish**.

Implementing the business logic

To implement the business logic for the WeatherForecast program:

1. Select **appClientModule** → **(default package)**.

2. Double-click **Main.java**.

3. Write the client implementation in the main(String[] args) method of the Java class.

   Example 3-1 shows sample code for the main Java class. You can find this code in the `WeatherForecast_TestApp_snippet.txt` file in the downloadable materials that you extracted.

   While you can expect errors because of unresolved type declarations, you can ignore them.

   Example 3-1   The main() method of Main.java

   ```
   try {
       IWeatherForecast port = new WeatherForecast();

       Calendar calendar = Calendar.getInstance();
       calendar.set(Calendar.YEAR, 2006);
       calendar.set(Calendar.MONTH, Calendar.JANUARY);
       calendar.set(Calendar.DAY_OF_MONTH, 7);

       Weather weather = port.getDayForecast(calendar);
       System.out.println(weather.toString());
   } catch(Exception ex) {
       ex.printStackTrace();
   }
   ```

4. Organize the imports by pressing Ctrl+Shift+O.

5. Save the file. All errors should be resolved.

Figure 3-4 shows the results.
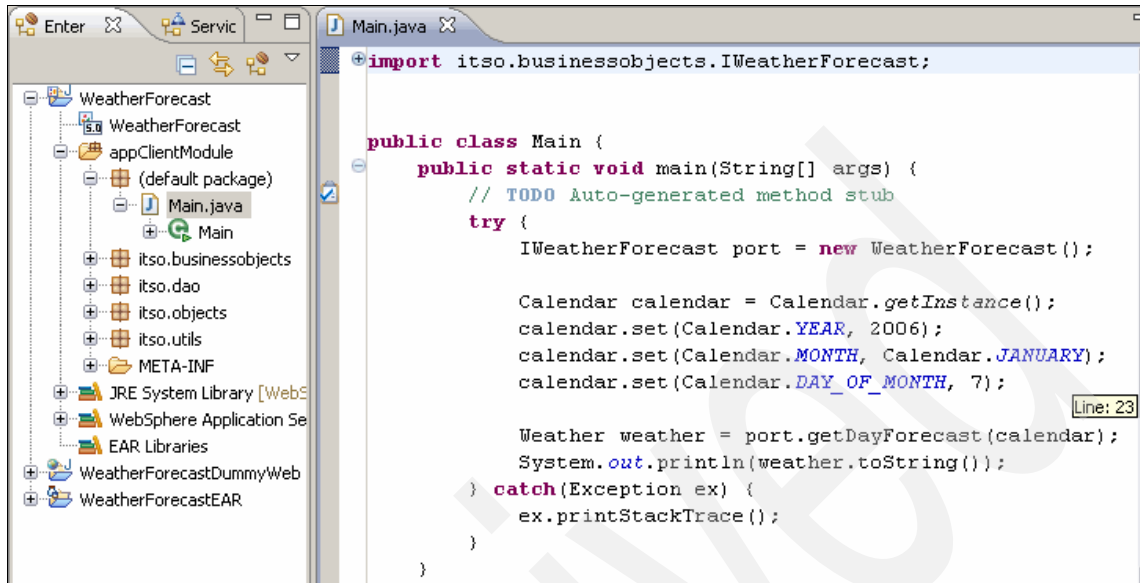
Figure 3-4   Main.java and itso package files

Configuring a JDBC data source in the EAR file

To configure a JDBC™ data source in the enhanced EAR file:

1. Right-click the enterprise application project (**WeatherForecastEAR**) and select **Java EE → Open WebSphere Application Server Deployment**.

2. In the enhanced EAR deployment descriptor editor that opens in the main area, under the Data source defined in the JDBC provider selected above section, click the **Add** button.

3. In the Create Data Source window, select **Version 5.0 data source** and **Derby JDBC Provider (XA)**. Click **Next**.

4. In the next window for the data source:

   a. For name, type `Weather WS Data Source`.
   b. For JNDI name, type `jdbc/weather`.
   c. Click **Next**.

5. In the Create Resource Properties window:

   a. Select the **databaseName** property.

   b. In the Value field, type the full path of the database directory. The path delimiter for the enhanced EAR is a forward slash (/). The path to the WEATHER directory is `/Database/WEATHER`.

     c. Click **Finish**.

6. Save the enhanced EAR.

Deploying and testing the application

Start the test environment and deploy the EAR file of the application client project into it:

1. In the Servers view, right-click the **WebSphere Application Server v7.0** instance and select **Start**. Wait until the server has achieved a *started* state.

2. Right-click the server instance again and select **Add and Remove Projects**.

3. In the Add and Remove Projects window, select the enterprise application that contains the application client project from the left pane. In this case, we select **WeatherForecastEAR**. Click the **Add** button.

4. Click **Finish**.

5. Go back to the Servers view and see whether the server has achieved a `Synchronized` status.

To run the application client project:

1. Expand **appClientModule** → **(default package)**.

2. Right-click **Main.java** and select **Run As** → **Run Configurations**.

3. In the left pane of the Run Configurations window (Figure 3-5), select **WebSphere Application Server v7.0 Application Client**. In the top row of icons, select **New launch configuration**.
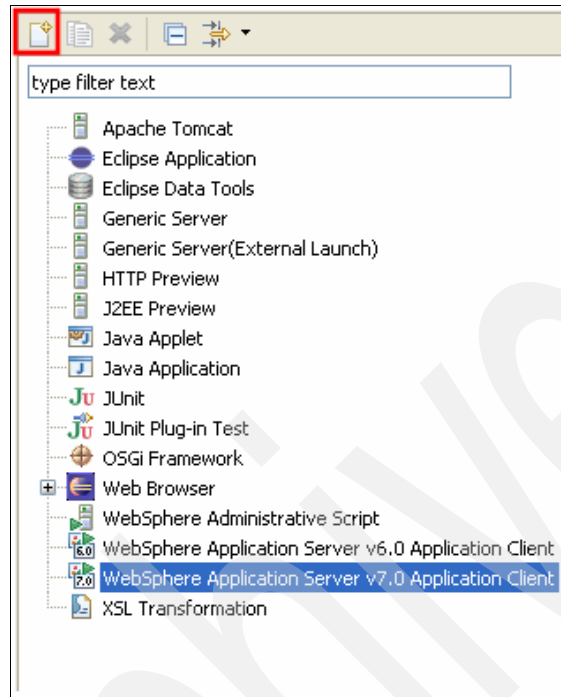
Figure 3-5   Launch configurations

4. In the right pane of the Run Configurations window:

   a. In the Name field, enter a new launch configuration name. In this example, we type WeatherForecast.

b. On the Application tab (Figure 3-6):

   i.   Select the EAR and application client module.

   ii. Ensure that the provider URL points to the bootstrap port of your test server.

   iii. Click **Apply**.
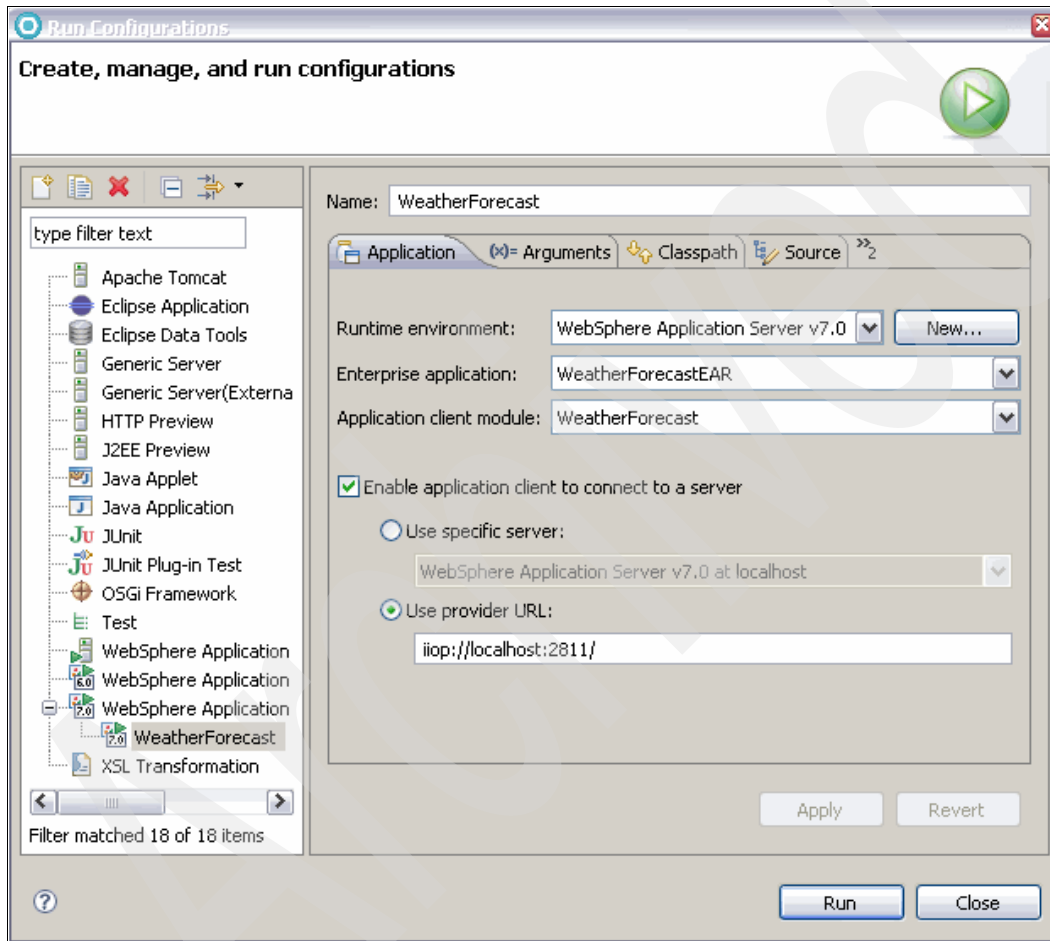
Figure 3-6   Application tab for launching the configuration

c. Click the **Classpath** tab (Figure 3-7), then:

   i. Select **User Entries**.

   ii. Click the **Add External JARs** button.
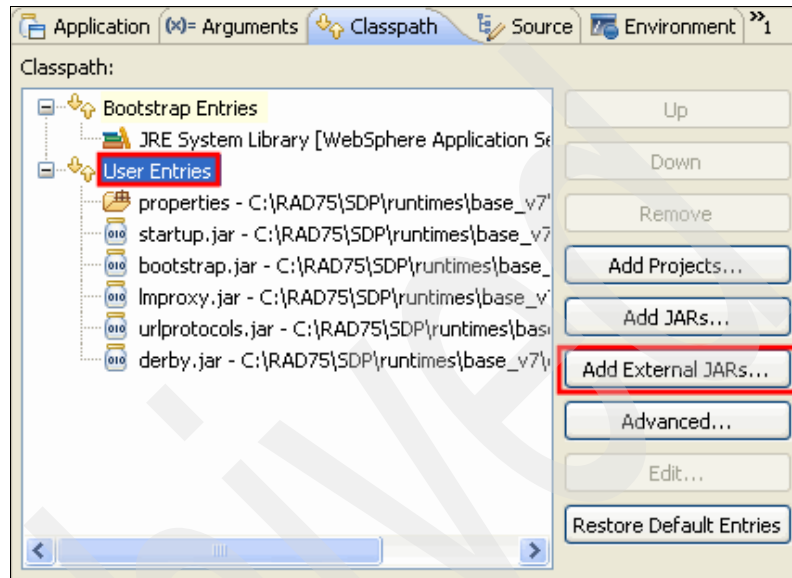
Figure 3-7   Adding the derby.jar file to the classpath

   iii. Browse to the `rad_root`/SDP/runtimes/base_v7/derby/lib/derby.jar
      directory and select the **derby.jar** file to add it to the classpath of the
      application client project. Click **Open**.

   iv. Click **Apply**.

d. Click **Run**. The results are displayed in the Console view (Figure 3-8)
   based on the data that is retrieved from the weather database.
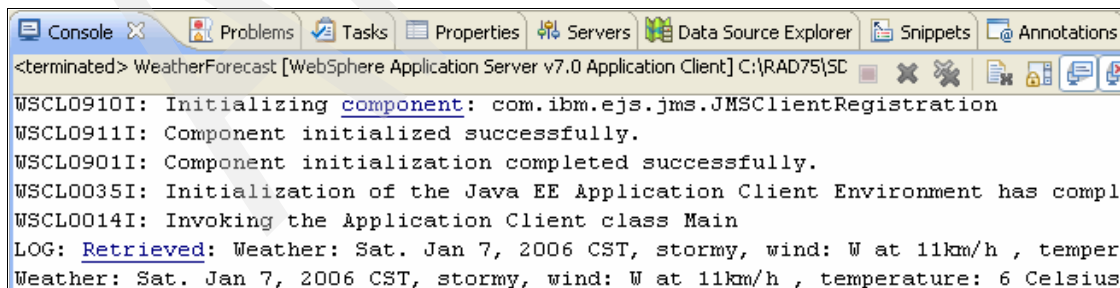
Figure 3-8   WeatherForecast application results

4

Developing Web services applications

In this chapter we explain how to develop Web services applications by using specific techniques that are common to most application development tools. We use these techniques to create Web services that will be deployed to the WebSphere Application Server V7.0 run time. To help illustrate the development process, we use the WeatherForecast sample application as an example.

This chapter contains the following topics:

4.1 Web services development environment

Web services rely heavily on standards and specifications. Therefore, a suitable development environment is essential for creating the Web-services-enabled applications and the Web services themselves.

4.1.1 Web services development tools

An effective development environment is made up of tools and utilities that aid in the programming tasks of the application developer. At a minimum, the development environment should make the appropriate API available to the developer.

Enterprise platforms for Java have included Web services as part of their API offering since Java 2 Platform, Enterprise Edition (J2EE) V1.4. The latest iteration of this standard, Java Platform, Enterprise Edition (Java EE) V5, continues to support Web services with its current version. This means that any development environment for Java EE includes support for the creation of Web services by using the Java API for XML Web services (JAX-WS) and the Java Specification Request (JSR) 109 V1.2 specifications.

Java Standard Edition V6

Java Standard Edition (Java SE) V6 is the specification for the core runtime environment for Java-based platforms including Java EE V5. Java SE V6 includes the JAX-WS API and the default implementations of the command-line tools that are used in Web services development, such as `wsimport` and `wsgen`. Java SE V6 also includes the classes for creating *endpoint publishers* that allow a Web service to run on a standalone Java Runtime Environment (JRE) without being deployed to an application server.

WebSphere Application Server V7.0

WebSphere Application Server V7.0 is the latest version of the IBM enterprise platform that is updated to the Java EE 5 standard, which includes the new Web services specifications. The JAX-WS API and class definitions are available through this runtime environment. Aside from this, WebSphere Application Server also provides several command-line tools that are used in the development of Web services. Such tools include `wsimport`, `wsgen`, `schemagen`, and `xjc`.

> **Web services command-line tools**: WebSphere Application Server V7.0 has
> its own implementation of the same Web services command-line tools that the
> Java SE 6 development kit provides. The artifacts that are generated by the
> Java SE 6 version of these tools are generally portable across different
> runtime environments. However, for seamless integration with the WebSphere
> Application Server platform, use the tools that are provided with WebSphere
> Application Server.

Ant

Another Neat Tool or Ant has been a long-time favorite among enterprise Java
programmers. It uses an XML file to manage an automated build process for
particularly complicated assembly tasks, such as those used in Java EE and
Web services. WebSphere Application Server has its own version of this utility
called `ws_ant` with additional functionalities such as module deployment, starting
and stopping of the server, and execution of administration scripts.

4.1.2 Integrated development environments and Web services

An integrated development environment (IDE) simplifies programming tasks for
application developers. An IDE plays an important role in Web services
development because of the complex XML constructs that are used by these
kinds of applications. Programming Web services is error prone and inefficient to
do manually. Using an IDE increases a developer's productivity.

Rational Application Developer

For WebSphere Application Server V7.0, the preferred IDE for application
development is Rational Application Developer. Rational Application Developer is
built on the Eclipse software platform and features a wide array of tools that are
used for developing, assembling, and deploying applications in Java EE and Web
services.

Two variants of Rational Application Developer are available for use with
WebSphere Application Server V7.0:

► Rational Application Developer V7.5
► Rational Application Developer Assembly and Deploy

Rational Application Developer V7.5 is a full-featured IDE with a complete set of
application development tools for the Java EE environment. Rational Application
Developer Assembly and Deploy is the Java EE application assembly tool that
comes with WebSphere Application Server V7.0 and features only a minimal set
of tools.

Table 4-1 summarizes the fundamental differences between these variants.

Table 4-1   WebSphere Application Server V7.0 IDEs

	Rational Application Developer V7.5	**Rational Application Developer Assembly and Deploy**
Primary purpose	Full-featured Java EE IDE	Java EE application assembly tool for WebSphere Application Server V7.0
Development tools	Complete Java EE development toolset	Minimal Java EE development toolset
Web services development tools	JAX-WS and Java API for XML-based remote procedure call (JAX-RPC)	JAX-RPC only
Test environment	Embedded WebSphere Application Server V7.0 test environment	External WebSphere Application Server V7.0 installation

Development and test environment for the sample application: All examples in this chapter use the Rational Application Developer Assembly and Deploy tool that comes with WebSphere Application Server V7.0. (You must have installed *both* prior to performing the examples.) By using this tool, we demonstrate how most Web service IDEs operate internally. The only exception is for the examples in 4.4, "EJB Web services" on page 197, which use Rational Application Developer V7.5.

In practice, perform Web services development by using a full-featured IDE such as Rational Application Developer V7.5.

4.1.3 Setup for the Web services development examples

The `Chapter4/ch04_app_dev.zip` file contains a `Files` folder that has the artifacts that we use in the development process outlined in this chapter. To follow the instructions in this chapter, extract these files to a temporary directory for use in the example.

The `ch04_app_dev.zip` archive includes the following files:

► The `itso` folder contains the source files for the WeatherForecast sample application in the Java package directory structure that is used for the *bottom-up* Web services development example.

► The `jms_setup.py` file contains the administrative script that is used to create Java Message Services (JMS) resources for the Enterprise JavaBeans (EJB) Web service example.

► The `WeatherForecast_java_snippet.txt` file contains the code that is used to implement the business logic for the *top-down* Web services development example.

► The `WeatherForecast_JMS_snippet.txt` file contains the code that is used to implement the business logic for the EJB Web service example.

► The `WeatherForecastService.wsdl` and `WeatherForecastService_schema1.xsd` files are the Web Services Description Language (WSDL) and schema files that are used as the service definition for the *top-down* Web services development example.

► The `WeatherWSTest_java_snippet.txt` file contains the code that is used to implement the business logic for the thin client example.

- ► The `WeatherWSTest_JMS_snippet.txt` file contains the code that is used to implement the business logic for the asynchronous client of the EJB Web service example.
- ► The `WeatherWSTest_jsp_snippet.txt` file contains the code that is used to implement the business logic for the JSP™-managed client example.

4.2 Server-side Web services development

A distributed system programming model separates an application into server-side and client-side components. The server and client components are independent of one another and can be developed by different teams of programmers.

Web services development follows this distributed programming paradigm. The examples in this section show strategies in server-side Web services development that are supported by the JAX-WS programming model.

4.2.1 Web services development from a WSDL file

Web services development can be initiated from the information from a service definition. This information is stored in a file by using the WSDL XML format. The WSDL file is parsed, and the information is used to construct skeleton Java code that developers can use to write the business logic in. Making Web services in this manner is known as *top-down development*. Development of Web services in this way involves the following tasks:

1. Acquiring or creating the WSDL file
2. Generating the skeleton code (from a command line)
3. Implementing the Web service
4. Configuring the enhanced EAR file
5. Deploying the Web service
6. Testing the Web service (from an external browser)

We explain each of these tasks in the sections the follow.

Acquiring or creating the WSDL file

In practice, you can create the WSDL file by using an XML tool or you can use an existing WSDL file. In this example we use an existing WSDL file to develop the Web service.

> **Note:** In this example, Rational Application Developer Assembly and Deploy is used to merely *view* the contents of the WSDL file using its WSDL editor. If you choose to forego this step, proceed to "Generating the skeleton code (from a command line)" on page 171.

To create a generic project to hold the WSDL for viewing:

1. In the workspace, switch to the **Java** perspective.

2. Select **File** → **New** → **Project**.

3. Expand **General** and select **Project**. Click **Next**.

4. In the Project name field, type `WeatherForecastProject` and click **Finish**. A new generic project is created and is displayed on the workspace.

5. Right-click the project in the workspace and select **New** → **Folder**.

6. In the New Folder window, for folder name, type `wsdl`, then click **Finish**. A new folder is created under the `WeatherForecastProject`.

Import the existing WSDL file into the Rational Application Developer Assembly and Deploy workspace:

1. Under WeatherForecastProject, right-click the **wsdl** folder and select **Import**.

2. In the Import window, expand **General** and select **File System**. Click **Next**.

3. In the next window:

   a. Browse to the directory of the WSDL and schema files (included with the `ch04_app_dev.zip` archive).

   b. Select the `WeatherForecastService.wsdl` and `WeatherForecastService_schema1.xsd` files.

   c. Click **Finish**.

View the imported files by using the WSDL editor:

1. Double-click the **WeatherForecastService.wsdl** file.

2. In the WSDL editor that opens in the main area (Figure 4-1), in the upper right corner, click **View** and select **Advanced**. The advanced view mode of the WSDL editor shows the parts of the WSDL in native XML notation as opposed to a functional notation in the simplified view mode.
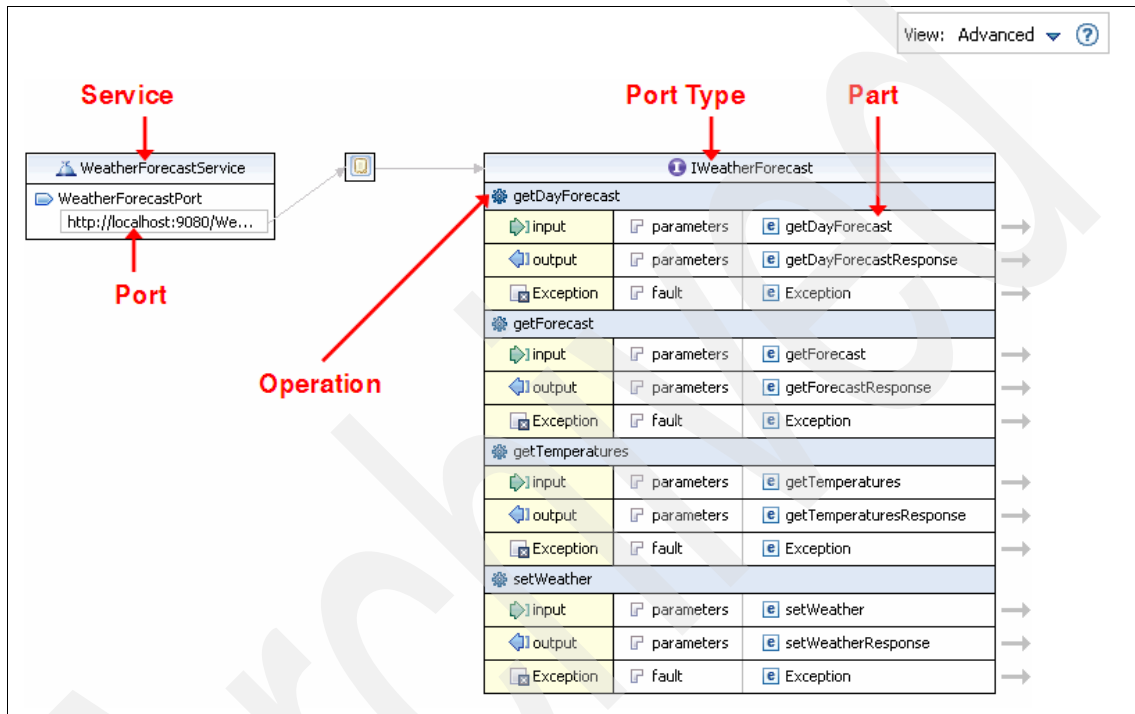
Figure 4-1   WSDL editor

Usually, when XML is involved in any application development undertaking, such as in Web services, it is preferable to use a tool to craft or modify the file. Using such a tool reduces the errors that are encountered in dealing with the usually complex XML structures such as WSDL files.

The WSDL editor tool is available in both Rational Application Developer Assembly and Deploy and in Rational Application Developer V7.5. You can use it to create a new WSDL file or to view or modify an existing one.

Click the arrow link to the right of the `getDayForecast` variable in the WSDL editor.

The schema editor opens and shows the mapping between the WSDL part with the data type in the XML Schema Definition (XSD) file (Figure 4-2). If the

style attribute has a value of `document` in the binding section of the WSDL, the WSDL data types are defined in a separate schema (XSD) document. This schema document is specified in an import element in the WSDL. The Web services tools access the XSD file when the WSDL is parsed. Therefore, always keep the WSDL and XSD *together*.
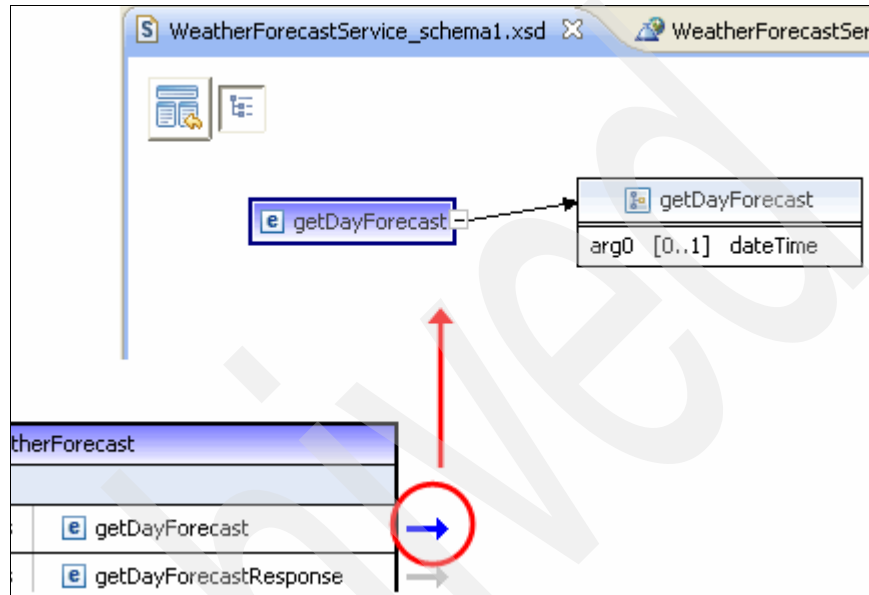
Figure 4-2   Schema editor

3. Return to the WSDL editor and click a graphic area on the WSDL. Click the **Properties** view (Figure 4-3) to see the values from the selected area. If the Properties view is not present, switch to the **Java EE** perspective.
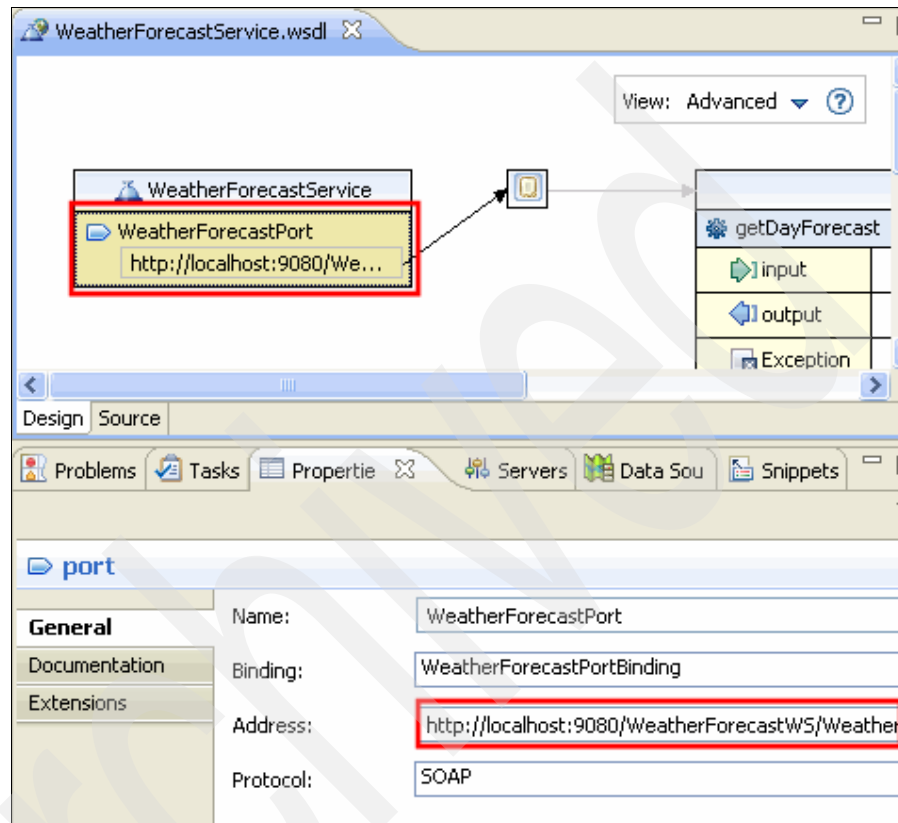
Figure 4-3   Properties view

Generating the skeleton code (from a command line)

For the top-down development strategy, **wsimport** is used to generate the skeleton code from which to build the Web service implementation. Figure 4-4 shows a graphic view of how **wsimport** fits into the process and the components it generates.
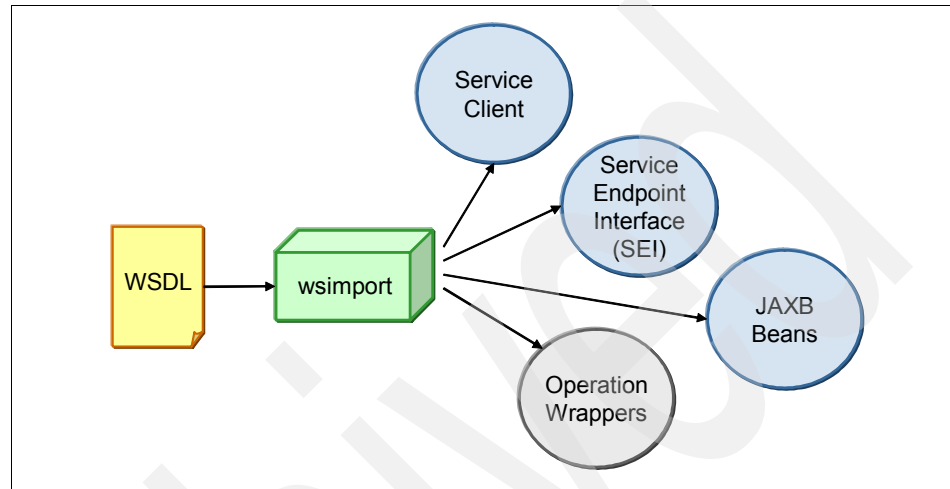
Figure 4-4   wsimport process

wsimport steps: In Rational Application Developer V7.5, the steps for using **wsimport** are incorporated into the Web service Wizard that automates most of the Web services development steps. For most of the examples in this chapter we perform the **wsimport** steps from the command line because the Web service wizard is not included in Rational Application Developer Assembly and Deploy. However, the examples in 4.4, "EJB Web services" on page 197, use of this tool because it uses Rational Application Developer V7.5.

To generate the skeleton code for the Web service by using the WSDL file:

1. Open a console window and navigate to *WAS_HOME*/bin.

2. Type the **setupCmdLine** command to set up the environment variable *WAS_PATH*.

3. On a command line, enter the following command to set the path:

   ```
   set PATH=%PATH%;%WAS_PATH%
   ```

4. Change to the directory of the WSDL (`WeatherForecastService.wsdl`) and schema (`WeatherForecastService_schema1.xsd`) files. These files are included with the `ch04_app_dev.zip` file.

5. On a command line, enter the **wsimport** command as follows on the WSDL file:

   `wsimport -verbose -keep WeatherForecastService.wsdl`

   The following options are used:

   – The verbose option provides a trace output of the operations being performed by the command

   – The keep option retains the `*.java` source files that are generated by the command.

The **wsimport** command generates the several artifacts in an `itso.businessobjects` package (Figure 4-5). These files (in the `itso` folder) are generated in the same location as the WSDL file.
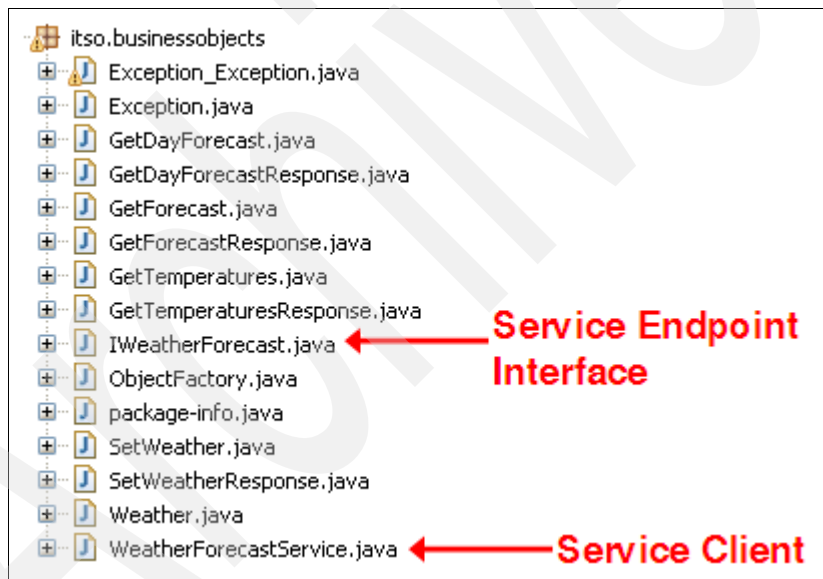
Figure 4-5   The wsimport generated artifacts

The following artifacts are generated:

► The service endpoint interface (SEI) provides the interface definition for the Web service implementation

► The service client is used in client programs.

- ► The request and response wrappers encapsulate the operations that are defined in the WSDL file. Each operation specified in the WSDL has a *operation_name*.java and *operation_name*Response.java file generated for it.

- ► The complex data types that are defined in the WSDL have representations of their own.

- ► Exceptions are specified for the faults defined for each operation defined in the WSDL.

- ► An *ObjectFactory* is used for Java API for XML Binding (JAXB) purposes.

- ► A *package-info* encapsulates the Java package format that is derived from the WSDL namespace.

> **itso package contents:** The contents of the itso package generated by **wsimport** are different from the those in the ch04_app_dev.zip file. The package name was derived by **wsimport** from the WSDL file.

Implementing the Web service

After the skeleton code is generated by using **wsimport**, you must write the business logic of the Web service to a Java class known as the service implementation bean (SIB).

To create a Dynamic Web Project for the Web service:

1. In Rational Application Developer Assembly and Deploy, switch to the **Java EE** perspective if it is not already open.

2. Select **File** → **New** → **Dynamic Web Project**.

3. In the New Dynamic Web Project window, for project name, type WeatherForecastWS, and for EAR project name, type WeatherForecastWSEAR. Click **Finish**.

4. If prompted to switch perspectives, click **No**.

Import the **wsimport** generated artifacts (in the itso folder) into the Dynamic Web Project:

1. Expand **Java Resources** on the Dynamic Web Project (WeatherForecastWS). Right-click **src** and select **Import**.

2. Expand **General** and select **File System**. Then click **Next**.

3. In the File system window (Figure 4-6 on page 174):

   a. Click **Browse** and go to the directory of the files created by **wsimport** (the itso package).

   b. Import the itso package created by **wsimport**.

c. Click the **Filter Types** button to import only the *.java source files.

d. Click **Finish**.
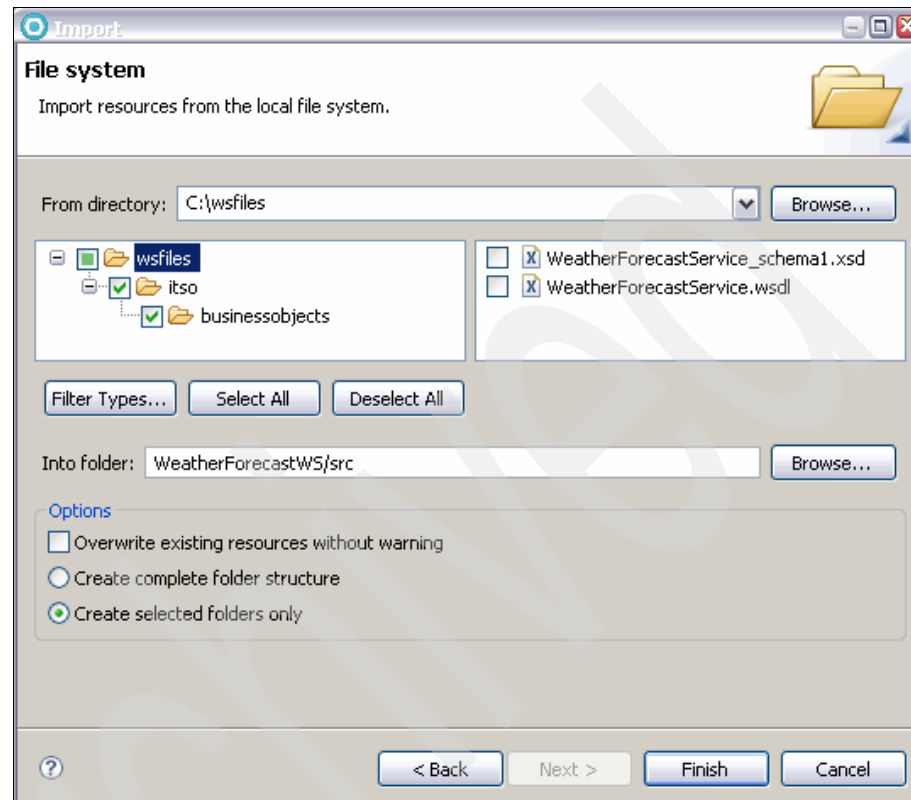
Figure 4-6   Importing the files that are generated by the wsimport tool

Create the SIB Java class that will implement the Web service business logic:

1. Right-click the imported package **itso.businessobjects** and select **New →
   Class**.

2. In the New Java Class wizard (Figure 4-7):

   a. In the Name field, type `WeatherForecast`.

   b. Next to the Interfaces section, click **Add** and select the **WeatherForecast**
      SEI class as the implemented interface.
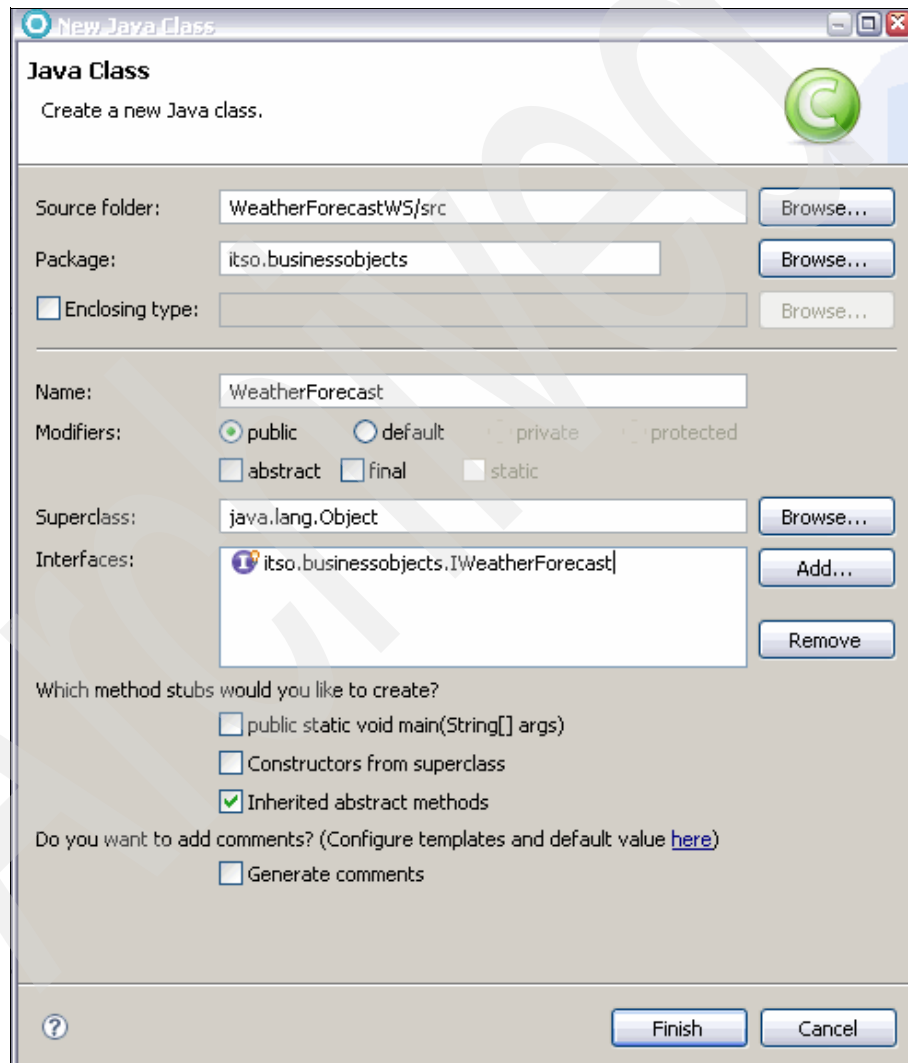
Figure 4-7   Creating a new Java class

c. Click **Finish**.

In JAX-WS, it is not necessary for the SIB Java class to implement the interface type of the SEI. Only the annotation attribute in the SIB source code must refer to the SEI by name. However, for Java style purposes, this example has the SIB Java class implement the SEI Java interface.

The Java editor opens in the main area of Rational Application Developer Assembly and Deploy.

3. In the SIB class's source code (WeatherForecast.java), add the @WebService annotation with the endpointInterface attribute set to the SEI class name, itso.businessobjects.IWeatherForecast. Place this before the class declaration (Figure 4-8).

```
@WebService(endpointInterface="itso.businessobjects.IWeatherForecast")
public class WeatherForecast implements IWeatherForecast {
```

Figure 4-8   SIB annotation

4. Organize imports by pressing Ctrl+Shift+O.

5. Write business logic into the getDayForecast() method of the SIB class (WeatherForecast.java).

> **The getDayForecast() method:** For this example, the getDayForecast() method is the only one required to be implemented. The following service methods of WeatherForecast.java are optional for implementation per the user's discretion.
>
> ► getForecast()
> ► getTemperatures()
> ► setWeather()

Example 4-1 shows the code for the getDayForecast() operation of the WeatherForecast SIB. You can also find this code in the WeatherForecast_java_snippet.txt file in the ch04_app_dev.zip archive file. This code fragment uses Java Database Connectivity (JDBC) to look up and return data from the weather database by using a given a date. Notice that JAX-WS tools use the XMLGregorianCalendar for XML representations of dateTime.

Example 4-1   The getDayForecast() method of WeatherForecast.java

```
public Weather getDayForecast(XMLGregorianCalendar arg0)
    throws Exception_Exception {
    Connection con = null;
```

```
            PreparedStatement pm = null;
            Weather result = null;

            try{
                InitialContext ic = new InitialContext();
                DataSource ds = (DataSource) ic.lookup("jdbc/weather");

                con = ds.getConnection();
                pm = con.prepareStatement(
                    "SELECT * FROM ITSO.SANJOSE WHERE WEATHERDATE = ?");
                Date sqlDate =
                    new
Date(arg0.toGregorianCalendar().getTime().getTime());
                pm.setDate(1, sqlDate);
                ResultSet rs = pm.executeQuery();
                while (rs.next()) {
                    GregorianCalendar cal = new GregorianCalendar();
                    cal.setTime(rs.getDate("WEATHERDATE"));

                    XMLGregorianCalendar xmlCal =
                        DatatypeFactory.newInstance().
                            newXMLGregorianCalendar(cal);

                    result = new Weather();
                    result.setDate(xmlCal);
                    result.setCondition(rs.getString("CONDITION"));
                    result.setTemperatureCelsius(rs.getInt("TEMPERATURE"));
                    result.setWindDirection(rs.getString("WINDDIR"));
                    result.setWindSpeed(rs.getInt("WINDSPEED"));
                    result.setDbflag(true);
                }
            } catch (NamingException nmex) {
                nmex.printStackTrace(System.err);
            } catch (SQLException e) {
                e.printStackTrace(System.err);
                result = null;
            } catch (DatatypeConfigurationException dtypconfex) {
                dtypconfex.printStackTrace(System.err);
            } finally {
                try {
                    if (pm != null)
                        pm.close();
                    if (con != null)
                        con.close();
                } catch (SQLException ex) {
```

```
                    ex.printStackTrace(System.err);
            }
        }
        return result;
    }
```

6. Organize imports by pressing Ctrl+Shift+O.

   For the code shown in Example 4-1 on page 176, choose the following
   classes to resolve the import declarations:

   – javax.sql.ResultSet
   – javax.sql.DataSource
   – javax.xml.datatype.DatatypeFactory
   – java.sql.Date
   – java.sql.Connection

7. Save the file. All errors should be resolved.

Configuring the enhanced EAR file

The WebSphere Application Server uses a proprietary set of configuration data
to supplement the information stored in the standard Java EE deployment
descriptor. This is known as the *enhanced enterprise archive (EAR) feature*,
which includes configuration information for JDBC providers and data sources.

To configure the JDBC data source in the enhanced EAR file:

1. Right-click the Enterprise Application Project (**WeatherForecastWSEAR**)
   and select **Java EE → Open WebSphere Application Server Deployment**.

2. In the enhanced EAR deployment descriptor editor that opens in the main
   area, in the Data source defined in the JDBC provider selected above section,
   click the **Add** button.

3. In the Create Data Source window, select **Derby JDBC Provider (XA)** and
   click **Next**.

4. In the next window, for the data source, in the Name field type `Weather WS`
   `Data Source`, and in the JNDI name field type `jdbc`. Click **Next**.

5. In the Create Resource Properties window:

   a. Select the **databaseName** property.

   b. In the Value field, enter the full path of the database directory. Note that
      the path delimiter for the enhanced EAR is a forward slash (/).

   > **Note:** If the weather database is set up as explained in "Set up the
   > WEATHER database (Derby)" on page 540, the value for this filed
   > should be `C:/Database/WEATHER`.

c. Click **Finish**.

6. Close and save the enhanced EAR file.

Deploying the Web service

Rational Application Developer Assembly and Deploy uses an external WebSphere Application Server V7.0 installation as an attached test environment. in this section we explain how to set up a WebSphere Application Server V7.0 server definition inside Rational Application Developer Assembly and Deploy.

Note: As mentioned in the beginning of this chapter, WebSphere Application Server V7.0 must be installed for these examples so that a server definition can be created for it in Rational Application Developer Assembly and Deploy. The server definition that we create in this section is used in all examples in this chapter, except in 4.4, "EJB Web services" on page 197, which uses the embedded test environment of Rational Application Developer V7.5.

Create a server definition for a WebSphere Application Server V7.0 installation:

1. In Rational Application Developer Assembly and Deploy, switch to the **Java EE** perspective if it is not already open.

2. In the Servers view, right-click the empty space and select **New → Server**.

3. In the New Server window (Figure 4-9), for Select the server type, select **WebSphere Application Server v7.0**. Next to the Server runtime environment section, click the **Add** link.
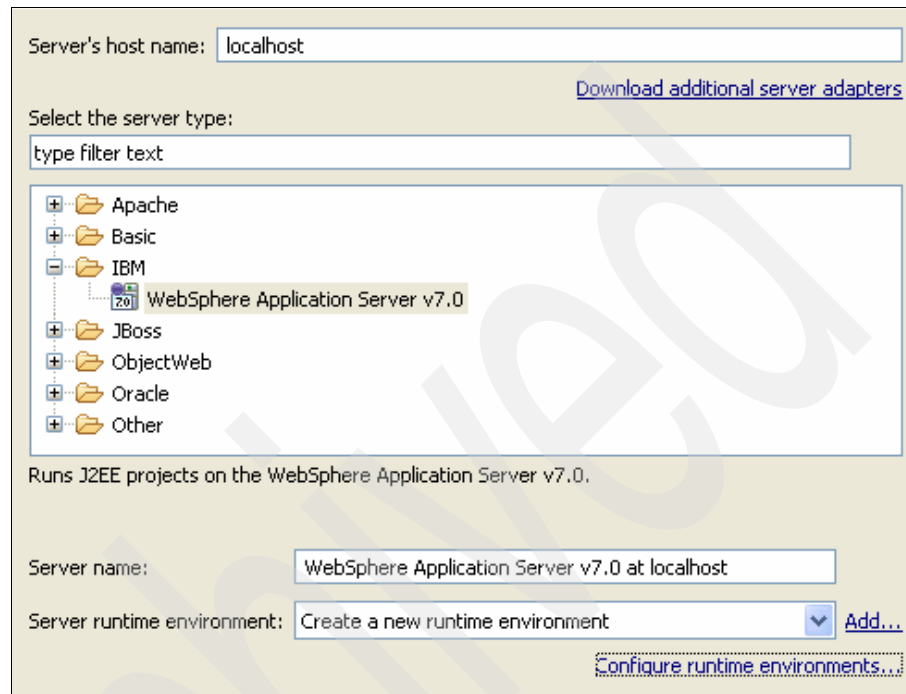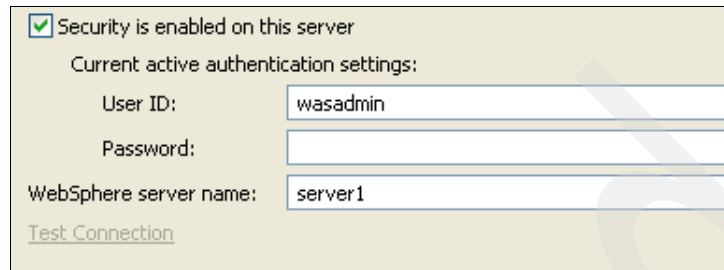
Figure 4-9   New Server window

4. In the New Server Runtime Environment window, under the Installation directory, browse to the *WAS_HOME* directory. Click **Finish**.

5. Back in the New Server window, click **Next**.

6. As shown in Figure 4-10, if Security is enabled on this server is selected, enter the user ID and password. You can test the settings by clicking the **Test Connection** link. Click **Finish**.

Figure 4-10   Entering the server credentials

A new server definition for WebSphere Application Server V7.0 is created in the Servers view.

To deploy the Web service to the attached WebSphere Application Server V7.0:

1. In the Servers view, right-click the **WebSphere Application Server v7.0 at localhost** server definition and select **Start**.

   If not previously present, the Console view is displayed, and the WebSphere Application Server V7.0 startup trace commences.

2. In the Console view (Figure 4-11), wait for the line reads `Server server1 open for e-business` to display. In the Servers view, wait for the state heading for the server to read `Started`.
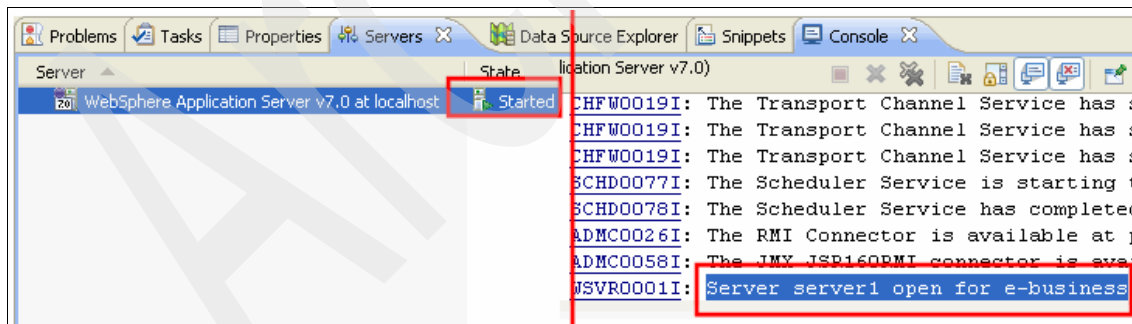
Figure 4-11   Server started

3. In the Servers view, right-click the server definition and select **Add and Remove Projects**.

4. In the Add and Remove Projects window:

  a. Select the Enterprise Application Project to be deployed (**WeatherForecastWSEAR**).

  b. Click the **Add** button and wait for the project to be displayed under Configured projects.

  c. Click **Finish**.

5. In the Servers view, right-click the server definition and select **Publish**. Wait for the status heading in the Servers view to read `Synchronized`. At this stage, the Web service is deployed.

Testing the Web service (from an external browser)

A simple way to test a deployed Web service is to exploit the fact that it must expose its WSDL interface for clients. You can access the exposed WSDL by using a Web browser, which is convenient because a client application is not required to perform this test.

To test the ability to access the deployed Web service's WSDL file by using a browser, open a browser and enter the following URL:

```
http://localhost:9080/WeatherForecastWS/WeatherForecastService?wsdl
```

Figure 4-12 shows the WSDL of the Web service. When the request is sent, the URL on the address bar of the browser changes to a direct access URL, for example, `../WeatherForecastService.wsdl`.

Figure 4-12   Accessing the Web service WSDL

> **Alternative:** You can also test deployed Web services by using the Web Services Explorer, which we explain in 4.5.1, "The Web Services Explorer" on page 214.

4.2.2 Web services development from an existing Java bean

Web services development can also begin with business logic that is already in a Java bean, or in this case, with a working Java application in a package. A minimal amount of code made through Java annotations is inserted into the bean, after which the bean is introspected and the Web service interface is derived.

This strategy for creating Web services is known as *bottom-up development*. Development of a Web service by using this strategy involves the following tasks:

1. Annotating the Java bean (from a text editor)
2. Generating the Web service interface (from a command line)
3. Completing the Web service implementation
4. Configuring the enhanced EAR file
5. Deploying and testing the Web service (from an external browser)

We explain each of these tasks in the sections the follow.

Annotating the Java bean (from a text editor)

Before a Web service interface can be generated, you must annotate the existing Java bean. In this example, the Java bean that is being used implements an interface, which must also be annotated.

Annotate the existing Java bean's interface as the Web service's SEI:

1. Open the `IWeatherForecast.java` SEI in *any text editor*. This file is in the `/itso/businessobjects` directory of the `ch04_app_dev.zip` file.

2. Add the `@WebService` annotation and the appropriate import statement to the interface (Figure 4-13).

Figure 4-13   Annotating the interface class

3. Save and close the file.

To annotate the existing Java bean as the Web service's SIB:

1. Open the `WeatherForecast.java` SIB class in *any text editor*. This file is in the `/itso/businessobjects` directory of the `ch04_app_dev.zip` file.

2. Add the `@WebService` annotation with SEI class name, `itso.businessobjects.IWeatherForecast`, that is specified in the `endpointInterface` attribute. Also add the appropriate `import` statement (Figure 4-14).


```java
import java.util.List;

import javax.jws.WebService;

/**
 * Business object for the Weather Forecast service.
 */
@WebService(endpointInterface="itso.businessobjects.IWeatherforecast")
public class WeatherForecast implements IWeatherForecast {
    /**
     * Default constructor.
```

Figure 4-14   Annotating the Java bean

3. Save and close the file.

Generating the Web service interface (from a command line)

You must now compile the annotated Java bean and its accompanying classes and run them through the **wsgen** command to generate the service interface. Figure 4-15 illustrates where the **wsgen** tool fits in this task.
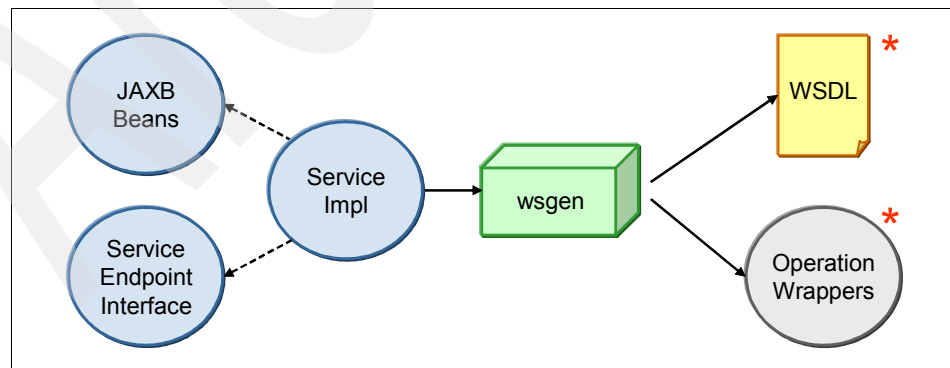
Figure 4-15   The role of the wsgen tool for generating the Web service interface

> **wsgen steps and the Web service wizard:** In Rational Application Developer V7.5, the `wsgen` steps are incorporated into the Web service wizard, which automates most of the Web services development steps. For most of the examples in this chapter, we perform the `wsgen` steps from a command line because the Web service wizard is not included in Rational Application Developer Assembly and Deploy. However, the examples in 4.4, "EJB Web services" on page 197, use this tool because it uses Rational Application Developer V7.5.

To generate the Web service's interface artifacts from a Java bean:

1. Open a console window and navigate to *WAS_HOME*/bin.

2. On a command line, enter the `setupCmdLine` command to set up the environment variable *WAS_PATH*.

3. On a command line, enter the following command to set the path:

   `set PATH=%PATH%;%WAS_PATH%`

4. Navigate to the directory of the `itso` folder. This folder is in the `ch04_app_dev.zip` file and where the annotated Java classes are ("Annotating the Java bean (from a text editor)" on page 184).

5. On a command line, enter the following command to compile the annotated Java bean in its package directory:

   `javac ./itso/businessobjects/*.java`

6. On a command line, enter the `wsgen` command on the compiled classes:

   `wsgen -verbose -keep -wsdl -cp .`
   `itso.businessobjects.WeatherForecast`

   The options used are summarized as follows:

   – The verbose option provides a trace output of the operations that are being performed by the command.

   – The keep option retains the *.java source files that are generated by the command.

   – The wsdl option generates a WSDL file from the service implementation.

   – The cp option specifies the Java class path for this command to use.

The `wsgen` command generates the same artifacts as the `wsimport` command. Such artifacts include the request/response wrappers and complex type implementations. Figure 4-16 shows a list of these artifacts. The difference is that the SEI and SIB are already present before the command was run on the annotated Java bean. If the -wsdl option was specified, a WSDL file is also derived from the Java bean.
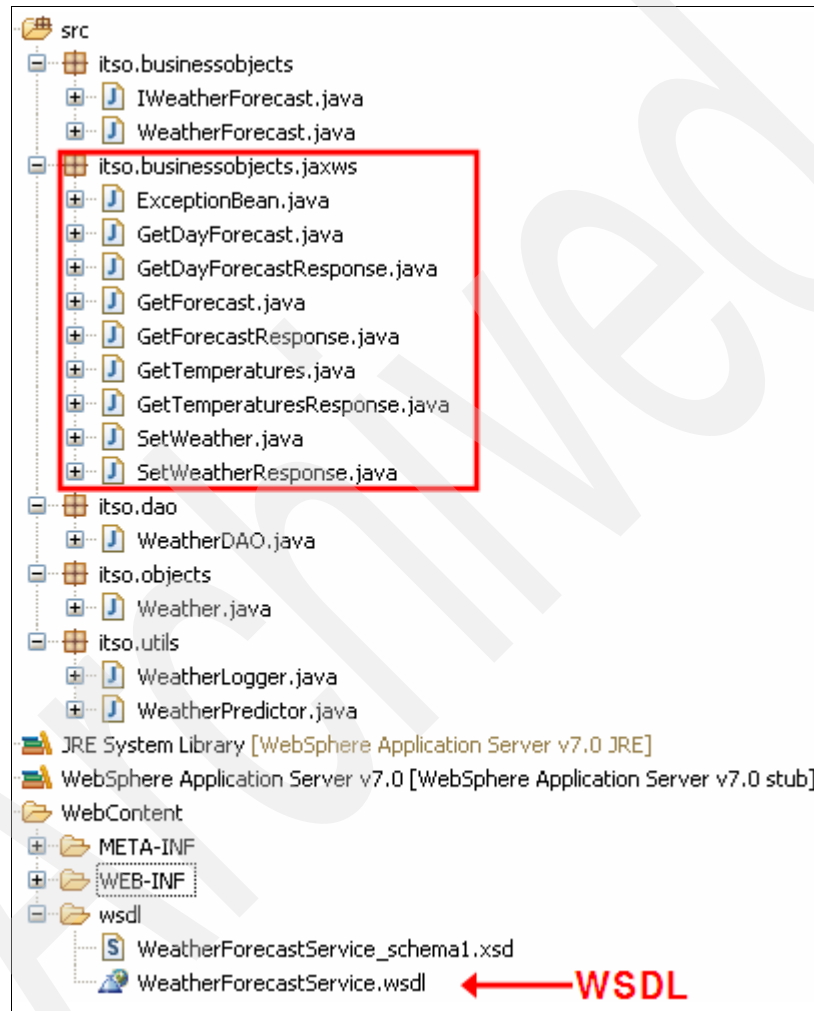
Figure 4-16   The wsgen generated artifacts

Completing the Web service implementation

Before proceeding, undeploy and delete the previous Web service projects (see 4.2.1, "Web services development from a WSDL file" on page 166) from the

Rational Application Developer Assembly and Deploy workspace to avoid conflicts:

1. In the Servers view, right-click the server definition and select **Add and Remove Projects**.

2. In the Add and Remove Projects window, select the Enterprise Application Project to be removed (**WeatherForecastWSEAR**). Click the **Remove** button and wait for the project to be displayed under Available projects. Click **Finish**.

3. In the Servers view, right-click the server definition and select **Publish**. Wait for the status heading in the Servers view to read Synchronized.

4. In the Servers view, right-click the server definition and select **Stop**. Wait for the state heading in the Servers view for the server to read Stopped.

5. Press Ctrl and select the **WeatherForecastWS** and **WeatherForecastWSEAR** projects, right-click, and select **Delete**.

6. In the Delete Resources dialog box, select **Delete project contents on disk** and click **OK**.

Import the generated artifacts into a Dynamic Web Project in Rational Application Developer Assembly and Deploy and assemble into an EAR file.

Create a Dynamic Web Project for the Web service:

1. In Rational Application Developer Assembly and Deploy, switch to the **Java EE** perspective if it is not already open.

2. Select **File** → **New** → **Dynamic Web Project**.

3. In the New Dynamic Web Project window, for project name type WeatherForecastWS, and for EAR project name type WeatherForecastWSEAR. Click **Finish**.

4. If prompted to switch perspectives, click **No**.

Import the wsgen artifacts (including the annotated Java bean) into the Dynamic Web Project:

1. Expand **Java Resources** in the Dynamic Web Project (WeatherForecastWS). Right-click **src** and select **Import**.

2. Expand **General** and select **File System**. Then click **Next**.

3. In the File system window:

   a. Click **Browse** and go to the directory of the itso folder (with the annotated Java bean and the artifacts created by wsgen).

   b. Import the itso folder created by wsimport.

c. Click the **Filter Types** button to import only the *.java source files.

d. Click **Finish**.

Configuring the enhanced EAR file

After assembling the Web service into an enterprise application, the JDBC data source settings are configured by using the enhanced EAR file. The following steps are the same as in the top-down approach example in 4.2.1, "Web services development from a WSDL file" on page 166.

1. Use the WebSphere Application Server Deployment editor to configure the enhanced EAR file.

2. Configure a JDBC data source for the Derby database.

3. Save the enhanced EAR file.

Deploying and testing the Web service (from an external browser)

Assuming that you created the server definition as explained in "Deploying the Web service" on page 179, after the EAR is properly configured, deploy it to the attached WebSphere Application Server V7.0:

1. Deploy the EAR file to the attached WebSphere Application Server V7.0 instance (see "Deploying the Web service" on page 179).

2. Test the Web service by accessing its WSDL file from a browser (see "Testing the Web service (from an external browser)" on page 182).

4.3 Developing clients for Web services

In the next series of examples we show techniques for client-side Web services development. Now that the server-side component is in place, the client programs can be created to access the Web service.

4.3.1 Creating a managed Web service client

In this example we develop a managed client to access a Web service. A managed client is an application that resides in the same server run time as the server-side Web service.

Development of a managed Web service client involves the following tasks:

1. Generating the client skeleton code (from a command line)
2. Writing the client application
3. Deploying and running the client application

We explain each of these tasks in the sections the follow.

Generating the client skeleton code (from a command line)

When we used `wsimport` in the top-down example in 4.2.1, "Web services development from a WSDL file" on page 166, a service client class (WeatherForecastService.java) was generated along with the other artifacts (see Figure 4-5 on page 172). This service client was ignored in that example. Now the service client is the crucial piece in the development of the client application.

By using the WSDL (`WeatherForecastService.wsdl`) and schema (`WeatherForecastService_schema1.xsd`) files from the `ch04_app_dev.zip` file, follow the same steps in "Generating the skeleton code (from a command line)" on page 171.

> **WSDL:** In an actual development scenario, the WSDL that is used to generate the client comes from the Web service provider. Usage of this WSDL ensures that the Web service contract, as specified in the WSDL, is fulfilled in the client-side application.

Writing the client application

Before writing the client application, a working WeatherForecastWSEAR project must be deployed in the attached WebSphere Application Server V7.0 in Rational Application Developer Assembly and Deploy. This project can come from either 4.2.1, "Web services development from a WSDL file" on page 166, or 4.2.2, "Web services development from an existing Java bean" on page 183. Follow the instructions in "Deploying the Web service" on page 179 to deploy the project to the defined server.

In this section, the service client is used in a simple JavaServer™ Pages (JSP) page (acting as a managed client) that is deployed in a Web application. Create a Dynamic Web Project for the Web service client:

1. In Rational Application Developer Assembly and Deploy, switch to the **Java EE** perspective if it is not already open.

2. Select **File** → **New** → **Dynamic Web Project**.

3. In the New Dynamic Web Project window, for project name type `WeatherForecastWebClient`, and for EAR project name type `WeatherForecastWebClientEAR`. Click **Finish**.

4. If prompted to switch perspectives, click **No**.

Import the `wsimport`-generated artifacts (`itso` folder) into the Dynamic Web Project:

1. Expand **Java Resources** in the Dynamic Web Project (`WeatherForecastWebClient`). Right-click **src** and select **Import**.

2. Expand **General** and select **File System**. Then click **Next**.

3. In the File system window:

   a. Click **Browse** and go to the directory of the files created by `wsimport` (the `itso` package).

   b. Import the `itso` package created by `wsimport`.

   c. Click the **Filter Types** button to import only the *.java source files.

   d. Click **Finish**.

Modify the URLs in the service client Java class to reference the Web service WSDL:

1. In the imported package under **Java Resources** → **src**, double-click the **WeatherForecastService.java** service client to open it in the Java editor.

2. Modify the client code to correct the value of the WSDL URLs, as shown in Figure 4-17. The URL for the example looks like this:

   ```
   http://localhost:9080/WeatherForecastWS/WeatherForecastService?wsdl
   ```

   > **URL to access the WSDL**: This URL is the same one that is used to access the WSDL from a browser in "Testing the Web service (from an external browser)" on page 182.

```
@WebServiceClient(
    name =
        "WeatherForecastService",
    targetNamespace =
        "http://businessobjects.itso/",
    wsdlLocation =
        "http://localhost:9080/WeatherForecastWS/WeatherForecastService?wsdl")

public class WeatherForecastService
    extends Service
{

    private final static URL WEATHERFORECASTSERVICE_WSDL_LOCATION;

    static {
      URL url = null;
        try {
            url =
              new URL("http://localhost:9080/WeatherForecastWS/WeatherForecastService?wsdl");
```

Figure 4-17   Modifying the service client

3. Save the WeatherForecastService.java service client.

Create a client JSP in the Dynamic Web Project:

1. In the Rational Application Developer Assembly and Deploy workspace, under WeatherForecastWebClient, right-click **Web Content** and select **New → Other**.

2. In the Select a Wizard window, expand **Web** and select **JSP**. If the JSP option is not available, select the **Show All Wizards** box at the bottom of the window. Click **Next**.

3. In the JavaServer Page window, for the parent folder select **WebContent** and for the file name type WeatherWSTest.jsp. Click **Finish**.

4. Write the client implementation on the JSP.

    Example 4-2 shows the code for the JSP of the client Web application. This code is also written in the WeatherWSTest_jsp_snippet.txt file that is included with the ch04_app_dev.zip file as part of the additional materials for this book. (See Appendix A, "Additional material" on page 537.)

    The crucial section here lies in the *scriptlet*, which is the area delimited by <%...%>. The service client, WeatherForecastService, is used to obtain an instance of IWeatherForecast by using the getWeatherForecastPort() method. From this instance, the operations of the Web service, such as getDayForecast(), are invoked.

    Example 4-2   The WeatherWSTest.jsp file

```
<%@ page language="java"
    contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">

<%@page import="itso.businessobjects.WeatherForecastService"%>
<%@page import="itso.businessobjects.IWeatherForecast"%>
<%@page import="itso.businessobjects.Weather"%>
<%@page import="javax.xml.datatype.DatatypeFactory"%>
<%@page import="java.util.GregorianCalendar"%><html>

<head>
<meta http-equiv="Content-Type"
    content="text/html; charset=ISO-8859-1">
<title>Weather Web service Test</title>
</head>

<body>
```

```
<%
    WeatherForecastService service =
        new WeatherForecastService();
    IWeatherForecast port = service.getWeatherForecastPort();

    Weather weather = port.getDayForecast(
        DatatypeFactory.newInstance().newXMLGregorianCalendar(
            "2006-01-07T00:00:00"));

    out.print("The weather is " +
        weather.getCondition() + " ");

    out.print("with temperature at " +
        weather.getTemperatureCelsius() + " deg. C ");

    out.print("and winds gusting at " +
        weather.getWindSpeed() + " MPH ");

    out.println("from a direction of " +
        weather.getWindDirection() + ".");
%>
</body>
</html>
```

5. Save the file.

Deploying and running the client application

After the EAR file is properly configured, deploy it to the attached WebSphere Application Server V7.0. To deploy the file, the server definition must already be properly created, as explained in "Deploying the Web service" on page 179.

1. In Rational Application Developer Assembly and Deploy, in the Servers view, right-click the **WebSphere Application Server v7.0 at localhost** server definition and select **Start**. Wait for the server to indicate that it is started.

2. In the Servers view, right-click the server definition and select **Add and Remove Projects**.

3. In the Add and Remove Projects window, select the Enterprise Application Project to be deployed (**WeatherForecastWebClientEAR**). Click the **Add** button and wait for the project to be displayed under Configured projects. Click **Finish** and wait for the server to indicate that it is synchronized.

4. Under WeatherForecastWebClient → WebContent, right-click **WeatherWSTest.jsp** and select **Run As** → **Run on Server**.

5. In the Run On Server window, highlight **WebSphere Application Server v7.0 at localhost** and click **Finish**. The embedded Web browser of Rational Application Developer Assembly and Deploy is displayed with the results (Figure 4-18).
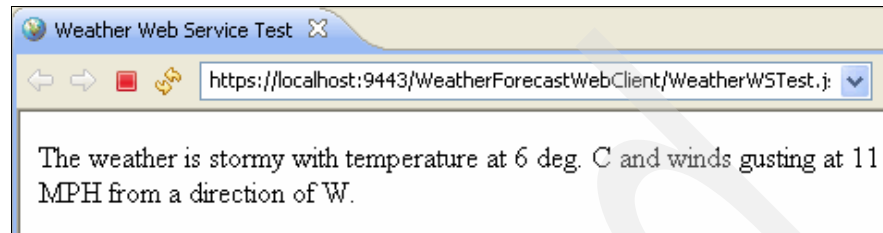
Figure 4-18   Client Web application results

4.3.2 Creating a Web service thin client

WebSphere Application Server V7.0 provides a way by which Web services can be invoked by clients that are not on the same server run time. This application model is known as an *unmanaged client*.

Development of a Web service thin (unmanaged) client involves the following tasks:

1. Generating the skeleton code (from a command line)
2. Writing the client application
3. Running the client application

We explain each of these tasks in the sections that follow.

Generating the skeleton code (from a command line)

As in the client example from 4.3.1, "Creating a managed Web service client" on page 189, use the `wsimport` command to generate the artifacts that are required to develop the client application.

By using the WSDL (`WeatherForecastService.wsdl`) and schema (`WeatherForecastService_schema1.xsd`) files from the `ch04_app_dev.zip`, follow the same steps in "Generating the skeleton code (from a command line)" on page 171.

Writing the client application

Before you write the client application, ensure that a working WeatherForecastWSEAR project is deployed in the attached WebSphere Application Server V7.0 in Rational Application Developer Assembly and Deploy. This project can come from either 4.2.1, "Web services development from a

WSDL file" on page 166, or 4.2.2, "Web services development from an existing Java bean" on page 183. Use the instructions in "Deploying the Web service" on page 179 to deploy the project to the defined server.

In this example, we use a standalone Java application as a client. The programming logic is the same. The only major difference is the presence of a JAR class library that is included in the client application.

Create a Java project for the thin client application:

1. In Rational Application Developer Assembly and Deploy, switch to the **Java** perspective.

2. Select **File** → **New** → **Project**.

3. In the Select a Wizard window, expand **Java** and select **Java Project**. Click **Next**.

4. In the Create a Java Project window, for project name type `WeatherForecastThinClient`. Click **Next**.

5. In the Java Settings window, click the **Libraries** tab. Click the **Add External JARs** button.

6. Navigate to the `runtimes` folder of *was_home*, which is the WebSphere Application Server directory. Select **com.ibm.jaxws.thinclient_7.0.0.jar** and click **Open**. The JAR file is displayed in the list of referenced libraries.

> **Note:** The `com.ibm.jaxws.thinclient_7.0.0.jar` file enables a client application to act as an unmanaged client.

7. Back in the Java Settings window, click **Finish**.

Modify the WSDL URLs in the service client Java class:

1. Under **Java Resources** → **src**, in the imported package (`itso.businessobjects`), double-click the **WeatherForecastService.java** service client to open it in the Java editor.

2. Modify the code to use the correct WSDL URLs. The URL for the example looks like this:

   `http://localhost:9080/WeatherForecastWS/WeatherForecastService?wsdl`

   This is the same step as in "Writing the client application" on page 190. See Figure 4-17 on page 191 for details.

3. Save the service client Java class.

Create the client Java program:

1. Under WeatherForecastThinClient, right-click **src** and select **New → Class**.

2. In the New Java Class wizard:

   a. In the Name field, type WeatherWSTest.

   b. Under Which method stubs would you like to create?:

      i. Select **public static void main(String[] args)** to let the wizard generate a main() method for this Java class.

      ii. Clear **Inherited abstract methods** to declutter the generated code of other methods that are not needed for this example.

   c. Click **Finish**.

3. Write the client implementation in the main(String[] args) method of the Java class.

   The code in Example 4-3 is for the main method of the Java class of the thin client application. The code is also in the WeatherWSTest_java_snippet.txt file that is included with the ch04_app_dev.zip file.

   The program logic is the same as in the JSP file of the previous client example. That is, the Web service's operations are invoked through the service client class, WeatherForecastService.

Example 4-3   The main(String[] args) method of WeatherWSTest.java

```
public static void main(String[] args) {
    try {
        WeatherForecastService service = new
WeatherForecastService();
        IWeatherForecast port = service.getWeatherForecastPort();

        Weather weather = port.getDayForecast(
        DatatypeFactory.newInstance().newXMLGregorianCalendar(
            "2006-01-07T00:00:00"));

        System.out.print("The weather is " +
            weather.getCondition() + " ");
        System.out.print("with temperature at " +
            weather.getTemperatureCelsius() + " deg. C ");
        System.out.print("and winds gusting at " +
            weather.getWindSpeed() + " MPH ");
        System.out.println("from a direction of " +
            weather.getWindDirection() + ".");
    } catch(Exception_Exception ex_ex) {
        ex_ex.printStackTrace();
```

```
            } catch(DatatypeConfigurationException dtypeconfex) {
                dtypeconfex.printStackTrace();
            }
        }
```

4. Organize the imports by pressing Ctrl+Shift+O.

5. Save the file. All errors should be resolved.

Running the client application

With the client implemented, the application can be run from any Java run time.
To run the thin client Java application, right-click the main Java class
(**WeatherWSTest.java**) and select **Run As** → **Java Application**. The results are
displayed in the Console view (Figure 4-19) based on the data that is retrieved
from the weather database.

> **Resulting values**: The values of the result are the same as the managed client
> example in 4.3.1, "Creating a managed Web service client" on page 189,
> because the same parameter is used to invoke the Web service operation.


```
Console ✕    Problems  @ Javadoc  Declaration                           ■ ✕ ✕  ▤ ▤ ▤ ▤
<terminated> WeatherWSTest [Java Application] C:\RADAD\SDP\jdk\bin\javaw.exe (May 3, 2009 10:18:41 PM)
Retrieving document at 'http://localhost:9080/WeatherForecastWS/WeatherForecastService/'.
Retrieving schema at 'WeatherForecastService_schema1.xsd', relative to 'http://localhost:9080/WeatherF
The weather is stormy with temperature at 6 deg. C and winds gusting at 11 MPH from a direction of W.
```
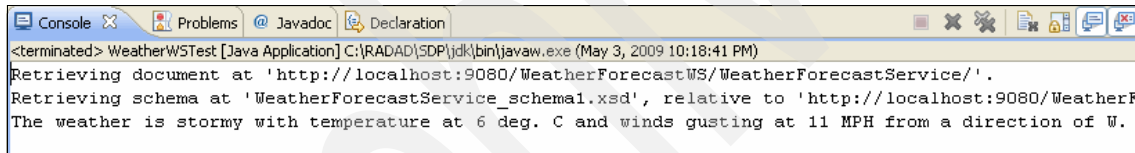
Figure 4-19   Client Java application results

4.4 EJB Web services

EJBs are Java EE 5 components that encapsulate business logic and have the
capability of being deployed for distributed system architectures. EJBs have a
distributed application execution model that is similar to Web services minus the
XML-based protocols. In fact, for the JAX-WS programming model, a particular
kind of EJB, the *session bean*, can be used to create Web services.

A distinct advantage of using an EJB implementation is the ability to use the JMS
as the Web service transport. In an HTTP transport, the client blocks when the
service request is sent and waits until the response is received. In a JMS
transport, control is given back to the client when the request is sent to a *queue*.
Therefore, the client is able to continue operation. The response is sent to
another queue to hold the message until the client can retrieve it. JMS provides

Web services a true asynchronous invocation mechanism, as illustrated in Figure 4-20.
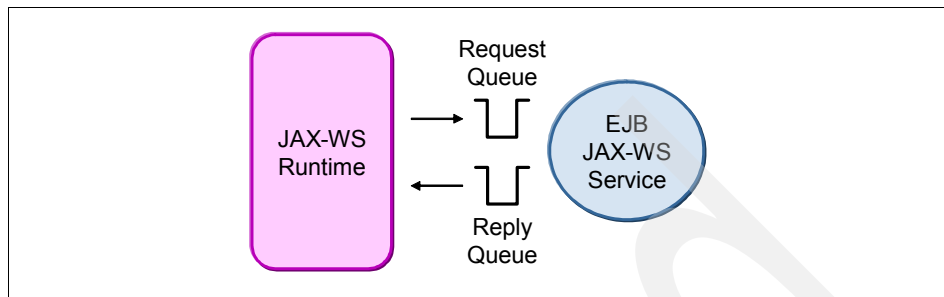
Figure 4-20   JMS EJB Web service transport

> **Note:** This example uses the full version of Rational Application Developer V7.5 with the complete set of development tools. These tools include the Web service wizard that automates most of the development processes seen in earlier examples in this chapter.

4.4.1 Creating an EJB Web service

In the example in this section we use a *top-down* approach to develop an EJB Web service. We use a WSDL file to start the process, similar to what we did in an earlier example. The difference is that the Web service wizard of Rational Application Developer is used for the development effort and the resulting Web service uses an EJB with a JMS transport.

Development of an EJB Web service involves the following tasks:

1. Setting up the JMS resources
2. Running the Web service wizard
3. Implementing the EJB Web service
4. Configuring the enhanced EAR file

We explain each of these tasks in the sections that follow.

Setting up the JMS resources

Because the EJB that is being developed uses JMS, you must set up the appropriate JMS resources in the WebSphere Application Server to provide this capability. To do this, you can use an administrative script.

The WebSphere Application Server has the ability to configure resources by using a *Jython* language script through a facility called *wsadmin*. Rational

Application Developer has the ability to edit, debug, and run Jython scripts for
wsadmin.

> **Test environment:** Rational Application Developer V7.5 includes the
> WebSphere Application Server V7.0 test environment, which is embedded into
> the IDE. This example uses the test environment as the development run time.

Create a Jython project for the administrative script and WSDL:

1. In the Rational Application Developer, right-click the **WebSphere Application
   Server v7.0** instance in the Servers view and select **Start**. Wait until the
   server has achieved a *started* state.

2. Select **File → New → Other**.

3. Expand **Jython** and select **Jython Project**. Click **Next**.

4. For project name, type WeatherForecastSetup. Click **Finish**.
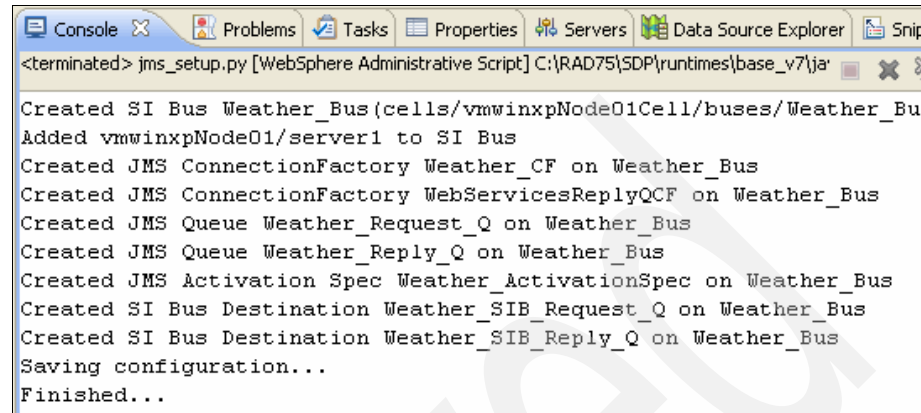
Import the administrative script and WSDL into the Jython project:

1. Right-click the **Jython Project** and select **Import**.

2. In the Import window, select **General → File System**. Click **Next**.

3. Browse for the directory of the administrative script (jms_setup.py), WSDL
   (WeatherForecastService.wsdl), and schema
   (WeatherForecastService_schema1.xsd) files. Import the files. Click **Finish**.

Run the administrative script:

1. Right-click the administrative script (*.py file) and select **Run As →
   Administrative Script**.

2. In the Edit Configuration window, in which you indicate the run time that the
   script will use, for scripting run time:

   a. Use the selector to choose **WebSphere Application Server v7.0**.
   b. Use the selector to choose a **WebSphere profile**.
   c. Click **Run**.

The administrative script runs, and the results are displayed in the Console view (Figure 4-21).


```
Console ☒  🔲 Problems  📋 Tasks  🔲 Properties  🔧 Servers  🗄 Data Source Explorer  📄 Snip
<terminated> jms_setup.py [WebSphere Administrative Script] C:\RAD75\SDP\runtimes\base_v7\ja  ▣  ✖  ☒
Created SI Bus Weather_Bus(cells/vmwinxpNode01Cell/buses/Weather_Bus
Added vmwinxpNode01/server1 to SI Bus
Created JMS ConnectionFactory Weather_CF on Weather_Bus
Created JMS ConnectionFactory WebServicesReplyQCF on Weather_Bus
Created JMS Queue Weather_Request_Q on Weather_Bus
Created JMS Queue Weather_Reply_Q on Weather_Bus
Created JMS Activation Spec Weather_ActivationSpec on Weather_Bus
Created SI Bus Destination Weather_SIB_Request_Q on Weather_Bus
Created SI Bus Destination Weather_SIB_Reply_Q on Weather_Bus
Saving configuration...
Finished...
```

Figure 4-21   Administrative script results

Note: The sample script `jms_setup.py` creates the following JMS resources that are needed for the JMS Web service:

- ► Request queue (JNDI name: `jms/Weather_Request_Q`)
- ► Connection factory (JNDI name: `jms/Weather_CF`)
- ► ActivationSpec (JNDI name: `eis/Weather_ActivationSpec`)

---

3. In the Servers view, right-click the server and select **Restart**.

   Restarting the server finalizes the configurations that the script performed on the messaging resources.

Running the Web service wizard

As mentioned earlier, this example uses the top-down approach, which is similar to the example in 4.2.1, "Web services development from a WSDL file" on page 166. However, this time we use the Web service wizard of Rational Application Developer V7.5. This tool automates most of the Web services development process and is less error prone than manually using **wsimport** or **wsgen**.

Similar to previous example, a WSDL is required to begin the process and provide a service interface. A WSDL gives a *location attribute* for the <soap:address> element. The value for this attribute is a URL. The specified protocol (HTTP or JMS) determines the transport that is used for the Web service. This becomes relevant later in the process.

To initiate the Web service Wizard and configure server-side and client-side properties:

1. Right-click the WSDL file and select **Web services** → **Generate Java bean skeleton**.

2. In the Web service wizard (Figure 4-22):

   a. For Web service type, select **Top down EJB Web service**.

   b. In the Configuration (server-side) section:

      i. For server, select **WebSphere Application Server v7.0**.
      ii. For Web service runtime, select **IBM WebSphere JAX-WS**.
      iii. For service project, type WeatherForecastJMS.
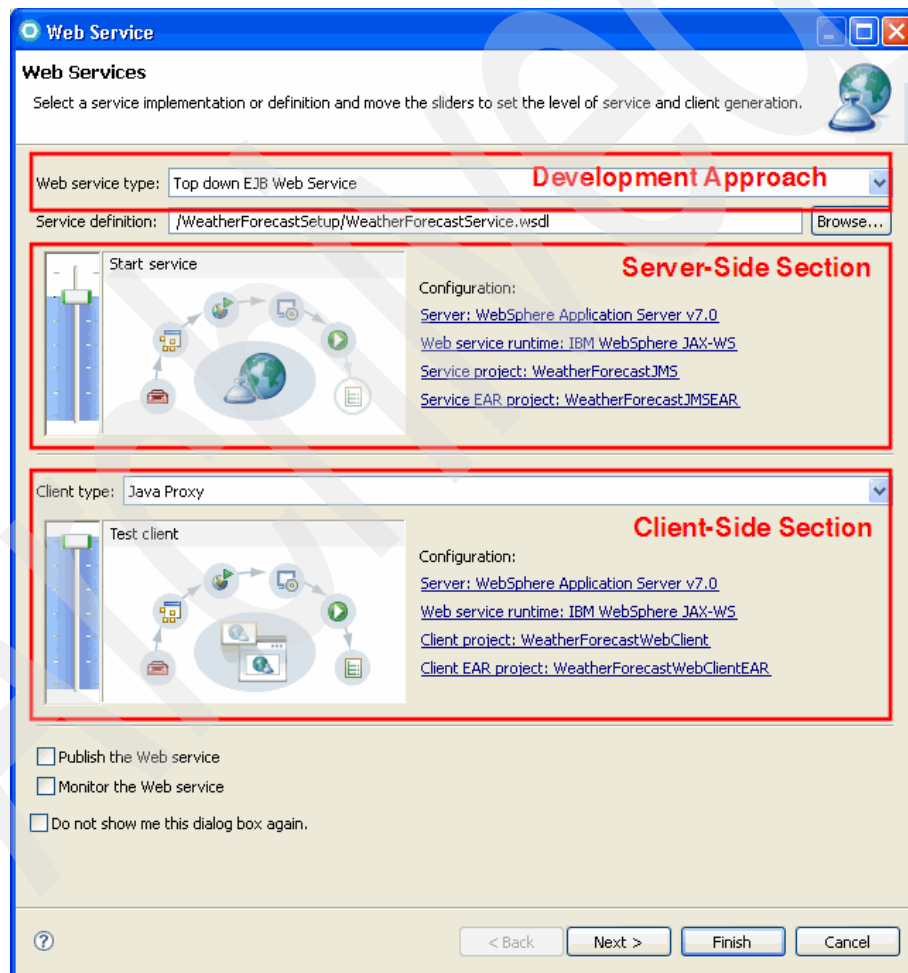
Figure 4-22   The Web service wizard

c. For client type, select **Java Proxy**.

d. In the Configuration (client-side) section:

    i. Generate a test client for the generated Web service by pushing the slider to the top-most position until it reads `Test client`. The client-side slider is set to *Client only* by default.

    ii. For client project, type `WeatherForecastWebClient` (use the Dynamic Web Project).

    iii. For the Client EAR project, type `WeatherForecastWebClientEAR`. Click **Next**.

To bind the implementation to JMS resources in the Web service Wizard:

1. On the WebSphere JAX-WS Top Down EJB Web service Configuration page, under WSDL bindings, select **Switch to JMS binding**.

   Because of the value of the <soap:address> location attribute, the WSDL bindings field is set to HTTP. This step ignores this setting and generates the service based on the JMS transport instead.

   Click **Next**.

2. On the WebSphere JAX-WS JMS Binding Configuration page, for destination JNDI name (request queue) type `jms/Weather_Request_Q`, and for JMS connection factory (connection factory) type `jms/Weather_CF`. Click **Next**.

3. On the WebSphere JAX-WS Router Project Configuration page, for activationSpec JNDI name (ActivationSpec), type `eis/Weather_ActivationSpec`.

   A router module handles the transport for EJB-based Web services. For the HTTP protocol, the router is a Web module and uses a servlet. A JMS router uses a message-driven bean (MDB) EJB module.

   The WSDL bindings are now set for JMS and to use a JMS router module. The HTTP router field has been disabled.

   Click **Next**.

To complete the Web service wizard processing:

1. On the Test Web service page, click **Next**.

2. On the WebSphere JAX-WS Web Service Client Configuration page, click **Next**.

3. On the Web service Client Test page, ensure that the test facility is set to **JAX-WS JSPs**. Click **Finish**.

> **Adding the EAR file to the test environment**: The Web service wizard automatically adds (deploys) the created EAR file to the WebSphere Application Server V7.0 test environment. It starts the server as part of the wizard's development process.

Implementing the EJB Web service

After using the Web service wizard, the skeleton EJB code is generated. The EJB Web service SIB Java source file opens in the main area. Note the annotations for both EJB and Web services.

To write the Web service implementation into the EJB skeleton:

1. Expand **WeatherForecastJMS** → **src** → **itso.businessobjects.** Double-click **WeatherForecastPortBindingImpl.java** to open it.

2. Implement the getDayForecast() method.

> **Note:** For this example, only the getDayForecast() method is required to be implemented. The following service methods of WeatherForecastPortBindingImpl.java are optional for implementation per the user's discretion:
>
> ► getForecast()
> ► getTemperatures()
> ► setWeather()

The code in Example 4-4 is for the getDayForecast() operation of the WeatherForecastPortBindingImpl SIB. You can also find the body of the code written in the `WeatherForecast_JMS_snippet.txt` file that is included with the `ch04_app_dev.zip` file. This code fragment is similar to the one used in the top-down (non-EJB) example in 4.2.1, "Web services development from a WSDL file" on page 166.

Errors as a result of unresolved type declarations are expected and should be ignored in the meantime.

Example 4-4   The getDayForecast() method of WeatherForecastPortBindingImpl.java

```
public Weather getDayForecast(XMLGregorianCalendar arg0)
    throws Exception_Exception {
    Connection con = null;
    PreparedStatement pm = null;
    Weather result = null;

    try{
        InitialContext ic = new InitialContext();
```

```java
            DataSource ds = (DataSource) ic.lookup("jdbc/weather");

            con = ds.getConnection();
            pm = con.prepareStatement(
                "SELECT * FROM ITSO.SANJOSE WHERE WEATHERDATE = ?");
            Date sqlDate =
                new
Date(arg0.toGregorianCalendar().getTime().getTime());
            pm.setDate(1, sqlDate);
            ResultSet rs = pm.executeQuery();
            while (rs.next()) {
                GregorianCalendar cal = new GregorianCalendar();
                cal.setTime(rs.getDate("WEATHERDATE"));

                XMLGregorianCalendar xmlCal =
                    DatatypeFactory.newInstance().
                        newXMLGregorianCalendar(cal);

                result = new Weather();
                result.setDate(xmlCal);
                result.setCondition(rs.getString("CONDITION"));
                result.setTemperatureCelsius(rs.getInt("TEMPERATURE"));
                result.setWindDirection(rs.getString("WINDDIR"));
                result.setWindSpeed(rs.getInt("WINDSPEED"));
                result.setDbflag(true);
            }
        } catch (NamingException nmex) {
            nmex.printStackTrace(System.err);
        } catch (SQLException e) {
            e.printStackTrace(System.err);
            result = null;
        } catch (DatatypeConfigurationException dtypconfex) {
            dtypconfex.printStackTrace(System.err);
        } finally {
            try {
                if (pm != null)
                    pm.close();
                if (con != null)
                    con.close();
            } catch (SQLException ex) {
                ex.printStackTrace(System.err);
            }
        }
```

```
        return result;
    }
```

3. Organize imports by pressing Ctrl+Shift+O.

   For the code in Example 4-4 on page 203, choose the following classes to resolve the import declarations:

   – java.sql.ResultSet
   – javax.sql.DataSource
   – javax.xml.datatype.DatatypeFactory
   – java.sql.Date
   – java.sql.Connection

4. Save the file. All errors should be resolved.

5. In the Servers view, right-click the WebSphere Application server test environment and select **Publish**.

Configuring the enhanced EAR file

After implementing the EJB Web service, configure the JDBC data source settings by using the enhanced EAR file:

1. Right-click the Enterprise Application Project for the Web service (**WeatherForecastJMSEAR**) and select **Properties**.

2. In the Properties window, select **Project Facets**.

> **Important:** You must configure the Project Facets of WeatherForecastJMSEAR to access the WebSphere Application Server Deployment editor.

3. Select **WebSphere Applications (Co-existence)** and **WebSphere Applications (Extended)**. Click **Apply**, then click **OK**.

4. Configure the enhanced EAR file by using the WebSphere Application Server Deployment editor.

5. Configure a JDBC data source for the Derby database.

6. Save the enhanced EAR file.

7. In the Servers view, right-click the **WebSphere Application server test environment** and select **Publish**.

4.4.2 Testing a Web service with a synchronous client

The Web service wizard can develop both the Web service (server-side) and the client. The wizard creates the skeleton EJB Web service project and a client

project. It also provides an option to create a test client within the client project. The previous example did this (see "Running the Web service wizard" on page 200) and created JSP clients (because the client application is a Dynamic Web Project) that can be used to test the implemented Web service.

Running the Web Services Test Client JSPs

The Web Services Test Client for a client Web application is a JSP page created by the Web service wizard that is used to test the developed Web service. This page is in a Dynamic Web Project that can be used to test the Web service and create Web applications that can access the Web service.

The Web Services Test Client that was generated in 4.4.1, "Creating an EJB Web service" on page 198, performs a synchronous invocation of the EJB Web service, as illustrated in Figure 4-23.
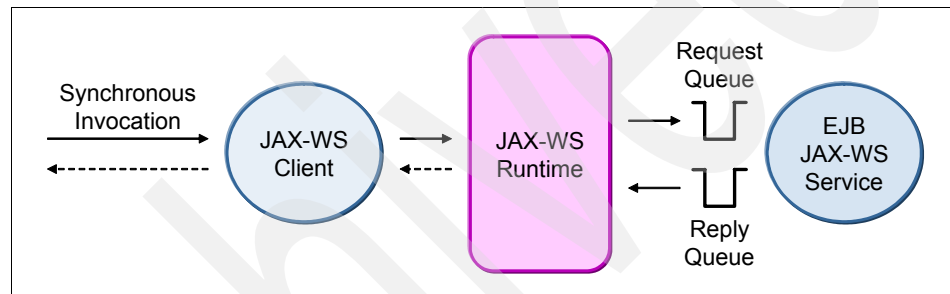
Figure 4-23   Synchronous invocation of the JMS Web service

To test the Web service by using the Web Services Test Client:

1. Select the Dynamic Web Project created by the Web service Wizard (**WeatherForecastWebClient**). Expand **WebContent** → **sampleWeatherForecastPortProxy**. Right-click **TestClient.jsp**. Select **Run As** → **Run on Server**.

2. In the Run On Server window, select **WebSphere Application Server v7.0** as the server to use. Click **Finish**.

3. In the Web Services Test Client (Figure 4-24), which is displayed in the main area:

   a. In the Methods pane, click the link for the **getDayForecast** operation.

   b. In the right pane, in the arg0 field, type `2006-01-07T00:00:00` as the parameter for the operation.

   c. Click **Invoke**.

> **Resulting value**: The values of the result should be the same as in the previous example (4.3.2, "Creating a Web service thin client" on page 194) because the same parameter is used to invoke the Web service operation.
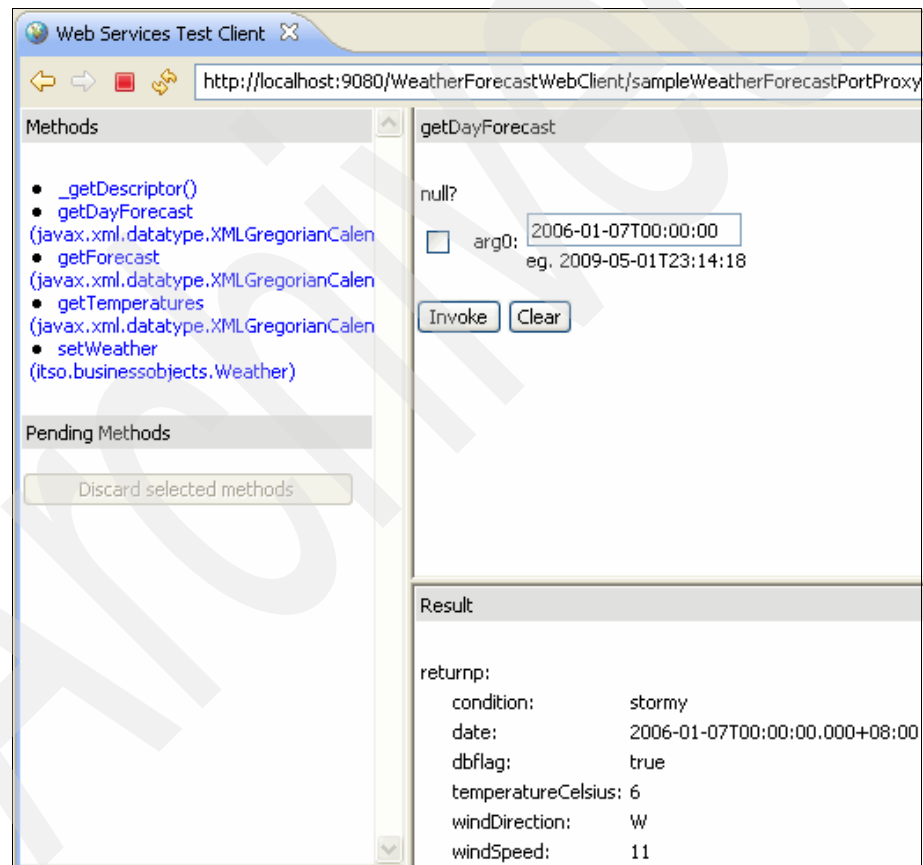
Figure 4-24   Web Services Test Client results

4.4.3 Creating an asynchronous client

The last example in 4.4.2, "Testing a Web service with a synchronous client" on page 205, used a Web application to do a synchronous invocation on a JMS Web service. This is forgivable in a testing environment but makes little sense in practice because the JMS transport is asynchronous by nature.

In the example shown in this section we create a client that uses the generated asynchronous methods of the client to invoke the JMS Web service, as illustrated in Figure 4-25.
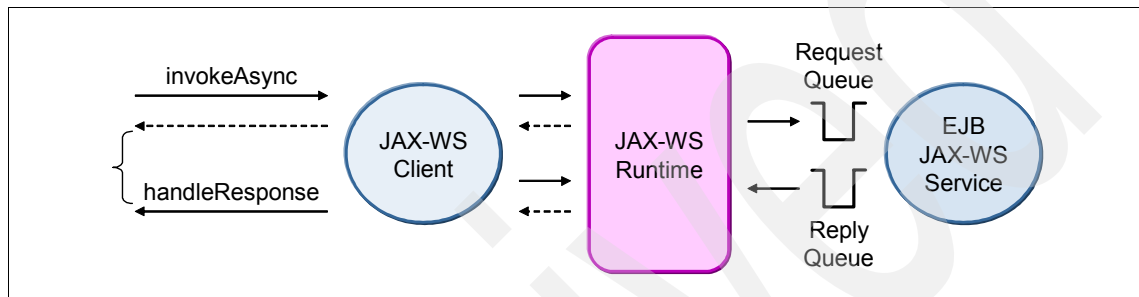
Figure 4-25   Asynchronous invocation of the JMS Web service

Development of an asynchronous EJB Web service client involves the following tasks:

1. Running the Web service wizard
2. Implementing the EJB Web service
3. Running the client application

We explain each of these tasks in the sections the follow.

Running the Web service wizard

In "Running the Web service wizard" on page 200, the WSDL used to generate the Web service artifacts is the same one that is used to generate the client application. The same actions are done here except that the Web service wizard is used to generate the client application.

To initiate the Web service wizard for client development only:

1. In Rational Application Developer, switch to the **Java EE** perspective if it is not already open.

2. On the client's Dynamic Web Project (`WeatherForecastWebClient`), expand **WebContent** → **WEB-INF** → **wsdl**, right-click the **WeatherForecastService.wsdl** file, and select **Web services** → **Generate Client**.

The WSDL used here is in the Dynamic Web Project that was created by using the Web service wizard. This WSDL has been updated by the Web service wizard with a JMS binding for the location attribute of the `<soap:address>` element.

3. In the Web service wizard (Figure 4-26), specify the properties for a client configuration. Under Configuration, for client project (use Application Client Project) type `WeatherForecastJMSClient,` and for client EAR project type `WeatherForecastJMSClientEAR`.

> **Application client project:** An application client project is a standalone application that acts as an unmanaged client. Application client projects are supported in the Java EE 5 specification and are packaged into an EAR file similar to Dynamic Web Projects and EJB projects. Rational Application Developer has the tools to develop and test application client projects.

Click **Next**.
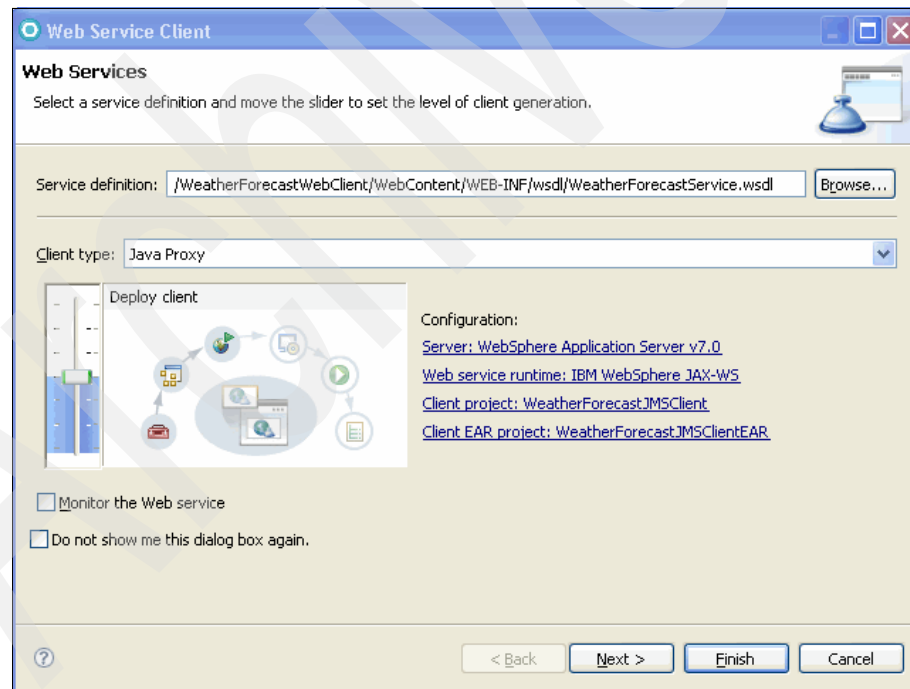
Figure 4-26   The Web service wizard (client only)

4. In the WebSphere JAX-WS Web service Client Configuration window, select **Enable asynchronous invocation for generated client**.

JAX-WS allows for the creation of clients that are able to invoke Web services in an asynchronous manner. This type of client is best suited for JMS Web services since the execution model of message-based services is an asynchronous one.

Because we selected the asynchronous client option, the Web service wizard has created *<operation_name>*Async() methods for the generated proxy class, as shown in Figure 4-27.

Click **Finish**.

Figure 4-27   Asynchronous methods

Implementing the EJB Web service

After the Web service wizard executes and the skeleton code for the client is created, the business logic of the client application is written to the client project.

To write the client program in the application client project:

1. In the application client project (WeatherForecastJMSClient), expand **src** →
   (default package) and double-click **Main.java**.

2. Implement the client into the main(String[] args) method of Main.java.

   The code in Example 4-5 is for the main method of the Java class of the
   application client. You can also find this code in the
   WeatherWSTest_JMS_snippet.txt file that is included in the ch04_app_dev.zip
   file.

   The program invokes the selected operation asynchronously by using the
   getDayForecastAsync() method. The async method returns a response object
   that can be polled while the client's operation continues (in this case,
   displaying a string repeatedly). When the Web service places a reply on
   queue, the polling loop ends and the response is retrieved.

   Errors that result because of unresolved type declarations are expected and
   should be ignored in the meantime.

Example 4-5   The main(String[] args) method of Main.java

```
public static void main(String[] args) {
    try {
        WeatherForecastPortProxy proxy =
            new WeatherForecastPortProxy ();
        javax.xml.ws.Response<GetDayForecastResponse> resp =
            proxy.getDayForecastAsync(DatatypeFactory.
                newInstance().newXMLGregorianCalendar(
                    "2006-01-07T00:00:00"));
        // Poll for the response.
        while (!resp.isDone()) {
            System.out.println(
                "Waiting for asynchronous response...");
            // Wait for 0.2 seconds.
            Thread.sleep(200);
        }
        GetDayForecastResponse gdfr = resp.get();
        Weather weather = gdfr.getReturn();
        System.out.println(
            "getDayForecast async invocation complete.");
        System.out.print("The weather is " +
            weather.getCondition() + " ");
        System.out.print("with temperature at " +
            weather.getTemperatureCelsius() + " deg. C ");
        System.out.print("and winds gusting at " +
            weather.getWindSpeed() + " MPH ");
```

```
                    System.out.println("from a direction of " +
                        weather.getWindDirection() + ".");
                } catch (InterruptedException e) {
                    System.out.println(e.getCause());
                } catch(DatatypeConfigurationException dtypeconfex) {
                    dtypeconfex.printStackTrace();
                } catch(ExecutionException execex) {
                    execex.printStackTrace();
                }
            }
```

3. Organize the imports by pressing Ctrl+Shift+O.

4. For the code in Example 4-5 on page 211, select
   javax.xml.datatype.DatatypeFactory to resolve the import declarations.

5. Save the file. All errors should be resolved.

Running the client application

Application client projects for WebSphere Application Server run on a specialized
environment called the *WebSphere Application Server client runtime or
application client*. This client is packaged together with the WebSphere
Application Server and can recognize deployment of an EAR file.

Rational Application Developer has an embedded client run time in the IDE that
can be configured by using the Run Configurations window.

To run the client application:

1. In the Application Client Project, expand **src** → **(default package)**, right-click
   Main.java, and select **Run As** → **Run Configurations**.

2. In the Run Configurations window (Figure 4-28), select **WebSphere Application Server v7.0 Application Client**. From the top row of icons, select **New launch configuration**.
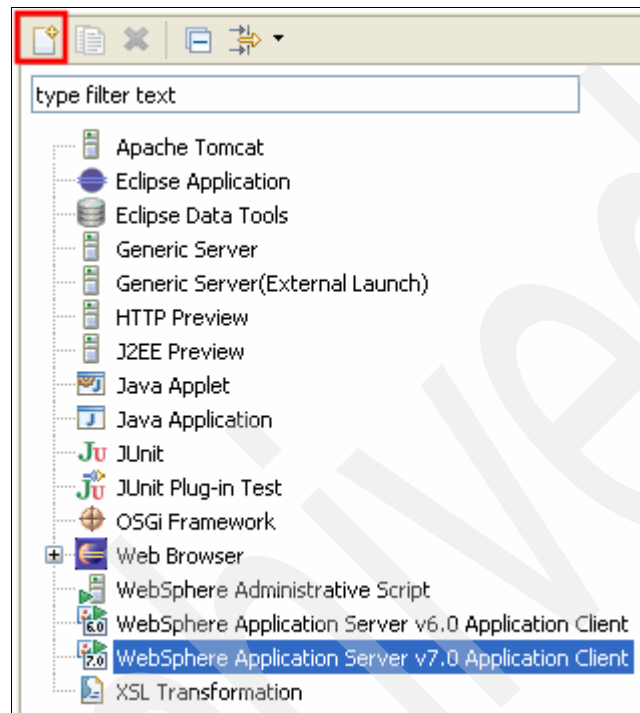
Figure 4-28   Launch configuration

3. In the Name field, type `WeatherForecastJMSClient` as the new launch configuration name.

4. Click **Apply**.

5. Click **Run**.

The results are displayed in the Console view (Figure 4-29), based on the data retrieved from the weather database.

> **Resulting values**: The values of the result should be the same as in the previous example because the same parameter is used to invoke the Web service operation.

```
WSCL0911I: Component initialized successfully.
WSCL0901I: Component initialization completed successfully.
WSCL0035I: Initialization of the Java EE Application Client Environment has completed.
WSCL0014I: Invoking the Application Client class Main
Retrieving document at 'file:/C:/Documents and Settings/Administrator/My Documents/workspace/.metadat
Retrieving schema at 'WeatherForecastService_schema1.xsd', relative to 'file:/C:/Documents and Settin
Waiting for asynchronous response...
Waiting for asynchronous response...
Waiting for asynchronous response...
Waiting for asynchronous response...
Waiting for asynchronous response...
Waiting for asynchronous response...
Waiting for asynchronous response...
Waiting for asynchronous response...
Waiting for asynchronous response...
getDayForecast async invocation complete.
The weather is stormy with temperature at 6 deg. C and winds gusting at 11 MPH from a direction of W.
```

Figure 4-29   Asynchronous client results

4.5 Testing and monitoring Web services

Rational Application Developer includes the capability to test and monitor Web services during its execution.

4.5.1 The Web Services Explorer

Rational Application Developer uses the Web application *Web Services Explorer* tool to test deployed Web services. This tool executes a Web service's operations and displays the results. The tool offers a better alternative for the service provider than creating a client just to test the developed service.

To run the Web Services Explorer in Rational Application Developer:

1. From the main menu, select **Run** → **Launch the Web Services Explorer**.

> **Alternative for launching Web Services Explorer**: You can also run the
> Web Services Explorer from the Services view. Right-click a listed Web
> service and select **Test with Web Services Explorer**.

2. In the Web Services Explorer view, which opens in the main area
    (Figure 4-30), in the upper right corner, click the WSDL Page icon to use a
    WSDL service definition to locate the Web service (inset in Figure 4-30).
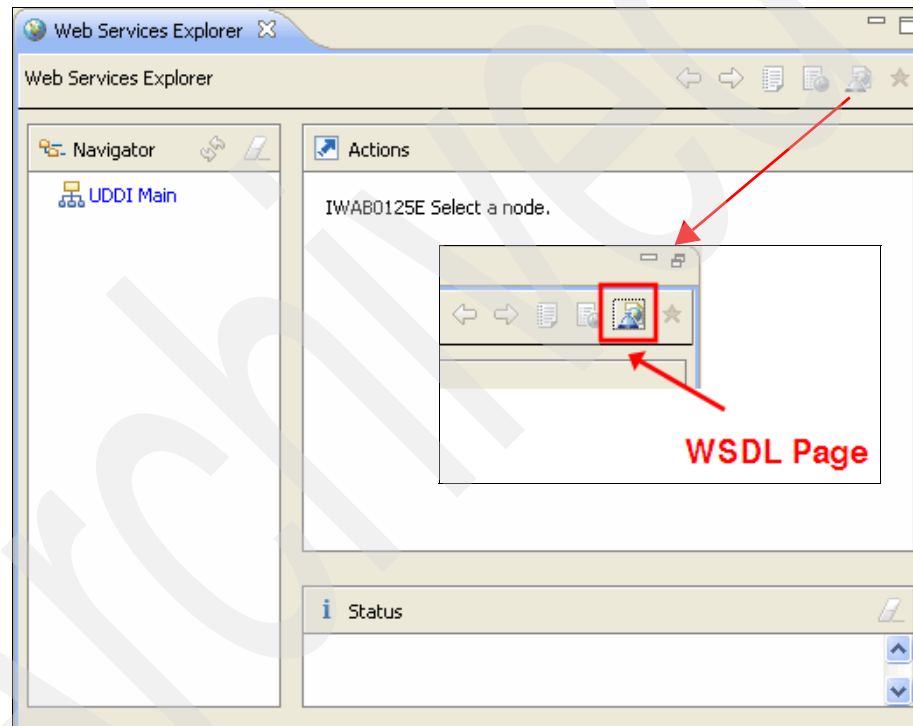
Figure 4-30   Web Services Explorer

3. In the Navigator pane, click the **WSDL Main** link. In the main area, in the
    Actions pane, under Open WSDL, click the **Browse** link (Figure 4-31).
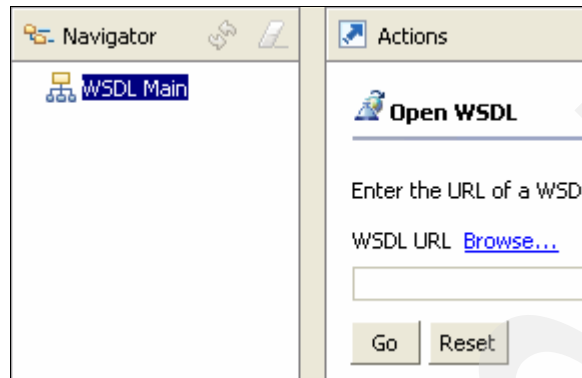
Figure 4-31   Browsing for a WSDL file

4. In the WSDL Browser dialog box (Figure 4-32), for Workspace Projects,
    select a project on the workspace that contains the WSDL of the service to be
    tested. Click **Go**.
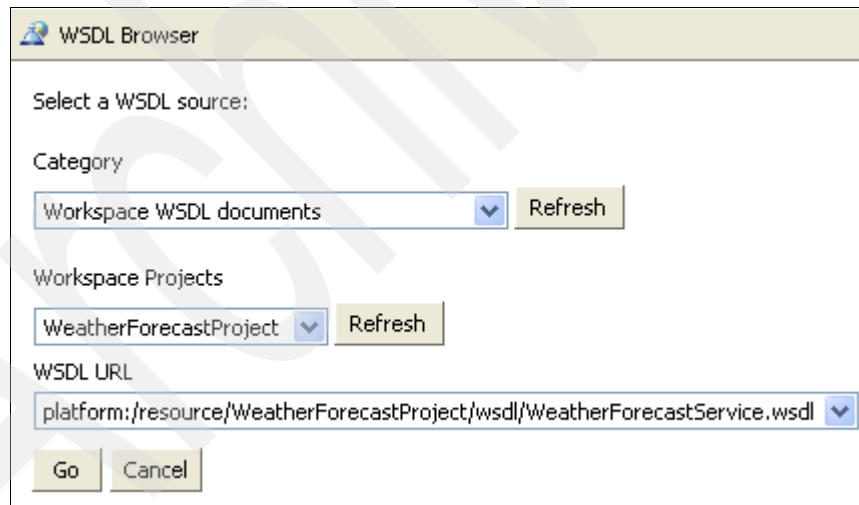
Figure 4-32   WSDL Browser dialog box

5. In the main area (Actions pane), click **Go**.

Test the operations of the selected Web service:

1. In the Actions pane, for WSDL Binding Details (Figure 4-33), under the listing of operations of the Web service, click an operation.

> **Endpoints field URL value**: The Endpoints field should have a correct URL value. If not, you can add a new endpoint value, which is similar to modifying the URL values in the service client class in "Writing the client application" on page 194.
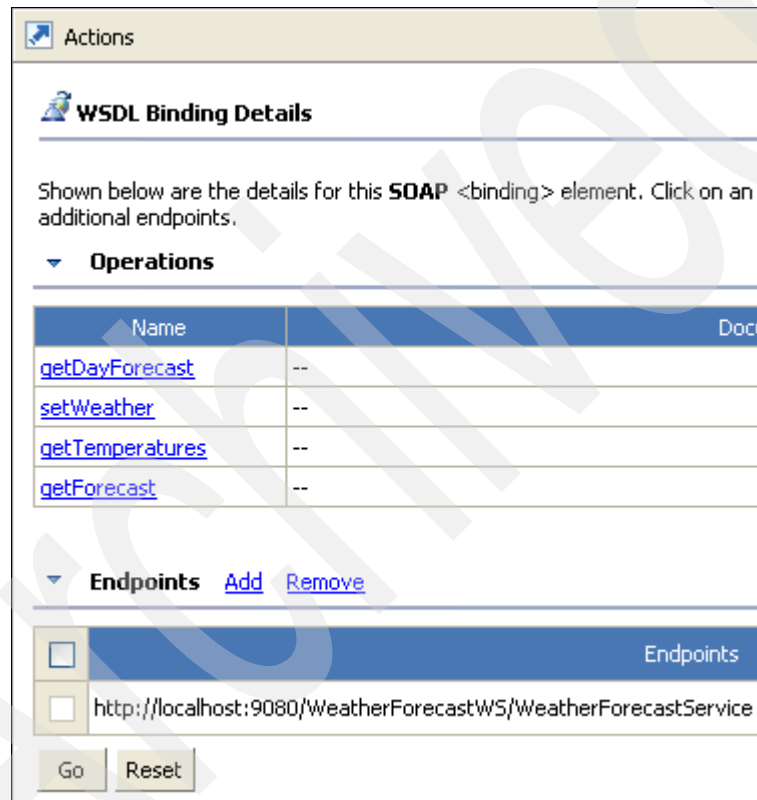
Figure 4-33   WSDL Binding Details in the Actions pane

2. On the Invoke a WSDL Operation page (Figure 4-34), notice that the URL entered is displayed under Endpoints. Under Body, enter the parameters of a Web service operation, then click **Go**.

Figure 4-34   Invoking a WSDL operation

The results are displayed in the Status pane (Figure 4-35).
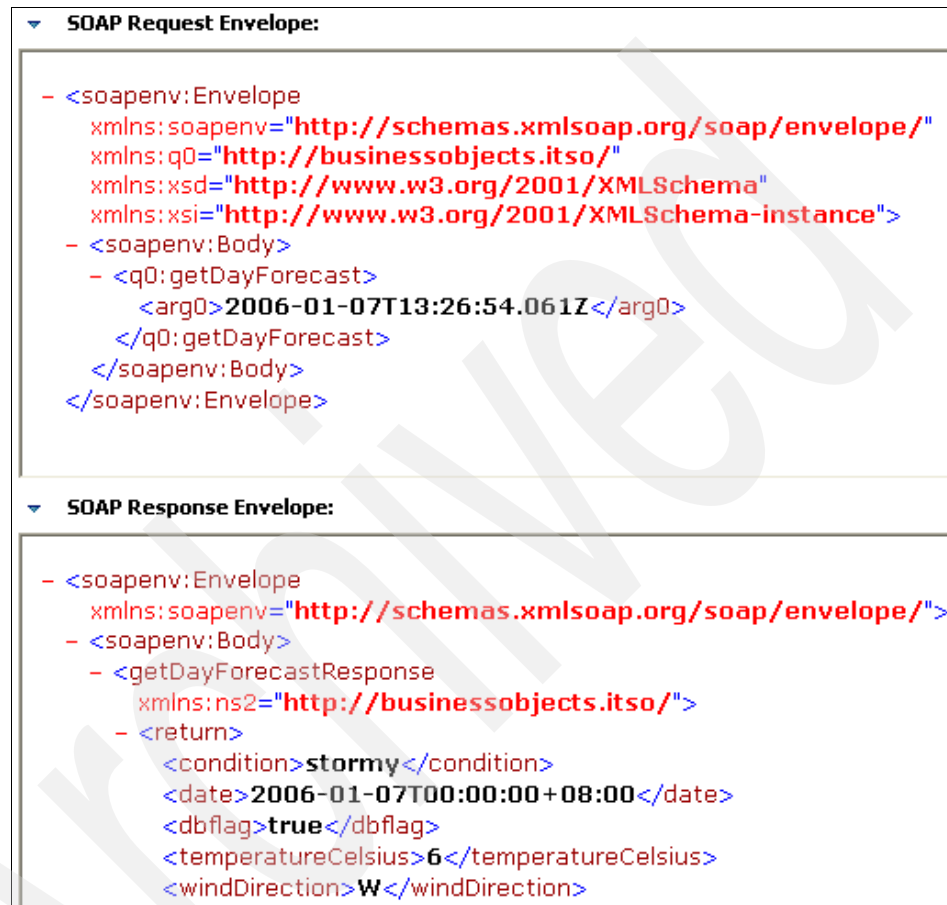
Figure 4-35   Results of the WSDL operation

3. On the Invoke a WSDL Operation page (Figure 4-34 on page 218), click the **Source** link.

You can view the request and response SOAP messages (Figure 4-36).

Figure 4-36   Viewing the SOAP messages

4.5.2 The TCP/IP Monitor

The TCP/IP Monitor is a feature of Rational Application Developer that allows the real-time inspection of requests and responses between the client and the server. You can use TCP/IP Monitor to monitor the SOAP messages of the deployed Web service as it interacts with a client.

The TCP/IP Monitor operates by redirecting the request message intended for the Web service through the configured port of the TCP/IP Monitor. The TCP/IP

Monitor then sends the request message through the port that is designated for the Web service and relays the response back to the client. Figure 4-37 illustrates this process.
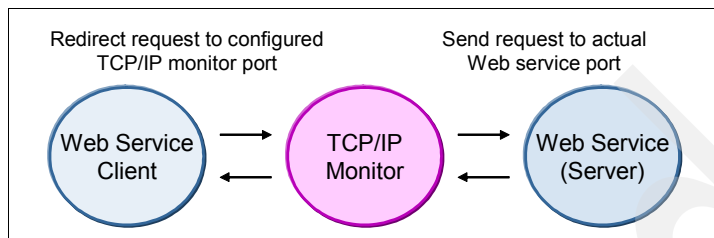
Figure 4-37   TCP/IP Monitor operation

To use the TCP/IP Monitor, you must modify the client to use the *local monitoring port* of the TCP/IP Monitor. In Rational Application Developer, you do this task during the development of the Web service client by selecting the **Monitor the Web service** check box in the Web Service Wizard for clients, as shown in Figure 4-38. This option configures the client's sending port and the local monitoring port of the TCP/IP Monitor to have the same value.
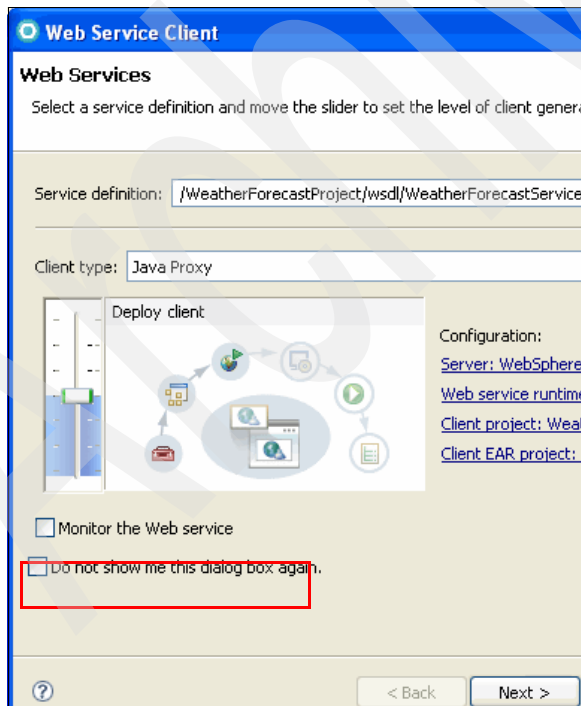
Figure 4-38   Monitor the Web service check box

When the generated client executes, the SOAP messages that it sends are redirected to the TCP/IP Monitor. Figure 4-39 shows the TCP/IP Monitor view of Rational Application Developer.

> **Request and Response panes**: You can set the Request and Response panes in the TCP/IP Monitor view for XML to properly view the SOAP messages of Web services.
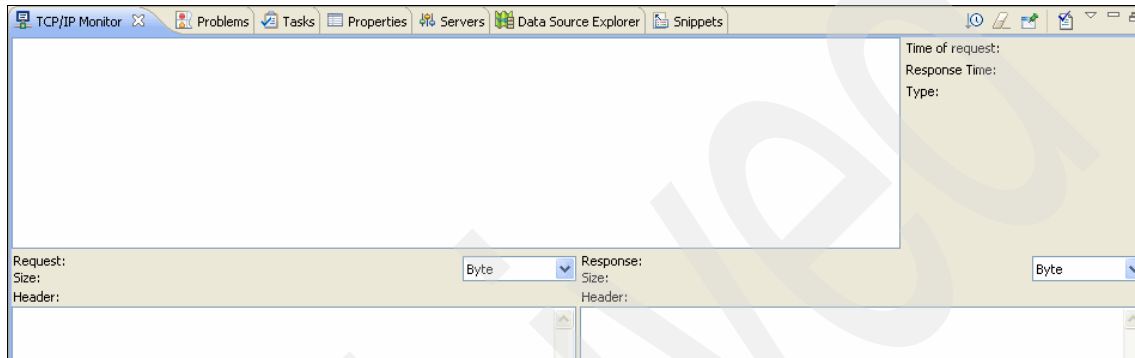
Figure 4-39   TCP/IP Monitor view

You can also use TCP/IP Monitor for existing (already developed) Web service clients. However, you must modify them at the source code level to use the local monitoring port.

> **Note:** To modify an existing Web service client's sending port, you must have a deep understanding of the JAX-WS API for clients. This is demonstrated in Chapter 2, "Web services programming model" on page 59 (see Example 2-28 on page 93).

To configure the TCP/IP Monitor for an existing client:

1. In Rational Application Developer, select **Window** → **Show View** → **Other**.

2. In the Show View dialog box (Figure 4-40), in the type filter text field, type `TCP/IP Monitor`. Select **TCP/IP Monitor** and click **OK**.
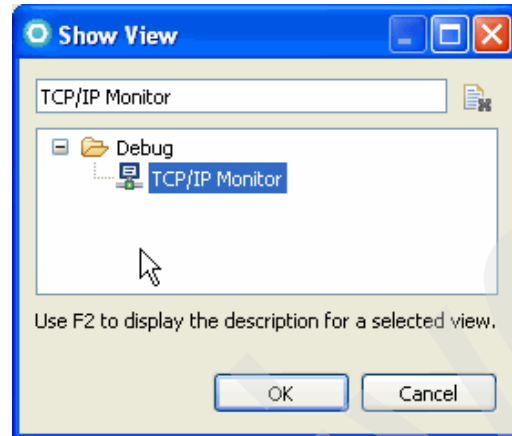
Figure 4-40   Show View dialog box

3. In the TCP/IP Monitor view in the workspace of Rational Application Developer, right-click the top pane and select **Properties**.

4. In the Preferences dialog box (Figure 4-41), click the **Add** button.
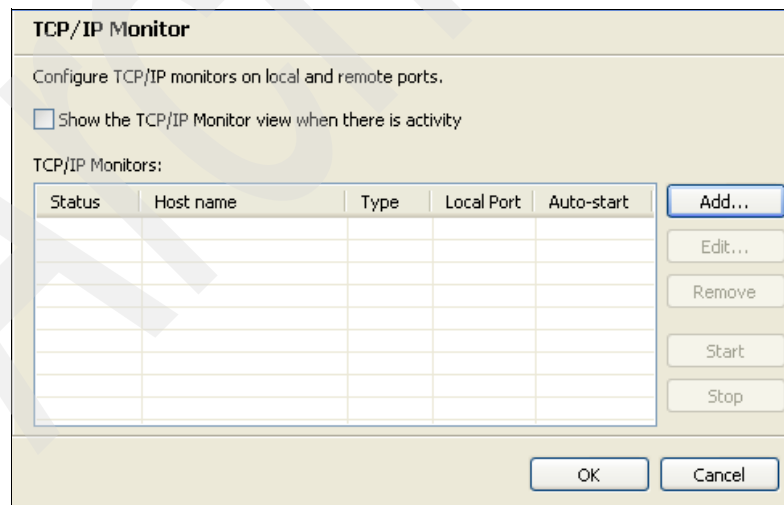
Figure 4-41   TCP/IP Monitor preferences

5. In the New Monitor dialog box (Figure 4-42), for Local monitoring port, enter a value that is the same as the one used by the existing client. For host name enter the name of the server where the service is deployed, and for port type the port of the service. Click **OK**.
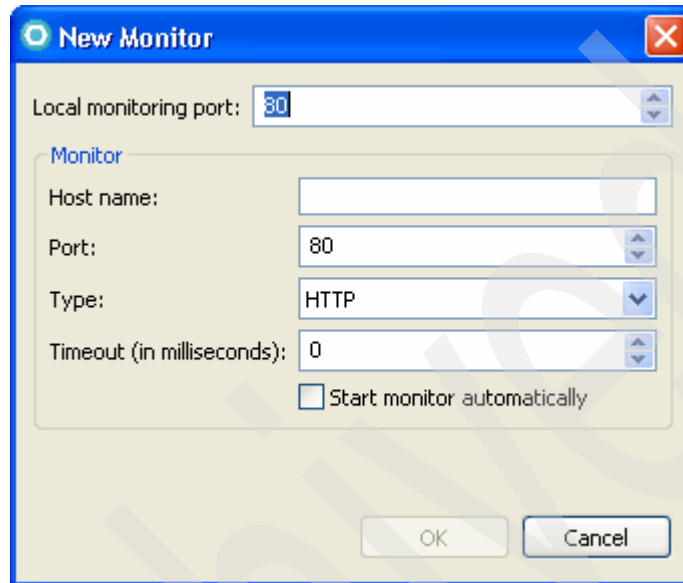
Figure 4-42   New Monitor dialog box

6. Back in the Preferences dialog box, select the new monitor and click the **Start** button.

5

Web services administration

In this chapter we discuss the aspects of WebSphere Application Server V7.0 system administration that relate specifically to Web services. We begin with the basic concept of WebSphere Application Server system administration. Then we discuss the task of deploying Web services and configuring them for the runtime environment. We also explain the configuration of WebSphere Application Server runtime resources. Finally, we discuss the testing and monitoring of Web services.

This chapter contains the following topics:

5.1 WebSphere Application Server administration

WebSphere Application Server system administration involves the management of resources that allow the server run time to maintain operations. For this purpose, WebSphere Application Server exposes interfaces through which system administrators can perform their tasks.

Note: This chapter only touches on the WebSphere Application Server administration concepts. The intent is to introduce concepts to WebSphere administrators that are specific to Web service applications and to application developers that might be dealing with a test environment.

For this reason, we assume that we are dealing with a single-server environment. However, these concepts can be applied to a deployment manager in a distributed server environment.

5.1.1 Administrative facilities

WebSphere Application Server provides the following general facilities for system administration:

► Commands

WebSphere Application Server provides a set of commands that perform general system administration tasks. These tasks include starting and stopping the server.

► Administrative console

The administrative console is the most common interface used for WebSphere Application Server system administration. It is a secure Web application that is accessed through a Web browser.

► `wsadmin`

`wsadmin` is the system administration command interpreter for WebSphere Application Server. It is used primarily for automating system administration tasks using scripts.

5.1.2 Administration basics

WebSphere Application Server system administration requires basic skills for operating the server run time. Regardless of role, all users are expected to know the basic operational commands and facilities of WebSphere Application Server.

WebSphere Application Server organizes runtime resources around the concept of a *node*. The node is used as the building block for robust systems with a flexible architecture because a single node can operate in a *standalone* capacity or be grouped together into a *cell*. Administrative functions, consequently, are focused on the *node*. The discussions in this chapter assume a standalone application server configuration on a single node.

Basic operational commands

When WebSphere Application Server is installed, a node is created with one server instance, server1, which is used as the default application server. The node and all its administrative facilities are in the installation directory. The installation directory varies depending on the platform or what was specified by the user during installation, but it is referred to here as *WAS_HOME*.

To start the WebSphere Application Server default server, type the following command:

```
startServer server1
```

To stop the WebSphere Application Server, type the following command:

```
stopServer server1
```

Situations can arise when a user might need to determine whether the default server is currently running. In this case, you can type the following command:

```
serverStatus server1
```

These commands are in the *WAS_HOME/bin* directory.

> **Administrative security**: If administrative security is enabled, the -username and -password options can be used with these commands for authentication.

The log files

WebSphere Application Server records its internal operations in log files that are viewed for monitoring or troubleshooting purposes. For the default server of the node named *node_name*, these files are in the
WAS_HOME/profiles/*node_name*/logs/server1 directory.

The general output of the WebSphere Application Server run time can be viewed in the `SystemOut.log` file. Figure 5-1 shows a sample of content from this file.

```
************* Start Display Current Environment *************
WebSphere Platform 7.0.0.0 [ND 7.0.0.0 r0835.03] running with process n
Host Operating System is Windows XP, version 5.1 build 2600 Service Pac
Java version = 1.6.0, Java Compiler = j9jit24, Java VM name = IBM J9 VM
was.install.root = C:\WAS70
user.install.root = C:\WAS70\profiles\AppSrv01
Java Home = C:\WAS70\java\jre
ws.ext.dirs = C:\WAS70/java/lib;C:\WAS70\profiles\AppSrv01/classes;C:\w
Classpath = C:\WAS70\profiles\AppSrv01/properties;C:\WAS70/properties;C
Java Library path = C:\WAS70\java\jre\bin;.;C:\WAS70\bin;C:\WAS70\java\
************* End Display Current Environment *************

[5/4/09 8:59:20:968 CST] 0000000c EJBContainerI I   WSVR0057I: EJB jar
[5/4/09 8:59:21:062 CST] 0000000b webapp         I com.ibm.ws.webcontain
[5/4/09 8:59:21:156 CST] 0000000b WASSessionCor I SessionContextRegistr
[5/4/09 8:59:21:296 CST] 0000000d MBeanDescript I   ADMN1216I: One or m
[5/4/09 8:59:21:390 CST] 0000000d EJBContainerI I   WSVR0037I: Starting
[5/4/09 8:59:21:421 CST] 0000000d EJBContainerI I   CNTR0167I: The serv
[5/4/09 8:59:21:453 CST] 0000000b MBeanDescript I   ADMN1216I: One or m
[5/4/09 8:59:21:468 CST] 0000000d EJBContainerI I   WSVR0057I: EJB jar
[5/4/09 8:59:21:703 CST] 0000000b webcontainer  I com.ibm.ws.wswebconta
[5/4/09 8:59:21:718 CST] 0000000d webapp         I com.ibm.ws.webcontain
[5/4/09 8:59:21:750 CST] 0000000d WASSessionCor I SessionContextRegistr
[5/4/09 8:59:21:843 CST] 0000000d servlet        I com.ibm.ws.webcontain
[5/4/09 8:59:22:062 CST] 0000000d servlet        I com.ibm.ws.webcontain
[5/4/09 8:59:22:109 CST] 0000000d servlet        I com.ibm.ws.webcontain
[5/4/09 8:59:22:125 CST] 0000000d webcontainer  I com.ibm.ws.wswebconta
[5/4/09 8:59:22:234 CST] 0000000b webapp         I com.ibm.ws.webcontain
[5/4/09 8:59:22:281 CST] 0000000b WASSessionCor I SessionContextRegistr
[5/4/09 8:59:22:359 CST] 0000000d ApplicationMg A   WSVR0221I: Applicat
[5/4/09 8:59:22:359 CST] 0000000d CompositionUn A   WSVR0191I: Composit
[5/4/09 8:59:22:593 CST] 0000000b webcontainer  I com.ibm.ws.wswebconta
[5/4/09 8:59:22:640 CST] 0000000c webapp         I com.ibm.ws.webcontain
[5/4/09 8:59:22:734 CST] 0000000c WASSessionCor I SessionContextRegistr
```

Figure 5-1   SystemOut.log sample

Opening the administrative console

Most system administration tasks are done through the administrative console, which can be accessed through a Web browser. When the WebSphere Application Server is running, the administrative console is available by entering the following URL:

`http://localhost:9060/ibm/console`

Note: If administrative security is enabled, the login page requires a user name and a password. If administrative security is not enabled, it requires only a user name for logging purposes.

The port number will vary for your environment. If in doubt, you can access the console from the First Steps menu the first time. You can open the menu by entering the following command:

WAS_HOME/profiles/*node_name*/firststeps/firststeps.bat (.sh)

Using wsadmin

To automate system administration functions, WebSphere Application Server provides the `wsadmin` script interface. The `wsadmin` facility uses the *Jython* script language to run administrative tasks as an interactive shell (default), inline on the console with the -c option, or through a script file with the -f option. Rational Application Developer has the ability to edit, debug, and run Jython administrative scripts for the application server.

> **Important:** You *must* ensure that the application server is started before you use the `wsadmin` interface.

5.2 Web services deployment

The Java Platform Enterprise Edition (EE) 5 standard includes support for the Web service specifications. As such, both the Web service server components and their clients can be deployed into Java EE 5 run times such as WebSphere Application Server V7.0. To do this, Web services must be packaged into Java EE deployment modules and installed by using the enterprise application deployment process.

The standard deployment unit for Java EE 5 enterprise applications, including WebSphere Application Server, is the enterprise archive (EAR) file. The EAR file is a compressed file that includes the component modules of an enterprise application.

The standard EAR file consists of the following major modules:

► Web module

This module contains the resources of a Web application including servlets and JavaServer Pages (JSP). It is packaged into a Web archive (WAR) file.

► EJB module

This module contains Enterprise JavaBeans (EJB) components. It is packaged into an EJB Java archive (EJB-JAR) file.

► Application client module

This module contains a standalone application that can access server components. It runs on a client run time and is packaged into a JAR file.

Table 5-1 shows how Web service-related components are packaged and deployed in an EAR file.

Table 5-1   Web service components packaging

Web service component	Deployment submodule	Main deployment module	Runtime environment
Web service (server)	WAR	EAR	Application server
EJB Web service (server)	EJB-JAR	EAR	Application server
Web service Web client	WAR	EAR	Application server
Web service EJB client	EJB-JAR	EAR	Application server
Web service application client	JAR	EAR	Client run time

Hint: Not all Web service components must be packaged in the same EAR deployment file.

Figure 5-2 illustrates how Web-service-related components are packaged.
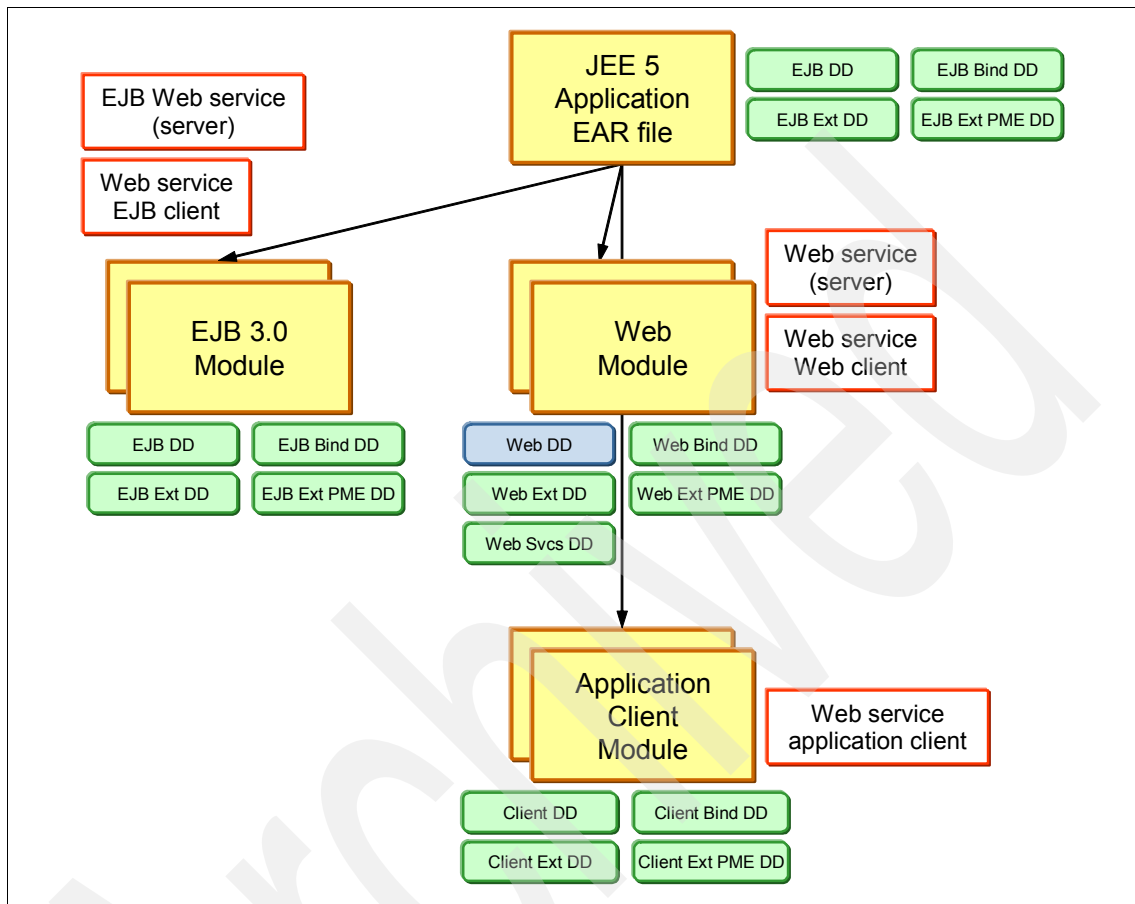
Figure 5-2   Web service packaging

Managing deployed Web services

After the EAR file is assembled, Web services are treated by WebSphere
Application Server similarly to any other enterprise application. The most
common facility for managing enterprise applications in WebSphere Application
Server is the administrative console. In this section we describe the
administrative console and its features for managing enterprise applications and
Web services.

Deploying a Web service EAR file

Management of enterprise applications begins with the installation or deployment of the EAR file into the application server. To deploy an enterprise application using the administrative console take the following steps.

> **Standard versus complex deployment**: The following steps are for a standard deployment and use default options for the process. Complex deployments might require you to use different options depending on the situation.

1. Using a Web browser, open the administrative console and log in. In the left navigation pane, select **Applications** → **New Application**. In the right pane, click **Install** to install a new application.

2. On the Path to the new application page, select **Local file system** and click **Browse**. Browse for the EAR file to be deployed. Click **Next**.

3. On the How do you want to install the application? page, select **Fast Path**. Click **Next**.

4. On the Select installation options page, accept the default values and click **Next**.

> **Deploy Web services option**: On the Select installation options page, selecting the Deploy Web services option ensures that the `wsdeploy` command is run against the EAR file that is being deployed. Selecting this option makes the JAX-RPC Web services in the application compatible with the WebSphere Application Server run time. Because most of the examples used in this book use the new Java API for XML Web Services (JAX-WS) standard, we do not use this option.

5. On the Map modules to servers page, accept the default values and click **Next**.

6. On the Summary page, click **Finish**.

7. After you see the `Application` *name* `installed successfully` message, click the **Save** link to commit the changes to the master configuration (Figure 5-3).
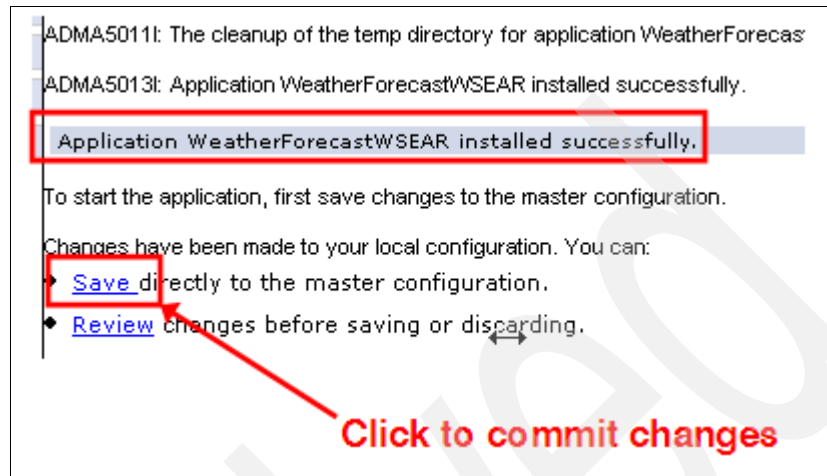
Figure 5-3   Successful installation

Saving changes to the master configuration: In WebSphere Application Server, you must commit any configuration changes made through the administrative console by saving the settings to the master configuration. You must do this for all areas of WebSphere Application Server that are managed through the administrative console.

The Enterprise Applications page

When deployed, the enterprise application is listed on the Enterprise Applications page of the administrative console (Figure 5-4). You can access this page by navigating to **Applications** → **Application Types** → **WebSphere enterprise applications**. The following functions are some of the more relevant enterprise application management functions that are available:

Start	Makes the enterprise application active
Stop	Disables the enterprise application
Install	Operates the same as selecting **Applications** → **New Application**
Uninstall	Removes the enterprise application
Update	Updates portions of the enterprise application
Export	Assembles the enterprise application EAR file for export
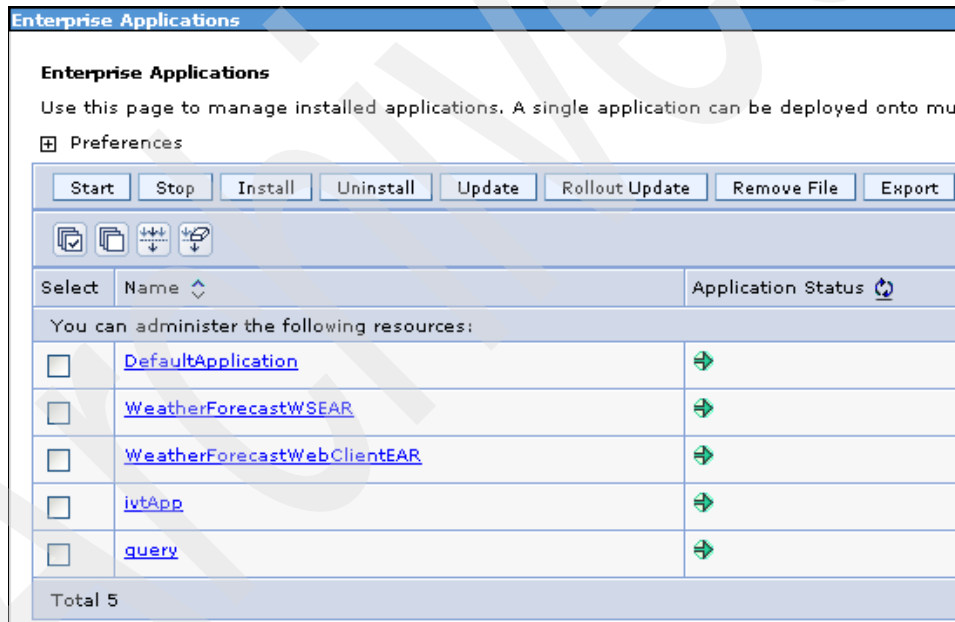
Figure 5-4   Enterprise Applications collection

Starting a deployed enterprise application

To start an application by using the management functions of the Enterprise Application page:

1. In the left navigation pane, select **Applications** → **Application Types** → **WebSphere enterprise applications**.

2. In the right pane (Figure 5-5), select the enterprise application to start and click the **Start** button.
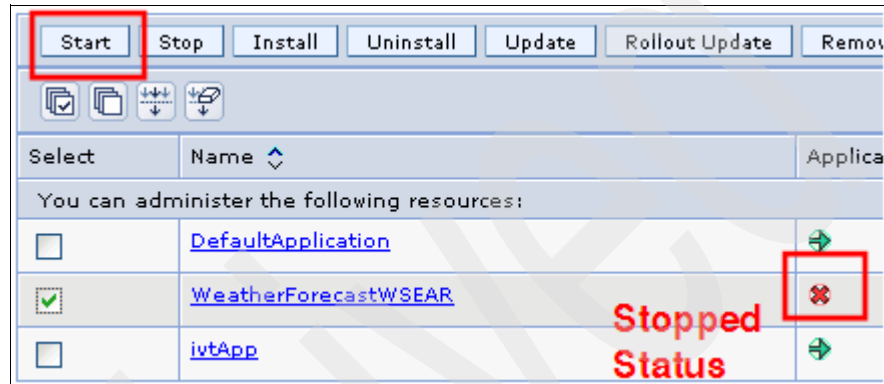
Figure 5-5   Starting the application

3. Wait for a message indicating that the application has started successfully (Figure 5-6).
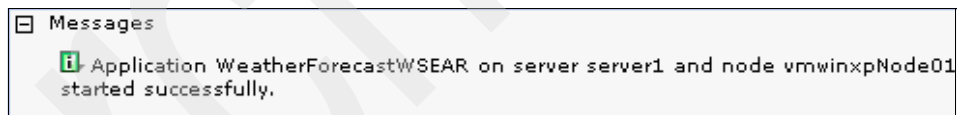
Figure 5-6   Successful start message

Uninstalling an enterprise application

To uninstall an enterprise application and remove it from WebSphere Application Server:

1. In the left navigation bar, select **Applications** → **Application Types** → **WebSphere enterprise applications**.

2. In the right pane, select the enterprise application to be uninstalled and click the **Uninstall** button in the top row of the collection.

3. Click **OK** to confirm the removal of the enterprise application.

4. Click the **Save** link to commit the changes to the master configuration.

5.3 Web services configuration

With WebSphere Application Server administrative facilities, the deployment and management of Web services components (server and client) and the configuration of the properties of these components are allowed. As in the previous section, the administrative console is used to illustrate the capabilities of WebSphere Application Server regarding this aspect.

5.3.1 Configuring Web service server-side settings

To configure the settings for Web services, the administrator must drill down to a specific enterprise application to access the enterprise application settings page (Figure 5-7). To open the enterprise application settings page:

1. In the left navigation pane of the administrative console, click **Applications** → **Application Types** → **WebSphere enterprise applications**.

2. In the right pane, click the link that represents an enterprise application. See the highlighted box in Figure 5-7.

Figure 5-7   Application settings for a Web service (server)

Web services configuration items

The application settings page of the WebSphere Application Server's administrative console has links to facilities that are used to modify the following Web services properties:

► Service providers
► Service provider policy sets and bindings
► Reliable messaging state
► Provide endpoint URL information
► Publish WSDL files

Service providers

The Service providers link leads to a page that lists all service providers (Web services) that are deployed in the application (Figure 5-8). This page provides administrators with the option to start or stop the listeners of the listed service providers.

Figure 5-8   Service providers

Service provider policy sets and bindings

The Service provider policy sets and bindings link references a page that lists the components for a particular service provider (Web service). See Figure 5-9. On this page, you can attach or detach policy sets and assign bindings. Chapter 6, "Policy sets" on page 261, describes these functions in more detail.

Figure 5-9   Service provider policy sets and bindings

Reliable messaging state

The Reliable messaging state link opens a page where administrators can manage the WS-ReliableMessaging properties of the Web service (Figure 5-10).
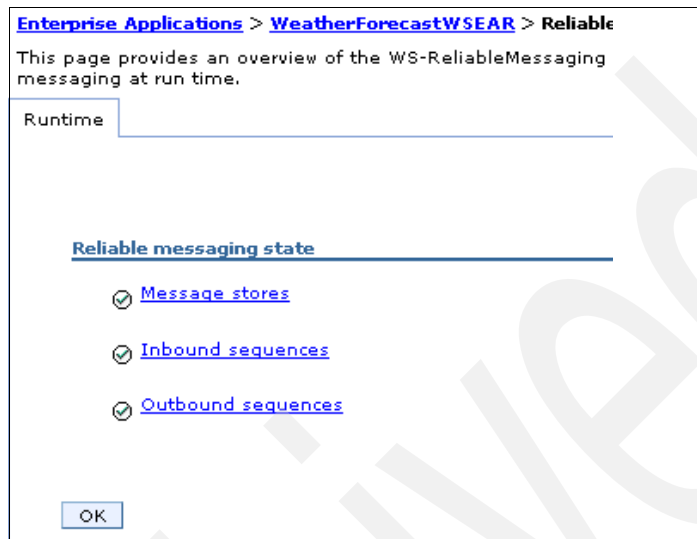
Figure 5-10   Reliable messaging state

Provide endpoint URL information

The application settings page provides two links that are used to manage endpoint URLs for Web services—one link is for HTTP endpoints and the other link is for Java Message Service (JMS) or EJB endpoints. With both links, administrators can modify the endpoint URL for a particular Web service.

To access the application settings, select **Applications** → **Application types** → **WebSphere enterprise applications**. Click the application name to open the configuration page. The links are in the Web Services Properties section.

By clicking the **Provide JMS and EJB endpoint URL** information link, you access the page for JMS and EJB endpoint URL information (Figure 5-11).

Figure 5-11   Provide JMS and EJB endpoint URL information

By clicking **Provide HTTP endpoint URL information**, you access the page for HTTP endpoint URL information (Figure 5-12).

Figure 5-12   Provide HTTP endpoint URL information

Publish WSDL files

The Publish WSDL files link leads to a page that allows administrators to download the published WSDL files of a Web service. Published WSDL files are often used by application developers in the Web service development process. Figure 5-13 shows an illustration of this page.
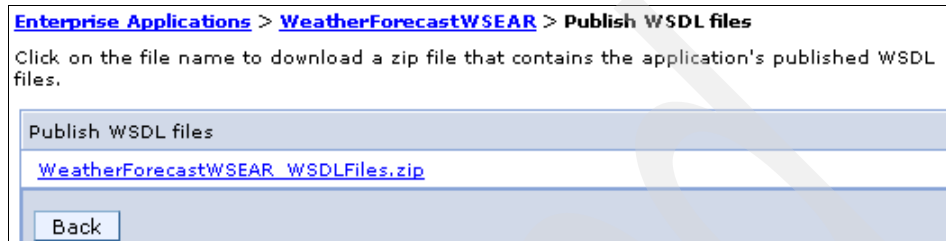
Figure 5-13   Publish WSDL files

5.3.2 Configuring Web service client settings

Configuring the settings for Web services clients provides similar functions but with fewer options. Selecting an enterprise application with Web service client components leads to the application settings page (Figure 5-14).
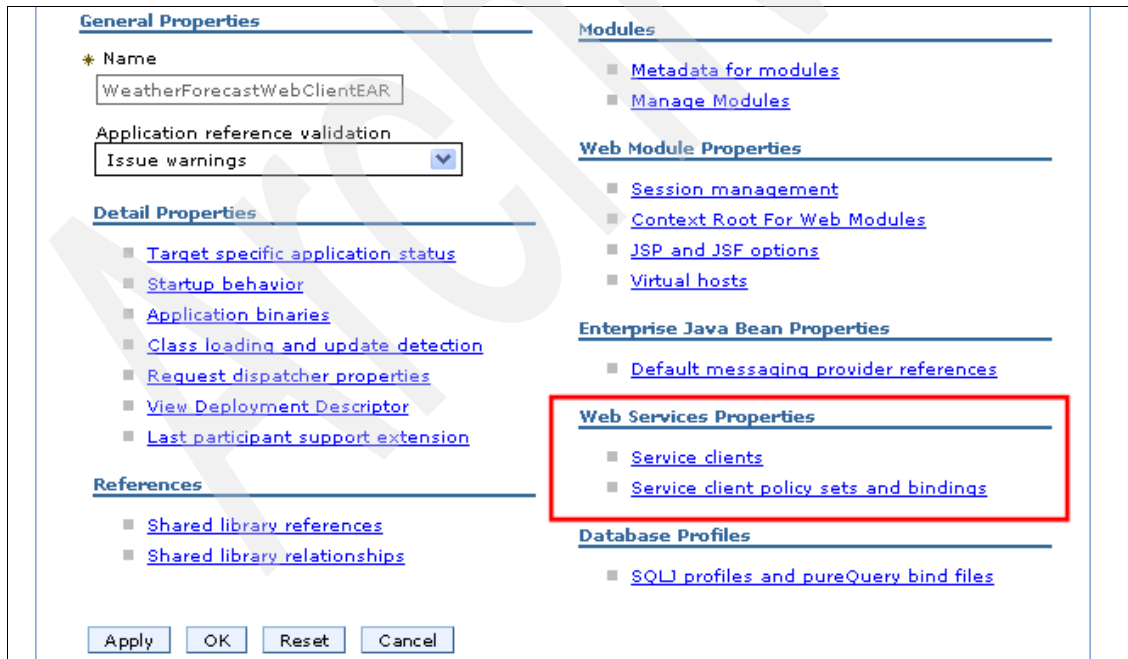
Figure 5-14   Application settings for a Web service client

Web service clients configuration items

Unlike in the Web services server-side settings list, there are only two configuration options available for client-side settings:

- ► Service clients
- ► Service client policy sets and bindings

Service clients

The Service clients link opens a page that lists all service clients that are deployed in the application (Figure 5-15). This page provides additional links that reference the specific modules that are related to a particular service client.

Figure 5-15   Service clients

Service client policy sets and bindings

The Service client policy sets and bindings link opens a page that is similar to the server-side version of this link (Figure 5-16). The page lists the components for a particular service client. You can attach or detach policy sets and assign bindings. Chapter 6, "Policy sets" on page 261, describes these functions in detail.

Figure 5-16   Service client policy sets and bindings

5.4 Managing Web service resources

Web services often require access to resources that are managed by the WebSphere Application Server. Part of the role of a system administrator includes the configuration of these resources to make them available to the deployed Web service.

5.4.1 Configuring JDBC resources

One advantage of Web services is its ability to abstract data resources, which include relational databases. The application server manages connections to Java Database Connectivity (JDBC)-compliant relational database providers.

One method of defining JDBC resources is to configure them within the Web service's deployment unit by using a WebSphere Application Server feature known as the *enhanced EAR file*. Application developers typically do this by using the enhanced EAR file editors of Rational Application Developer. See "Configuring the enhanced EAR file" on page 178 for an example.

This chapter focuses on tasks performed by the administrator. When an enhanced EAR file is not used, the administrator uses the WebSphere administrative tools to define the JDBC provider and data source for the resource. System administrators must ensure that the runtime configuration matches the expected values that are used in the Web service implementation. To be more specific, the data source that is used in the Web service (specified during development time) must match the data source that is configured (before deployment) in the server by its Java Naming and Directory Interface (JNDI) name.

Using the administrative console
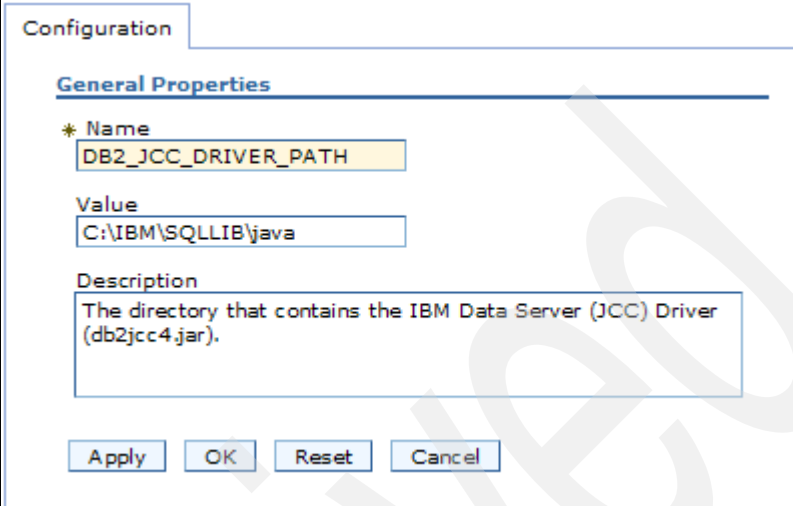
To configure the JDBC settings for a DB2® database provider, use the administrative console.

> **For more information**: For more information about defining JDBC resources and examples using other JDBC providers, see *WebSphere Application Server V7 Administration and Configuration Guide*, SG24-7615.

First set the environment variables for the DB2 JDBC drivers:

1. Select **Environment → WebSphere Variables**.

2. Select the scope of this resource. In a standalone server environment, it is sufficient to create the data source at the server level with a value of:

   Node=*node_name*, Server=server1

3. Locate and click the **DB2_JCC_DRIVER_PATH** entry.

4. On the Configuration page (Figure 5-17), in the value field, enter the path to the DB2 JDBC driver. Click **OK**.

Figure 5-17   Configuring the DB2 driver path

5. Repeat the process for the DB2_JCC_DRIVER_NATIVEPATH variable. For DB2, it should use the same path as DB2_JCC_DRIVER_PATH.

The user ID and password that are required to access the database are specified in a *J2C authentication data* entry. To create the authentication data:

1. Select **Security** → **Global Security**. Expand **Authentication** → **Java Authentication and Authorization Service** and select **J2C authentication data**.

2. Click **New**.

3. On the Properties page (Figure 5-18):
   a. Provide a name for the new alias.
   b. Enter the user ID and password that will provide access to the database.
   c. Click **OK**.

Figure 5-18   Creating a JAAS authentication alias

To create a JDBC provider that targets a DB2 database:

1. Expand **Resources** → **JDBC** and select **JDBC Providers**.

2. Select the scope of this resource. In a standalone server environment, it is sufficient to create the data source at the server level with a value of:

   Node=*node_name*, Server=server1

   Ensure that the environment variables set earlier are in the same scope.

3. Click the **New** button.

4. In the Configuration dialog box (Figure 5-19), select the general properties for the JDBC provider:

   a. For database type, select **DB2**.
   b. For provider type, select **DB2 Using IBM JCC Driver**.
   c. For implementation type, select **XA data source**.
   d. For name, type `DB2 Using IBM JCC Driver (XA)`.

   > **Driver**: A DB2 XA-capable JDBC Driver is used in this example. If the application does not require two-phase commit capabilities, use the regular driver.

   e. Click **Next**.

Figure 5-19   Creating a DB2 JDBC provider

5. The next panel allows you to enter database provider classpath information. In this case, the defaults are acceptable. Click **Next**.

6. On the Summary page, click **Finish**.

Create the data source:

1. Select **Resources** → **JDBC** → **JDBC Providers**.

2. Select the **DB2 Using IBM JCC Driver (XA)** that was defined earlier, and under Additional Properties, select **Data Sources**.

3. Click **New**.

4. Add the new data source (Figure 5-20):

   a. For data source name, enter a value for administrative purposes.

   b. For JNDI name, enter the same name that was used in the Web service implementation to obtain a JDBC connection. Click **Next**.

Figure 5-20   Basic data source properties

5. Enter the properties for the data source (Figure 5-21):

   a. For driver type, which represents the JDBC level of the driver being used, select **4**. A type of 4 JDBC driver is a pure Java-implemented network (thin client) driver.

   b. For database name, specify the name of the database that is being accessed.

   c. For server name, enter the name of the host of the database.

   d. For port number, enter the post number that is being used by the database.

   e. Select **Use this Data Source in container-managed persistence (CMP)**. Selecting this option means that the data source being created should be configured for use with CMP entity EJBs. Click **Next**.

Figure 5-21   Data source database properties

6. On the Setup security aliases page, click **Next**.

7. On the Summary page, click **OK**.

8. Save the configuration.

9. Test the connection. Select the data source and click **Test Connection**.

5.4.2 Configuring JMS resources

Web services can use JMS as the transport protocol to achieve an asynchronous mode of operation. The JMS resources used by the Web service are managed

by WebSphere Application Server. These resources must be properly configured to ensure the correct operation of the Web service that uses them.

Using the administrative console

JMS resources can be configured directly by using the administrative console or through an administrative script by using the `wsadmin` facility. In this section we show how to configure JMS resources by using the administrative console.

> **More information:** For more information about defining a service integration bus and JMS resources, see *WebSphere Application Server V7 Administration and Configuration Guide*, SG24-7615.

The messaging infrastructure of WebSphere Application Server, as illustrated in Figure 5-22, requires several resources that you must create for JMS transport of Web services:

► The *service integration bus*, which is the primary messaging structure of WebSphere Application Server

► A *connection factory* that allows Java programs to access messaging points on the service integration bus

► A *destination*, which binds the messaging point (queue or topic) to the service integration bus

► A *queue* or *topic*, which is the messaging point itself

► An *ActivationSpec*, which is required by JCA to access the messaging system of WebSphere Application Server
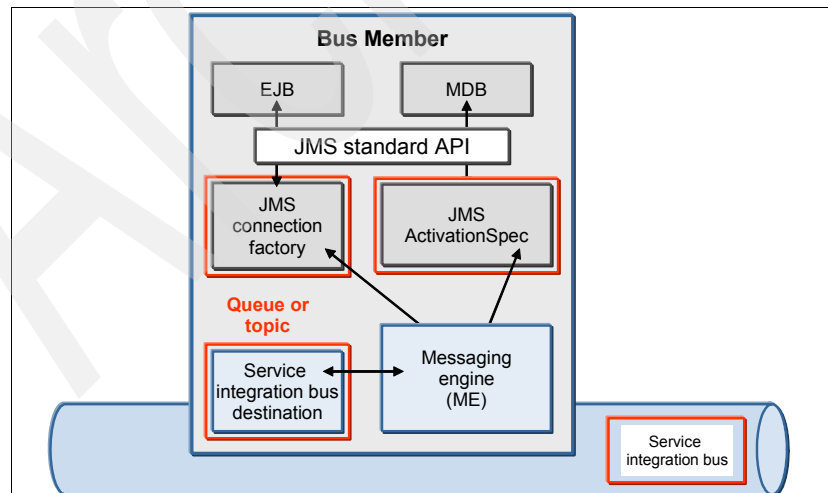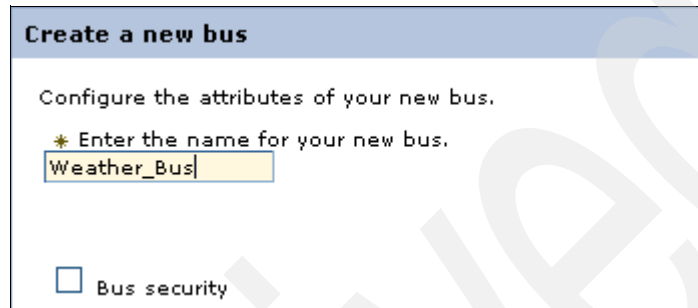
Figure 5-22   WebSphere messaging infrastructure

To create a service integration bus in the WebSphere Application Server and add a bus member for it:

1. In the WebSphere Application Server administrative console, select **Service Integration** → **Buses** and click the **New** button.

2. Under Create a new bus (Figure 5-23), for the name of the new service integration bus, type `Weather_Bus`. Clear **Bus security**, because it is not needed for a test environment, and click **Next**.
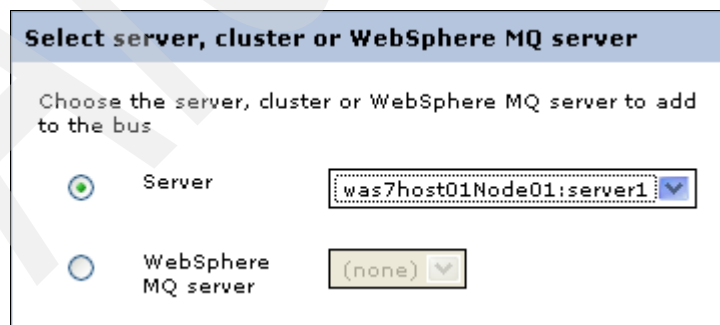
Figure 5-23   Creating a new service integration bus

3. Click **Finish**.

4. Save the changes to the master configuration.

5. When the bus collection is displayed, click the recently created **Weather_Bus** to view its properties.

6. Under Topology, click **Bus members**. On the Bus members collection, click **Add**.

7. As shown in Figure 5-24, select **Server**. Ensure that `server1` is displayed. Click **Next**.
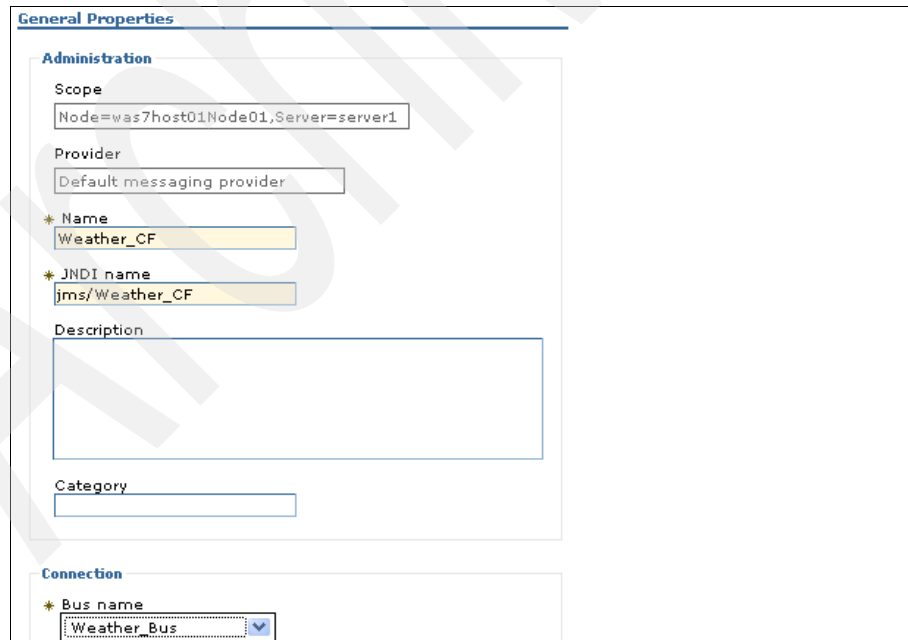
Figure 5-24   Adding bus members

8. For the following steps, accept the default configurations and click **Next** until you reach the Step 2: Summary page.

9. On the Step 2: Summary page, click **Finish**.

10. Save the changes to the master configuration.

To create a JMS queue connection factory:

1. In the WebSphere Application Server administrative console, select **Resources** → **JMS** → **Queue Connection Factories**.

2. Select the scope of this resource. In a standalone server environment, it is sufficient to create this at the server level with a value of Node=*node_name*, Server=server1. Click the **New** button.

3. Select the **Default messaging provider** and click **OK**.

4. Define the queue connection factory properties (Figure 5-25):

   a. For name, type `Weather_CF`.

   b. For JNDI name, type `jms/Weather_CF`, which will be used by the Web service to connect to the service integration bus.

   c. For bus name, select **Weather_Bus**, which you defined earlier.

   d. Click **OK**.

Figure 5-25   Queue connection factory properties

5. Save the properties to the master configuration.
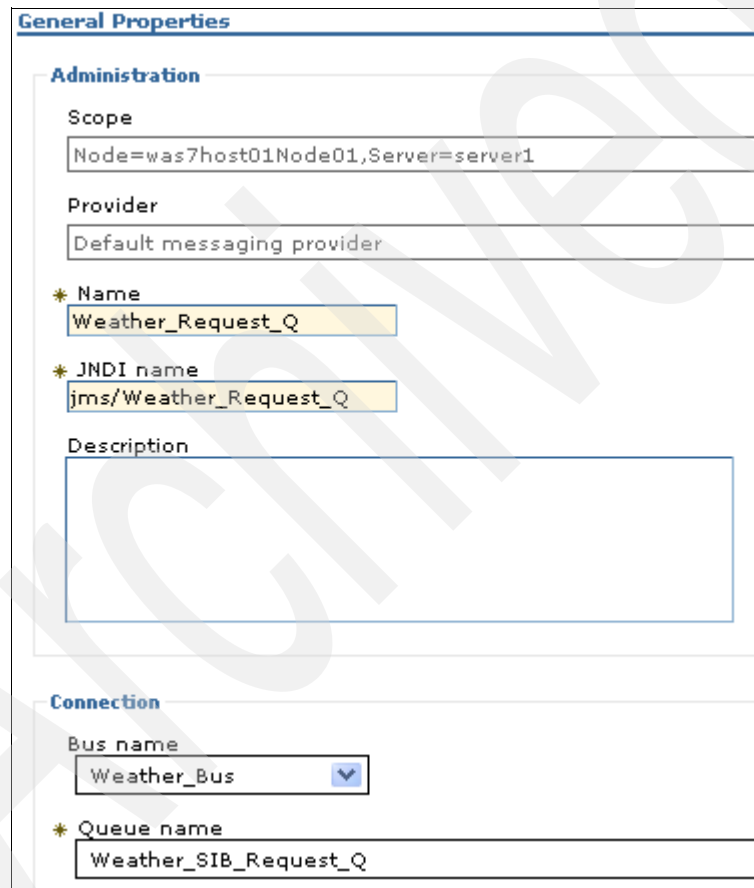
To create a destination for the service integration bus:

1. In the WebSphere Application Server administrative console, select **Service Integration** → **Buses** and click **Weather_Bus** to view its properties.

2. Under Destination resources, click **Destinations**.

3. Under Destinations collection, click **New**.

4. Select **Queue** to create a queue messaging point. Click **Next**.

5. In the Identifier field, type `Weather_SIB_Request`, which is the queue's name. Click **Next**.

6. Under Bus member, ensure that server1 is displayed. Server1 is where the destination will be located. Click **Next**.

7. In the summary step, click **Finish**.

8. Save the changes to the master configuration.

To create a queue and bind it to the previously defined destination:

1. In the WebSphere Application Server administrative console, select **Resources** → **JMS** → **Queues**.

2. Select the scope of this resource. In a standalone server environment, it is sufficient to create this at the server level with a value of Node=*node_name*, Server=server1. Click the **New** button.

3. Select the **Default messaging provider** and click **OK**.

4. Define the queue general properties (Figure 5-26):

   a. For name, type `Weather_Request_Q`.

   b. For JNDI name, type `jms/Weather_Request_Q`, which is used by the Web service to connect to this queue.

   c. For bus name, select **Weather_Bus**, which you defined earlier.

   d. For queue name, select **Weather_SIB_Request_Q**, which you defined earlier as a service integration bus destination.

   e. Click **OK**.

Figure 5-26   Queue general properties

5. Save the changes to the master configuration.

To create an ActivationSpec:

1. In the WebSphere Application Server administrative console, select **Resources → JMS → Activation specifications**.

2. Select the scope of this resource. In a standalone server environment, it is sufficient to create this at the server level with a value of Node=*node_name*, Server=server1. Click the **New** button.

3. Select the **Default messaging provider** and click **OK**.

4. Define the ActivationSpec properties (Figure 5-27):

    a. For name, type `Weather_ActivationSpec`.

    b. For JNDI name, type `eis/Weather_ActivationSpec`, which is used by the Web service to connect to this queue using JCA.

    c. For destination type, select **Queue**.

    d. For destination JNDI name, type `jms/Weather_Request_Q`, which you defined earlier for the queue resource.

    e. For bus name, select **Weather_Bus**, which you defined earlier.

    f. Click **OK**.

Figure 5-27   ActivationSpec properties

5. Save the changes to the master configuration.

> **wsadmin versus administrative console for setting up JMS resources**: In "Setting up the JMS resources" on page 198, we use the `wsadmin` facility to create JMS resources that are specific to that example. In the previous example, we performed the same task, but by using the administrative console for Web services in general. The outputs (created resources) of the two methods are not the same.

5.5 Tracing Web services

WebSphere Application Server uses a diagnostic trace facility to capture information regarding specific components in the server run time. This information is recorded in the `trace.log` file (default name) and is in the `WAS_HOME`/profiles/`node_name`/logs/server1 directory, which is the same as the other WebSphere Application Server log files.

The diagnostic trace facility presents fine-grained information about runtime components. On the Change Log Detail Levels page, administrators can configure the trace for the specific components to log. To access this page, in the WebSphere Application Server administrative console, select **Troubleshooting** → **Logs and trace** → **server** → **Diagnostic Trace** → **Change Log Detail Levels**.

To specify components to include in the trace, specify a trace string in the entry field, as shown in Figure 5-28.
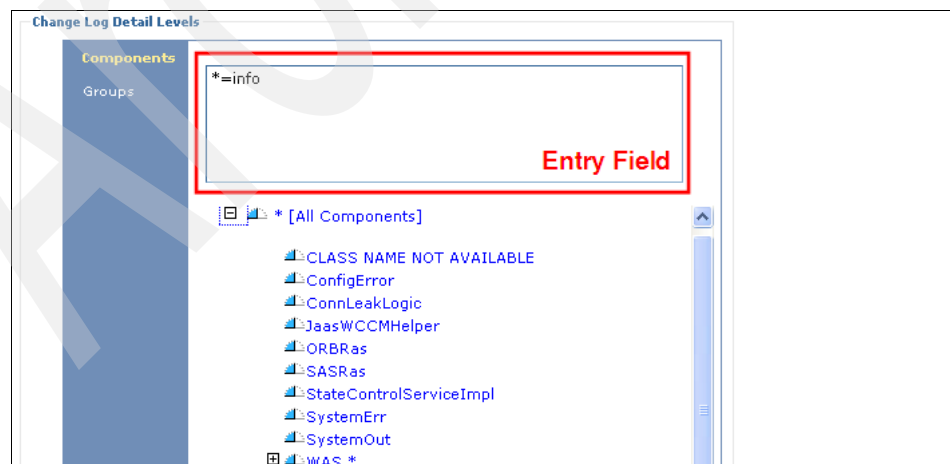
Figure 5-28   Change Log Detail Levels page

The trace string must have the following format:

component_item1=detail_level:component_item2=detail_level

Consider the following trace string as an example:

```
*=info:com.ibm.ws.websvcs.*=all
```

This means that all components will be logged in the trace file at an *info* detail level and all messages for components under com.ibm.ws.websvcs will be logged.

The following list of trace strings is related to Web services:

► com.ibm.ws.websvcs.* (JAX-WS integration layer with WebSphere)
► com.ibm.ws.websvcs.transport.jms.* (JMS transport)
► com.ibm.ws.policyset.* (policy sets)
► com.ibm.ws.wssecurity.* (WS-Security)
► com.ibm.ws.addressing.* (WS-Addressing)
► com.ibm.ws.wstx.* (WS-Transactions)
► org.apache.axis2.* (Axis2 run time)
► org.apache.axis2.jaxws.* (JAX-WS run time)

You can also enter trace strings by selecting a component item, right-clicking, and specifying the log detail level for the component (Figure 5-29).
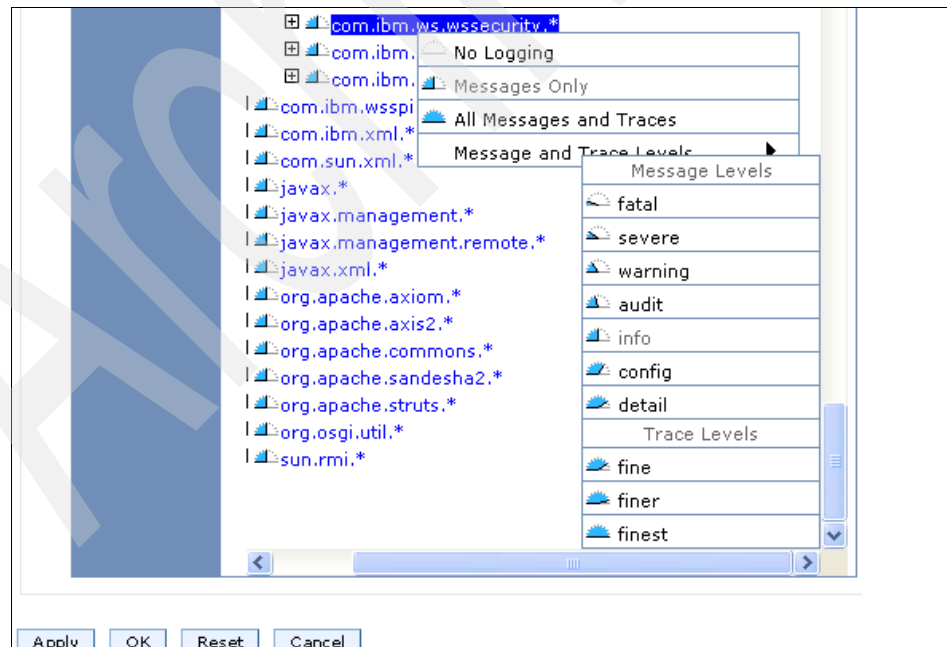
Figure 5-29   Specifying trace strings from the component list

When all trace strings are entered, click **OK** or **Apply** and save them to the master configuration. Restart the WebSphere Application Server and access the Web service by using a client.

The `trace.log` file is now generated, with a record of all information for the components specified and at the detail levels given. Figure 5-30 shows an example `trace.log` file.


```
[10/1/08 15:11:41:296 CDT] 00000021 JMSUtils        >  findSOAPAction Entry
[10/1/08 15:11:41:296 CDT] 00000021 JMSUtils        3   Got this soapAction from MessageContext.g
[10/1/08 15:11:41:296 CDT] 00000021 JMSUtils        <  findSOAPAction, soapActionString= echoOper.
[10/1/08 15:11:41:296 CDT] 00000021 JMSUtils        3   contentType text/xml; charset=UTF-8
[10/1/08 15:11:41:296 CDT] 00000021 JMSUtils        3   writing to MessagEFormatter with followin
[10/1/08 15:11:41:296 CDT] 00000021 JMSUtils        <  writeMessage() Exit
[10/1/08 15:11:41:296 CDT] 00000021 SOAPOverJMSSe 3   Setting SOAPJMS_contentType on JMSMessage
[10/1/08 15:11:41:296 CDT] 00000021 SOAPOverJMSSe 3   Setting SOAPJMS_targetService on JMSMessa
[10/1/08 15:11:41:296 CDT] 00000021 SOAPOverJMSSe 3   Setting SOAPJMS_requestIRI on JMSMessage
[10/1/08 15:11:41:296 CDT] 00000021 SOAPOverJMSSe 3   Setting SOAPJMS_bindingVersion on JMSMess
[10/1/08 15:11:41:296 CDT] 00000021 JMSUtils        >  findSOAPAction Entry
[10/1/08 15:11:41:296 CDT] 00000021 JMSUtils        3   Got this soapAction from MessageContext.g
[10/1/08 15:11:41:296 CDT] 00000021 JMSUtils        <  findSOAPAction, soapActionString= echoOper.
[10/1/08 15:11:41:296 CDT] 00000021 SOAPOverJMSSe 3   Permanent reply queue (stub or client-bin
[10/1/08 15:11:41:296 CDT] 00000021 SOAPOverJMSSe 3   Using permanent reply-to queue jms/SJT_Re
[10/1/08 15:11:41:296 CDT] 00000021 SOAPOverJMSSe 3   Set the reply-to queue and started the qu
[10/1/08 15:11:41:296 CDT] 00000021 SOAPOverJMSSe 3   About to send message:

    JMSMessage class: jms_bytes
    JMSType:          null
    JMSDeliveryMode:  2
    JMSExpiration:    0
    JMSPriority:      4
    JMSMessageID:     null
    JMSTimestamp:     -1
    JMSCorrelationID: null
    JMSDestination:   null
    JMSReplyTo:       queue://SJT_SIB_Reply_Q
    JMSRedelivered:   false
      JMS_IBM_MsgType: 1
      SOAPJMS_contentType: text/xml; charset=UTF-8
      SOAPJMS_targetService: EchoServicePortType
      SOAPJMS_requestIRI: jms:jndi:jms/SJT_Request_Q?jndiConnectionFactoryName=jms/SJT_CF&targetS
      SOAPJMS_soapAction: echoOperation
      SOAPJMS_bindingVersion: 1.0
3c736f6170656e763a456e76656c6f706520786d6c6e733a736f6170656e763d22687474703a2f2f
736368656d61732e786d6c736f61702e6f72672f736f61702f656e76656c6f70652f223e3c736f61
70656e763a426f64793e3c6e73323a6563686f537472696e67496e70757420786d6c6e733a6e7332
3d22687474703a2f2f636f6d2f69626d2f7761732f777373616d6d6d65672f7365692f6563686f2f22
3e3c6563686f496e707574743e68656c6c6f20776f726c64213c2f6563686f496e707574743e3c2f6e73
323a6563686f537472696e6765746f7075743e3c2f736f6170656e763a426f64793e3c2f736f61706
6e763a456e76656c6f70653e

Message send options:
deliveryMode=1, priority=4, timeToLive=300000
[10/1/08 15:11:41:296 CDT] 00000021 SOAPOverJMSSe 3   CHARACTER_SET_ENCODING : UTF-8
[10/1/08 15:11:41:296 CDT] 00000021 SOAPOverJMSSe 3   Sent the message, messageID=ID:a363493f47
[10/1/08 15:11:41:296 CDT] 00000021 SOAPOverJMSSe 3   About to commit QueueSession (send)
[10/1/08 15:11:41:296 CDT] 00000025 JMSListenerMD >  Common JMSListenerMDB.onMessage() Entry
```

Figure 5-30   A trace.log example

Part 3

Advanced concepts

259

6

Policy sets

In this chapter we explore the policy set feature in WebSphere Application Server V7. The use of policy sets can greatly reduce the amount of configuration that must be done for Web services by providing reusable configurations.

We begin by describing the motivation to use policy sets and the policy set definitions. Then we guide you through samples to administer policy sets in WebSphere Application Server. This includes applying policy sets to your Web services by using a default policy set and a custom policy set. We also show you how to configure an application-specific binding and a general binding for your Web service. Finally, we explore the support for policy set tools in Rational Application Developer V7.5.

This chapter contains the following topics:

6.1 Motivation

The motivation for using policy sets is three-fold:

► Proliferation of specifications

The proliferation of Web service specifications and the interdependencies between them makes the configuration of qualities of service for a Web service a large task. This task is best tackled in a step that is separate from the implementation of interfaces, clients, and services.

► Multi-vendor environment

The configuration task is all the more daunting in a multivendor environment because of the need to match client and server configurations. The use of policy sets to separate the configuration parameters for Web services from the implementation of the clients and services is the first step in simplifying configuration in a multivendor environment. The second step is to standardize the way in which the configuration is expressed (for example, by using Web services Description Language (WSDL)), so that configuration files are exchangeable.

► Development and management of qualities of service

Separating policy sets from the services is preferable to a development and management perspective. The skills to configure policies are different from defining and implementing services. An expert in a particular area develops and maintains policy sets, which are combined and applied to Web services administratively, without any necessity to redevelop or redeploy the clients or the services themselves.

Managing the complexity of Web services configurations

Working with Web services has certain difficulties. One challenge in particular is the abundance of standards associated with Web services and the complexity that this adds when configuring Web services. Configuration is made more difficult because there are few defaults, and configuration data must be re-entered for each Web service separately. Web services are relatively difficult to locate and manage in a WebSphere Application Server environment.

Each Web service application can contain multiple services. Each service can expose multiple endpoints. Finally, each endpoint has one or more operations associated with it, which can lead to a large number of configuration points for a single application, as illustrated in Figure 6-1.
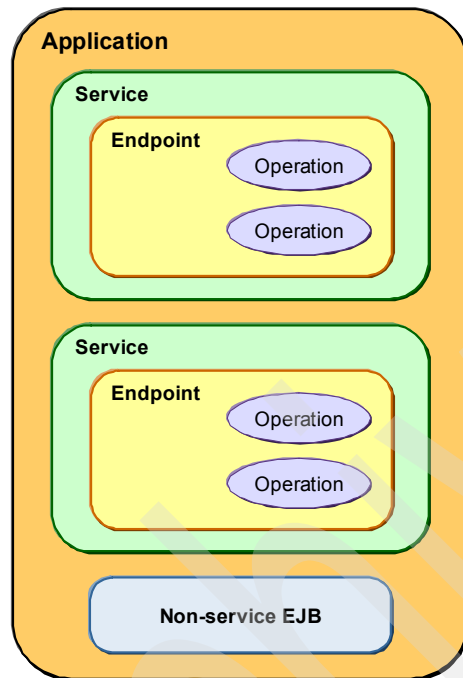
Figure 6-1   Configuration points for a Web service application

In previous releases of WebSphere Application Server, each service had its own quality of service (QoS) separately defined. In many cases, QoS configurations were not reusable. You had to repeat the configuration for each service, which is error-prone and difficult to manage.

As more qualities of service are added to the Web service with additional Web services specifications, such as WS-Addressing, WS-ReliableMessaging, and WS-Security, there is a requirement to manage more information about the Web service effectively.

In addition, there are relationships between the various qualities of service. For example, to prevent a sequence attack against WS-ReliableMessaging, you must use WS-SecureConversation to establish a secure context between the two parties. Managing the individual configuration of each set of the qualities of service is a daunting task.

Managing a single set of configurations for these qualities of service is a much simpler model. This is especially true if you can relate these configuration groupings to a well-defined name and reuse it in different services and across multiple application servers. Managing a combination of WS-ReliableMessaging and WS-SecureConversation to secure WS-ReliableMessaging headers is a good example of how combining related policies into a single policy set helps to make the management of configurations easier. See 10.4, "Secure conversation example" on page 496, for an example.

6.2 Overview of policy sets

In WebSphere Application Server V7, you can configure and apply a QoS to deployed Java API for XML Web services (JAX-WS) service providers and service clients by using policy sets. Policy sets reduce the complexity of configuring Web services in WebSphere Application Server. By providing reusable configurations, administrators can deploy and configure Web services applications more quickly. The policy sets also provide a template for new users, which show them how to properly configure WebSphere Application Server for specific qualities of service.

6.2.1 Qualities of service

With the widespread adoption of Web services as the key technology for implementing a service-oriented architecture (SOA), QoS has become a major priority for service providers and consumers. Qualities of service cover an entire range of requirements that match the requirements of service consumers with those of the service providers. This includes reliability, security, accessibility, availability, interoperability, performance, transaction, management, and so on.

6.2.2 Policy set definitions

Figure 6-2 illustrates the elements of policy sets. We describe these elements in the sections that follow.
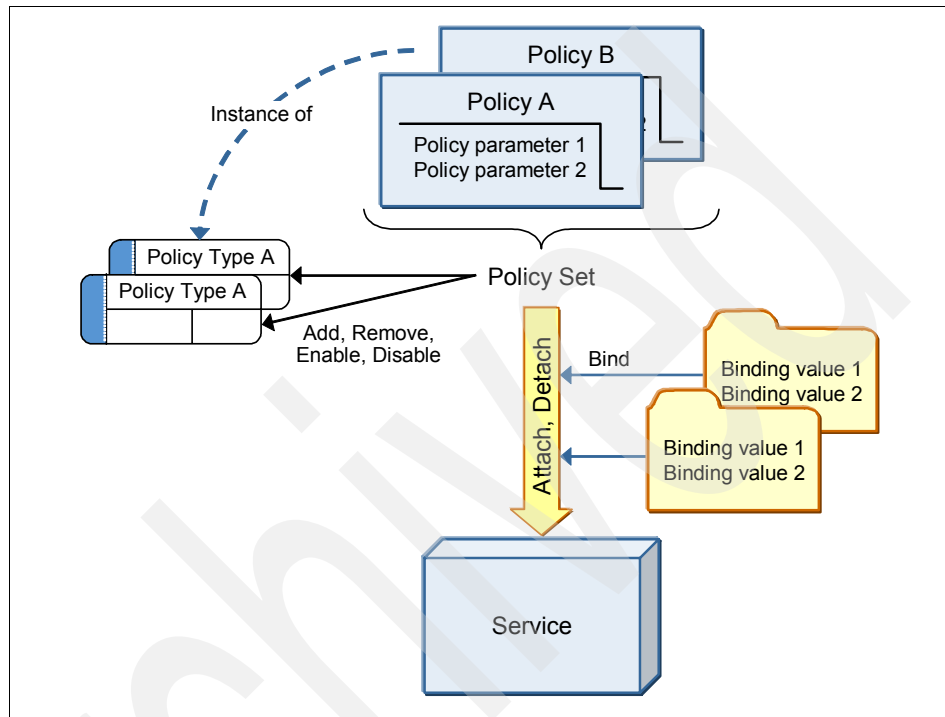
Figure 6-2   Policy sets

Policy type

A *policy type* is a single type of QoS that is defined by a set of assertions. Policy types can also be defined based on specific Web services standards. Policy types include the following examples:

- ▶ WS-Security
- ▶ WS-Addressing
- ▶ WS-ReliableMessaging
- ▶ HTTPS
- ▶ WS-Transaction

A policy type definition is based on WS-Policy standard language, for example, the WS-SecurityPolicy type is based on the WS-SecurityPolicy standard from the Organization for the Advancement of Structured Information Standards (OASIS).

Policy

A *policy* is a named, configured instance of policy type. A policy does not include environment-specific or platform-specific information such as a key for signing, keystore information, or persistent store information. These types of information are defined in the binding.

Binding

A *binding* is the topology-specific configuration of a QoS. It contains environment-specific and platform-specific information, such as keys for signing, keystore information, or persistent store information, which indicates that bindings are not normally shared. In WebSphere Application Server V7, there are two types of bindings:

► Application-specific bindings
► General bindings

Application-specific binding

You can create application-specific bindings only at a policy set attachment point. These bindings are specific to and constrained to the characteristics of the defined policy. Application-specific bindings are capable of providing configuration for advanced policy requirements, such as multiple signatures. However, these bindings are only reusable within an application. Furthermore, application-specific bindings have limited reuse across policy sets.

When you create an application-specific binding for a policy set attachment, the binding begins in a completely unconfigured state. You must add each policy that you want to override the default binding and fully configure the bindings for each policy that you add. For a Web services Security (WS-Security) policy, some high-level configuration attributes such as TokenConsumer, TokenGenerator, SigningInfo, or EncryptionInfo can be obtained from the default bindings if they are not configured in the application-specific bindings.

General binding

General bindings are new for WebSphere Application Server V7. They were created because of the success of reusing policy sets. These bindings can be configured to be used across a range of policy sets and can be reused across applications and for trust service attachments. Although general bindings are highly reusable, they are unable to provide configuration for advanced policy requirements, such as multiple signatures. There are two types of general bindings:

► General provider policy set bindings
► General client policy set bindings

The reason for having separate provider and client bindings is that the same binding cannot be reused for the provider and client in cases such as WS-Security. For the client, the private key is different from the server's private key. Therefore, separate binding files must be used in production.

General provider and client bindings are not linked to a particular policy set. They provide configuration information that you can reuse across multiple applications. You can create and manage general provider and client policy set bindings, and then select one of each binding type to use as the default for an application server. Setting the server default bindings is useful if you want the services that are deployed to a server to share binding configuration.

You can also share a binding configuration by assigning the binding to each application that is deployed to the server or by setting default bindings for a security domain and assigning the security domain to one or more servers. You can specify default bindings for your service provider or client that are used at the global security (cell) level, for a security domain, for a particular server.

Policy set

A *policy set* is a collection of policy types that are configured and associated with a given Web service provider or requester. A policy set is either predefined or user-defined. Policy types are added, removed, enabled, or disabled in the policy set. Policy sets are attached or detached to and from Web services and clients.

A policy set consists of a collection of policies of different types. For example, the Reliable Secure Profile (RSP) default policy set consists of instances of the WS-Security, WS-Addressing, and WS-ReliableMessaging policy types. A policy set is identified by a unique name that is unique across the cell.

WebSphere Application Server V7 ships with the following predefined policy sets:

- ► LTPA WSSecurity default
- ► Kerberos V5 HTTPS default
- ► SSL WSTransaction
- ► Username SecureConversation
- ► Username WSSecurity default
- ► WS-Addressing default
- ► WSHTTPS default
- ► WS-I RSP (Network Deployment)
- ► WS-ReliableMessaging persistent

These default policy sets can be used as starting points and best practices. The application server also provides other default policy sets that you can use or customize. To use the additional policy sets, you must import them from the default repository.

6.2.3 Using policy sets

After policy sets are created, they are associated with bindings that tailor specific details about the policy to the application (Figure 6-3). The combination of the policy set and the binding is applied to Web services or their clients.
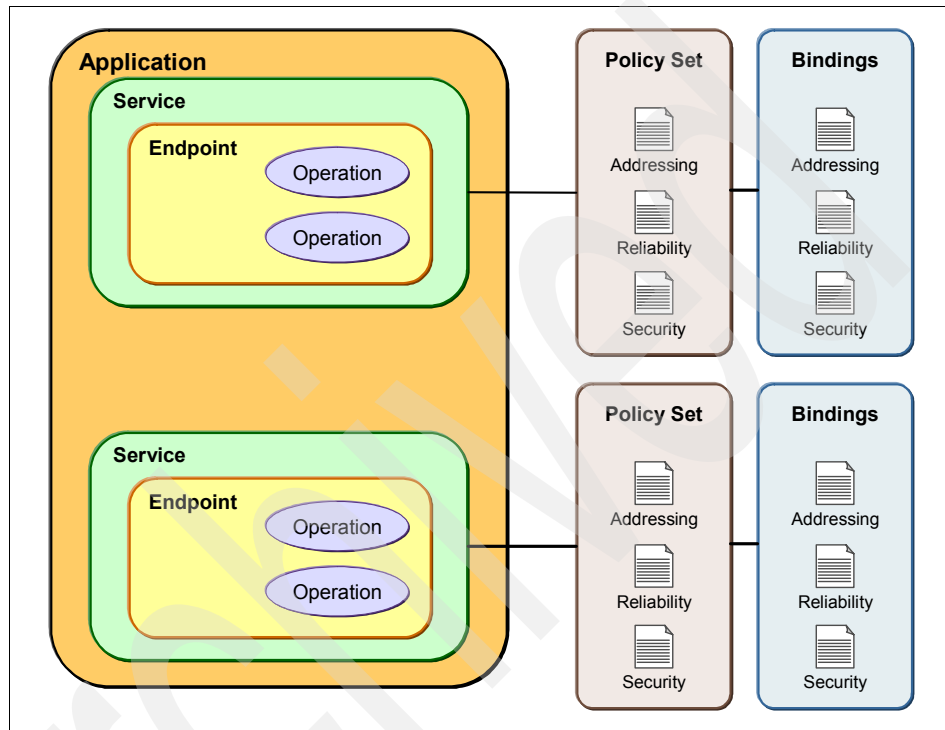
Figure 6-3   Attaching policy sets and bindings to Web services

6.3 New in WebSphere Application Server V7

WebSphere Application Server V7 enhances the usability and consumability of policy sets by providing the following new features:

► The ability to define multiple cell-level general bindings and to reuse these bindings across multiple applications

This capability significantly enhances the ease of use for binding configuration.

► The ability to choose default bindings for each security domain

- ► The ability to import a policy set from the WebSphere Application Server administrative console
- ► The ability to import a policy set that already exists by renaming the newly imported policy set

  This capability makes it easier for users to share policy set configurations across systems.
- ► The ability to import only default policy sets that are needed and to delete default policy sets, since they can be re-imported, if necessary
- ► The ability to import and export a general cell-level binding and to copy an existing general cell-level binding
- ► The ability to attach policy sets to WS-Notification service client endpoints

6.4 Policy set administration

In the following sections we explain how to manage and administer policy sets in a WebSphere Application Server V7 environment, which includes how to perform the tasks in the following sections:

- ► 6.4.2, "Viewing policy sets" on page 271
- ► 6.4.3, "Attaching a policy set to a Web service" on page 273
- ► 6.4.4, "Using a customized policy set" on page 284
- ► 6.4.5, "Configuring the application-specific bindings" on page 291
- ► 6.4.6, "Configuring general bindings" on page 306

First we briefly discuss the life cycle of a policy set.

6.4.1 Policy set life cycle

Rather than starting from scratch, create new policy sets by copying existing ones. Then modify the copies to configure them to your QoS requirements.

You attach a policy set to an application, service, endpoint, or operation either at deployment or after an application is deployed. When a policy set is attached, the application must be restarted to pick up the configuration changes.

Unless the child resources of an application are attached directly to another policy set, a policy set associated with a resource at any level is inherited by any resources underneath that resource. An *application-level attachment* is inherited by all child services, endpoints, and operations. A *service-level attachment* is inherited by all child endpoints and operations. An *endpoint-level attachment* is inherited by all child operations.

Existing user-defined policy sets are modifiable. However, because of unintended side effects, make a copy of a policy set and work with the copy. It is safer to work with an unattached policy set, such as a new copy. However, modifying an attached policy set alters the configuration for a deployed application, although the changes are not made until the application is restarted.

If the changes were intended to affect a particular application, and the application is not restarted for days or weeks, perhaps on a different shift, then there is a greater possibility for confusion. To help alert you to any problems, a warning message informs the administrator that specific endpoints will be affected. After changes are made, the associated application must be restarted for the changes to take effect.

6.4.2 Viewing policy sets

To view policy sets from the WebSphere administrative console:

1. Log in to the console.

2. In the left navigation area, expand **Services** → **Policy sets** (Figure 6-4). Two types of policy sets are listed:

   – *Application policy sets* are used by application resources.

   – *System policy sets* are used by system resources, such as the Security Token Service. See Chapter 10, "WS-SecureConversation" on page 471, to learn more about system policy sets.

Figure 6-4   Application policy sets

Default policy set bindings are also provided and listed here to minimize the configuration required by the policy sets.

Select **Application policy sets**.

The right pane shows the available policy sets, including a description of the qualities of service provided by the policy set. From this pane, you can create, copy, delete, import, and export policy sets. You are only allowed to modify user policy sets.

When you click a policy set name, the configuration opens. If the policy set is a user policy set, the fields on the configuration pages are editable. If the policy set is provided as a default, the policy set cannot be changed.

Reviewing the list of the policy sets

Default policy sets are provided with WebSphere Application Server V7 so that users can immediately and quickly configure most qualities of service (for example, the WS-I RSP policy set enables WS-ReliableMessaging, WS-Addressing, and WS-Security). To view the WS-I RSP policy set, click **WS-I RSP** on the page shown in Figure 6-4 on page 271. Figure 6-5 shows the result.

Figure 6-5   Details of the WS-I RSP policy set

As you can see, the WS-I RSP policy set consists of the WS-Addressing, WS-ReliableMessaging, and WS-Security policy types.

6.4.3 Attaching a policy set to a Web service

In this section we take you through an example of applying the WS-Addressing policy set. We also monitor the SOAP messages to confirm that the WS-Addressing policy set is successfully applied.

Preparing for the sample

The example application used in this chapter is the WeatherJavaBean application. It is similar to the application discussed in 4.2.2, "Web services development from an existing Java bean" on page 183.

The instructions in this chapter assume that you are using Rational Application Developer V7.5 with its integrated WebSphere Application Server V7 test environment. To follow along with the instructions, you can download the sample application, import it into a workspace, and install it to the test environment.

Downloadable material: The examples in this chapter use the WeatherJavaBean application. This application is included in the download material for this book in the `WeatherBase/WeatherWebService.zip` archive.

The project interchange file contains the following projects:

► WeatherBase: contains the core weather classes used by the applications (See 3.1.1, "The WeatherForecast application packages" on page 148.)

► WeatherJavaBeanServer: the Web service provider application

► WeatherJavaBeanWebClient: the Web service client application

For information about downloading the material, see Appendix A, "Additional material" on page 537.

For information about importing the application into your workspace, installing it on the server, and testing it, see "Using the WeatherJavaBean application" on page 543.

Attaching a WS-Addressing policy set

To apply the WS-Addressing default policy set to the Web services and Web services client:

1. In the administrative console (Figure 6-6) expand **Services** → **Service providers** to display a list of Web services.
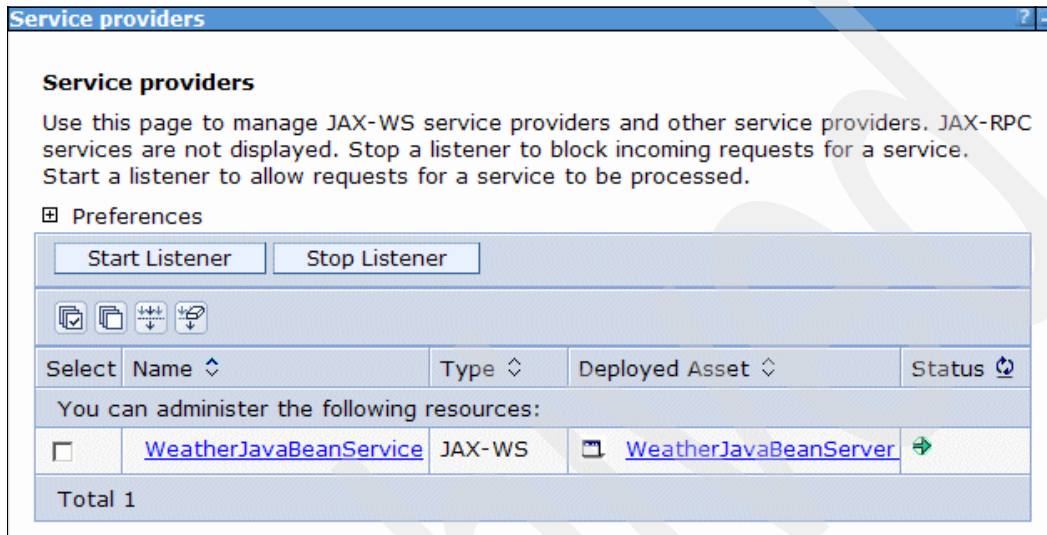
Figure 6-6   Selecting the WeatherJavaBeanService to configure it with a policy

2. In the right pane, click **WeatherJavaBeanService** to configure it with a policy.

3. On the Policy Set Attachments page:

   a. From the list of services, endpoints, and operations, select **WeatherJavaBeanService** (Figure 6-7).

   b. Click **Attach Policy Set**.

Figure 6-7   Attaching a policy set to the Weather Web service

A Web service has three levels of generality:

- Service
- Endpoint
- Operation

The service level is the most general, and the operation level is the most specific. The most specific attachment that applies is used for a given invocation of a Web service. For example, if you create attachments to both a Web service and an operation in that service, invocations of the operation use the attachment for the operation, but invocations of other operations use the attachment for the service.

In this example we apply the policy at the service level by attaching a policy set to the Weather Web service.

c. From the list of available policy sets to attach, select **WSAddressing default** (Figure 6-8).

Figure 6-8   Attaching a WSAddressing default policy set

The WS-Addressing default policy set is applied to the WeatherJavaBeanService (Figure 6-9).

Figure 6-9   Attached WSAddressing default policy set

4. Save the changes.

5. In the left navigation pane, under Services, select **Service clients**.

6. From the list of service clients, click **WeatherJavaBeanService** (Figure 6-10).
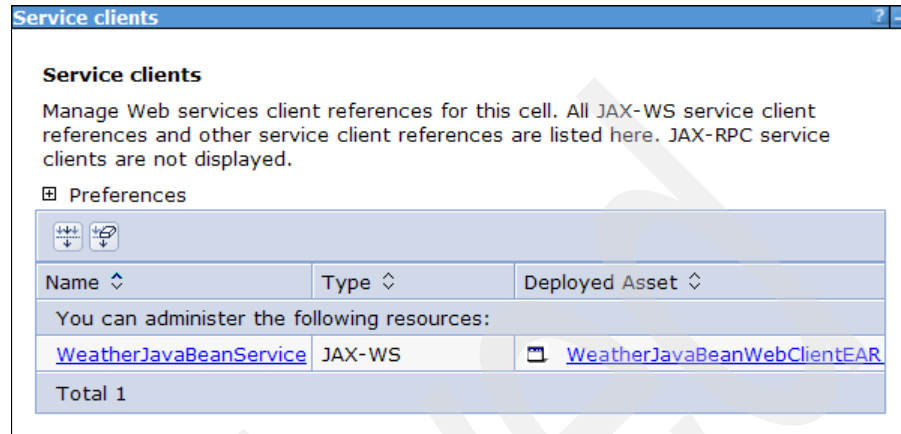
Figure 6-10   Service clients page

7. On the configuration page (Figure 6-11), select **WeatherJavaBeanService**. Click **Attach Client Policy Set**, and from the list of available policy sets to attach, select **WSAddressing default**.
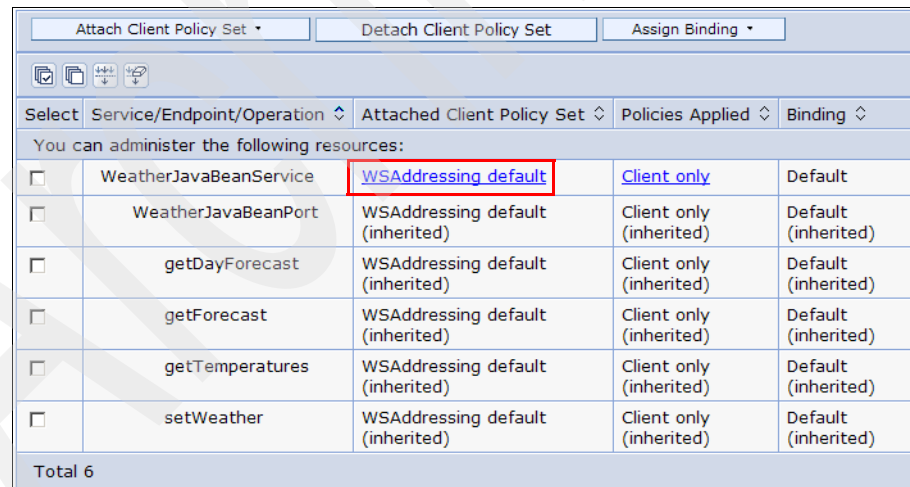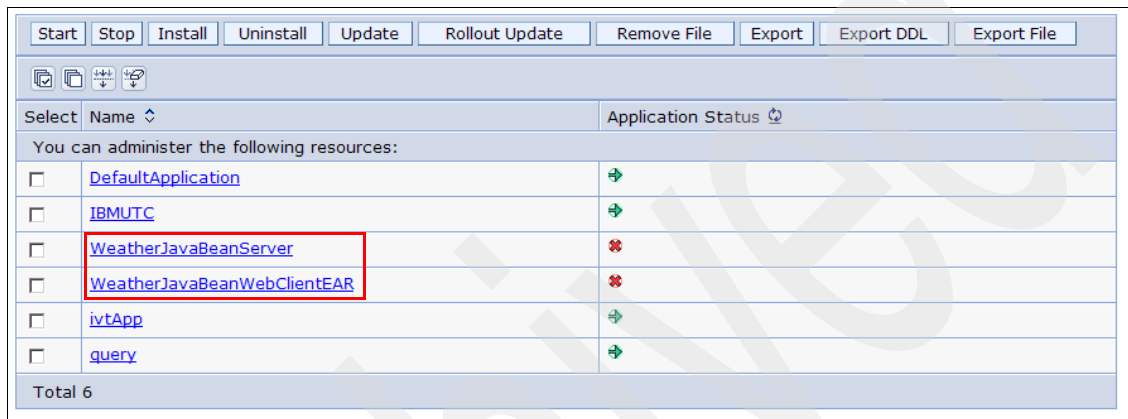
Figure 6-11   Attaching the WS-Addressing default to the client

8. Save the changes to the master configuration.

9. Restart the service and the client for the configuration changes to take effect.

10.On the next page:

    a. In the left navigation pane, select **Applications** → **Application Types** → **WebSphere Enterprise Applications**.

    b. In the right pane (Figure 6-12):

        i. Select **WeatherJavaBeanServer** and **WeatherJavaBeanWebClientEAR** and click **Stop**.

       ii. Select both applications again and click **Start**.

Figure 6-12   Stopping and then starting WeatherJavaBeanServer and WeatherJavaBeanWebClientEAR

The WS-Addressing default policy set is applied to the weather Web service and Web service client. To ensure that the policy has taken effect, monitor the SOAP traffic to see whether WS-Addressing information is added to the SOAP message.

Monitoring the SOAP traffic

Rational Application Developer provides a TCP/IP monitor that you use to monitor SOAP traffic over HTTP. The TCP/IP monitor acts as an intermediary between a Web service and its client. The client calls the TCP/IP address of the monitor rather than the SOAP endpoint. The monitor is configured to forward the request to the endpoint, and if it is a two-way request/reply, return the response to the start point, which is the client, unless a different endpoint for the reply is used.

To monitor the SOAP traffic:

1. Start Rational Application Developer if it is not already running.

2. Select **Window** → **Preferences** → **Run/Debug** → **TCP/IP Monitor** and click **Add**. The New Monitor window opens.

Alternatively, on the Servers tab, right-click the server name and select
Monitoring → **Properties** → **Add**. Then the Monitoring Ports window opens.

Although the windows are different, you can achieve the same result with
either one. In this example, we use the New Monitor window.

3. In the New Monitor window (Figure 6-13):

  a. In the Local monitoring port field, specify a unique port number on your
     local machine that is not used by any process (for example, 9089).

  b. In the Host name field, type `localhost`.

  c. For port, specify the port number of your WebSphere Application Server
     V7 for Web services provider.

  d. Select **Start monitor automatically**.

  e. Click **OK**.
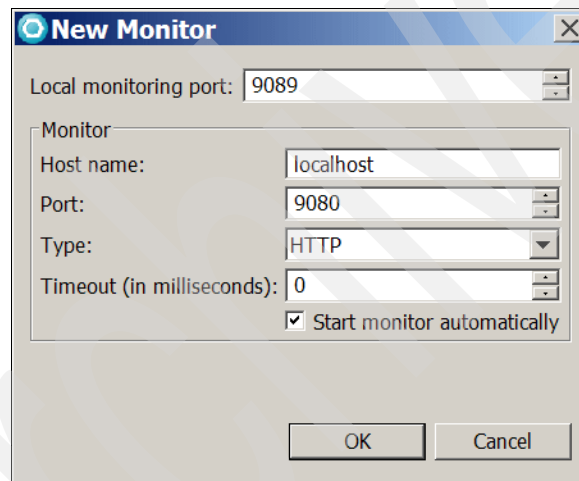
Figure 6-13   Creating a new TCP/IP monitor

4. Click **OK** again to close the preferences window.

The TCP/IP Monitor starts (Figure 6-14).
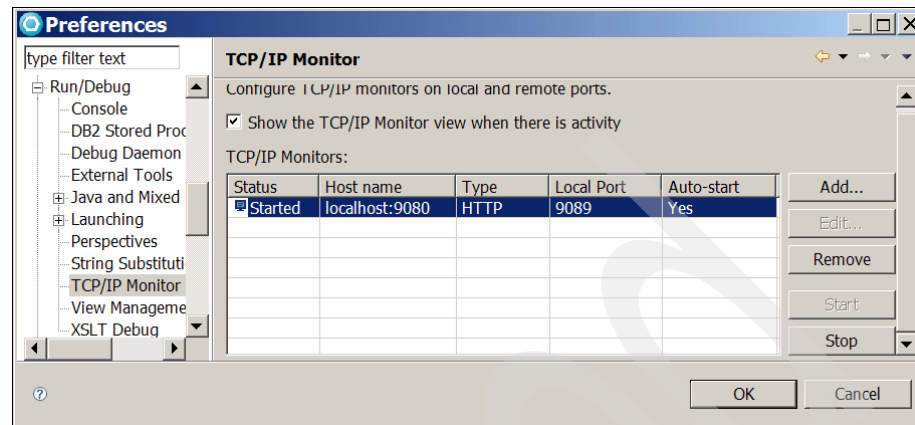
Figure 6-14   TCP/IP monitor started

5. Open a Web browser and run the sample JSP client. For example, if your server is running at port 9080, enter the following URL:

    ```
    http://localhost:9080/WeatherJavaBeanWebClient/sampleWeatherJavaBean
    PortProxy/TestClient.jsp
    ```

6. When the sample Web service JSP client opens in the browser, in the bottom pane, change the endpoint from 9080 to 9089. This value points to the port to which the TCP/IP monitor is listening.

As shown on the Web service Test Client page (Figure 6-15), in the Quality of Service pane, the URL is as follows:

```
http://localhost:9089/WeatherJavaBeanWebClient/sampleWeatherJavaBean
PortProxy/TestClient.jsp
```
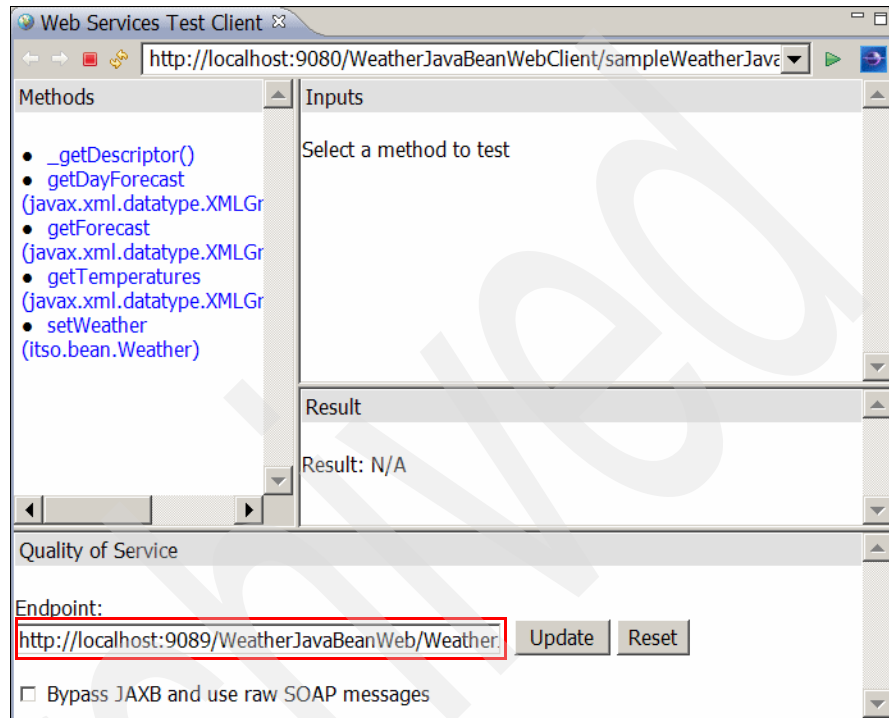

Figure 6-15   Updating the Web service endpoint

7. On the Web service Test Client page, in the Methods pane, click the **getDayForecast** method. In the upper right pane, type a value for arg0. You can copy and paste the example value provided by the JSP client. In this case, we use 2009-04-10T16:22:19. Click **Invoke**. Figure 6-16 shows the result.
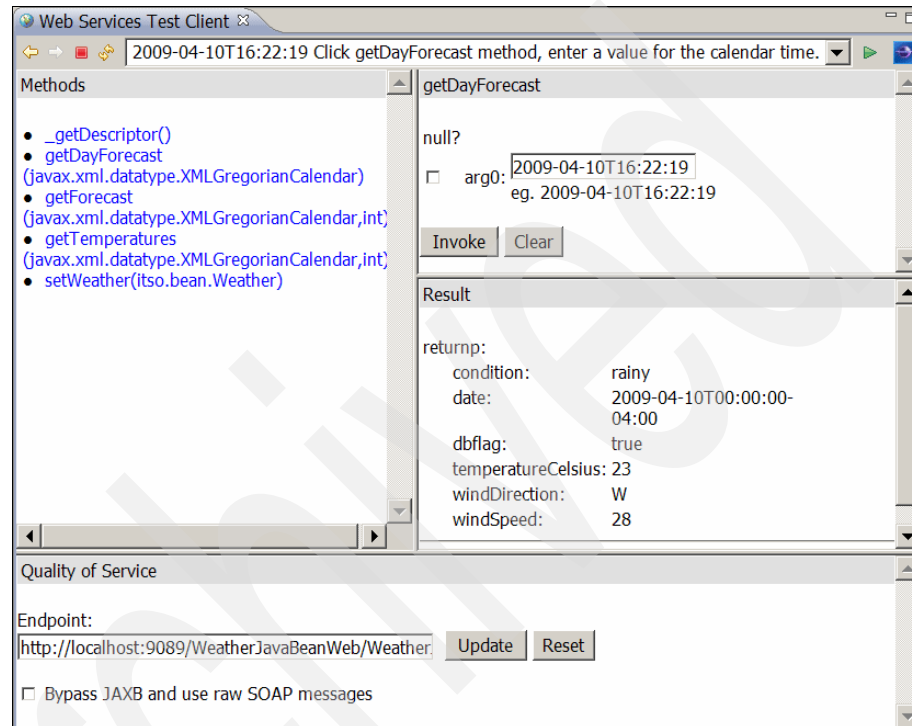
Figure 6-16   Web services result

8. In the TCP/IP Monitor view, which opens automatically, in the upper right corner, select **XML** for the request and response to view its contents (Figure 6-17).


```
Request: localhost:9089
Size: 547 (1021) bytes
Header: POST /WeatherJavaBeanWeb/WeatherJavaBeanService HTTP/1.1

<soapenv:Header xmlns:wsa="http://www.w3.org/2005/08/addressing">
   <wsa:To>http://localhost:9089/WeatherJavaBeanWeb/WeatherJavaBeanService</wsa:To>
   <wsa:MessageID>urn:uuid:A77482D9676F4609611239395506977</wsa:MessageID>
   <wsa:Action>http://bean.itso/WeatherJavaBeanDelegate/getDayForecastRequest</wsa:Action>
</soapenv:Header>
<soapenv:Body>
   <ns2:getDayForecast xmlns:ns2="http://bean.itso/">
     <arg0>2009-04-10T16:22:19</arg0>
   </ns2:getDayForecast>
</soapenv:Body>
</soapenv:Envelope>
```

Figure 6-17   Using TCP/IP Monitor to watch the SOAP traffic

Look at the WS-Addressing information in the SOAP request:

► <wsa:to> specifies the destination of the SOAP message.
► <wsa:MessageID> is the unique identifier for the SOAP message.
► <wsa:Action> is the in-envelope version of the SOAP HTTP Action header.

This information shows that the WS-Addressing policy set is successfully applied to the Weather Web service and its client.

Cleaning up the sample

To proceed with the next sample, we must restore the application to the original state. If you are using a standalone server, to detach the WS-Addressing policy:

1. Detach the service provider's policy set:

   a. In the administrative console, expand **Services** → **Service providers**.
   b. Click **WeatherJavaBeanService**.
   c. Select **WeatherJavaBeanService**.
   d. Click **Detach Policy Set**.
   e. Click **Save**.

2. Detach the service client's policy set:

   a. Expand **Services** → **Service clients**.
   b. Click **WeatherJavaBeanService**.
   c. Select **WeatherJavaBeanService**.
   d. Click **Detach Client Policy Set**.
   e. Click **Save**.

3. Restart the applications:

   a. In the navigation pane, select **Applications** → **Application Types** → **WebSphere Enterprise Applications**.

   b. Select **WeatherJavaBeanServer** and **WeatherJavaBeanWebClientEAR**.

   c. Click **Stop**.

   d. After the applications stop, click **Start**.

If you are using Rational Application Developer and its integrated WebSphere Application Server:

1. Right-click **WebSphere Application Server V7.0** and select **Add and Remove Projects**.

2. In the Add and Remove Projects window, select **Remove All** to uninstall the Web service and the client.

   Note that after you uninstall the Web service and the client application, the specific policy set and the binding that are attached with the Web service application are also gone.

3. Right-click **WebSphere Application Server V7.0** and select **Add and Remove Projects**.

4. In the Add and Remove Projects window, select **Add All** to install the Web service and the client.

You now have a fresh Web service and client without the policy set and binding attached.

6.4.4 Using a customized policy set

WebSphere Application Server V7 ships several default policy sets that are configured for use and are available as a template that you can copy and customize to suit your applications. In this example, we apply a custom policy set to the WeatherJavaBean application. Here is a summary of the requirements:

► Enable WS-Addressing, providing a transport-neutral way to uniformly address Web services and messages.

► Encrypt the message body of the SOAP request by using RSA encryption. The SOAP response is not required to be encrypted.

► Message integrity is not required.

None of the default policy sets matches these requirements exactly. Note that the WS-Security default policy set most closely meets our needs. Therefore, we use the WS-Security default policy set as a template to create our own policy set.

Keystores for the sample

Two keystores that contain the keys are provided with this book as part of the downloadable material:

- ▶ The receiver.jks keystore
- ▶ The sender.jks keystore

Table 6-1 indicates the properties of the keystores.

> **Downloadable material:** You can find the keystore files in the downloadable material in the `Chapter6/PolicySet.zip` file. For more information see Appendix A, "Additional material" on page 537.
>
> To use the files as we do in this section, extract the keystore files and place them in the location indicated in the first row of Table 6-1.

Table 6-1   Keystore properties

Property	Service	Client
Key store path	`C:\ITSO7758\PolicySet\` `receiver.jks`	`C:\ITSO7758\PolicySet\` `sender.jks`
Key store type	JKS	JKS
Key store password	itso	itso
Key alias	mark	henry
Distinguished name	cn=server,o=IBM, c=US	cn=client,o=IBM, C=US
Certificate file	server.arm	client.arm

When the Web service consumer sends a SOAP request to the Web service provider, it encrypts the message with the Web service provider's public key. When the Web service provider receives the encrypted message, it uses its own private key to decrypt the message.

For our example, *Henry* represents the Web services consumer and *Mark* represents the Web services provider. Henry uses Mark's public key to encrypt the SOAP request, and Mark uses his own private key to decrypt the SOAP request (Figure 6-18).
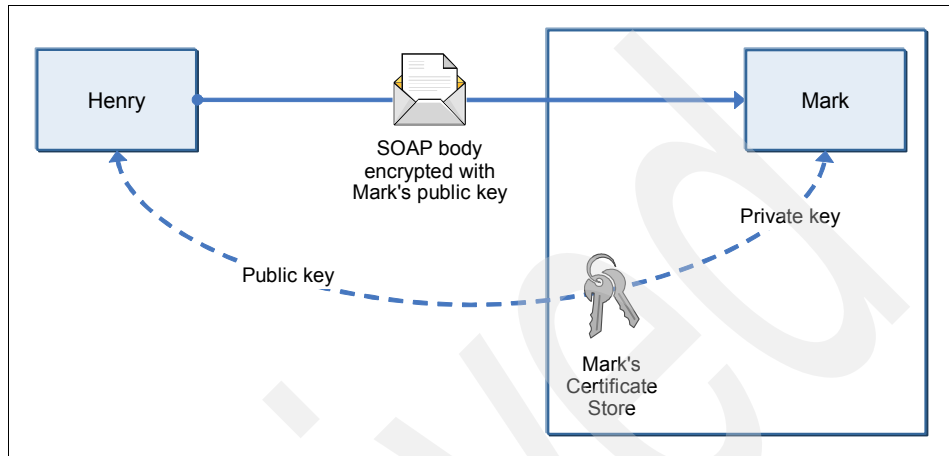
Figure 6-18   Customized policy set example scenario

Creating the custom policy set

To create our custom policy set, in the WebSphere administrative console:

1. In the left navigation pane, select **Services** → **policy sets** → **Application policy sets**.

2. In the right pane, click **New**.

    a. In the Name field, type `ITSO WSSecurity`.
    b. Under the Policies section, click **Add** and select **WS-Addressing**.
    c. Under the Policies section, click **Add** and select **WS-Security**.
    d. Click **OK**.
    e. Click **Save**.

3. Because this scenario encrypts only the body, update the WS-Security policy:

    a. Click **ITSO WSSecurity** to open the policy set.

    b. Click **WS-Security**.

c. Click the **Main policy** button to open the Main policy configuration page (Figure 6-19).

Figure 6-19   Main policy page

d. On the Main policy page (Figure 6-19 on page 287), click the **Request message part protection** link. This opens the panel shown in Figure 6-20.

Figure 6-20   Request message part protection page

e. For this scenario, apply only message encryption. The digital signature is not required.

i. Under the Integrity protection section (Figure 6-20), select **app_signparts** and click **Delete**. Then click **Save**.

ii. Under Encrypted parts, select **app_encparts** and click **Edit**.

Figure 6-21   Editing the encrypted parts

iii. Because you encrypt the body only, remove the UsernameToken encryption in the SOAP header. In the Elements in Part section (Figure 6-21 on page 289), select each XPath expression and then click **Remove**. The results are shown in Figure 6-22.
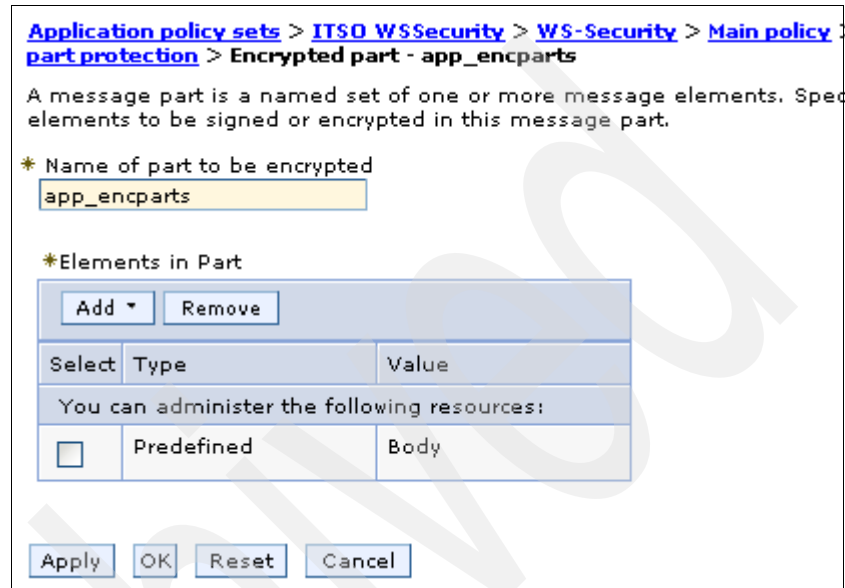

Figure 6-22   Encrypted message body

iv. Click **Apply** and then click **Save**.

f. In the navigation path, click **Main policy** to return to the Main policy page.

g. Click **Response message part protection**. The response message does not have to be secured.

h. On the Response message part protection page:

i. In the Confidentiality section, select **app_encparts** and click **Delete**.

ii. In the Integrity protection section, select **app_signparts** and click **Delete**.

iii. Click **Save**.

You have now created the customized policy set. Next attach the customized policy set to the Web service and client.

Attaching the custom policy set to the Web service

To attach the ITSO WSSecurity policy set to the WeatherJavaBean application:

1. In the navigation pane, click **Services** → **Service providers**.

2. Click **WeatherJavaBeanService**.

3. Select **WeatherJavaBeanService**. Click **Attach Policy Set** and from the list, select **ITSO WSSecurity**.

4. Click **Save**.

You have now attached the policy set and binding to the Web service.

Assigning the custom policy set to the Web service client

To attach the custom policy set to the Web service client:

1. In the navigation pane, click **Services** → **Service clients**.

2. Click **WeatherJavaBeanService**.

3. Select **WeatherJavaBeanService**. Click **Attach Client Policy Set** and from the list, select **ITSO WSSecurity**.

4. Click **Save**.

You have now attached the policy set and binding to your Web service client.

6.4.5 Configuring the application-specific bindings

In this section, you configure the application-specific binding for the WeatherJavaBeanService service. You configure XML encryption for the request message. On the client side, you use Mark's public key to encrypt the SOAP request. On the service side, you use Mark's private key to decrypt the inbound message (Figure 6-18 on page 286).

Configuring the Web service binding

In this section, you configure the custom binding for the WeatherJavaBeanService service. You configure the XML encryption to decrypt the request message by using Mark's private key.

To configure the Web service binding:

1. In the administrative console, expand **Services** → **Service providers**.

2. Click **WeatherJavaBeanService** to open the configuration page.

3. Check **WeatherJavaBeanService**, click **Assign Binding**, and select **New Application Specific Binding** (Figure 6-23).

Figure 6-23   Selecting New Application Specific Binding

4. On the next page:

a. For the bindings configuration name, type `ITSO-service`.

b. Click **Add** and select **WS-Addressing**. Click **OK**.

c. Click **Add** and select **WS-Security**. This opens the pane shown in
    Figure 6-24. From this window you will configure the WS-Security
    bindings.
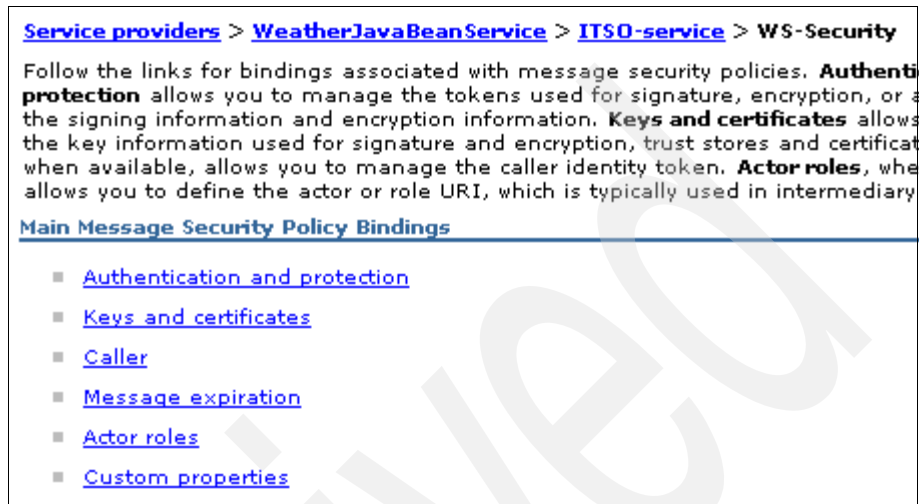
Figure 6-24   WS-Security policy bindings

> **Server protection token**: In the steps that follow, you configure the server
> protection token. First, you configure the protection token to encrypt the
> request message, and then you configure the way in which the message is
> encrypted.

5. To begin the configuration of the server protection token click **Authentication
    and protection**.

6. On the Authentication and protection page (Figure 6-25), because the Web services provider only has to decrypt the SOAP request, only configure the asymmetric encryption consumer. Click **AsymmetricBindingRecipientEncryptionToken0**.

Figure 6-25   Authentication and protection page

7. Verify that the JAAS login is **wss.consume.x509** and click **Apply** to generate a callback handler binding (Figure 6-26).

Figure 6-26   Generate callback handler binding

8. Click **Callback handler** (which is now available) to open the configuration page (Figure 6-27).

Figure 6-27   Callback handler configuration page

9. On the Callback Handler configuration page:

   a. In the Certificates section, make sure that **Trust any certificate** is selected.

   b. In the Keystore section, for the name, select **Custom** and then click **Custom keystore configuration**.

10.On the Custom keystore configuration page (Figure 6-28):

   a. In the Keystore section, for full path, enter the name for the `receiver.jks` keystore of `C:\ITSO7758\PolicySet\receiver.jks`.

   b. For type, select **JKS**.

   c. For password and confirm password, type `itso`.

   d. Under Key, in the Name field, type `cn=server,o=IBM, C=US`.

   e. In the Alias field, type `mark`.

   f. For key password and confirm password, type `itso`.

Figure 6-28   Custom keystore configuration

g. Click **OK** to close the panel.

11. Click **OK** again to close the callback handler configuration page.

12. Click **OK** to close the AsymmetricBindingRecipientEncryptionToken0 configuration.

    You return to the page shown in Figure 6-25 on page 294. The AsymmetricBindingRecipientEncryptionToken0 is now configured.

13. Click **Save**.

14. For the other three protection tokens, change their status to *configured*. It is not necessary to adjust their configuration.

    a. Click **AsymmetricBindingInitiatorEncryptionToken0**. Click **OK**.
    b. Click **AsymmetricBindingInitiatorSignatureToken0**. Click **OK**.
    c. Click **AsymmetricBindingRecipientSignatureToken0**. Click **OK.**
    d. Click **Save**.

    > **Configuring the server request message encryption protection**: Under Request message signature and encryption protection, request:app encparts must still be configured. The next series of steps performs this configuration.

15. In the Authentication and protection page (see Figure 6-25 on page 294), under Request message signature and encryption protection, click **request:app encparts**.

    a. In the Name field, type `req-enc-part` and click **Apply**.

    b. Under Key information, click **New**.

       i. Enter `req-enc-keyinfo` for the name.

       ii. Ensure that **AsymmetricBindingRecipientEncryptionToken0** is selected for token generator or consumer name.

       i. Click **OK**.

    c. Under Key information, select **req-enc-keyinfo**. Click **Add**, and then click **OK**.

    d. Click **Save**.

You have now configured the binding for the Web services (Figure 6-29).

Figure 6-29   WS-Security configured for the service

Configuring the WeatherJavaBeanService client bindings
In this section, you configure the custom binding for the WeatherJavaBeanService client. You configure the XML encryption to encrypt the request message by using Mark's public key. Again, there are two parts to the configuration. The first is the protection token and second is the message configuration.

Configuring the client protection token

To configure the client protection token:

1. In the navigation pane, click **Services** → **Service clients**.

2. Click **WeatherJavaBeanService**.

3. Check the **WeatherJavaBeanService**. Click **Assign Binding** and select **New Application Specific Binding**.

    a. Type ITSO-client as the name.
    b. Click **Add** and select **WS-Security**.

4. In the WS-Security configuration page, click **Authentication and protection**.

5. In the authentication and protection configuration panel, click **AsymmetricBindingRecipientEncryptionToken0**.

    a. Verify that JAAS login is **wss.generate.x509**.
    b. Click **Apply** to generate a callback handler binding.

6. Click **Callback handler**.

    a. In the Keystore section, select **Custom** and then click **Custom keystore configuration**.

    b. On the Custom keystore configuration page:

        i. Enter the full path name for the sender.jks keystore, such as C:\ITSO7758\PolicySet\sender.jks.

        ii. For type, select **JKS**.

        iii. For password and confirm password, type itso.

        iv. Under Key, in the Name field, type cn=client,o=IBM,C=US.

        v. In the Alias field, type mark.

        vi. Click **OK** to close the custom keystore configuration.

7. Click **OK** to close the callback handler configuration.

8. Click **OK** to close the AsymmetricBindingRecipientEncryptionToken0 configuration.

9. Click **Save**. The AsymmetricBindingRecipientEncryptionToken0 is now configured.

10. For the other three protection tokens, change their status to *configured*. You do not have adjust their configuration.

    a. Click **AsymmetricBindingInitiatorEncryptionToken0** and click **OK**.
    b. Click **AsymmetricBindingInitiatorSignatureToken0** and click **OK**.
    c. Click **AsymmetricBindingRecipientSignatureToken0** and click **OK**.
    d. Click **Save**.

The results are shown in Figure 6-30.

Figure 6-30   WS Security authentication and protection custom bindings

Configuring client request message encryption protection

To configure the client request message encryption protection:

1. Under Request message signature and encryption protection (Figure 6-30 on page 301), click **request:app encparts**.

   a. In the Name field, type req-enc-part.

   b. Under Key information, click **New**.

      On the next page:

      i. For name, type req-enc-keyinfo.

      ii. Under Type, select **Key identifier**.

      iii. For token generator or consumer name, ensure that **AsymmetricBindingRecipientEncryptionToken0** is selected.

      iv. Click **OK** to close the key information configuration panel.

2. Click **OK** again to close the request message signature and encryption protection configuration.

3. Click **Save**.

Restarting the Web service and client

Restart the service and the client application for the configuration changes to take effect:

1. In the navigation pane, select **Applications** → **Application Types** → **WebSphere Enterprise Applications**.

2. Select **WeatherJavaBeanServer** and **WeatherJavaBeanWebClientEAR**.

3. Click **Stop**.

4. After the applications stop, click **Start**.

The *ITSO WSSecurity* custom policy set is now applied to the Web service and its client. To ensure that the policy set has taken effect, examine the SOAP traffic to see whether WS-Security information is added to the SOAP message.

Monitoring the SOAP traffic

You use the TCP/IP monitor shipped with Rational Application Developer to watch the SOAP traffic to make sure that the SOAP request is encrypted. To monitor the SOAP traffic, do the following tasks:

1. Start Rational Application Developer if it is not already running.

2. Click **Window** → **Preferences** → **Run/Debug** → **TCP/IP Monitor**.

3. Select the monitor that was created in "Monitoring the SOAP traffic" on page 278.

4. Click **Start** if the status is *stopped*.

5. Open a Web browser and enter a URL with the host name and port set as follows:

   ```
   http://<hostname>:port/WeatherJavaBeanWebClient/sampleWeatherJavaBea
   nPortProxy/TestClient.jsp
   ```

   For example, if your server is running at port 9080, you might enter the following URL:

   ```
   http://localhost:9080/WeatherJavaBeanWebClient/sampleWeatherJavaBean
   PortProxy/TestClient.jsp
   ```

6. In the bottom pane of the sample Web service JSP client that opens in the browser, change the endpoint from 9080 to 9089. This endpoint points to a port to which the TCP/IP monitor is listening. The URL must be displayed as follows:

   ```
   http://localhost:9089/WeatherJavaBeanWebClient/sampleWeatherJavaBean
   PortProxy/TestClient.jsp
   ```

7. Click the **getDayForecast** method and enter a value for arg0. You can copy and paste the example value that is provided by the JSP client, such as 2009-04-10T16:22:19. Click **Invoke**.

8. In the TCP/IP monitor view, which opens automatically, select the **XML** format for the request and response to view its contents. Example 6-1 shows the (formatted) SOAP request.

> **Tip:** A quick way to format the XML results is to create a new file with a .xml extension in Rational Application Developer and paste the XML trace into it. Right-click the contents and select **Source → Format**.

Example 6-1   Encrypted SOAP request message

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
    <soapenv:Header xmlns:wsa="http://www.w3.org/2005/08/addressing">
        <s:Security xmlns:d="http://www.w3.org/2000/09/xmldsig#"
            xmlns:e="http://www.w3.org/2001/04/xmlenc#"

xmlns:s="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd"

xmlns:u="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
            soapenv:mustUnderstand="1">
            <u:Timestamp>
                <u:Created>2009-04-13T15:58:37.828Z
                </u:Created>
            </u:Timestamp>
            <e:EncryptedKey>
```

```
                <e:EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5" />
                <d:KeyInfo>
                    <s:SecurityTokenReference>
                        <s:KeyIdentifier

EncodingType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0
#Base64Binary"

ValueType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509S
ubjectKeyIdentifier">vmFljXX4Bpp4awaZeM3pL9nPf3s=
                        </s:KeyIdentifier>
                    </s:SecurityTokenReference>
                </d:KeyInfo>
                <e:CipherData>
                    <e:CipherValue>NRt ... =</e:CipherValue>
                </e:CipherData>
                <e:ReferenceList>
                    <e:DataReference URI="#w_20" />
                </e:ReferenceList>
            </e:EncryptedKey>
        </s:Security>
        <wsa:To>
            http://localhost:9089/WeatherJavaBeanWeb/WeatherJavaBeanService
        </wsa:To>
        <wsa:MessageID>urn:uuid:68ADA74396C3B417C61239638321837
        </wsa:MessageID>
        <wsa:Action>
            http://bean.itso/WeatherJavaBeanDelegate/getDayForecastRequest
        </wsa:Action>
    </soapenv:Header>
    <soapenv:Body>
        <e:EncryptedData xmlns:e="http://www.w3.org/2001/04/xmlenc#"
            Id="w_20" Type="http://www.w3.org/2001/04/xmlenc#Content">
            <e:EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#aes128-cbc" />
            <e:CipherData>
                <e:CipherValue>NOv ... /O=</e:CipherValue>
            </e:CipherData>
        </e:EncryptedData>
    </soapenv:Body>
</soapenv:Envelope>
```

You should see WS-Addressing information in the SOAP header. The SOAP message body is encrypted. Therefore, the ITSO WSSecurity custom policy set is successfully applied.

Example 6-2 shows the unencrypted response message.

Example 6-2   SOAP response

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
    <soapenv:Header xmlns:wsa="http://www.w3.org/2005/08/addressing">
        <s:Security

xmlns:s="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd"

xmlns:u="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
            soapenv:mustUnderstand="1">
            <u:Timestamp>
                <u:Created>2009-04-13T15:58:37.937Z
                </u:Created>
            </u:Timestamp>
        </s:Security>
        <wsa:Action>
            http://bean.itso/WeatherJavaBeanDelegate/getDayForecastResponse
        </wsa:Action>
        <wsa:RelatesTo>urn:uuid:68ADA74396C3B417C61239638321837
        </wsa:RelatesTo>
    </soapenv:Header>
    <soapenv:Body>
        <ns2:getDayForecastResponse xmlns:ns2="http://bean.itso/">
            <return>
                <condition>sunny</condition>
                <date>2009-04-13T00:00:00-04:00</date>
                <dbflag>true</dbflag>
                <temperatureCelsius>17</temperatureCelsius>
                <windDirection>NW</windDirection>
                <windSpeed>26</windSpeed>
            </return>
        </ns2:getDayForecastResponse>
    </soapenv:Body>
</soapenv:Envelope>
```

The response has the WS-Addressing correlation information for routing the reply in the SOAP header. Note that the uuid matches the request. The SOAP header also contains security tags, although they are not used.

Cleaning up the sample

To proceed with the next sample, restore the application to the original state. Follow the instructions in "Cleaning up the sample" on page 283 to clean up the sample.

6.4.6 Configuring general bindings

General bindings can be configured to be used across a range of policy sets. They can also be reused across applications and for trust service attachments. Since binding configuration can be time consuming and complex, this capability can reduce the number of bindings that must be configured. Users can configure a binding once and reuse it for multiple policy sets, even if the policy sets are for resources in different applications.

The general bindings that ship with WebSphere Application Server are initially set as the default bindings. However, you can choose a different binding as the default or change the level of binding that should be used as the default, for example, from cell-level binding to server-level binding. Default bindings are used when no application-specific binding or trust service binding is assigned to a policy set attachment.

In this section you configure a general binding and then apply it to the WeatherJavaBeanService service. You also configure XML encryption for the request message. On the client side, you use Mark's public key to encrypt the SOAP request. On the service side, you use Mark's private key to decrypt the inbound message.

Creating the general provider policy set binding

To create the general provider policy set binding:

1. In the administrative console (Figure 6-31 on page 307):

   a. In the left navigation pane, expand **Services** → **Policy sets** and click **General provider policy set bindings**.

      WebSphere Application Server V7 includes provider and client sample bindings for testing purposes. In the bindings, the product provides sample values for supporting tokens for different token types, such as the X.509 token and the username token. The bindings also include sample values for message protection information for token types such as X.509. Both provider and client sample bindings can be applied to the applications that are attached with a policy set. The sample bindings can be used as a base by making a copy of the bindings and then modifying the copy for your application needs. For example, you must change the key and keystore settings to ensure security and modify the binding settings to match your environment.

   b. In the right pane, select **Provider sample** and then click **Copy**.
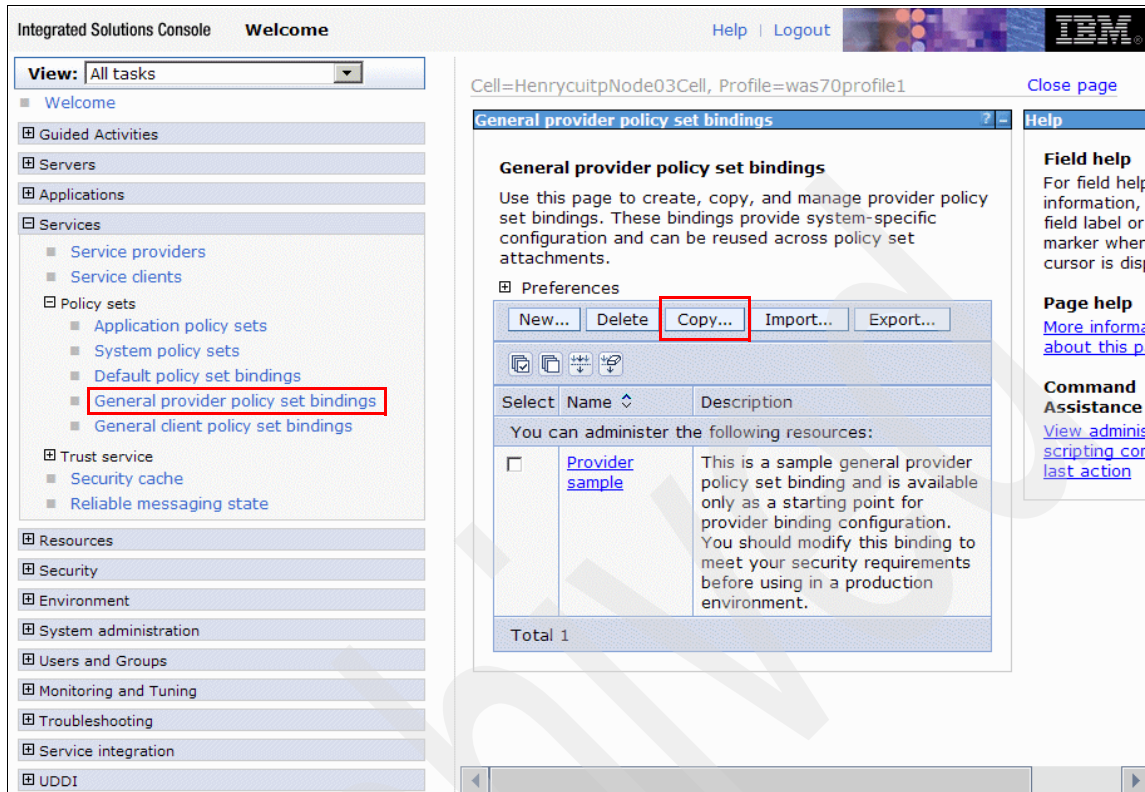
Figure 6-31   General provider policy set bindings

2. On the next page, for name, type `ITSO provider binding`. Click **OK** and then click **Save**.

3. Click **ITSO provider binding**.

4. On the next page for the binding, from the list of policies, click **WS-Security**.

5. Click **Authentication and protection**.

6. On the service side, use Mark's private key to decrypt the inbound message. Click **con_encx509token** (Figure 6-32).

Figure 6-32   Updating the decryption for the service provider

7. Click **Callback handler**.

8. Click **Custom keystore configuration**.

   On the Custom keystore configuration page (Figure 6-28 on page 297):

   a. In the Keystore section, for full path, update the name to point to the new receiver.jks keystore, for example C:\ITSO7758\PolicySet\receiver.jks.

   b. For type, select **JKS**.

   c. For password and confirm password, type `itso`.

   d. Under Key, in the Name field, type `cn=server, o=IBM, C=US`.

   e. In the Alias field, type `mark`.

   f. For password and confirm password, type `itso`.

   g. Click **OK** to close the custom keystore configuration.

9. Click **OK** again to close the callback handler configuration.

10. Click **OK** to return to the list of general provider policy set bindings.

11. Save the configuration.

Creating the general client policy set binding

To create the general client policy set binding:

1. In the administrative console, select **Services → Policy sets → General client policy set bindings**.

2. Select **Client sample** and then click **Copy**.

3. For name, type ITSO client binding. Click **OK**.

4. Click **Save**.

5. Click **ITSO client binding**.

6. Click **WS-Security**.

7. Click **Authentication and protection**.

8. On the client side, use Mark's public key to encrypt the SOAP request. Click **gen_encx509token**.

9. Click **Callback handler**.

10. Click **Custom keystore configuration**.

    On the Custom keystore configuration page:

    a. Enter the full path name for the sender.jks keystore, such as C:\ITSO7758\PolicySet\sender.jks.

    b. Select **JKS** as the type.

    c. For password and confirm password, enter itso.

    d. Select **JKS** as the type.

    e. For password and confirm password, enter itso.

    f. Under Key, in the Name field, type cn=client, o=IBM, C=US.

    g. In the Alias field, type mark.

    h. Click **OK** to close the custom keystore configuration.

11. Click **OK** to close the callback handler configuration.

12. Click **OK** again to return to the list of general client policy set bindings.

13. Click **Save**.

Applying the policy set and binding to the service and client

Attach the ITSO WSSecurity policy set and the general binding to the WeatherJavaBean application:

1. Assign the policy set to the weather forecast Web service:

   a. In the navigation pane, click **Services → Service providers**.
   b. Click **WeatherJavaBeanService**.

  c. Select **WeatherJavaBeanService**.

  d. Click **Attach Policy Set**, and from the list select **ITSO WSSecurity**.

  e. Select **WeatherJavaBeanService** again.

  f. Click **Assign Binding**, and from the list select **ITSO provider binding**.

  g. Save the configuration.

2. Assign the policy set to the client:

  a. In the navigation pane, click **Services** → **Service clients**.

  b. Click **WeatherJavaBeanService.**

  c. Select **WeatherJavaBeanService**.

  d. Click **Attach client policy set**, and from the list select **ITSO WSSecurity**.

  e. Select **WeatherJavaBeanService**.

  f. Click **Assign Binding**, and from the list select **ITSO client binding**.

3. Save the configuration.

4. Restart the application for the configuration changes for the general binding to take effect:

  a. In the navigation pane, select **Applications** → **Application Types** → **WebSphere Enterprise Applications**.

  b. Select **WeatherJavaBeanServer** and **WeatherJavaBeanWebClientEAR**.

  c. Click **Stop**.

  d. After the applications stop, click **Start**.

5. Follow the instructions in "Monitoring the SOAP traffic" on page 278 to test your application. You should see WS-Addressing information in the SOAP header. The SOAP message body is encrypted.

Exporting the customized policy set and general binding

After you configure the custom policy set and the general binding, you can export them from one system to another. Both the policy set and the binding are exported as an archive file, which can be imported to another system by using the import policy set and binding function. With this approach, you can configure the file once and reuse it for multiple systems. You can also import the policy set and the binding into a development environment such as Rational Application Developer. This is extremely useful, as the binding configuration can be time consuming and complex.

To export the customized policy set and general binding:

1. In the administrative console, expand **Services** → **Policy sets** → **Application policy sets**.

2. Select **ITSO WSSecurity** and click **Export**.

3. Select **ITSO WSSecurity.zip** and click **Save** to save the file to your local drive.

4. Expand **Services** → **Policy sets** → **General provider policy set bindings**.

5. Select **ITSO provider binding** and click **Export**.

6. Click **ITSO provider binding.zip** and click **Save** to save the file to your local drive.

7. Expand **Services** → **Policy sets** → **General client policy set bindings**.

8. Select **ITSO client binding** and click **Export**.

9. Click **ITSO client binding.zip** and then click **Save** to save the file to your local drive.

6.4.7 Exploring the integration with multiple security domains

WebSphere Application Server V7 offers multiple security domain support. With this feature you can create multiple security configurations and assign them to different applications in WebSphere Application Server processes. By creating multiple security domains, you can configure different security attributes for both administrative and user applications within a cell environment. You can configure each application to use a different security configuration by assigning the security domain to the server, cluster, or service integration bus that hosts the application. For instructions on how to configure multiple security domains, see "Achieving Web services interoperability between the WebSphere Web services Feature Pack and Windows® Communication Foundation, Part 2: Configure and test WS-Security" in IBM developerWorks at the following address:

http://publib.boulder.ibm.com/infocenter/wasinfo/V7r0/topic/com.ibm.
websphere.nd.multiplatform.doc/info/ae/ae/csec_sec_multiple_domains.html

In support of multiple security domains, each of the general bindings is scoped to a security domain (Figure 6-33). With domain scoping, applicable configuration settings in the bindings, such as Java Authentication and Authorization Service (JAAS) logins, are constrained based on the configuration attributes of the assigned domain. The default domain for bindings is the global security domain. The bindings scoped to the global security domain are available for all attachments, regardless of the domain in which the attached resource resides.

Figure 6-33   Multiple security domains for the general binding

6.4.8 Configuring policy sets by using wsadmin scripting

The WebSphere Application Server `wsadmin` tool gives you the ability to run scripts by using the JACL and Jython scripting languages. You can use the `wsadmin` tool to configure application or system policy sets for Web services. For instructions about how to configure policy sets by using `wsadmin` scripting, see

the "Configuring application and system policy sets for Web services using wsadmin scripting" topic in the WebSphere Application Server V7 Information Center at the following address:

http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/topic/com.ibm.
websphere.nd.multiplatform.doc/info/ae/ae/txml_wsfpsecurewebservices.html

6.5 Rational Application Developer support

Several policy sets are included with WebSphere Application Server V7, which are available in the Rational Application Developer workbench. Policy sets are attached to Web services and clients by using a wizard. All policy sets that are attached to the service are listed. More can be added or removed. Policy sets are also modifiable by using Rational Application Developer.

You can import policy sets and general bindings into Rational Application Developer. By doing so, you can work with policy sets and bindings that are exported from a production environment. You can also create application-specific client-side bindings with Rational Application Developer.

In this section we show how to use the tools that ship with Rational Application Developer to apply a policy set to the weather forecast Web service.

6.5.1 Importing the policy set and general binding into the workspace

To import policy sets into your workspace:

1. From the main menu, click **File** → **Import**.

2. In the Import window (Figure 6-34), expand **Web services** and select **WebSphere Policy Sets**. Click **Next**.
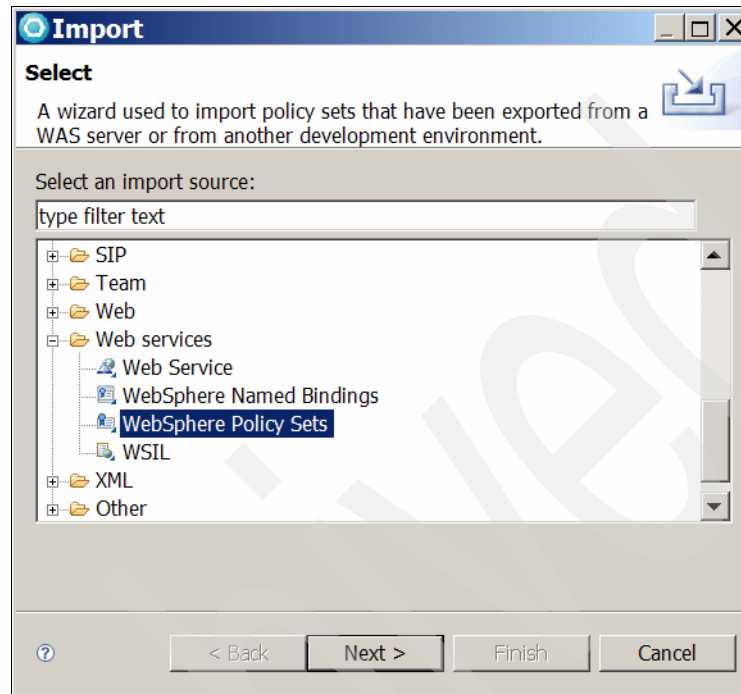
Figure 6-34   Importing WebSphere Policy Sets

3. Click **Browse** and select the **ITSO WSSecurity.zip** file that you created in "Exporting the customized policy set and general binding" on page 310. Click **Finish**.

To import the general Web service provider binding into your workspace:

1. From the main menu, click **File → Import**.

2. In the Import window, expand **Web services** and select **WebSphere Named Bindings**. Click **Next**.

3. Click **Browse** and select the **ITSO provider binding.zip** file that you created in "Exporting the customized policy set and general binding" on page 310. Click **Finish**.

To import the general Web service client binding into your workspace:

1. From the main menu, click **File → Import**.

2. In the Import window, expand **Web services** and select **WebSphere Named Bindings**. Click **Next**.

3. Click **Browse** and select the **ITSO client binding.zip** file that you created in "Exporting the customized policy set and general binding" on page 310. Click **Finish**.

To verify that the policy set and the general binding are imported successfully, click **Window** → **Preferences**. In the left pane of the Preferences window (Figure 6-35), select **Service Policies**. In the right pane of the Preferences window, you see the policy set and general bindings.
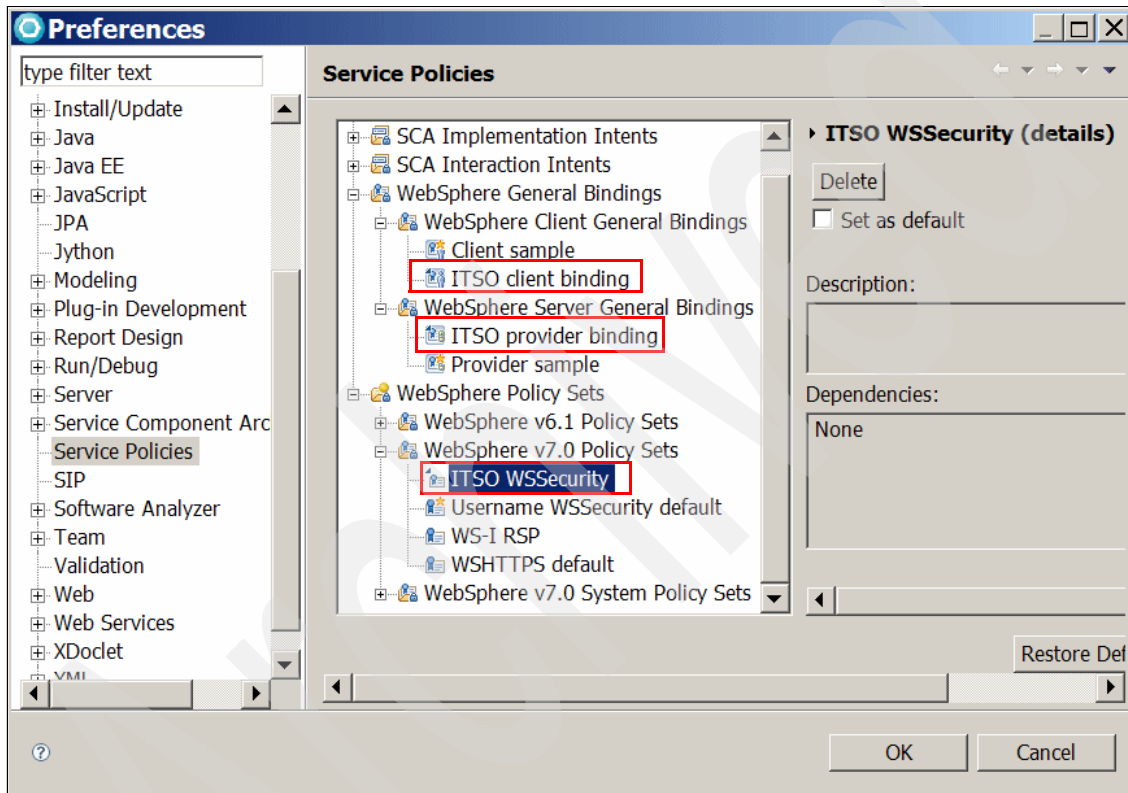
Figure 6-35   Service Policies in the Preferences window

6.5.2 Attaching a policy set and general binding to a service provider

You can apply a policy set at the service, port, or operation level of a Web service. Different policy sets can be applied to various endpoints and operations within a single Web service. However, the service and client must have the same policy set settings. This example applies a policy set to all operations.

To attach a policy set and general binding to a service provider:

1. In the Services view, expand the **JAX-WS** folder. Right-click the **WeatherJavaBeanService** service entry and select **Manage Policy Set Attachment** (Figure 6-36). The Service Side Policy Set Attachment Wizard opens.
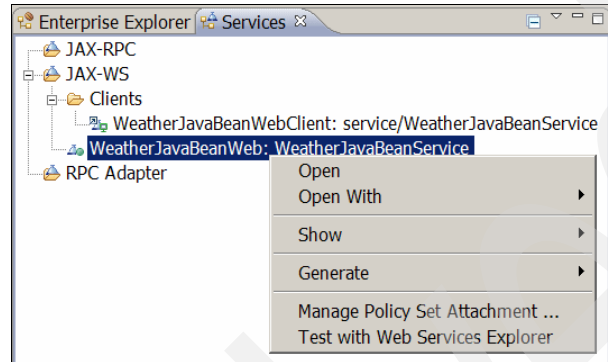
Figure 6-36 Selecting Manage Policy Set Attachment

2. In the Service Side Policy Set Attachment for WebSphere window (Figure 6-37):

   a. For application, select **WeatherJavaBeanServer**.

      The Application section lists all endpoints for this Web service that are already attached to a policy set. This table is currently empty because you have not attached any service endpoint to any policy set yet.

   b. Click **Add**.
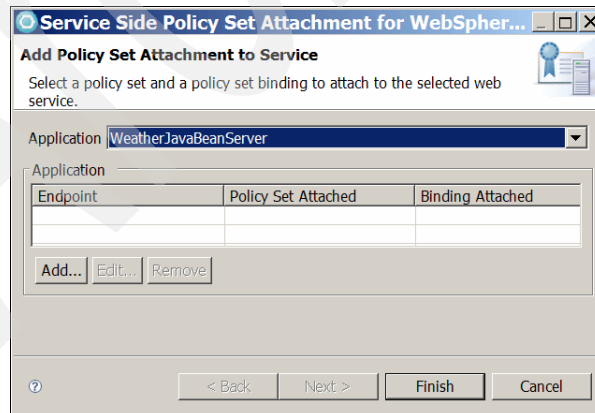
Figure 6-37 Add Policy Set Attachment to Service panel

3. In the End Point Definition Dialog window (Figure 6-38):

   a. Select the service endpoint to which to attach a policy set. You can attach the policy set to the entire service, to a specific endpoint, or to a specific operation.

      For this example, for endpoints, attach the policy set to the entire service. Therefore, accept the default of <all endpoints>.

   b. For policy set, select **ITSO WSSecurity**.

   c. For binding, select **ITSO provider binding**.

   d. Click **OK**.
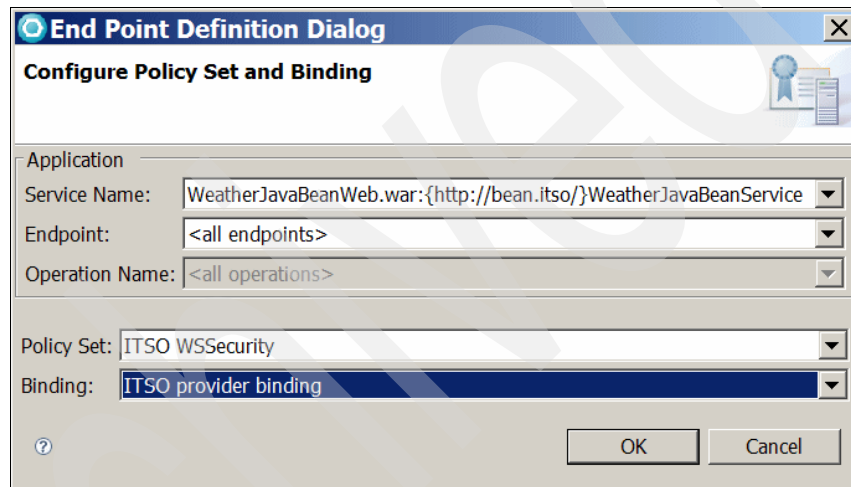
Figure 6-38   End Point Definition Dialog window

4. Click **Ignore** to ignore the WS-I warning message.

When you return to the Add Policy Set Attachment to Service panel (Figure 6-39), the entry that you just created is now in the application table.
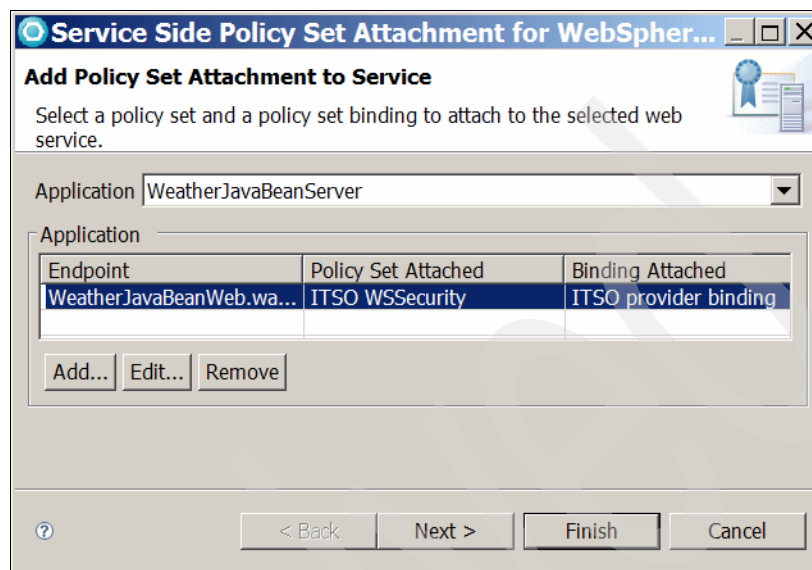
Figure 6-39   Policy set applied to the service

5. In the Add Policy Set Attachment to Service panel, click **Finish**.

After a policy set is attached to a Web service, a `policyAttachments.xml` file is generated in the `WeatherJavaBeanServer\META-INF` folder (Example 6-3). This file is appended for each additional policy set setting that is added to any service within the EAR file.

Example 6-3   The policyAttachments.xml file

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<psa:PolicySetAttachment
xmlns:psa="http://www.ibm.com/xmlns/prod/websphere/200605/policysetattachment"
xmlns:ps="http://www.ibm.com/xmlns/prod/websphere/200605/policyset">
    <psa:PolicySetReference name="ITSO WSSecurity" id="com.ibm.ast.ws.local.qos.policyset.id1">
        <psa:PolicySetBinding scope="domain" name="ITSO provider binding"/>
        <psa:Resource
pattern="WebService:/WeatherJavaBeanWeb.war:{http://bean.itso/}WeatherJavaBeanService"/>
    </psa:PolicySetReference>
</psa:PolicySetAttachment>
```

6.5.3 Attaching policy set and general binding to Web service client

To attach the policy set and the general binding to the Web service client:

1. In the Services view (Figure 6-40), expand **JAX-WS** → **Clients**. Right-click **WeatherJavaBeanService** and select **Manage policy set Attachment**.

   The Client Side Policy Set Attachment Wizard opens.
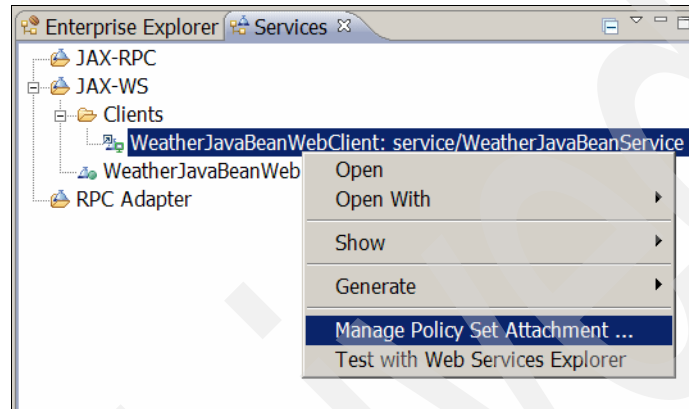
Figure 6-40   Attaching a policy set to the client

2. In the Configure Policy Acquisition for Web service Client panel, click **Next**.

3. In the Add Policy Set Attachment to Web service Client panel, click **Add**.

4. In the End Point Definition Dialog window (Figure 6-41), apply the ITSO WSSecurity policy set to the client. The policy sets and bindings configuration for our Web service client must match the service to function correctly.

   a. For policy set, select the **ITSO WSSecurity** policy set.
   b. In the Binding field, select **ITSO client binding**.
   c. Click **OK**.
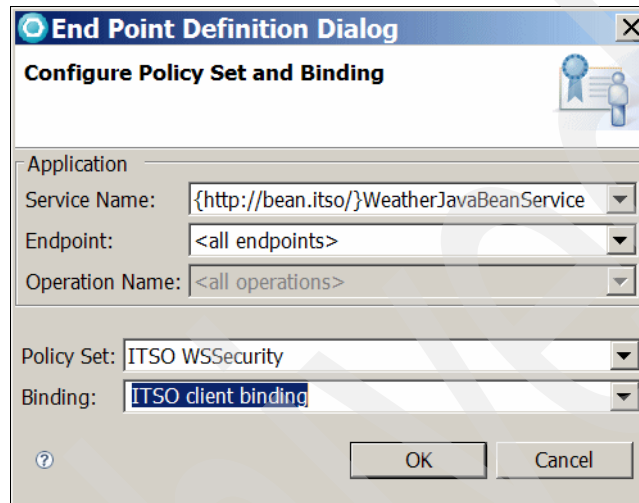
Figure 6-41   End Point Definition Dialog window for the client

5. Click **Ignore** to the message that is displayed.

6. Click **Finish**.

After a policy set is attached to a Web service, a `clientPolicyAttachments.xml` file is generated in the `WeatherJavaBeanWebClientEAR\META-INF` folder (Example 6-4). This file is appended for each additional policy set setting that is added to any service within the EAR file.

Example 6-4   The clientPolicyAttachments.xml file

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<psa:PolicySetAttachment
xmlns:psa="http://www.ibm.com/xmlns/prod/websphere/200605/policysetattachment"
xmlns:ps="http://www.ibm.com/xmlns/prod/websphere/200605/policyset">
    <psa:PolicySetReference name="ITSO WSSecurity" id="com.ibm.ast.ws.local.qos.policyset.id1">
        <psa:PolicySetBinding scope="domain" name="ITSO client binding"/>
        <psa:Resource pattern="WebService:/{http://bean.itso/}WeatherJavaBeanService"/>
    </psa:PolicySetReference>
</psa:PolicySetAttachment>
```

7. Follow the instructions in "Monitoring the SOAP traffic" on page 278 to test your application. You should see WS-Addressing information in the SOAP header. The SOAP message body is encrypted.

Cleaning up the sample

To proceed with the next sample, remove the general binding for the client. To detach the client-side policy set and binding:

1. In the Services view, expand the **JAX-WS** → **Clients**. Right-click **WeatherJavaBeanService** and select **Manage policy set Attachment**.

2. In the window that opens, click **Next**.

3. Select **Remove** and click **Finish**.

6.5.4 Attaching policy set and application-specific binding to Web service client

Rational Application Developer supports the client-side application-specific binding. For example, if you want to use your own keystore and key to encrypt the SOAP request, a wizard is provided to configure such a custom binding. The policy set configuration wizard currently does not support Web service provider custom binding configurations.

To configure the client custom binding:

1. In the Services view, expand **JAX-WS** → **Clients**. Right-click **WeatherJavaBeanService** and select **Manage policy set Attachment**.

2. In the window that opens, click **Next**.

3. In the Add Policy Set Attachment to Web Service Client window, click **Add**.

4. In the End Point Definition Dialog window (Figure 6-42), in the Binding field, type `ITSO custom binding`. Click **OK**.
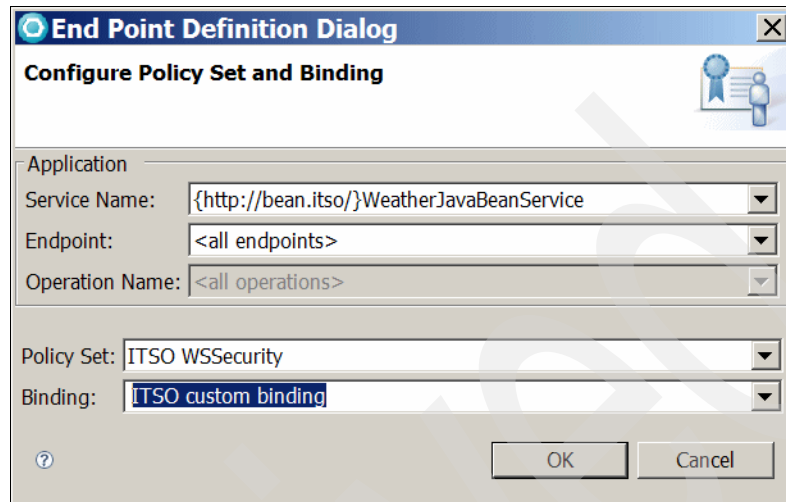
Figure 6-42   Configuring the application-specific binding

5. In the message window that opens, click **Ignore**.

6. In the Add Policy Set Attachment to Web Service Client panel (Figure 6-43), in the Bindings Configuration table, under Policy Type, select **WSSecurity**.

The policy types listed in this table are in the policy set. Any of these policies that require additional configuration information are marked. For the ITSO WSSecurity policy set, the WSAddressing policy does not have to be configured, while the WSSecurity policy must be configured. The warning message shown at the top of the panel is displayed because you have not configured the client-side binding.

Click **Configure**.
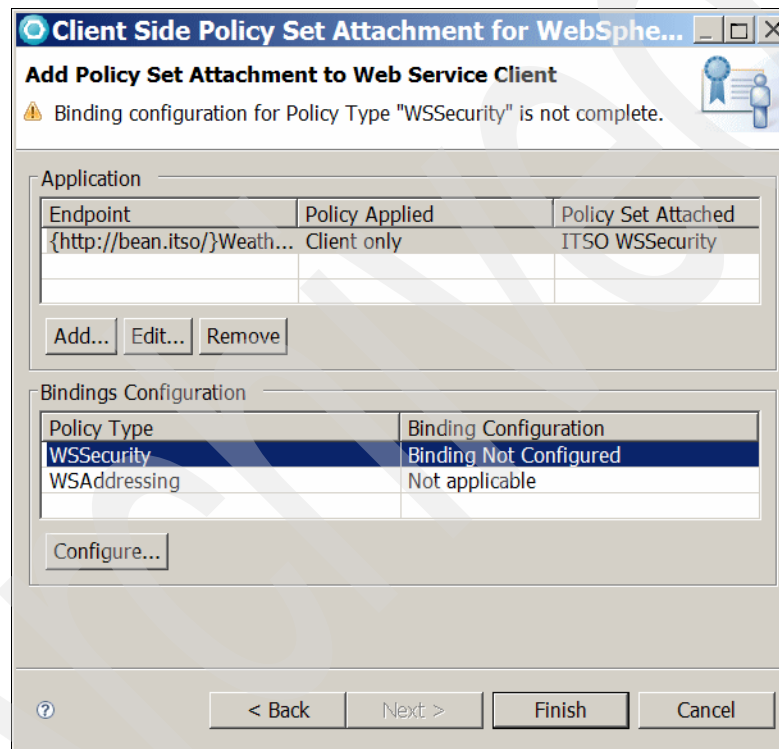
Figure 6-43   Configuration warning message

7. In the WSSecurity Binding Configuration panel (Figure 6-44), click **Key Store Settings**.
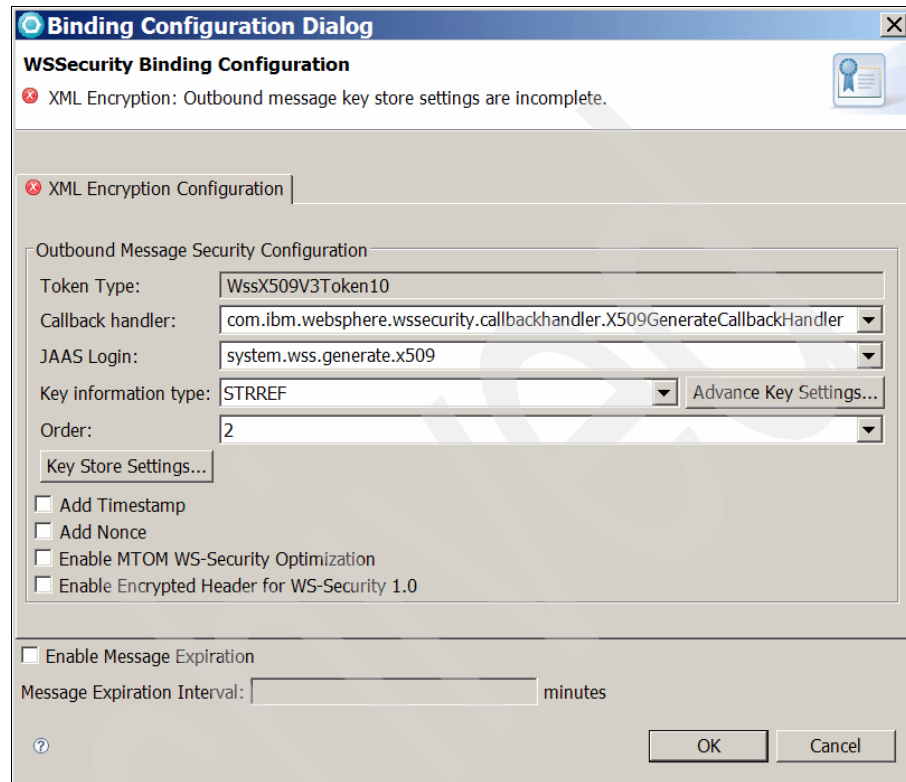
Figure 6-44   WSSecurity Binding Configuration panel

8. In the Key Store Settings Dialog window (Figure 6-45):
    a. For keystore path, type `C:\ITSO7758\sender.jks`.
    b. For keystore password, type `itso`.
    c. For keystore type, enter `JKS`.
    d. For key alias, type `mark`.
    e. Click **OK**.

Figure 6-45   Key store settings

9. In the WSSecurity Binding Configuration panel (Figure 6-44 on page 324), click **OK**.

10. In the next window, click **Finish**.

11. Test the application by following the instructions in "Monitoring the SOAP traffic" on page 302.

6.6 More information

The WebSphere Application Server Application V7 Information Center includes tremendous detail about policy sets. A good pace to start is with the "WebSphere Application Server documentation for Feature Pack for Web 2.0, Version 1.0 Fix Pack 2" topic at the following address:

http://publib.boulder.ibm.com/infocenter/wasinfo/V7r0/index.jsp?topic=/com.ibm.websphere.express.doc/info/exp/ae/cwbs_wsspsps.html

For complete instructions about applying the WS-Security policy set to JAX-WS Web services, see the developerWorks article "Achieving Web services interoperability between the WebSphere Web services Feature Pack and

Windows Communication Foundation, Part 2: Configure and test WS-Security" at the following address:

http://www.ibm.com/developerworks/websphere/library/techarticles/0712_levay/0712_levay.html

Also about developerWorks is the article "Using the WS-I Supply Chain Management application in WebSphere V6.1 Web services Feature Pack, Part 2: Apply WS-Security 1.0 to the JAX-WS SCM application," which shows a variety of WS-Security configurations using the policy set. You can find this article at the following address:

http://www.ibm.com/developerworks/websphere/library/techarticles/0801_zeitouni/0801_zeitouni.html

You can use the policy set for the JAX-WS Web services thin client. For a complete example of the usage from a Web services thin client, see *Web Services Feature Pack for WebSphere Application Server V6.1*, SG24-7618.

7

WS-Policy and WS-MetadataExchange

Web Services Policy (WS-Policy) is an interoperability standard that is used to describe and communicate the policies of a Web service so that service providers can export policy requirements in a standard format. Web Services Metadata Exchange (WS-MetadataExchange or WS-MEX) defines a mechanism to retrieve Web services metadata from a Web service endpoint.

In this chapter first we introduce the key concepts of WS-Policy. Then we explain the WS-Policy and WS-MEX support in WebSphere Application Server V7. We also guide you through samples of using WS-Policy and WS-MEX in WebSphere Application Server. Finally, we explore the tools support in Rational Application Developer V7.5.

The chapter contains the following topics:

7.1 Overview of the WS-Policy specification

The WS-Policy framework specification provides a way to describe and communicate the policies associated with Web services. A service provider can export its policy requirements in a standardized format. By doing so, service clients can combine these requirements with their own capabilities to establish the policies required for a specific interaction. This practice allows for the interoperability for quality of service (QoS) configurations and easier configuration of Web service clients.

7.1.1 WS-Policy concepts

WS-Policy introduces several key concepts. The following list of WS-Policy terms and their associated WS-Policy definitions is taken directly from the WS-Policy specification:

Policy   A potentially empty collection of policy alternatives.

Policy alternative   A potentially empty collection of policy assertions.

Policy assertion   Represents an individual requirement, capability, or other property of a behavior. For example, a policy assertion can require a username security token to be used for Web services authentication.

Policy attachment   A mechanism for associating a policy with one or more policy scopes.

Policy expression   An XML information set (Infoset) representation of a policy, either in a normal form or in an equivalent compact form.

Policy scope   A collection of policy subjects to which a policy can apply.

7.1.2 WS-Policy operators

Policies are used to convey a set of capabilities, requirements, and general characteristics of entities. These are generally expressible as a set of policy alternatives. Policy operators are used to group policy assertions into policy alternatives. WS-Policy introduces three operators that you might see in the WS-Policy XML for a service definition:

<wsp:ExactlyOne>   Specifies a list of possible policy alternatives, one of which must be adhered to in order to conform to the policy.

<wsp:All>   Specifies a list of policy alternatives, all of which must be adhered to in order to conform to the policy.

<wsp:Policy>   The semantics for <wsp:Policy> are the same as for <wsp:All>.

To compactly express complex policies, policy operators can be recursively nested. That is, one or more instances of wsp:Policy, wsp:All, and wsp:ExactlyOne can be nested within wsp:Policy, wsp:All, and wsp:ExactlyOne.

Example 7-1 shows a policy document that uses assertions that are defined in WS-SecurityPolicy.

Example 7-1   Sample WS-Policy document

```
(01) <wsp:Policy
        xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702"
        xmlns:wsp="http://www.w3.org/ns/ws-policy">
(02) <wsp:ExactlyOne>
(03)   <wsp:All>
(04)      <sp:SignedParts>
(05)         <sp:Header Namespace="http://www.w3.org/2005/08/addressing" />
(06)      </sp:SignedParts>
(07)   </wsp:All>
(08)   <wsp:All>
(09)      <sp:EncryptedParts>
(10)         <sp:Body />
(11)      </sp:EncryptedParts>
(12)   </wsp:All>
(13) </wsp:ExactlyOne>
(14) </wsp:Policy>
```

Essentially, a Web service that uses this policy file has one requirement for the clients: The client must either sign the message header or encrypt the message body. Let us look more closely at the sample in Example 7-1 on page 329:

► In Line (01), the `http://www.w3.org/ns/ws-policy` namespace (with the wsp prefix) defines the policy language, which is the generic syntax shared by all domains to define the policy. The policy is specific to the WS-SecurityPolicy domain as defined by the `http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702` namespace (noted by the sp prefix in Example 7-1 on page 329).

The WS-Policy specification creates namespaces that define the policy languages. The *policy language* is the generic syntax shared by all domains to define policy configurations. Each domain, such as Web services Security (WS-Security), Web services Transactions (WS-Transaction), or Web services Addressing (WS-Addressing), has its own set of predefined policy assertions that can be used within the generic policy language. The policy assertions describe the configurations that are needed to communicate with the service provider.

► Lines (03–07) represent one policy alternative for signing a message header.

► Lines (08–12) represent a second policy alternative for encrypting a message body.

► Lines (02–13) illustrate the ExactlyOne policy operator. Policy operators group policy assertions into policy alternatives. A valid interpretation of the policy in Example 7-1 on page 329 is that an invocation of a Web service either signs or encrypts the message body.

7.1.3 WS-PolicyAttachment

The Web services PolicyAttachment (WS-PolicyAttachment) specification defines two general-purpose mechanisms for associating policies with the subjects to which they apply. The policies can be defined as part of existing metadata about the subject. Alternatively, the policies can be defined independently and associated through an external binding to the subject.

To enable a Web services policy to be used with existing Web service technologies, the WS-PolicyAttachment specification describes the use of these general-purpose mechanisms with WSDL definitions and Universal Description, Discovery, and Integration (UDDI). Example 7-2 shows a Web Services Description Language (WSDL) with the WS-Addressing policy attached.

Example 7-2   WSDL with a policy attachment

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="WeatherJavaBeanService" targetNamespace="http://bean.itso/"
   xmlns="http://schemas.xmlsoap.org/wsdl/"
xmlns:wsp="http://www.w3.org/ns/ws-policy"

xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility
-1.0.xsd"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:tns="http://bean.itso/"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">
   ... ...
   <portType name="WeatherJavaBeanDelegate">
      <operation name="getDayForecast">
      ... ...
      </operation>
   </portType>
   <binding name="WeatherJavaBeanPortBinding" type="tns:WeatherJavaBeanDelegate">
      <soap:binding style="document"
         transport="http://schemas.xmlsoap.org/soap/http" />
      <wsp:PolicyReference URI="#429de5ff-5f0c-4742-9ce9-78ff93985bc1" />
      <operation name="getDayForecast">
      ... ...
      </operation>
   </binding>
   <service name="WeatherJavaBeanService">
      <port name="WeatherJavaBeanPort" binding="tns:WeatherJavaBeanPortBinding">
         <soap:address
            location="http://localhost:9080/WeatherJavaBeanWeb/WeatherJavaBeanService"
/>
      </port>
   </service>
   <wsp:Policy wsu:Id="429de5ff-5f0c-4742-9ce9-78ff93985bc1">
      <wsp:ExactlyOne>
         <wsp:All>
            <addressing:Addressing
               xmlns:addressing="http://www.w3.org/2007/05/addressing/metadata">
               <wsp:Policy>
                  <wsp:ExactlyOne>
```

```
                <wsp:All>
                </wsp:All>
              </wsp:ExactlyOne>
            </wsp:Policy>
          </addressing:Addressing>
        </wsp:All>
  ... ...
      </wsp:ExactlyOne>
    </wsp:Policy>
</definitions>
```

In Example 7-2 on page 331, the Web service operation "getDayForecast" that is being bound to the specified portType must enforce the Web service policy. To reference a policy expression within the WSDL, the wsu:Id attribute is used to identify a policy expression and a URI to this ID value for referencing this policy expression by using a `wsp: PolicyReference` element.

7.1.4 Policy intersection

Policy intersection is the process of comparing Web services policies for common alternatives. The interaction takes place when only both sides of an interaction agree on at least one policy alternative. This operation is typically used when a requester expresses its capabilities and requirements in policy form. The requester policy indicates the assertions that its local runtime infrastructure is capable of processing. The intersection of two policies gives zero or more alternatives on which both parties agree.

As an example of intersection, consider Example 7-3 in which you have a service provider policy (P1).

Example 7-3   Service provider policy P1

```
(01) <wsp:Policy
        xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702"
        xmlns:wsp="http://www.w3.org/ns/ws-policy" >
        <!-- Policy P1 -->
(02)    <wsp:ExactlyOne>
(03)      <wsp:All> <!-- Alternative A1 -->
(04)        <sp:SignedElements>
(05)          <sp:Body/>
(06)        </sp:SignedElements>
(07)        <sp:EncryptedElements>
(08)          <sp:Body/>
(09)        </sp:EncryptedElements>
(10)      </wsp:All>
```

```
(11)        <wsp:All> <!-- Alternative A2 -->
(12)          <sp:EncryptedParts>
(13)            <sp:Body />
(14)          </sp:EncryptedParts>
(15)        </wsp:All>
(16)      </wsp:ExactlyOne>
(17) </wsp:Policy>
```

Example 7-3 on page 332 indicates two policy alternatives:

- ▶ The first alternative (A1, lines 03–10) contains two policy assertions. One indicates that the SOAP body should be signed (lines 04–06). The other assertion (lines 07–09) indicates that the SOAP body should be encrypted.

- ▶ The second alternative (A2, lines 11–15) contains one assertion, which indicates that the SOAP body must be encrypted (lines 12–14).

Example 7-4 shows the service requester policy (P2).

Example 7-4   Service requester policy P2

```
(01) <wsp:Policy
        xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702"
        xmlns:wsp="http://www.w3.org/ns/ws-policy" >
        <!-- Policy P2 -->
(02)    <wsp:ExactlyOne>
(03)      <wsp:All> <!-- Alternative A3 -->
(04)        <sp:EncryptedParts>
(05)          <sp:Body />
(06)        </sp:EncryptedParts>
(07)      </wsp:All>
(08)      <wsp:All> <!-- Alternative A4 -->
(09)        <sp:SignedElements>
(10)          <sp:Body/>
(11)        </sp:SignedElements>
(12)      </wsp:All>
(13)    </wsp:ExactlyOne>
(14) </wsp:Policy>
```

Example 7-4 indicates two policy alternatives:

- ▶ The first alternative (A3, lines 03–07) contains one assertion, which indicates that the SOAP body must be encrypted (lines 04–06).

- ▶ The second alternative (A4, lines 08–12) contains one assertion, which indicates that the SOAP body must be signed (lines 12–14).

Because there is only one alternative (A2) in policy P1 with the same assertion type as another alternative (A3) in policy P2, the intersection is a policy with a single alternative that contains all of the assertions in A2 and in A3, as shown in Example 7-5.

Example 7-5   Policy intersection result

```
(01) <wsp:Policy
        xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702"
        xmlns:wsp="http://www.w3.org/ns/ws-policy" >
        <!-- Intersection of P1 and P2 -->
(02)    <wsp:ExactlyOne>
(03)      <wsp:All>
(04)        <sp:EncryptedParts>
(05)          <sp:Body />
(06)        </sp:EncryptedParts>
(07)        <sp:EncryptedParts>
(08)          <sp:Body />
(09)        </sp:EncryptedParts>
(10)      </wsp:All>
(11)    </wsp:ExactlyOne>
(12) </wsp:Policy>
```

Note that there are two assertions of the type sp:EncryptedParts, one from each of the input policies. In general, whether two assertions of the same type are compatible or the repetition is a redundancy depends on the domain-specific semantics of the assertion type. If the assertions have no parameters and the assertions in nested policy expressions have no parameters, multiple assertions of the type within a policy alternative in the intersection result have the same meaning as a single assertion of the type within the policy alternative.

Based on the semantics of multiple assertions of the EncryptedParts assertion type, as specified in the WS-SecurityPolicy specification, one of the sp:EncryptedParts assertions in Example 7-5 is redundant and can be removed from the result. The intersection of the two policies provides one alternative that the SOAP body must be encrypted.

7.2 WS-Policy support in WebSphere Application Server V7

WebSphere Application Server conforms to the WS-Policy Framework V1.5 specification. With this support, service providers can share their policy set

configurations in an interoperable format that is embedded in the WSDL document.

A policy represents the capabilities and requirements of a Web service, for example, whether a message is secure and how to secure it, and whether a message is delivered reliably and how this is achieved. You can communicate the policy configuration to any other client, service registry, or service that supports the WS-Policy specification, including non-WebSphere Application Server products in a heterogeneous environment. Clients can then use WSDL documents that contain this policy data to dynamically configure themselves at run time. You must administratively configure the service to embed the policy data in its WSDL document and configure client environments to dynamically use that information.

The following WS-Policy assertion specifications are supported in WebSphere Application Server V7:

► WS-Addressing

  http://www.w3.org/TR/ws-addr-core/

► WS-Transaction

  http://docs.oasis-open.org/ws-tx/wstx-wsat-1.1-spec-os/wstx-wsat-1.1
  -spec-os.html

► WS-ReliableMessaging

  http://docs.oasis-open.org/ws-rx/wsrm/200702

► WS-SecurityPolicy

  http://docs.oasis-open.org/ws-sx/ws-securitypolicy/v1.2/ws-security
  policy.html

7.2.1 Service provider policy sharing

A WebSphere Application Server service provider can share its current policy configuration through its WSDL. The policy configuration is in the standard WSDL WS-PolicyAttachment format so that it can be shared with other clients, service registries, or services that support the WS-Policy specification.

You can make the policy configuration of a Java API for XML Web services (JAX-WS) service endpoint available to share in two ways:

▶ Include the policy configuration of the service provider in the WSDL. The WSDL is then available for publishing, or to be obtained using an HTTP Get request.

▶ Enable the WS-MEX protocol so that the policy configuration of the service provider is included in the WSDL and is available to a WS-MetadataExchange GetMetadata request. An advantage of using the WS-MEX protocol is that you can apply message-level security to WS-MetadataExchange GetMetadata requests by using a suitable system policy set.

7.2.2 Service client policy acquisition

A WebSphere Application Server service client can obtain the policy of the service provider in the standard WS-PolicyAttachments format and use this information to establish a configuration that is acceptable to both the client and the service provider. The client can be configured dynamically, based on the policies that are supported by its service provider. A client can acquire the provider policy by using the following mechanisms:

▶ A WSDL publication
▶ An HTTP Get request
▶ In metadata returned by a WS-MetadataExchange request

7.2.3 Policy intersection in WebSphere Application Server

As explained previously, *policy intersection* is the comparison of a client policy and a provider policy to determine whether they are compatible. It entails the calculation of a new policy that complies with the requirements and capabilities of both the client policy and the provider policy.

When you obtain the policy of a service provider, you can choose to use the provider policy only or to use the client and the provider policy in WebSphere Application Server. Policy intersection has the following outcomes:

▶ When you specify *provider policy only*, the calculated policy is based on all the policies that the WebSphere Application Server client supports that are intersected by the provider policy. Effectively, the provider determines the policy as long as the client can support that policy. This policy configuration is available if the scope point (endpoint operation) where the provider policy is attached is not attached to a client policy set and does not inherit a policy set attachment from parent scope points.

► When you specify *client and provider policy*, the calculated policy is based on the policy that is acceptable to the client that is intersected by the provider policy. Effectively, the policy conforms to the client policy set, but it might be restricted further by the policies that are dictated by the provider.

The policy that is acceptable to the client is defined by the policy set that is either attached to the client scope point or that the client scope point inherits from a parent scope point. This policy configuration is available if the scope point (endpoint operation) where the provider policy is attached is attached to a client policy set or inherits a policy set attachment from parent scope points.

7.2.4 Relationship to policy sets

Policy sets provide reusable constructs that configure qualities of service that can be shared. The same configuration for policy sets can be used for the client and server side. You can combine a collection of policy documents in a policy set to address your requirements for qualities of service. Keep in mind that you might have to define the bindings for the policy set.

A *binding* is the specific configuration of a QoS. It contains the environment and platform-specific information, such as keys for signing, keystore information, or persistent information.

WS-Policy is a separate concept from policy sets. Policy sets are not inherently concerned with the WS-Policy specification. Instead, they deal with the specific configuration of Web services. Consider policy sets as a front end to WS-Policy. Policy sets provide a mechanism to specify policy configurations within a WebSphere Application Server environment, which is enough in certain environments. Policy sets do not provide a mechanism to communicate this policy configuration to non-WebSphere Application Server partners in a heterogeneous environment. Policy sets also do not provide a mechanism for the client to calculate an effective policy that is acceptable to both the service client and provider based the intersection of a list of client and provider policies. WS-Policy provides these extra capabilities.

7.3 WS-MetadataExchange

The WS-MEX specification defines a mechanism to retrieve Web services metadata from an Web service endpoint.

7.3.1 Overview of WS-MetadataExchange

Web services use metadata to describe what other endpoints need to know to interact with them. There are three types of metadata:

WS-Policy   Describes the capabilities, requirements, and general characteristics of Web services

WSDL   Describes abstract message operations, concrete network protocols, and endpoint addresses used by Web services

XML Schema   Describes the structure and contents of XML-based messages received by and sent by Web services

To bootstrap communication with Web services, the WS-MEX specification defines the following mechanisms to retrieve metadata:

► A WS-MetadataExchange request can be realized by sending a WS-Transfer Get request to a resource endpoint that represents the metadata of the actual endpoint. A WS-Transfer Get returns a one-time snapshot of the endpoint. When issued against the actual endpoint, this is a snapshot of the endpoint's data. To get a snapshot of the endpoint's metadata, a special metadata resource endpoint must be used.

► A WS-MetadataExchange request can be realized by issuing a WS-MetadataExchange GetMetadata request to the actual endpoint. The GetMetadata response can return the following information:

– A reference to the resource endpoint

The reference is resolved by using a WS-Transfer request (or equivalent requests such as WS-ResourceTransfer).

– The metadata inline in the response itself.

7.3.2 WS-MetadataExchange support

WebSphere Application Server V7 supports usage of the WS-MetadataExchange 1.1 GetMetadata request to return metadata in a response. A service provider can use this mechanism to make available WSDL that is annotated with WS-Policy information. That is, the service provider can share its policies. A service client can use this mechanism to obtain WSDL that is annotated with WS-Policy information from a service provider and then apply those policies. The policy configuration must be in the WS-PolicyAttachments format in the WSDL of the service provider.

WebSphere Application Server V7 does not provide full support for the WS-MEX specification. Rather, the implementation is focused on key support for

WS-Policy. It is not possible to explicitly drive WS-MEX in WebSphere Application Server V7. Instead, it is used purely for the export and acquisition of the WS-Policy by the run time.

7.3.3 Securing WS-MetadataExchange requests

You can secure WS-MetadataExchange requests by using transport-level security or message-level security (WS-Security). WS-MEX is the preferred mechanism (over HTTP GET) for policy exchange in cases where the metadata exchange requires message-level security because transport-level security is either not available on the application endpoint or is inadequate. An advantage of message-level security is that it provides end-to-end security by incorporating security features in the header of the SOAP message.

To provide message-level security for a GetMetadata request, you must attach a system policy set that contains only WS-Security or WS-Addressing policies. You can specify general bindings that are scoped either to the global domain or to the security domain of the service.

7.4 Applying WS-Policy and WS-MEX to the sample application

In this section we provide examples to show how to apply the WS-Policy and WS-MEX to our WeatherJavaBean application by using the WebSphere Application Server administrative console. You apply the Username WSSecurity policy set to the WeatherJavaBean application. After the policy set is applied to the provider, you can share its policy configuration in published WSDL. On the client side, the WSDL is obtained by using an HTTP Get request or the WS-MetadataExchange GetMetadata request. The client is configured dynamically based on the policies that are supported by the provider.

7.4.1 Preparing for the example

The example application used in this chapter is the WeatherJavaBean application. It is similar to the application discussed in 4.2.2, "Web services development from an existing Java bean" on page 183.

The instructions in this chapter assume that you are using Rational Application Developer V7.5 with its integrated WebSphere Application Server V7 test environment. To follow along with the instructions, you can download the sample application, import it into a workspace, and install it to the test environment.

Configuring WebSphere Application Server security

Because this example uses the Username WSSecurity policy set, you must also enable WebSphere Application Server security to authenticate the username token. You can enable the security either during or after profile creation. If you enable security during profile creation, you can skip to the next section. The federated repository is used as the default user repository at profile creation time.

To enable WebSphere Application Server security:

1. Start the deployment manager.

2. In the administrative console, expand **Security** → **Global security**.

3. Click **Security Configuration Wizard**.

4. Select **Enable application security** and click **Next**.

5. Select **Federated repositories** and click **Next**.

6. Enter the user ID and password to add to the repository for administration. In the test environment for this example we enter `admin` as the primary administrative user name and `admin` as the password. Click **Next**.

7. Click **Finish**.

8. Save the changes and restart the WebSphere environment.

Enabling security in Rational Application Developer

If you are using Rational Application Developer and its integrated WebSphere Application Server, to start the server from the workbench:

1. In the Servers view, double-click **WebSphere Application Server V7.0** to open the server configuration editor.

2. In the server configuration editor (Figure 7-1):

    a. Under Publishing settings for WebSphere Application Server, select **Run server with resources on Server**.

    b. Expand the **Security** section.

    c. Select **Security is enabled on this server**. Enter a user ID and password. The User ID and Password fields specify the administrator user of the WebSphere administrative console. These values must be the same as those entered in the Security Configuration wizard window.

    d. Select **Automatically trust server certificate during SSL handshake**.

Figure 7-1   Server editor settings

3. Save and close the server configuration editor.

Creating a general binding for the Username WSSecurity policy set

WebSphere Application Server V7.0 includes provider and client sample bindings for testing purposes. In the bindings, the product provides sample values for supporting tokens for different token types, such as the X.509 token and the username token. The bindings also include sample values for message protection

information for token types such as X.509. Both provider and client sample bindings can be applied to the applications that are attached with a policy set.

When using the Username WSSecurity default policy set, you must configure the user name and password for username token authentication separately from the security settings defined in the bindings. The sample binding does not include a user name or password for token authentication, because it is specific to the target deployed system. You must specify a valid user name and password in your environment by using the WebSphere administrative console. You can do this by copying the sample binding and customizing it:

1. In the administrative console, expand **Services** → **Policy sets** → **General client policy set bindings**.

2. Select **Client sample** and click **Copy**.

3. For name, type `ITSO Username WSSecurity binding` and click **OK**.

4. Click **ITSO Username WSSecurity binding** to edit the binding.

5. Click **WS-Security** → **Authentication and protection**.

6. In the Authentication tokens list, select **gen_signunametoken** to edit the username token settings.

7. In the Additional Bindings section at the bottom of the page, click **Callback handler**.

8. Enter the WebSphere administrator user name and password. Confirm the password and click **Apply** (Figure 7-2).

Figure 7-2   Configuring the username token

9. Click **Save**.

You have configured the client-side general binding to use for the WeatherJavaBean application.

7.4.2 Configuring a service provider to share its policy configuration

In this section you include the policy configuration of the service provider in the WSDL. The WSDL is then available for the client to obtain using an HTTP Get request.

To configure a service provider to share its policy configuration:

1. In the administrative console, expand **Services** → **Service providers**.

2. Click **WeatherJavaBeanService** to configure it with a policy.

3. On the next page:

   a. Select **WeatherJavaBeanService**. Click **Attach** and select **Username WSSecurity default**.

   b. Select **WeatherJavaBeanService**. Click **Assign Binding** and select **Provider sample**. In theory, you do not need to do this step, because the provider sample binding is set as the default binding. However, at the time at which this book was written, we had to do this manually. Figure 7-3 shows the result.

   c. Click the **Disabled** hyperlink to share its policy configuration. By default, the policy configuration is not available in its WSDL, which is why you see *Disabled* in the Policy Sharing column.

Figure 7-3   Assigning the policy set and binding

4. To include the policy configuration of the service provider in its WSDL so that it can be either published or obtained by using an HTTP Get request, select **Exported WSDL** (Figure 7-4). Click **OK**.

Figure 7-4   Share the policy using the exported WSDL

5. Click **Save**.

The policy configuration of the service provider is now available to its clients. The WSDL of the service provider contains the current policy configuration in the WS-PolicyAttachments format so that it is available to other clients, service registries, and services that support the WS-Policy specification. The link in the Policy Sharing column on the Service provider policy sets and bindings page changes to *Enabled* (Figure 7-5).

Select	Service/Endpoint/Operation ⇕	Attached Policy Set ⇕	Binding ⇕	Policy Sharing ⇕
	You can administer the following resources:			
☐	WeatherJavaBeanService	Username WSSecurity default	Provider sample	Enabled
☐	WeatherJavaBeanPort	Username WSSecurity default (inherited)	Provider sample (inherited)	Enabled (inherited)
☐	getDayForecast	Username WSSecurity default (inherited)	Provider sample (inherited)	Enabled (inherited)
☐	getForecast	Username WSSecurity default (inherited)	Provider sample (inherited)	Enabled (inherited)
☐	getTemperatures	Username WSSecurity default (inherited)	Provider sample (inherited)	Enabled (inherited)
☐	setWeather	Username WSSecurity default (inherited)	Provider sample (inherited)	Enabled (inherited)
Total 6				

Figure 7-5   Policy sharing enabled

7.4.3 Configuring client policy by using the service provider policy

A Web service client can obtain the policy configuration of a Web service provider and use this information to establish a policy configuration that is acceptable to both the client and the service provider. There are four scenarios when applying a policy to the Web service client:

► No policy

► Client policy

   The policy is calculated based on the policy set, which is a static client policy configuration. This option is available when a client policy set is attached.

► Provider policy

The policy is calculated based on a dynamically acquired provider policy. This option is available when a client policy set is not attached. In this scenario, the policy configuration is based on the provider's requirements.

► Client and provider policy

The policy is calculated based on a dynamically acquired provider policy and static client policy configuration. In this scenario, the policy configuration is based on the provider's policy requirements, but the client can configure restrictions as to what is acceptable. This option is available when a client policy set is attached.

In this section, we apply the provider only policy to the Web service client:

1. In the administrative console, expand **Services** → **Service clients**.

2. Click **WeatherJavaBeanService** to configure it with a policy.

3. Select **WeatherJavaBeanService**.

4. Because no client policy set is applied, in the Policies Applied column, click **None**.

5. On the Policies Applied page (Figure 7-6):

   a. Under Apply the following policies, select **Provider policy only**. By selecting this option, you can configure the client based solely on the policy of the service provider.

Figure 7-6   Applying the Provider only policy

b. To obtain the provider policy using an HTTP Get request, select **HTTP GET request**. By default, the HTTP Get request targets the URL for the service endpoint followed by "?WSDL".

c. Click **OK**.

6. Assign the customized client policy set binding to the Web service client:

a. Select **WeatherJavaBeanService**. Click **Assign Binding** and select **ITSO Username WSSecurity binding**.

b. Click **Save**.

The Web application client-side policy is calculated when it is required at run time, based on either the policy of the service provider or the client policy set and the policy of the service provider, depending on which option you selected. This calculated policy is known as the *effective policy* and is cached as a runtime configuration. The effective policy is used for subsequent outbound Web service requests to the endpoint or operation for which the dynamic policy calculation was performed. The policy set configuration of the client does not change.

Figure 7-7 shows that the provider only policy is now applied.

Select	Service/Endpoint/Operation ⬦	Attached Client Policy Set ⬦	Policies Applied ⬦	Binding ⬦
	You can administer the following resources:			
☐	WeatherJavaBeanService	None	Provider only	ITSO Username WSSecurity binding
☐	WeatherJavaBeanPort	None	Provider only (inherited)	ITSO Username WSSecurity binding (inherited)
☐	getDayForecast	None	Provider only (inherited)	ITSO Username WSSecurity binding (inherited)
☐	getForecast	None	Provider only (inherited)	ITSO Username WSSecurity binding (inherited)
☐	getTemperatures	None	Provider only (inherited)	ITSO Username WSSecurity binding (inherited)
☐	setWeather	None	Provider only (inherited)	ITSO Username WSSecurity binding (inherited)
Total 6				

Figure 7-7   Provider only policy now applied

7. Test the application by following the instructions in "Monitoring the SOAP traffic" on page 278. In the TCP/IP Monitor, the client first acquires the WSDL through the HTTP GET (Figure 7-8). The client policy calculations for a service are performed at the first invocation on that service. Calculated policies are cached in the client for performance.
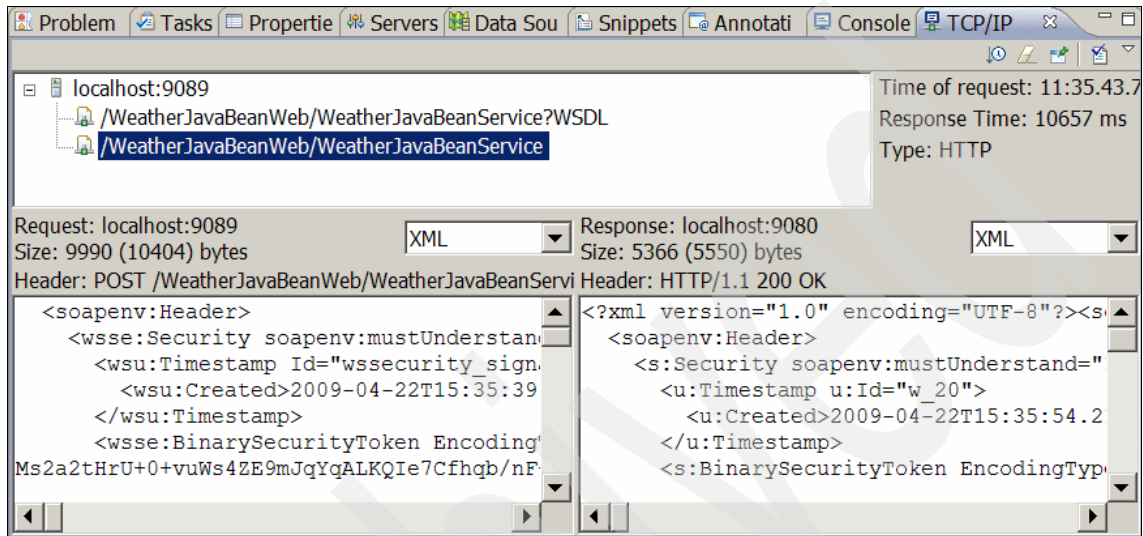
Figure 7-8   WS-Policy traffic

8. To see the policy-annotated WSDL, query the Web service endpoint as follows:

```
http://localhost:9080/WeatherJavaBeanWeb/WeatherJavaBeanService?wsdl
```

Example 7-6 shows the policy-annotated WSDL.

Example 7-6   WSDL with the user name WS-Security policy

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="WeatherJavaBeanService" targetNamespace="http://bean.itso/"
   xmlns="http://schemas.xmlsoap.org/wsdl/"
xmlns:wsp="http://www.w3.org/ns/ws-policy"

xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility
-1.0.xsd"
... ...>
   ... ...
</portType>
   <binding name="WeatherJavaBeanPortBinding" type="tns:WeatherJavaBeanDelegate">
      <soap:binding style="document"
```

```
            transport="http://schemas.xmlsoap.org/soap/http" />
        <wsp:PolicyReference URI="#95761ca1-23ba-4e85-abb8-9b3689405a08" />
        <operation name="getDayForecast">
            ... ...
        </operation>
    </binding>
    <service name="WeatherJavaBeanService">... ...</service>
    <wsp:Policy wsu:Id="95761ca1-23ba-4e85-abb8-9b3689405a08">
        <wsp:ExactlyOne>
            <wsp:All>
                <addressing:Addressing
                    xmlns:addressing="http://www.w3.org/2007/05/addressing/metadata">
                    <wsp:Policy>
                        <wsp:ExactlyOne>
                            <wsp:All>
                            </wsp:All>
                        </wsp:ExactlyOne>
                    </wsp:Policy>
                </addressing:Addressing>
            </wsp:All>
... ...
        </wsp:ExactlyOne>
        <sp:AsymmetricBinding
            xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
        </sp:AsymmetricBinding>
        <sp:Wss10 xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
            <wsp:Policy>
                <sp:MustSupportRefKeyIdentifier />
            </wsp:Policy>
        </sp:Wss10>
    </wsp:Policy>
... ...
    <wsp:Policy wsu:Id="a5a0a090-bf59-4906-bddd-725dd07281ea">
        <sp:SignedParts
            xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
            <sp:Body />
            <sp:Header Namespace="http://schemas.xmlsoap.org/ws/2004/08/addressing" />
            <sp:Header Namespace="http://www.w3.org/2005/08/addressing" />
        </sp:SignedParts>
        <sp:EncryptedParts
            xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
            <sp:Body />
        </sp:EncryptedParts>
        <sp:SignedEncryptedSupportingTokens
            xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
```

```
        </sp:SignedEncryptedSupportingTokens>
    </wsp:Policy>
... ...
</definitions>
```

7.4.4 Configuring service provider to share a policy by using WS-MEX

In this section, you configure the service provider to share its policy by using WS-MEX through the administrative console. The client is configured to use WS-MEX to acquire a provider's policy.

To enable the service provider to share its policy by using WS-MEX:

1. In the administrative console, expand **Services** → **Service providers**.

2. Click **WeatherJavaBeanService** to configure it with a policy.

3. Click the **Enabled** hyperlink to change the settings for the policy-sharing configuration.

4. To enable WS-MEX and make the policy configuration of the service provider available to a WS-MetadataExchange GetMetada request, select **WS-MetadataExchange request** (Figure 7-9). You can choose both the **Exported WSDL** and **WS-MetadataExchange request** options so that they are available to different client requests.

Figure 7-9   Policy sharing using WS-MEX

After you select the WS-MetadataExchange request option, the Attach as system policy to the WS-MetaExchange check box is available for you to select. You can apply message-level security to secure the WS-MetadataExchange GetMetadata request by attaching a system policy set. An advantage of message-level security is that it provides end-to-end security, which is important for the exchange of security metadata. For this example, we do not select it (the default).

To configure the client to use WS-MEX to acquire the provider's policy:

1. In the administrative console, expand **Services** → **Service clients**.

2. Click **WeatherJavaBeanService** to configure it with a policy.

3. Click **Provider policy only**.

4. To obtain the provider policy by using a WS-MetadataExchange GetMetadata request, click **WS-MetadataExchange request**. Click OK.

5. Click **Save**.

Test the Web services again. In the TCP/IP Monitor, the client issues a WS-MEX GetMetadata request to the actual Web service endpoint (Example 7-7). The dialect of the request is WSDL.

Example 7-7   WS-MetadataExchange GetMetadata request

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
   <soapenv:Header xmlns:wsa="http://www.w3.org/2005/08/addressing">
      <wsa:To>
         http://localhost:9089/WeatherJavaBeanWeb/WeatherJavaBeanService
      </wsa:To>
      <wsa:MessageID>urn:uuid:76EF2707340C93E6F81240589174816
      </wsa:MessageID>
      <wsa:Action>
         http://schemas.xmlsoap.org/ws/2004/09/mex/GetMetadata/Request
      </wsa:Action>
   </soapenv:Header>
   <soapenv:Body>
      <mex:GetMetadata xmlns:mex="http://schemas.xmlsoap.org/ws/2004/09/mex">
         <mex:Dialect>http://schemas.xmlsoap.org/wsdl/
         </mex:Dialect>
      </mex:GetMetadata>
   </soapenv:Body>
</soapenv:Envelope>
```

The GetMetadata response returns the WSDL with the policy information
(Example 7-8).

Example 7-8   WS-MetadataExchange GetMetadata response

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
    <soapenv:Header xmlns:wsa="http://www.w3.org/2005/08/addressing">
        <wsa:Action>
            http://schemas.xmlsoap.org/ws/2004/09/mex/GetMetadata/Response
        </wsa:Action>
        <wsa:RelatesTo>urn:uuid:76EF2707340C93E6F81240589174816
        </wsa:RelatesTo>
    </soapenv:Header>
    <soapenv:Body>
        <mex:Metadata xmlns:mex="http://schemas.xmlsoap.org/ws/2004/09/mex"
            xmlns:tns="http://bean.itso/">
            <mex:MetadataSection Dialect="http://schemas.xmlsoap.org/wsdl/"
                Identifier="{http://bean.itso/}WeatherJavaBeanService">
                <definitions name="WeatherJavaBeanService"
                    targetNamespace="http://bean.itso/"
xmlns="http://schemas.xmlsoap.org/wsdl/"
                    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:wsp="http://www.w3.org/ns/ws-policy"

xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility
-1.0.xsd"
                    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
                    ... Anotated wsdl content with WS-Policy info...
                </definitions>
            </mex:MetadataSection>
            <mex:MetadataSection Dialect="http://schemas.xmlsoap.org/wsdl/">
                <mex:Location>
                    http://localhost:9089/WeatherJavaBeanWeb/WeatherJavaBeanService?wsdl
                </mex:Location>
            </mex:MetadataSection>
        </mex:Metadata>
    </soapenv:Body>
</soapenv:Envelope>
```

Cleaning up the sample

To proceed with the next sample, restore the application to the original state. If you are using a stand-alone WebSphere Application Server, detach the username WS-Security policy:

1. Detach the service provider's policy set:

    a. In the administrative console, expand **Services** → **Service providers**.
    b. Click **WeatherJavaBeanService**.
    c. Select **WeatherJavaBeanService**.
    d. Click **Detach Policy Set**.
    e. Click **Save**.

2. Detach the service client's policy set:

    a. Expand **Services** → **Service clients**.
    b. Click **WeatherJavaBeanService**.
    c. Select **WeatherJavaBeanService**.
    d. Click **Detach Client Policy Set**.
    e. Click **Save**.

3. Restart the applications:

    a. In the left pane, select **Applications** → **Application Types** → **WebSphere Enterprise Applications**.

    b. Select **WeatherJavaBeanServer** and **WeatherJavaBeanWebClientEAR**.

    c. Click **Stop**.

    d. After the application stops, click **Start**.

If you are using Rational Application Developer and its integrated WebSphere Application Server, uninstall and re-install the Web service and client:

1. Right-click **WebSphere Application Server V7.0** and select **Add and Remove Projects**. In the Add and Remove Projects window, select **Remove All** and click **Finish** to uninstall the Web service and the client.

    After you uninstall the Web service and the client application, the specific policy set and the binding that are attached with the Web service application are also gone.

2. Right-click **WebSphere Application Server V7.0** and select **Add and Remove Projects**. In the Add and Remove Projects window, select **Add All** to install the Web service and the client.

    Now you have a fresh Web service and client without policy set and binding attached.

Exporting the general binding for username WS-Security

To proceed to the next sample, export the general binding from the administrative console so that you can import it into the Rational Application Developer. To export the general binding:

1. In the administrative console, expand **Services** → **Policy sets** and select **General client policy set bindings**.

2. Select **ITSO Username WSSecurity binding** and click **Export**.

3. Select the **ITSO Username WSSecurity binding.zip** file and click **Save** to save the file to your local drive.

7.5 Tools support

Rational Application Developer V7.5 provides tools support for WS-Policy and WS-MEX. It provides wizards that are equivalent to those provided by the administration for WS-Policy so that users can omit the step of logging into the administrative console to configure.

In this section we configure the service provider to share its policy configuration, and configure the client policy by using a service provider policy. We demonstrate these tasks by using the tools that ship with Rational Application Developer.

7.5.1 Importing the Web service general binding

Before you use the wizard to share the policy configuration, import the Web service general binding into your workspace:

1. Click **File** → **Import**.

2. In the Import window, expand **Web services** and select **WebSphere Named Bindings**. Click **Next**.

3. In the next window, click **Browse** and select the **ITSO Username WSSecurity binding.zip** file that you created in "Exporting the general binding for username WS-Security" on page 355. Click **Finish**.

4. Verify that the policy set and the general binding imported successfully. Click **File** → **Preferences** → **Service Policies**. You see `ITSO Username WSSecurity binding` listed in the Service Policies preferences.

7.5.2 Configuring a service provider to share its policy configuration

Configure a service provider to share its policy configuration:

1. In the Services view, expand **JAX-WS** → **Services**. Right-click **WeatherJavaBeanService** and select **Manage Policy Set Attachment**.

2. Click **Add**.

3. For policy set select **Username WSSecurity default**, and for binding select **Provider Sample**. Click **OK**.

4. In the message window that opens, click **Ignore**.

5. In the next window, click **Next**.

6. In the Configure Policy Sharing window, select the service and click **Configure**.

7. In the Configure Policy Sharing for Web Service panel (Figure 7-10), select **Share Policy Information via WSDL** and click **OK**.
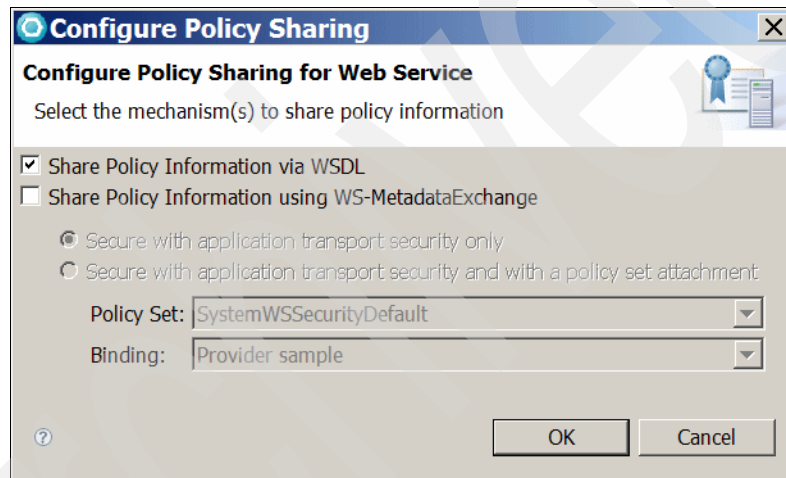
Figure 7-10   Choosing the Share Policy Information via WSDL option

8. In the warning message window that opens, click **Ignore**.

9. Click **Finish**.

After you configure the Web service project to share the policy information, a `wsPolicyServiceControl.xml` file is generated in the `WeatherJavaBeanServer\META-INF` folder (Example 7-9). With this file, the Web service developer can specify the policy acquisition information during tools time, thus omitting the requirement for an additional administrative step.

Example 7-9   The wsPolicyServiceControl.xml file

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
```

```
<ns2:WSPolicyServiceControl
xmlns:ns2="http://www.ibm.com/xmlns/prod/websphere/200709/WSPolicyServiceControl"
Version="1.0">
    <WSPolicyServiceControlReference
Resource="WebService:/WeatherJavaBeanWeb.war:{http://bean.itso/}WeatherJavaBeanServic
e/">
<WSPolicyAttachmentNamespace>http://www.w3.org/ns/ws-policy</WSPolicyAttachmentNamesp
ace>
        <ExportPolicySetConfigurationInWSDL>true</ExportPolicySetConfigurationInWSDL>
    </WSPolicyServiceControlReference>
</ns2:WSPolicyServiceControl>
```

7.5.3 Configuring the client policy by using a service provider policy

To configure the client policy by using the service provider policy:

1. In the Services view, expand **JAX-WS** → **Clients**. Right-click **WeatherJavaBeanService** and select **Manage Policy Set Attachment**.

2. In the Client Side Policy Set Attachment window (Figure 7-11), click the **Use Provider Policy** button.
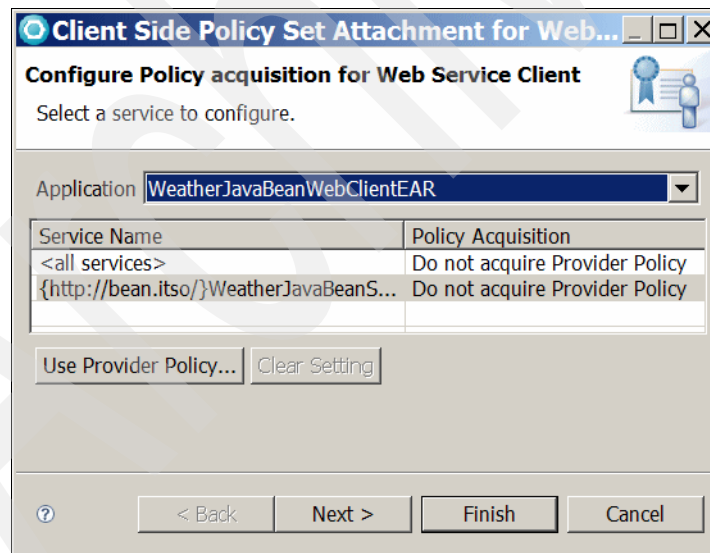
Figure 7-11   Clicking the Use Provider Policy button

3. In the Configure Policy acquisition for Web service Client window (Figure 7-12), select **HTTP Get request targeted at <default WSDL URL>**, and click **OK**.
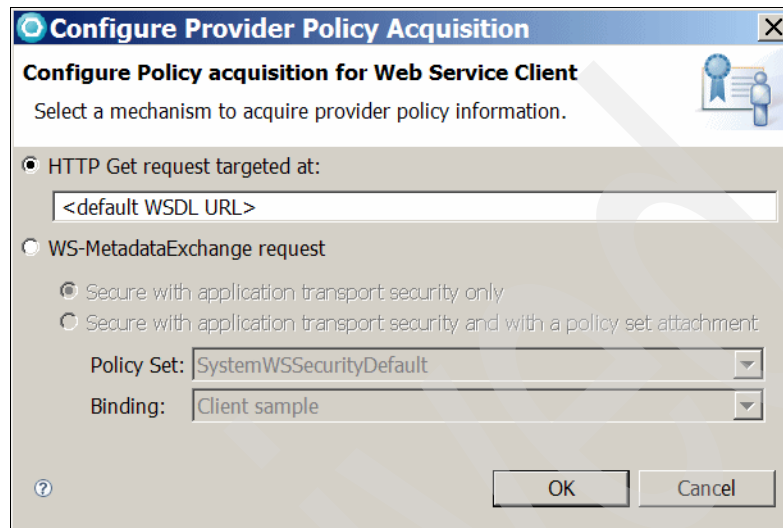
Figure 7-12   Selecting the HTTP Get request targeted at option

4. In the message window that opens, click **Ignore**. The Policy Acquisition field for the service changes to Acquire Provider Policy.

5. Back in the Client Side Policy Set Attachment window, click **Finish**.

6. After you configure policy acquisition for Web service client, a `wsPolicyClientControl.xml` file is generated in the `WeatherJavaBeanWebClient\META-INF` folder (Example 7-10). With this file, the Web service developer can specify the policy acquisition information during tools time, thus omitting the requirement for an additional administrative step.

Example 7-10   The wsPolicyClientControl.xml file

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:WSPolicyClientControl
xmlns:ns2="http://www.ibm.com/xmlns/prod/websphere/200709/WSPolicyClientControl"
Version="1.0">
    <WSPolicyClientControlReference
Resource="WebService:/{http://bean.itso/}WeatherJavaBeanService/">
        <ProviderPolicyAcquisition>
<PolicyAcquisitionClass>com.ibm.ws.wspolicy.acquisition.AcquireViaQWSDL</PolicyAcquis
itionClass>
        </ProviderPolicyAcquisition>
```

```
    </WSPolicyClientControlReference>
</ns2:WSPolicyClientControl>
```

7. In the WebSphere administrative console, expand **Services** →**Policy sets** → **Default policy set bindings**. In the **Default service client binding** drop down menu, choose **ITSO Username WSSecurity binding**. Click **Apply** and then **Save**.

8. Test the application by following the instructions in "Monitoring the SOAP traffic" on page 278. In the TCP/IP Monitor, you see that the client first acquires the WSDL through HTTP GET. The effective policy is used for subsequent outbound Web service requests to the endpoint or operation for which the dynamic policy calculation was performed.

7.6 More information

For more information, consult the following sources:

► The Web services Policy 1.5 - Framework specification

   http://www.w3.org/TR/ws-policy/

► The Web services Policy 1.2 - Attachment (WS-PolicyAttachment) specification

   http://www.w3.org/Submission/WS-PolicyAttachment/

► The WS-SecurityPolicy 1.2 specification

   http://docs.oasis-open.org/ws-sx/ws-securitypolicy/v1.2/ws-security
   policy.html

► The WS-MetadataExchange 1.2 specification draft

   http://specs.xmlsoap.org/ws/2004/09/mex/WS-MetadataExchange.pdf

Web services transaction specifications

This chapter introduces the Web Services Transaction (WS-Transaction) specifications and those that are supported in WebSphere Application Server. To illustrate the transactional Web services support, we provide simple examples that use Web Services AtomicTransaction (WS-AT) and Web Services BusinessActivity (WS-BA).

This chapter contains the following topics:

8.1 Overview of the WS-Transaction specifications

The WS-Transaction specifications define mechanisms for transactional interoperability between Web services domains and provide a means to compose transactional qualities of service into Web services applications. These specifications describe an extensible Web Services Coordination (WS-Coordination) framework and specific coordination types for the following transactions (Figure 8-1):

► Short duration and atomicity, consistency, isolation, durability (ACID) transactions (WS-AT)

► Longer running business transactions (WS-BA)
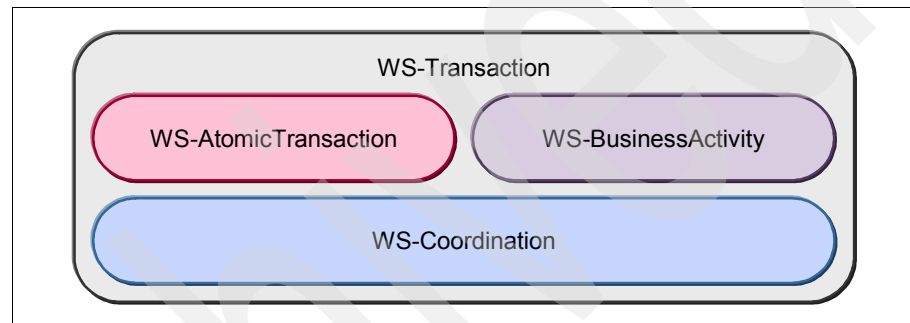
Figure 8-1   WS-Transaction block diagram

The WS-AT specification support in the application server provides transactional quality of service (QoS) to the Web services environment. Distributed Web services applications and the resources that they use can take part in distributed global transactions.

With Web services Business Activity (WS-BA) support in the application server, Web services on different systems can coordinate activities that are more loosely coupled than atomic transactions. Such activities can be difficult or impossible to roll back atomically and, therefore, require a compensation process if an error occurs.

Web Services Coordination (WS-Coordination) specifies a CoordinationContext and a registration service with which participant Web services can enlist to take part in the protocols that are offered by specific coordination types.

> **Note:** WebSphere Application Server V7 supports both the WS-Transaction 1.1 and the WS-Transaction 1.0 specifications. In this case, note the following points:
>
> ► Java API for XML-based Web services (JAX-WS) run time supports WS-Transaction 1.1 and 1.0 specifications.
>
> ► Java API for XML-based remote procedure calls (JAX-RPC) run time supports WS-Transaction 1.0 specifications only.

WebSphere Application Server V7 supports WS-Transaction 1.0 by default, but you can change it to 1.1. To access this setting, click **Servers** → **Server Types** → **WebSphere application servers** → *server_name*. In the right pane, select **Container Services** → **Transaction Service**.
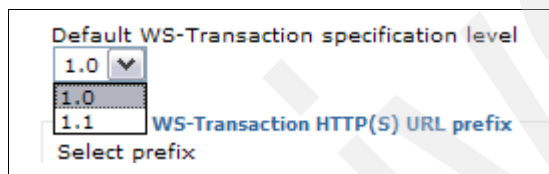
Figure 8-2   WS-Transaction 1.0 and 1.1 support

In the following sections, we discuss WS-Coordination at a high level, then WS-AT and WS-BA in detail with examples.

8.2 WS-Coordination

The WS-Coordination specification describes a framework for a coordination service (or coordinator) that consists of the following component services:

► An *activation service* with a *CreateCoordinationContext* operation that performs the following tasks:

a. Creates a new activity

b. Returns its coordination context that supports a certain coordination type, such as WS-AT

► A *registration service* with a *register* operation that performs the following tasks:

   a. Enables an application to register for a certain coordination protocol under the current coordination type.

   b. Exchanges endpoint references (EPRs). Each side of the coordination protocol (participant and coordinator) supplies an EPR.

► A set of coordination protocol services for each supported coordination type.

Figure 8-3 shows the sequence of the interaction between certain applications and a coordinator.
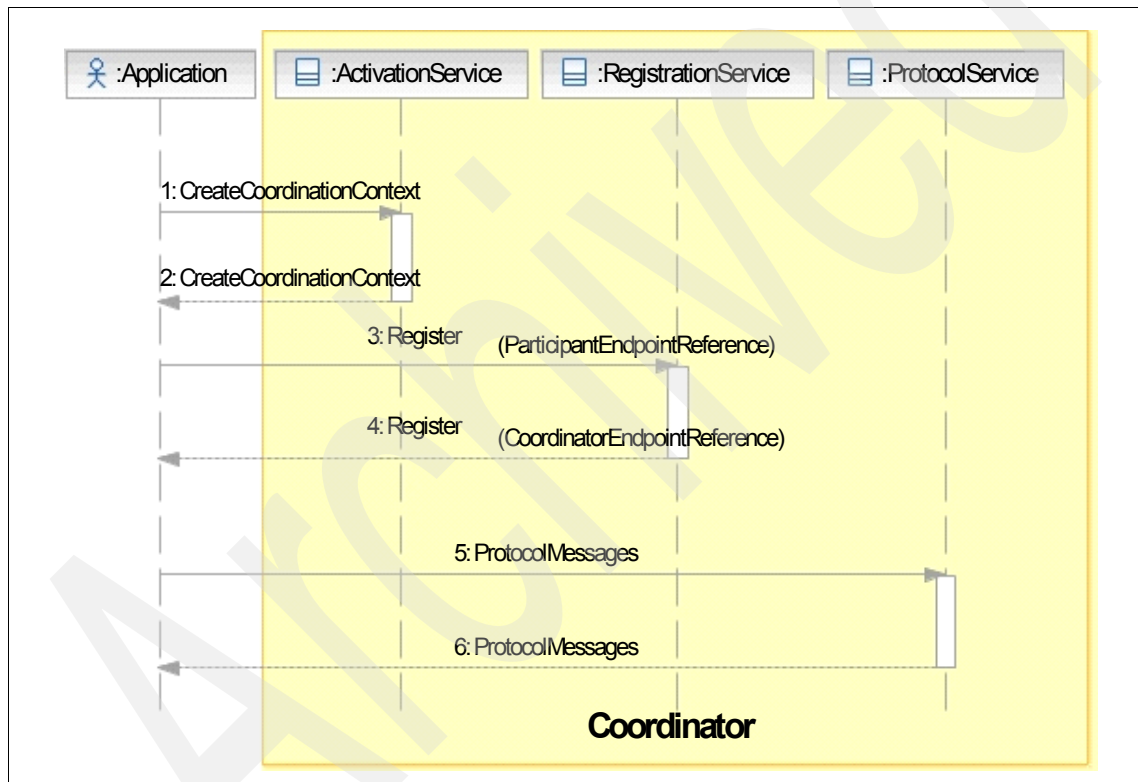
Figure 8-3   WS-Coordination sequence diagram

For more information about WS-Coordination, see the following Web address:

http://docs.oasis-open.org/ws-tx/wstx-wscoor-1.1-spec-os/wstx-wscoor-1.1-spec-os.html#_Toc160426472

8.3 WS-AtomicTransaction

The Web services specification provides the definition of the atomic transaction coordination type that is to be used with the extensible coordination framework described in the WS-Coordination specification. WS-AT is a specific coordination type that defines protocols for atomic transactions.

WS-AT support in the application server provides transactional QoS to the Web services environment. Distributed Web services applications and the resources they use can take part in distributed global transactions. WS-AT is a two-phase commit transaction protocol and is suitable for short-duration transactions only.

WS-AT support is an interoperability protocol that introduces no new programming interfaces for transactional support. Global transaction demarcation is provided by standard enterprise application use of the Java Transaction API (JTA) UserTransaction interface. If an application component that is running under a global transaction makes a Web services request, a WS-AT CoordinationContext is implicitly propagated to the target Web service, but only if the appropriate application deployment descriptors are set.

Figure 8-4 shows a transaction context that is shared between two WebSphere application servers for a Web services request that contains a WS-AT CoordinationContext.
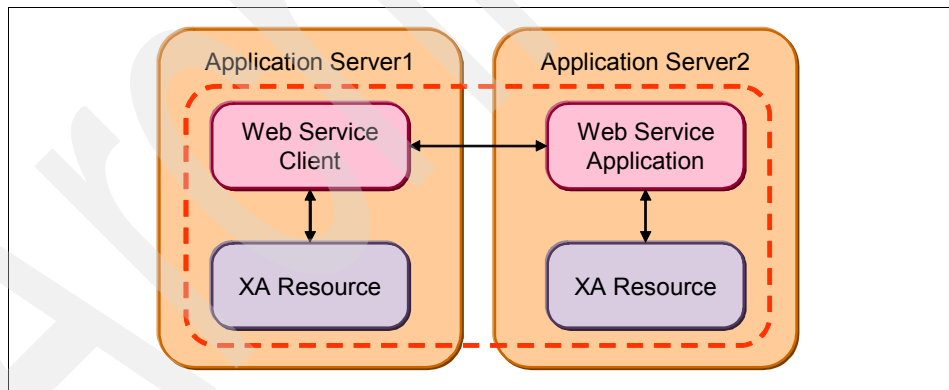
Figure 8-4   WS-AT support in WebSphere Application Server V7

8.3.1 Example of using WS-AtomicTransaction

> **Download material**: The completed example used in this section is included in the download materials in the `Chapter8/ws-at.zip` project interchange file for Rational Application Developer V7.5.
>
> This example uses a DB2 database. The instructions for setting up this database can be found in "Set up the WEATHER database (DB2)" on page 542.

The following example illustrates the action of WS-AT. It is composed of three components:

► The *WeatherEJB project* contains the JAX-WS Weather Enterprise JavaBeans (EJB) Web service, which inserts a record into the database. This service is developed by using the top-down method from the `WeatherJavaBeanService.wsdl` file.

► The *WeatherJavaBeanWeb project* contains the JAX-WS Weather JavaBean Web service, which inserts another record into the database. This service is developed by using the bottom-up method.

► The *WeatherJavaBeanWebClient project* contains the JAX-WS Weather JavaBean test client.

Figure 8-5 shows the sequence of the interactions between the WeatherEJB, WeatherJavaBeanWeb, and the WeatherJavaBeanWebClient components.
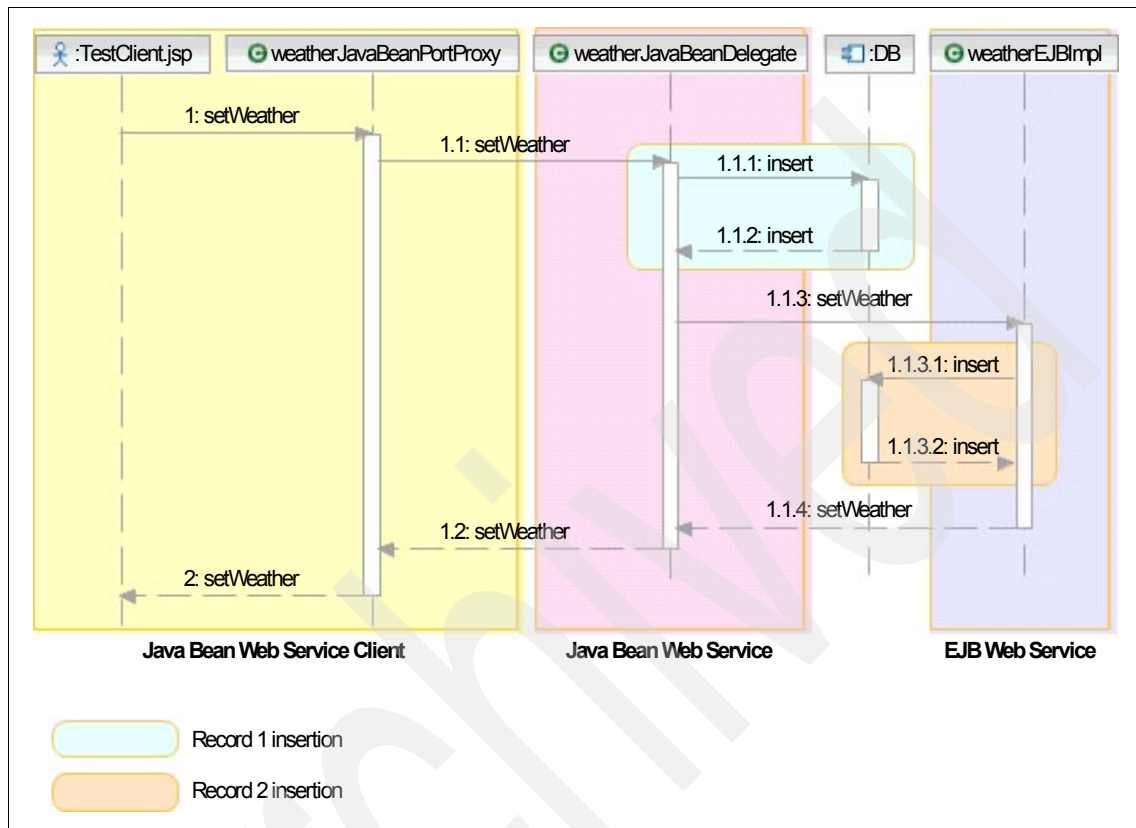
Figure 8-5   Web service example for atomic transaction

To experiment with atomic transactions, this example performs two database record insertions by using two applications:

► The `TestClient.jsp` file invokes the JavaBean Web service in the Web project and passes a weather object for insertion into the database.

► The JavaBean Web service then invokes the EJB Web service to perform another insertion operation. Note that WeatherJavaBeanWeb contains generated Java proxy classes for calling the EJB Web service.

These two database insertions are executed in one global transaction.

Weather EJB Web service

The Weather EJB Web service uses the WeatherBase utility project to insert a weather record into the database, as shown in Example 8-1.

Example 8-1   setWeather() method implementation in EJB Web service

```
public void setWeather(Weather arg0) throws Exception_Exception {
   WeatherDAO weatherDao = new WeatherDAO();
   itso.objects.Weather weather = new itso.objects.Weather();
   weather.setCondition(arg0.getCondition());
   weather.setDate(arg0.getDate().toGregorianCalendar());
   weather.setTemperatureCelsius(arg0.getTemperatureCelsius());
   weather.setWindDirection(arg0.getWindDirection());
   weather.setWindSpeed(arg0.getWindSpeed());
   weatherDao.insertWeather(weather);
   //throw new Exception_Exception("WeatherEJB insert failed", new
Exception());
}
```

The setWeather() method optionally throws an exception to simulate an error in the database insertion.

> **Global transaction**: WeatherDAO uses connections with automatic commit. By using a global transaction around the JDBC access, autocommit is disabled, and the commit is performed when the global transaction ends. The global transaction is carried from the JavaBean Web service to the EJB Web service when you activate the atomic transaction support.

Weather JavaBean Web service

The Weather JavaBean Web service (WeatherJavaBean) does the following actions:

► Starts a global transaction
► Inserts a weather object into the database
► Calls the EJB Web service (to insert another weather object)
► Commits the changes

Example 8-2 illustrates this Web service.

Example 8-2   setWeather() method implementation in JavaBean Web service

```
public void setWeather(Weather dayWeather) throws Exception {

    UserTransaction userTransaction = null;
    try {
```

```
            InitialContext context = new InitialContext();
            userTransaction = (UserTransaction) context
                    .lookup("java:comp/UserTransaction");
            // start transaction
            userTransaction.begin();

            //insert record in database
            WeatherForecast wfc = new WeatherForecast();
            wfc.setWeather(dayWeather);

            //insert record in database via EJB service
            WeatherJavaBeanPortProxy beanPortProxy = new
WeatherJavaBeanPortProxy();
            itso.ejbean.Weather weather = new itso.ejbean.Weather();
            XMLGregorianCalendar xl = DatatypeFactory.newInstance()
                    .newXMLGregorianCalendar(2009, 04, 24, 00, 00, 00, 00,
1);
            weather.setDate(xl);
            weather.setCondition("windy");
            weather.setTemperatureCelsius(22);
            weather.setWindDirection("W");
            weather.setWindSpeed(22);
            beanPortProxy.setWeather(weather);

            // commit transaction
            userTransaction.commit();
        } catch (Exception e) {
            throw new java.rmi.RemoteException("web service bean
transaction error: "
                    + e.getMessage());
        }

    }
```

Activating WS-Transaction support

Policy sets are used to simplify the configuration of the QoS for Web services
and clients. *Policy sets* are assertions about how Web services are defined. By
using policy sets, you can combine configurations for different policies. For more
information see Chapter 6, "Policy sets" on page 261.

More about policy sets: Policy sets are used with JAX-WS applications, but
not with JAX-RPC applications.

You can configure the way that a JAX-WS client or Web service handles WS-AT or Web Services Business Activity (WS-BA) context by configuring the WS-Transaction policy type. You can specify that the *client* must *send* context, can send context if it is available, or must not send context. You can also specify that the *Web service* must *receive* context, can receive context if it is available, or must not receive context.

WebSphere Application Server V7 provides default policy sets, including the SSL WSTransaction policy set that provides transactional integrity by using SSL. In this example, SSL will not be required, so the WSTransaction policy set is imported from a default repository of WebSphere Application Server:

1. Run the WebSphere Application Server administrative console.

2. Navigate to **Services** → **Policy sets** → **Application policy sets**.

3. On the Application policy sets page (Figure 8-6), select **Import** → **From Default Repository**.
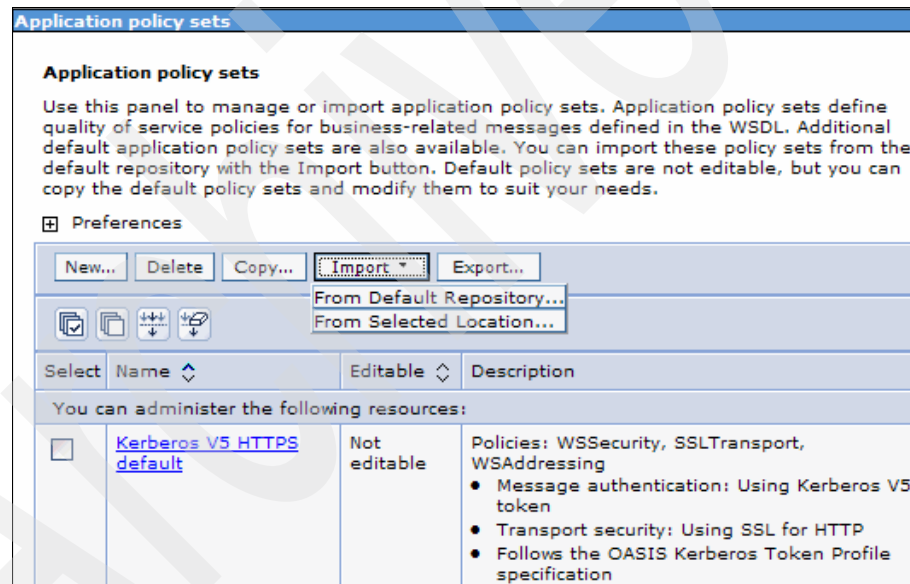
Figure 8-6   Importing policy sets

4. From the list of policy sets, select **WSTransaction** and click **OK**.

Now the WSTransaction policy set is imported into the list of supported application policy sets. To add the WSTransaction policy set to the development tool:

1. In the administrative console, select **WSTransaction**, which was imported, and click **Export**.

2. Click the **WSTransaction.zip** link to download this file (Figure 8-7).
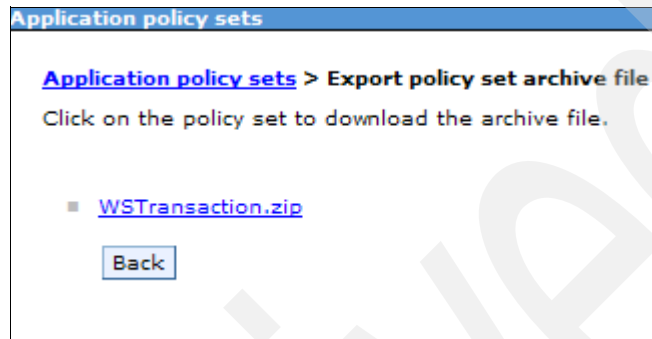
Figure 8-7   Exporting WSTransaction policy set

3. In Rational Application Developer, select **File** → **Import**.

4. In the Import window (Figure 8-8), expand **Web services** and select **WebSphere Policy Sets**. Click **Next**.
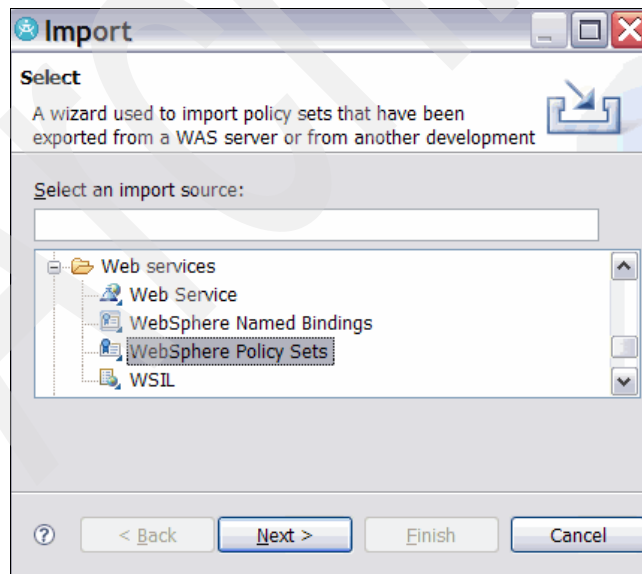
Figure 8-8   Importing the WSTransaction policy set

5. Browse to the **WSTransaction.zip** file and click **Finish** to import the file.

Now the workspace is synchronized with WebSphere Application Server policy sets and the WSTransaction policy set is successfully added to both.

To activate WSTransaction policy sets for the projects (clients and Web services):

1. In Rational Application Developer, select the Services view and expand the **JAX-WS** folder. The list of Web services and clients are displayed, as shown in Figure 8-9.
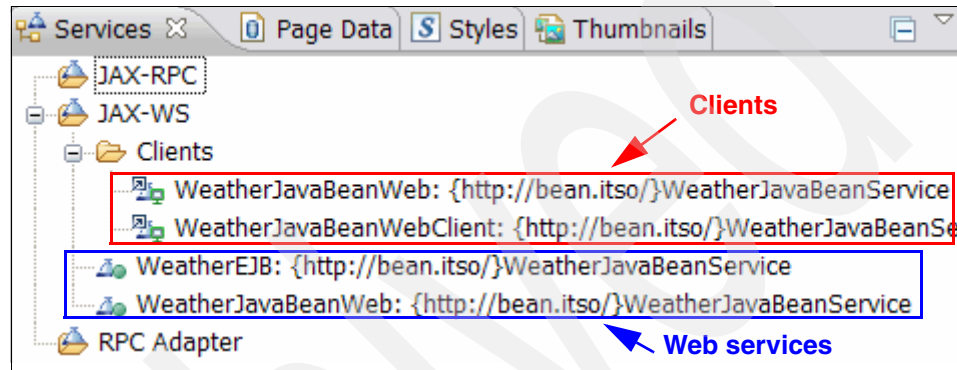
Figure 8-9   JAX-WS Web services and clients

2. For each Web service"

a. Right-click the Web service and select **Manage Policy Set Attachment**.
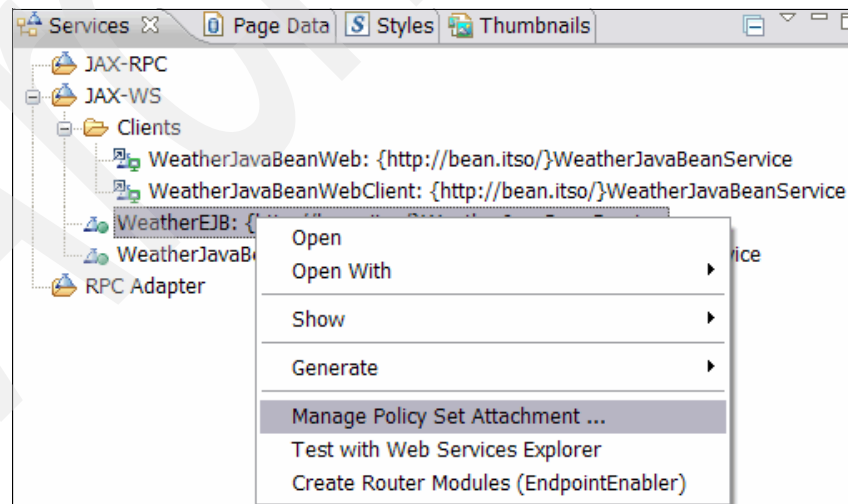
Figure 8-10   Attaching policy sets

b. Click **Add**.

　 c. In the End Point Definition Dialog window (Figure 8-11), for Policy Set, select **WSTransaction**. For the binding, select **<Default Binding>**. Click **OK**, then **Finish**.
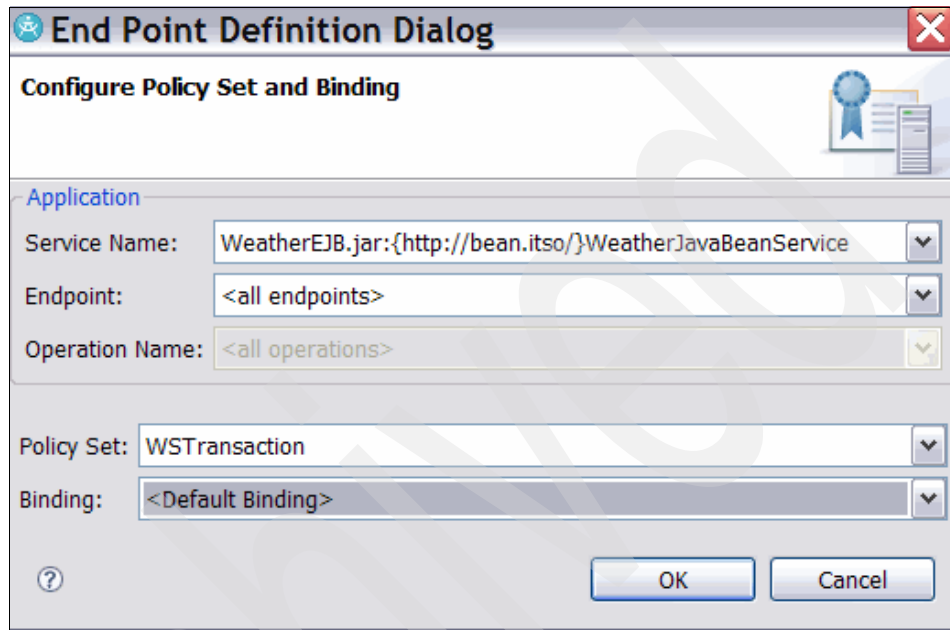
Figure 8-11   Adding WSTransaction policy set

3. For each Web service client:

　 a. Right-click each Web service client and select **Manage Policy Set Attachment**.

　 b. In the Configure Policy Acquisition for Web Service Client window, click **Next**.

　 c. Click **Add**.

　 d. In the Configure Policy Set and Binding window, select **WSTransaction** as the policy set. For the binding, select **<Default Binding>**. Click **OK**.

　 e. Click **Finish**.

Testing the application

At this point, we can test the application:

1. Deploy the following enterprise applications in the server:

   – WeatherEJBEAR
   – WeatherJavaBeanServer
   – WeatherJavaBeanWebClientEAR

2. Then run the `TestClient.jsp` file found in:

   `WeatherJavaBeanWebClient/WebContent/sampleWeatherJavaBeanPortProxy.`

   To start the client, right-click **TestClient.jsp** and select **Run As** → **Run on Server** (Figure 8-12).
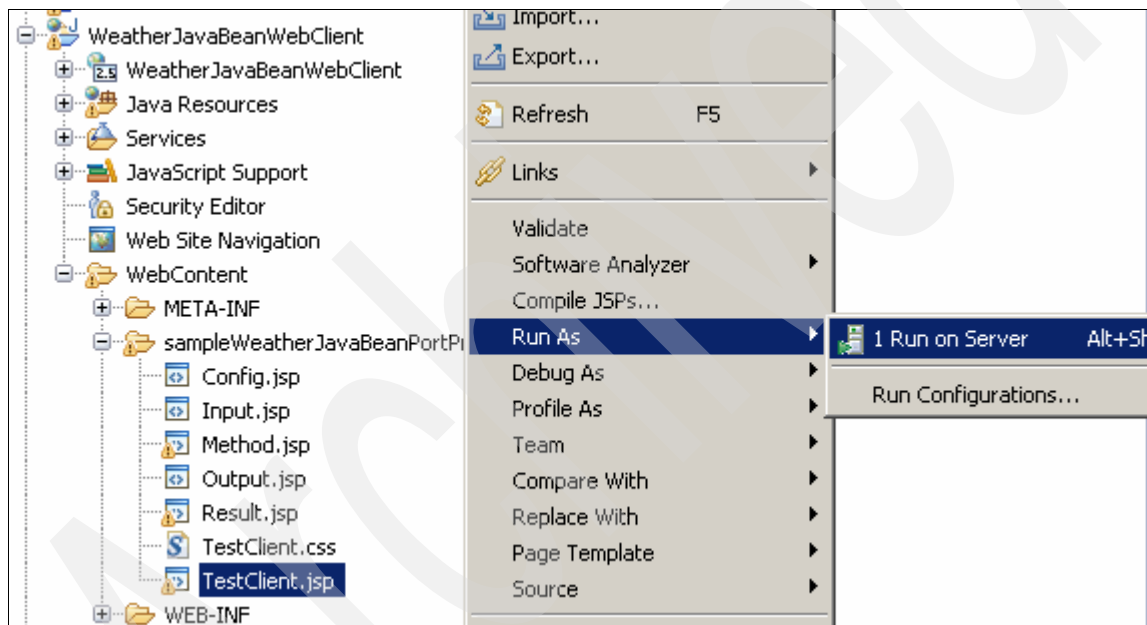
Figure 8-12   Run the test client

The test client will open as shown in Figure 8-14 on page 376. Note that you can modify the endpoint to use the WC_defaulthost port for your server.
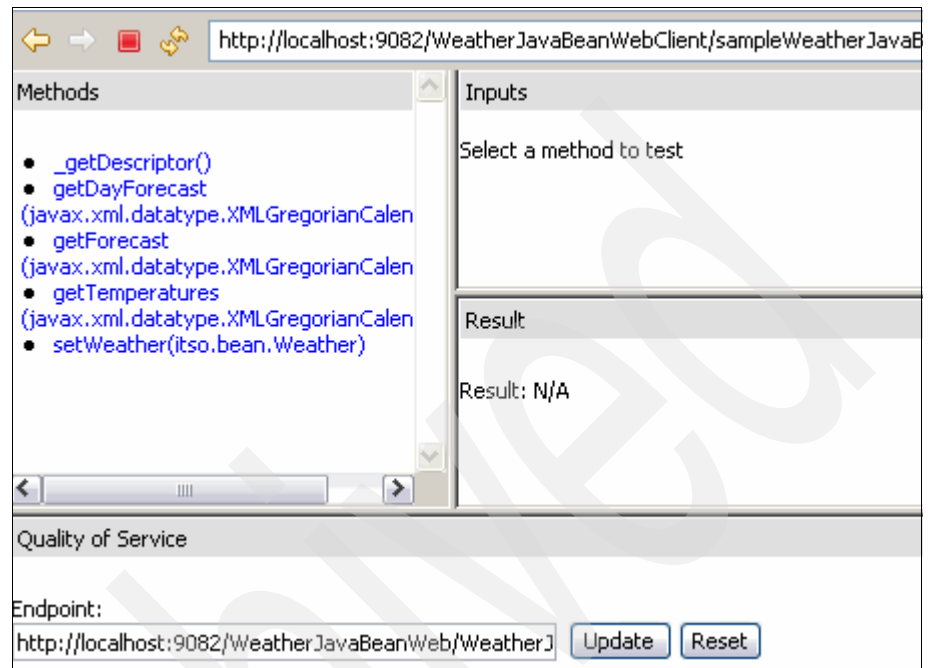
Figure 8-13 Test client

3. Click the **setWeather** method.

4. Complete the fields as shown in Figure 8-14. Be sure to enter a date that does not exist in the database.
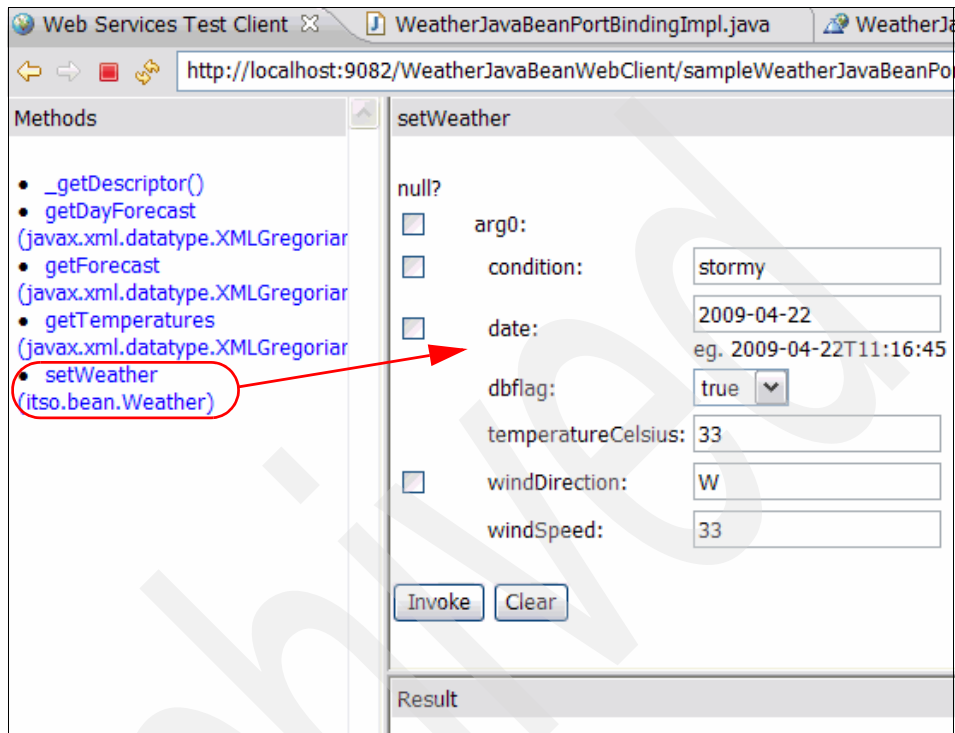
Figure 8-14   Running test client

5. Click **Invoke**. After running the setWeather() operation, the ITSO.SANJOSE table displays the records. There should be two new records, as shown in Figure 8-15.

Figure 8-15   Database output (two records inserted)

Testing atomic transactions

The atomic transaction is tested by using error simulation in the EJB Web service.

EJB Web service error simulation

To simulate an error in the EJB Web service:

1. Edit the setWeather() method of the WeatherEJB and activate the following statement:

```
throw new Exception_Exception("WeatherEJB insert failed", new
Exception());
```

2. Run the client. No records are inserted in the database. The Web service returns the following exception:

```
Exception: javax.xml.ws.soap.SOAPFaultException: web service bean
transaction error: WeatherEJB insert failed Message: web service
bean transaction error: WeatherEJB insert failed
```

8.3.2 SOAP messages for atomic transaction

The SOAP message passed from the JavaBean to the EJB Web service carries the WS-AT information in the header, as shown in Example 8-3.

Example 8-3   SOAP message for atomic transaction

```
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Header>
    <wscoor:CoordinationContext soapenv:mustUnderstand="1"
xmlns:wscoor="http://schemas.xmlsoap.org/ws/2004/10/wscoor">

<wscoor:Identifier>com.ibm.ws.wstx:0000012....</wscoor:Identifier>
      <wscoor:Expires>130000</wscoor:Expires>

<wscoor:CoordinationType>http://schemas.xmlsoap.org/ws/2004/10/wsat</ws
coor:CoordinationType>
      <wscoor:RegistrationService>
        <wsa:Address
xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing">https://IB
M-11B9...</wsa:Address>
        <wsa:ReferenceParameters
xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing">
          <websphere-wsat:instanceID
xmlns:websphere-wsat="http://wstx.Transaction.ws.ibm.com/extension">com
.ibm.ws.wstx:0000012...</websphere-wsat:instanceID>
          <wsaucf:RoutingInformation
xmlns:wsaucf="http://ucf.wsaddressing.ws.ibm.com">
            <wsaucf:Fragile>O...</wsaucf:Fragile>
          </wsaucf:RoutingInformation>
          <websphere-wsat:deferable
xmlns:websphere-wsat="http://wstx.Transaction.ws.ibm.com/extension">tru
e</websphere-wsat:deferable>
          <websphere-wsat:txID
xmlns:websphere-wsat="http://wstx.Transaction.ws.ibm.com/extension">com
.ibm.ws.wstx:0000012</websphere-wsat:txID>
          <wsaucf:VirtualHostName
xmlns:wsaucf="http://ucf.wsaddressing.ws.ibm.com">default_host</wsaucf:
VirtualHostName>
        </wsa:ReferenceParameters>
      </wscoor:RegistrationService>
    </wscoor:CoordinationContext>
  </soapenv:Header>
  <soapenv:Body>
```

```
        <ns2:setWeather xmlns:ns2="http://bean.itso/">
          <arg0>
            <condition>Windy</condition>
            <date>2009-04-24T00:00:00.000+00:01</date>
            <dbflag>false</dbflag>
            <temperatureCelsius>22</temperatureCelsius>
            <windDirection>W</windDirection>
            <windSpeed>22</windSpeed>
          </arg0>
        </ns2:setWeather>
      </soapenv:Body>
</soapenv:Envelope>
```
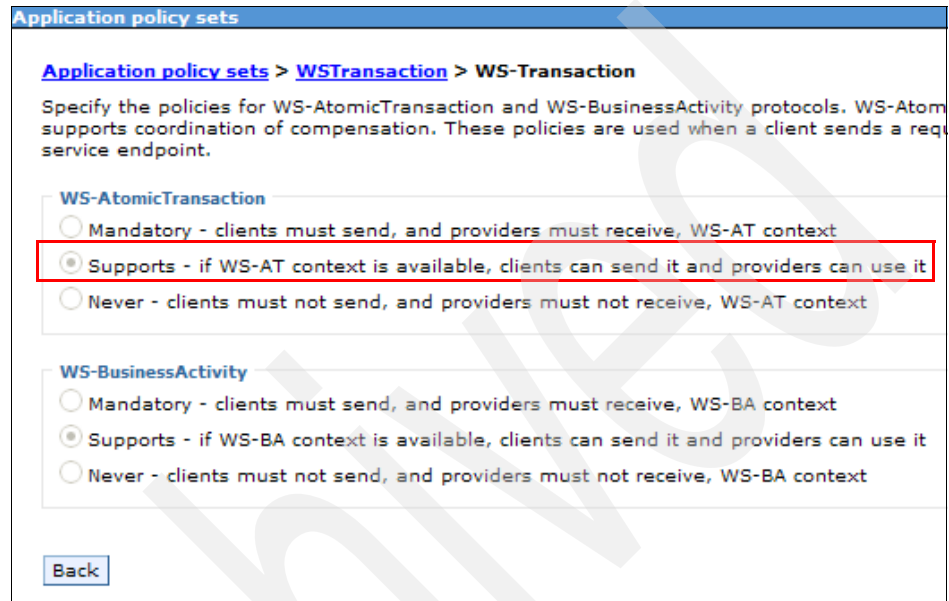
> **Note:** The SOAP message shown in Example 8-3 has information from the WS-Coordination, WS-Addressing, and WS-AT specifications.

8.3.3 WS-Transaction policy assertions

If you configure the policies for WS-Transaction protocol for a provider, this configuration affects the assertions that are included in any Web Services Description Language (WSDL) that is generated for the Web service with which the policy type is associated. The WS-Policy assertion that is used to describe the transactional requirements of a client or provider that uses WS-AT is *ATAssertion*.

Because the WS-Transaction policy type has a WS-AtomicTransaction (WS-AT) setting of Supports, a policy assertion is included in the WSDL. The WS-Transaction policy setting is determined from the administrative console (under **Services** → **Policy sets** → **Application policy sets** → **WSTransaction** → **WS-Transaction**). See Figure 8-16.

Figure 8-16   WS-Transaction setting

The application server can also parse, understand, and apply such assertions that are in WSDL.

Figure 8-17 shows the WSDL where the WS-AT ATAssertion indicates that an endpoint must be invoked with WS-AT context included in the request message and that the context can be in WS-Transaction 1.0 or 1.1 format. There are two namespaces and two assertions, one for each WS-Transaction specification level, using the WS-Policy ExactlyOne operator to show that the client must choose which specification level to use.

```
<wsdl:definitions targetNamespace="http://bean.itso/"
     xmlns:tns="http://bean.itso/"
     xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
     xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
     xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/
     oasis-200401-wss-wssecurity-utility-1.0.xsd">
        <wsp:Policy wsu:Id="ATPolicy">
           <wsp:ExactlyOne>
           <wsp:All>
           <wsat:ATAssertion wsp:Optional="true"
xmlns:wsat="http://schemas.xmlsoap.org/ws/2004/10/wsat"/>
        </wsp:All>
        <wsp:All>
           <wsat11:ATAssertion wsp:Optional="true"
xmlns:wsat11="http://docs.oasis-open.org/ws-tx/wsat/2006/06"/>
        </wsp:All>
        <!-- omitted assertions -->
           </wsp:ExactlyOne />
        </wsp:Policy>
        <!-- omitted elements -->
        <wsdl:binding name="WeatherJavaBeanPortBinding"
type="tns:WeatherJavaBeanDelegate">
           <!-- omitted elements -->
        <wsdl:operation name="setWeather">
              <wsp:PolicyReference URI="#ATPolicy"/>
              <!-- omitted elements -->
        </wsdl:operation>
     </wsdl:binding>
</wsdl:definitions>
```

WS-Transaction 1.0

WS-Transaction 1.1

The endpoint must be invoked with WS-AT context included in the request message

Figure 8-17   ATAssertion in the WSDL file

8.4 WS-BusinessActivity

A *business activity* is a collection of tasks that are linked together so that they have an agreed-upon outcome. Unlike atomic transactions, activities such as sending an e-mail can be difficult or impossible to roll back atomically and, therefore, require a compensation process in the event of an error. The WebSphere Application Server business activity support provides this compensation ability through business activity scopes.

WS-BA support is an implementation of the WS-BA and WS-Coordination specifications in WebSphere Application Server. These specifications define a set of protocols that enable Web service applications to participate in loosely coupled business processes. These processes are distributed across the heterogeneous Web service environment, with the ability to compensate actions if an error occurs. For example, an application that sends an e-mail cannot unsend it following a failure. However, the application can provide a business-level compensation handler that sends another e-mail that advises of the new circumstances.

In addition to supporting the WS-BA interoperability protocol, WebSphere Application Server provides a programming interface called the *Business Activity API* for creating business activities and compensation handlers. With this programming interface, you can specify compensation data and check or alter the status of a business activity.

8.4.1 Example of using WS-BusinessActivity

> **Download material**: The example used in this section is included in the download materials in the `Chapter8/ws-ba.zip` project interchange file for Rational Application Developer V7.5.
>
> This example uses a DB2 database. The instructions for setting up this database can be found in "Set up the WEATHER database (DB2)" on page 542.

The following example is used to illustrate the action of WS-BA. It is composed of two components:

► The WeatherBAEJB project contains the following session EJBs:

– WeatherUpdaterSoapBindingImpl session EJB that is exposed as JAX-WS Web service

This service is developed by using the top-down method from the `WeatherUpdater.wsdl` file.

- WeatherRaleigh session EJB that inserts a weather record into the database
- WeatherSanJose session EJB that inserts another weather record into the database

► The WeatherBAEJBClient project contains the JAX-WS Weather EJB test client.

Figure 8-18 shows a sequence that illustrates the interaction between these components. It also shows a business activity that spans multiple transactions.
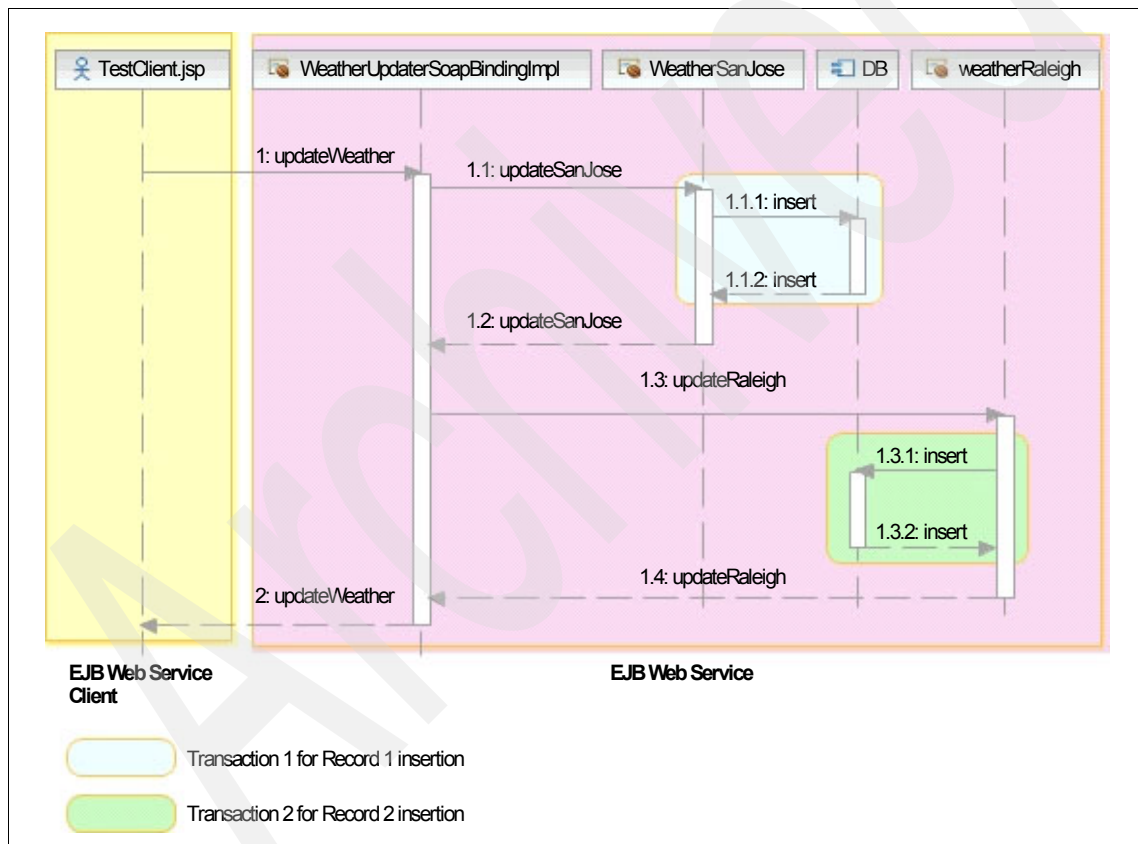
Figure 8-18   Web service example for business activity transaction

To experiment with business activity transactions, this example performs two database record insertions by using two applications:

- ▶ The TestClient invokes the WeatherUpdaterSoapBindingImpl session EJB as a Web service to insert multiple weather records.

- ▶ The WeatherUpdaterSoapBindingImpl session EJB calls two session EJBs (WeatherRaleigh and WeatherSanJose) to each insert a weather record. Each session EJB runs as its own transaction. This simulates a more realistic scenario where each session EJB invokes external services to perform work that cannot be done under one transaction.

If anything goes wrong in one of the transactions, updates must be rolled back by invoking compensation services for each session EJB.

8.4.2 Weather EJB Web service

Example 8-4 shows updateWeather() method in the WeatherUpdaterSoapBindingImpl session EJB. It invokes the WeatherRaleigh and WeatherSanJose session EJBs to each insert two weather records.

Example 8-4   updateWeather() method

```
public String updateWeather(XMLGregorianCalendar date) throws
Exception{
    Calendar date1 = Calendar.getInstance();
    date1.setTime(date.toGregorianCalendar().getTime());
    Calendar date2 = (Calendar)date1.clone();
    date2.roll(Calendar.DATE, true);

    String result1=null, result2=null;

    //Create a new context
    InitialContext ctx = new InitialContext();
    //use the context to lookup the remote interface.
    WeatherSanJoseRemote ejbSanJose = (WeatherSanJoseRemote)
ctx.lookup(WeatherSanJoseRemote.class.getName());
    WeatherRaleighRemote ejbRaleigh = (WeatherRaleighRemote)
ctx.lookup(WeatherRaleighRemote.class.getName());

    result1 = ejbSanJose.updateSanJose(date1);
    result2 = ejbRaleigh.updateRaleigh(date2);

    return "Updated weather: \n" + result1 + "\n" + result2 ;
}
```

The two other session beans are identical and insert a weather record, as shown in Example 8-5.

Example 8-5   updateSanJose() method

```
public String updateSanJose (Calendar date) throws Exception
{
    Weather w = new Weather(date);
    WeatherPredictor.calculateWeatherValues(w);
    WeatherDAO dao = new WeatherDAO();
    dao.deleteWeather(date);
    dao.insertWeather(w);
    System.out.println("Weather San Jose inserted: " + w);
    return "San Jose " + w;

}
```

8.4.3 Using the business activity support

To activate business activity support for the Weather application:

1. Activate the compensation service in the application server.
2. Use multiple transactions for the three EJBs.
3. Create a Service Data Object (SDO) for compensation.
4. Create the compensation classes.
5. Activate compensation for the session beans.
6. Register the session bean with the compensation service.

We explain each of these steps in the following sections.

> **Note:** All the EJBs in this example are EJB 3.0.

Activating the compensation service in the server

The compensation service required for business activities is not active by default in the server. To activate the compensation service:

1. Open the administrative console.

2. Expand **Servers** → **Server Types** → **WebSphere Application servers** and select the server.

3. Under Container Settings, select **Container Services** → **Compensation service**.

4. Under General Properties (Figure 8-19), select **Enable service at server startup** and click **OK**.
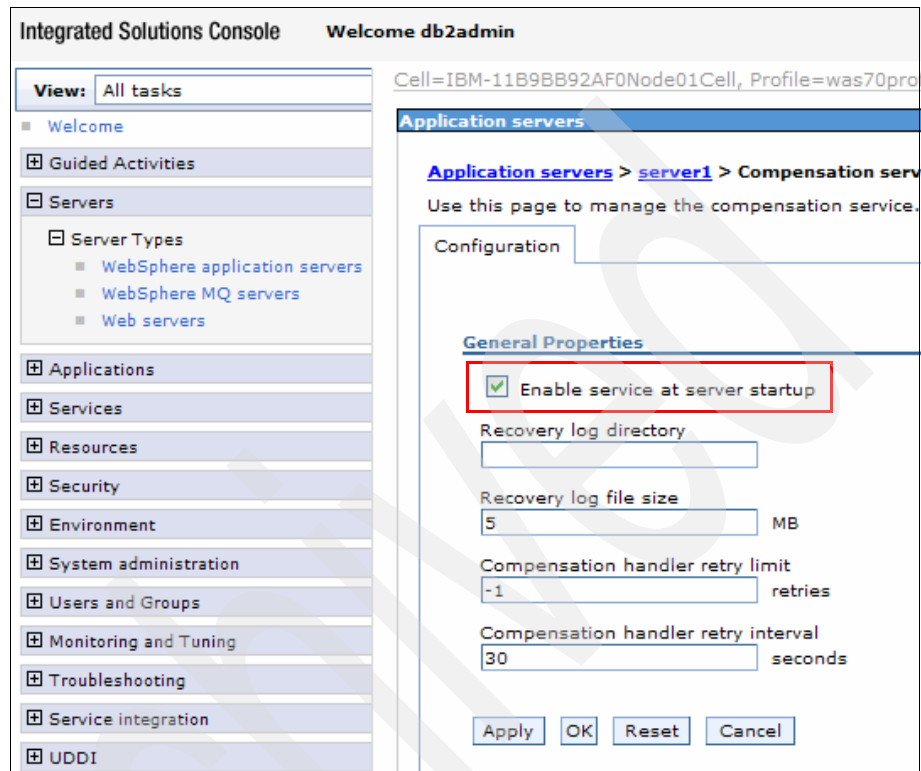
Figure 8-19   Activating compensation service

5. Save the configuration and restart the server. By default, the recovery log directory is in *profile_root*/recoveryLogs.

Using multiple transactions

To run the three EJBs in separate transactions, inject the annotations that are shown in Example 8-6 at the class level of each EJB to specify the default transaction management and attribute, respectively, for all EJB business methods.

Example 8-6   Transaction EJB annotations

```
@TransactionManagement(value=TransactionManagementType.CONTAINER)
@TransactionAttribute(value=TransactionAttributeType.REQUIRES_NEW)
```

Creating a service data object for compensation

To register a bean as a business activity, you must set up a compensation bean and a service data object (SDO) that holds the information necessary to perform the compensation. For our application, the SDO only requires the date of the weather record. This is enough to delete the inserted weather record for that date as a compensation of the insert.

Example 8-7 shows the compensation data bean. The CompensationData JavaBean is imported into the `itso.compensation` package.

Example 8-7   Compensation data JavaBean (SDO)

```
public class CompensationData {

    private DataObject weatherDate;

    public CompensationData(Calendar date, String location) {
        super();
        try {
            MetadataFactory mFactory = MetadataFactory.eINSTANCE;
            Metadata metadata = mFactory.createMetadata();
            Table table = metadata.addTable("ANYTABLE");
            table.setSchemaName("ITSO");
            Column dateColumn = table.addDateColumn("WeatherDate"); //key
            table.addStringColumn("Location");
            dateColumn.setNullable(false);
            table.setPrimaryKey(dateColumn);
            metadata.setRootTable(table);

            JDBCMediatorFactory medFactory =
JDBCMediatorFactory.soleInstance;
            JDBCMediator mediator = medFactory.createMediator(metadata);
            DataObject graph = mediator.getEmptyGraph();
            weatherDate = graph.createDataObject(0);
            weatherDate.setDate("WeatherDate", date.getTime());
            weatherDate.setString("Location", location);
            System.out.println("Created CompensationData: " + location + "
" + weatherDate.getDate("WeatherDate"));
        } catch (Exception e) {
            System.out.println("Cannot create CompensationData: " +
e.getMessage());
        }
    }
```

```
        public DataObject getSDO() { return weatherDate; }

}
```

In this code, a JDBCMediator is used to create the SDO for an imaginary table with two columns:

- ► WeatherDate, which is the date of the weather record
- ► Location, which indicates the location

The getSDO() method retrieves the SDO.

Creating the compensation classes

For each session bean of the business activity, define a compensation class that is invoked if something goes wrong and the data must be rolled back.

Example 8-8 shows the code of the RaleighCompensate class.

The three compensation classes, RaleighCompensate, SanJoseCompensate, and UpdaterCompensate, are imported into the itso.compensation package.

Example 8-8   Compensation class

```
public class RaleighCompensate implements CompensationHandler {

    public void close(DataObject arg0) throws
RetryCompensationHandlerException, CompensationHandlerFailedException {
        System.out.println("Compensate close for " +
arg0.getString("Location"));
    }

    public void compensate(DataObject arg0) throws
RetryCompensationHandlerException, CompensationHandlerFailedException {
        Calendar cal = Calendar.getInstance();
        cal.setTime(arg0.getDate("WeatherDate"));
        WeatherDAO dao = new WeatherDAO();
        boolean result = dao.deleteWeather(cal);
        if (result) System.out.println("Compensate: deleted weather for "
+ arg0.getString("Location"));
    }
}
```

A compensation class must provide the following methods:

► close(), which is called after a commit (can do some cleanup)

► compensate(), which is called to roll back the update (in our case, delete the weather record that was inserted)

The SDO is passed as a parameter and should have enough information to perform the rollback.

Note that the UpdaterCompensate bean has no work to perform.

Activating compensation for the session beans

To activate compensation and provide the name of the compensation class for each bean:

1. In the workspace, right-click the **WeatherBAEJB** project and select **Java EE → Generate WebSphere Programming Model Extensions Deployment Descriptor** (Figure 8-20).
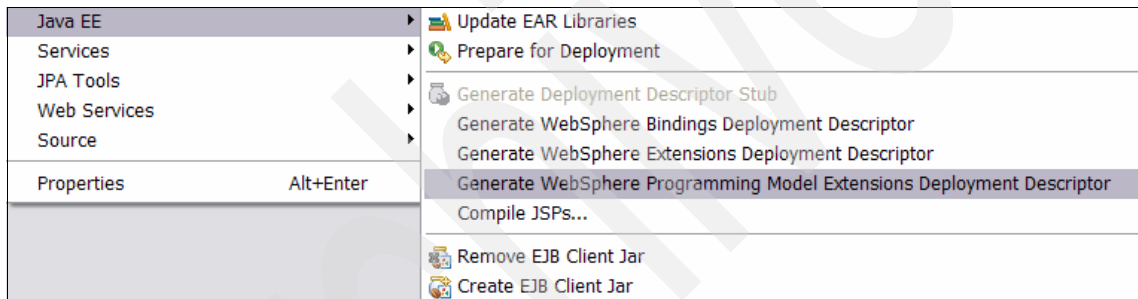
Figure 8-20   Generating the PME

2. Right-click **EJB PME** and select **Add** → **Compensation** (Figure 8-21).
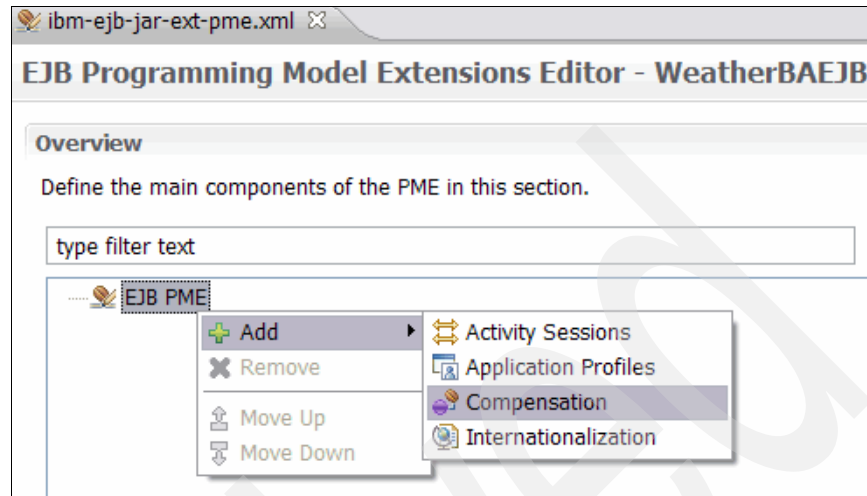
Figure 8-21   Adding compensation

3. Right-click **Compensation** and select **Add** → **Bean Policy** (Figure 8-22).
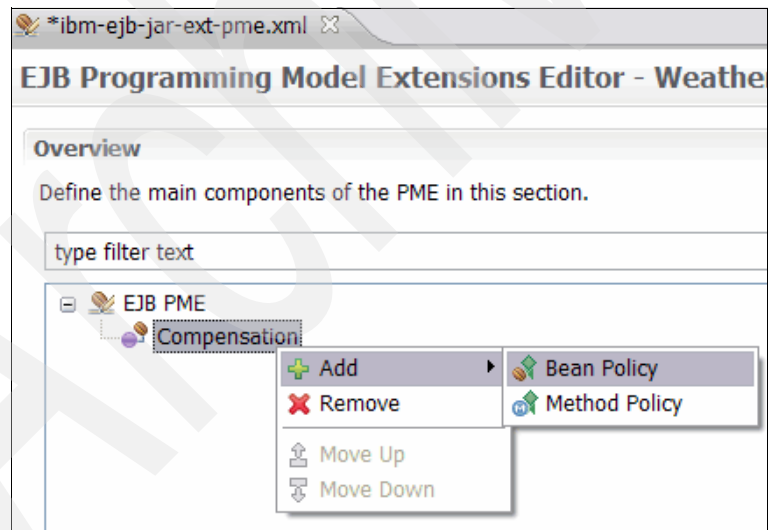
Figure 8-22   Adding bean policy

4. Change the Type field to `REQUIRED`, and for Class, browse to the `itso.compensation.RaleighCompensate` class as the compensation handler (Figure 8-23).
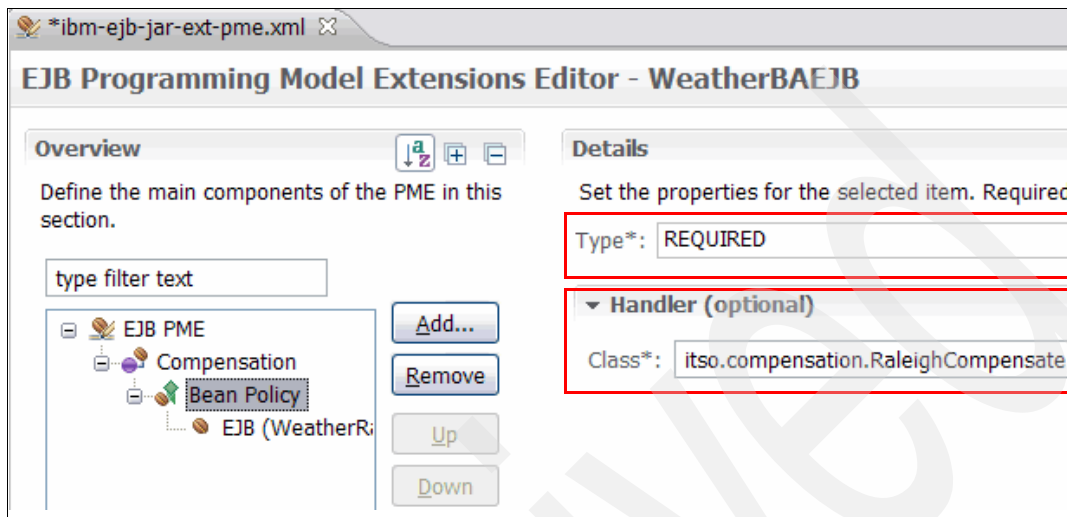
Figure 8-23   Configuring the bean policy

5. Click **EJB ()**, which is the child of Bean Policy, and set the EJB name to `WeatherRaleigh`.
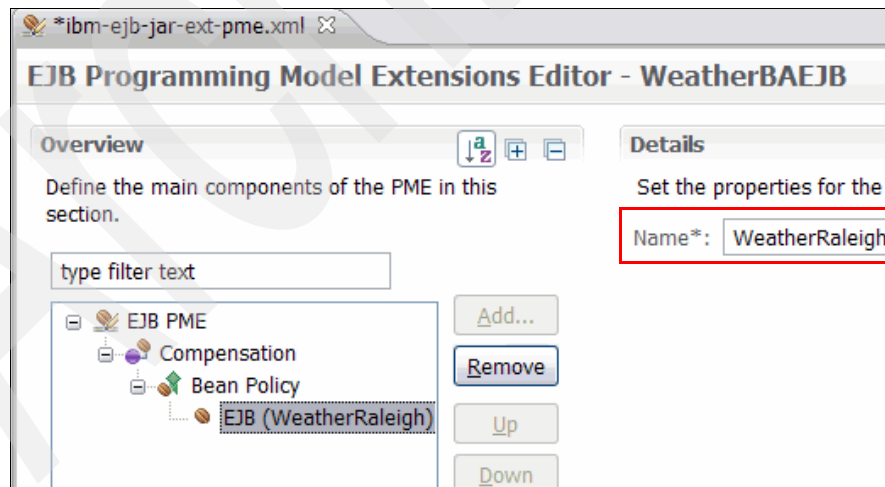
Figure 8-24   Setting the EJB name

6. Repeat the steps starting from step 3, but this time for the WeatherSanJose and WeatherUpdaterSoapBindingImpl EJB. Figure 8-25 shows the result.
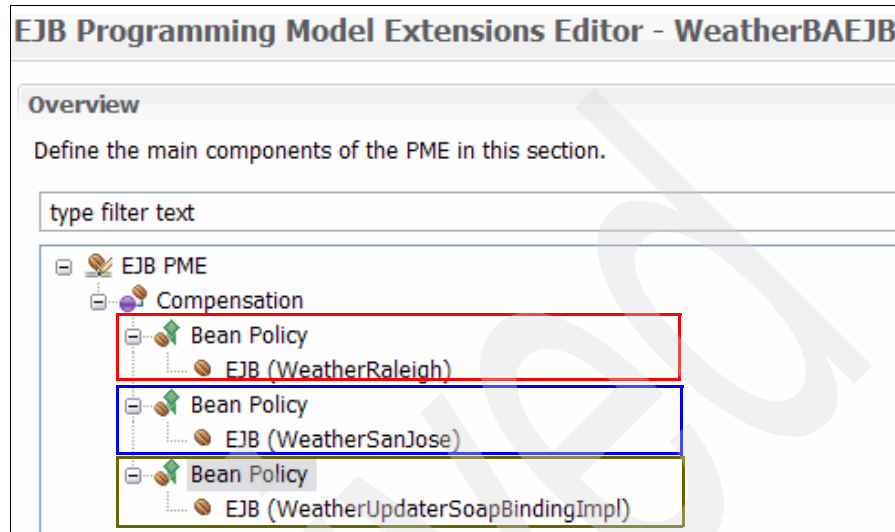
Figure 8-25   The result for adding all the bean policies

Registering the session bean with the compensation service

Finally, write the code in the session bean to register the EJB with the compensation service and set the SDO object for compensation.

Example 8-9 shows the updateSanJose() method with business activity support added.

Example 8-9   Session bean with business activity support

```
public String updateSanJose (Calendar date) throws Exception
{
    InitialContext ctx = new InitialContext();
    UserBusinessActivity uba = (UserBusinessActivity)
    ctx.lookup("java:comp/websphere/UserBusinessActivity");
    CompensationData sdo = new CompensationData(date, "San Jose");
    uba.setCompensationDataImmediate(sdo.getSDO());

    Weather w = new Weather(date);
    WeatherPredictor.calculateWeatherValues(w);
    WeatherDAO dao = new WeatherDAO();
    dao.deleteWeather(date);
    dao.insertWeather(w);
    System.out.println("Weather San Jose inserted: " + w);
```

```
    // rollback if month=October
    if (date.get(Calendar.MONTH) == 9)
        throw new Exception("Simulated abort in San Jose");

    return "San Jose " + w;
}
```

The method in Example 8-9 on page 392 does the following actions:

► Retrieves the UserBusinessActivity, initializes the SDO, and registers the SDO for compensation:

   – setCompensationDataImmediate activates the compensation immediately.

   – setCompensationDataAtCommit is used when a global transaction is present.

   These methods can be called multiple times with an updated SDO as the application makes changes to the database.

► For testing the compensation, it throws an exception when the month is October. In the updateRaleigh() method, the same is done for September.

Example 8-10 shows the updateWeather method with business activity support added. In the main session bean, any exceptions that occur in the called beans are caught to initiate a rollback.

Example 8-10   Session bean method in WeatherUpdater

```
public String updateWeather(XMLGregorianCalendar date) throws
Exception{
    Calendar date1 = Calendar.getInstance();
    date1.setTime(date.toGregorianCalendar().getTime());
    Calendar date2 = (Calendar)date1.clone();
    date2.roll(Calendar.DATE, true);

    String result1=null, result2=null;
    //Create a new context
    InitialContext ctx = new InitialContext();
    // business activity
    UserBusinessActivity uba = (UserBusinessActivity)
ctx.lookup("java:comp/websphere/UserBusinessActivity");
    CompensationData sdo = new CompensationData(date1, "Updater");
    uba.setCompensationDataImmediate(sdo.getSDO());

    //use the context to lookup the remote interface.
```

```
        WeatherSanJoseRemote ejbSanJose = (WeatherSanJoseRemote)
ctx.lookup(WeatherSanJoseRemote.class.getName());
        WeatherRaleighRemote ejbRaleigh = (WeatherRaleighRemote)
ctx.lookup(WeatherRaleighRemote.class.getName());

    try{
        result1 = ejbSanJose.updateSanJose(date1);
        result2 = ejbRaleigh.updateRaleigh(date2);
    } catch (Exception e) {
        e.printStackTrace();
        uba.setCompensateOnly();
    }
    // rollback if month=August
    if (date1.get(Calendar.MONTH) == 7)
    {
        throw new RemoteException("Simulated abort in Updater");
    }
    if (uba.isCompensateOnly())
        System.out.println("Weather updates will be compensated");
    else
        System.out.println("Updated weather: \n" + result1 + "\n" +
result2);

    return "Updated weather: \n" + result1 + "\n" + result2 ;
}
```

The method in Example 8-10 on page 393 shows the following actions:

▶ To initiate compensation when a rollback occurs in a called bean, we issue the setCompensateOnly method.

▶ If the date is in August, we simulate an exception in the main bean.

8.4.4 Application testing with business activity support

After making all the changes, run the Web service client.

Deploy the application on the server and run the TestClient.jsp file found in WeatherBAEJBClient/WebContent/sampleWeatherUpdaterProxy.

Test the following scenarios:

- Run the Web service with a July date. Both inserts should be committed.

- Run the Web service with an August date. All updates are compensated by simulating an exception in the main bean.

- Run the Web service with a September date. All updates are compensated because the Raleigh bean throws an exception.

- Run the Web service with an October date. The San Jose update is compensated. (The Raleigh bean has not been called yet.)

Watch the messages in the console. Example 8-11 shows the messages when all updates are compensated.

Example 8-11   Console messages

```
Created CompensationData: Updater Sat Aug 01 00:00:00 GMT+02:00 2009
Created CompensationData: San Jose Sat Aug 01 00:00:00 GMT+02:00 2009
Weather San Jose inserted: Weather: Sat. Aug 1, 2009 GMT+02:00, stormy,
wind: NE at 1km/h , temperature: 20 Celsius
Created CompensationData: Raleigh Sun Aug 02 00:00:00 GMT+02:00 2009
Weather Raleigh inserted: Weather: Sun. Aug 2, 2009 GMT+02:00, cloudy,
wind: NW at 17km/h , temperature: 33 Celsius
CNTR0020E: EJB threw an unexpected ..... Exception data:
java.rmi.RemoteException: Simulated abort in Updater
Compensate compensate for Updater
Compensate: deleted weather for San Jose
Compensate: deleted weather for Raleigh
```

8.5 More information

The WS-Transactions specifications are available from the following resources:

- WS- AtomicTransaction 1.1

  http://docs.oasis-open.org/ws-tx/wstx-wsat-1.1-spec-os/wstx-wsat-1.1-spec-os.html

- WS- AtomicTransaction 1.0

  http://www.ibm.com/developerworks/library/specification/ws-tx/#atom

- WS-BusinessActivity 1.1

  http://docs.oasis-open.org/ws-tx/wstx-wsba-1.1-spec-os/wstx-wsba-1.1-spec-os.html

- WS-BusinessActivity 1.0

  http://www.ibm.com/developerworks/library/specification/ws-tx/#ba

- WS-Coordination 1.1

  http://docs.oasis-open.org/ws-tx/wstx-wscoor-1.1-spec-os/wstx-wscoor
  -1.1-spec-os.html

- WS-Coordination 1.0

  http://www.ibm.com/developerworks/library/specification/ws-tx/#coor

In addition, consult the following topics in the WebSphere Application Server V7
Information Center:

- "Web services Atomic Transaction support in WebSphere Application Server"

  http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/index.jsp?topic
  =/com.ibm.websphere.base.doc/info/aes/ae/cjta_wstran.html

- "Web services Business Activity support in the application server"

  http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/topic/com.ibm.
  websphere.base.doc/info/aes/ae/cjta_wsba.html

- "Configuring a server to use business activity support"

  http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/index.jsp?topic
  =/com.ibm.websphere.base.iseries.doc/info/iseries/ae/tjta_wsba_enable
  .html

- "Transaction compensation and business activity support"

  http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/index.jsp?topic=/
  com.ibm.websphere.express.iseries.doc/info/iseriesexp/ae/cjta_wsbascope
  .html

- "Business activity API"

  http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/index.jsp?topic
  =/com.ibm.websphere.express.doc/info/exp/ae/rjta_wsba_api.html

9

WS-Notification

In this chapter we describe the support that is provided by WebSphere Application Server V7.0 and Rational Application Developer for WebSphere Software 7.5 for developing and configuring Web Services Notification (WS-Notification)-based applications.

In this chapter we begin by providing an overview of the WS-Notification specification. Next we discuss how WebSphere Application Server supports the WS-Notification specification. Then we explain how to configure a WS-Notification broker application and how to develop WS-Notification consumer and producer applications. In addition, we discuss the advanced features and configuration options for WS-Notification.

The WS-Notification examples in this chapter introduce WS-Notification development by using the new Java API for XML Web Services (JAX-WS) 2.1 programming model.

This chapter contains the following topics:

9.1 WS-Notification overview

WS-Notification can be described as publish/subscribe for Web services. More formally, WS-Notification is a family of related white papers and specifications that define a standard Web services approach to notification by using a topic-based publish/subscribe pattern.

The event-driven, or notification-based, interaction pattern is a commonly used pattern for interobject communications. Examples exist in many domains, for example, in publish/subscribe systems provided by message-oriented middleware vendors or in system and device management domains. This notification pattern is increasingly being used in a Web services context.

The white paper *Publish-Subscribe Notification for Web services*, available at the following address, introduces the concepts of notification patterns. It also sets the goals and requirements for the WS-Notification family of specifications.

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

WS-Notification consists of the following specifications:

► WS-BaseNotification

 http://docs.oasis-open.org/wsn/wsn-ws_base_notification-1.3-spec-os.pdf

► WS-BrokereredNotification

 http://docs.oasis-open.org/wsn/wsn-ws_brokered_notification-1.3-spec
 -os.pdf

► WS-Topics

 http://docs.oasis-open.org/wsn/wsn-ws_topics-1.3-spec-os.pdf

9.1.1 WS-BaseNotification

The WS-BaseNotification specification defines the basic roles that are required for publishing and subscribing. These roles are the *NotificationProducer* and *NotificationConsumer* roles. The NotificationProducer role is responsible for producing (publishing) notification messages to a NotificationConsumer.

To establish a relationship between the producer and the consumer, the NotificationProducer accepts subscription requests from, or on behalf of, a NotifcationConsumer. Such requests include, among other information, a definition of which topics the consumer wants to receive messages.

The following two styles of consumer are described within the specification:

- ► Push-style consumer

  A push-style consumer is sent notifications when the producer determines that messages are available for a topic to which the consumer is subscribed. The producer pushes messages to the consumer. The push-style consumer is required to expose a Web service endpoint to accept receipt of such messages.

- ► Pull-style consumer

  A pull-style consumer takes the opposite approach. Instead of the producer (or broker) sending messages to the consumer whenever they are available, the consumer controls the rate and timing of the message transfer by requesting messages from the producer when it requires them. The consumer pulls messages from the producer.

  To facilitate the ability to pull messages from a producer (or broker), a mechanism called a *pull point* is defined. Before subscribing to the producer, a pull style consumer requests that the producer create a pull point. In response, the producer returns an endpoint reference for the pull point. When the consumer subsequently subscribes to the producer, it specifies the pull point endpoint reference as the consumer reference. This endpoint reference indicates to the producer that messages destined for the consumer should be sent to the pull point.

  At a time of its choosing, the consumer then retrieves (pulls) the messages from the pull point. The roles that are necessary for handling pull points are defined by the *CreatePullPoint* and *PullPoint* port types.

Finally, the WS-BaseNotification specification defines roles for handling the lifetime management of subscriptions (SubscriptionManager and PausableSubscriptionManager). The port types for these roles allow a subscription to be paused, resumed, renewed, and deleted (unsubscribed).

All roles are specified in the form of Web services Description Language (WSDL) 1.1 port types and associated operations, messages, and XML schema definitions. You can find detailed descriptions of the roles and the WSDL and XML schemas that are used in the WS-BaseNotification specification.

9.1.2 WS-BrokeredNotification

The WS-BrokeredNotification specification builds on the concepts defined in the WS-BaseNotification specification to describe a *NotificationBroker* role. A NotificationBroker is an intermediary between a NotificationProducer and a NotificationConsumer.

The NotificationBroker role includes the following benefits, among others:

▶ Allows applications that do not expose Web service endpoints to publish messages

From the consumer point of view, the necessary Web service endpoints for creating and managing subscriptions are provided by the broker.

▶ Reduces the number of messages sent by a producer

The producer can send an individual message to the broker who then potentially distributes it to multiple consumers.

▶ Reduces the number of messages sent to a consumer

The broker can consolidate notifications from multiple producers that match the requirements of a consumer subscription, into a single call to the notify operation of the NotificationConsumer.

▶ Anonymizes notifications so that consumers and producers are unaware of each other's identity

Demand-based publishing is an important pattern introduced by the WS-BrokeredNotification specification. In this pattern, a producer registers with the broker before it publishes messages. As part of the registration process, the producer indicates that it is interested in knowing whether subscribers exist for the topics to which it publishes. In return, the broker subscribes to the producer and uses the subscription as an indication of the demand for messages. By pausing the subscription, the broker indicates that no active subscriptions exist. Therefore, the producer can decide to temporarily stop publishing messages. When the subscription is resumed by the broker, once again, demand for messages exists.

Figure 9-1 shows the basic interactions that occur for a brokered WS-Notification system.
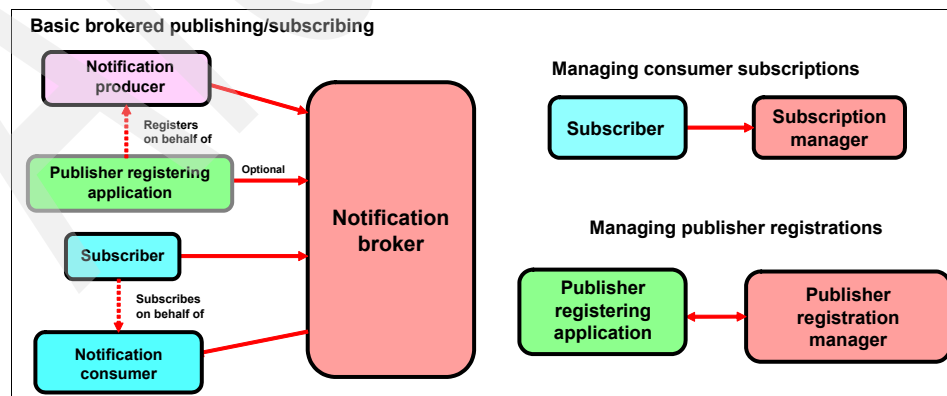
Figure 9-1   Basic brokered WS-Notification interactions

> **Separation of entities**: The separation of the NotificationBroker,
> SubscriptionManager, and PublisherRegistrationManager entities shown is for
> clarity only and is not a requirement of the WS-Notification specification.

9.1.3 WS-Topics

The WS-Topics specification defines the terms and mechanisms for
discriminating between items of interest when subscribing or publishing. These
items (or *topics* as they are called) can be organized into hierarchies and
grouped by XML namespaces. The WS-Topic specification defines a convention
for referring to a single topic or group of topics, called a *topic expression*.

Topic expressions contain two pieces of information, a component that dictates
the style of the content in the expression, known as a *dialect*, and the content of
the expression itself. The WS-Topics specification defines three standard dialects
of topic expressions:

► Simple topic expressions

This is a basic style of topic expression in which the only allowed expressions
are QNames. That is, only root topics (those with no parent topic) can be
referred to by simple topic expressions. There is no topic hierarchy or wild cards.

A simple topic expression has the following dialect value:

```
http://docs.oasis-open.org/wsn/t-1/TopicExpression/Simple
```

The following example shows a simple topic expression:

```
tns1:stock       Indicates a topic named "stock" in a namespace
                 corresponding to the prefix "tns1"
```

► Concrete topic expressions

This topic dialect extends the simple topic expression pattern to allow topic
hierarchies by using the forward slash (/) character to indicate a child of a
relationship. This topic dialect does not allow any wild cards, and a valid simple
topic expression is automatically valid in the concrete topic expression dialect.

A concrete topic expression has the following dialect value:

```
http://docs.oasis-open.org/wsn/t-1/TopicExpression/Concrete
```

The following example shows a concrete topic expression:

```
tns1:stock/IBM    Indicates a subtopic named "IBM" of topic "stock"
                  in a namespace correponding to the prefix "tns1"
```

► Full topic expressions

This topic dialect extends the concrete topic expression dialect to include the
concepts of wild cards and conjunction. It is based on a subset of the XPath

location path expressions and describes how expressions of this type can be evaluated by using the XML document representation of a topic space. Wild cards in topics are achieved by using the XPath style asterisk (*) and dot (.) characters, with conjugation described using the pipe (|) operator.

A full topic expression has the following dialect value:

```
http://docs.oasis-open.org/wsn/t-1/TopicExpression/Full
```

The following examples are full topic expressions:

tns1:stock/*   Indicates all subtopics of the topic named *stock* in the namespace that corresponds to the prefix *tns1*.

tns1:cars|tns2:boats   Indicates the topic named *cars* in the namespace that corresponds to the prefix *tns1* or the topic named *boats* in the namespace that corresponds to the prefix *tns2*.

Additional terms defined by WS-Topics include topic trees (a hierarchical grouping of topics), topic namespaces (a hierarchical grouping of topics under the same namespace), and topic sets (the set of topics supported by a producer or broker).

For full details about all concepts, and additional examples of topic expressions, see the WS-Topics specification document.

9.2 WS-Notification in WebSphere Application Server

WebSphere Application Server V7 supports Version 1.3 of the WS-Notification family of specifications. In the following sections we discuss the resources that are provided within the application server to support the use of WS-Notification.

9.2.1 Core WS-Notification resources

The WS-Notification resources are services and service points.

WS-Notification services
A WS-Notification service provides the ability to expose some or all of the messaging resources that are defined on a service integration bus (SIB) for use by WS-Notification applications. It encapsulates the Web service and messaging resources necessary for the application server or cluster to act as a WS-Notification broker application.

Usually, you configure one WS-Notification service for an SIB, but you can configure more than one.

A WS-Notification service references three SIB inbound services:

► Notification broker inbound service

This inbound service exposes operations that are defined by the NotificationBroker port type from WS-BrokeredNotification and the CreatePullPoint and PullPoint port types from WS-BaseNotification. These port types define the functions that are necessary to subscribe consumers and publish messages.

► Subscription manager inbound service

This inbound service exposes operations that are defined by the PausableSubscriptionManager port type from WS-BaseNotification. This port type defines the function necessary to manage the lifetime of a consumer subscription.

► Publisher registration manager inbound service

This inbound service exposes operations that are defined by the PublisherRegistrationManager port type from WS-BrokeredNotification. This port type defines the function necessary to manage the lifetime of a publisher registration.

All three inbound services support the GetResourceProperty and SetTerminationTime operations that are defined by the WS-ResourceProperties specification. These operations allow properties of WS-Notification service resources to be queried and, for some resources such as subscriptions, allow the termination time to be set.

WS-Notification service types: Version 6.1 versus Version 7.0

WebSphere Application Server V7 introduces the following types of WS-Notification services. Upon creation of a WS-Notification service, you must make a choice between the two types:

► Version 6.1
► Version 7.0

With the Version 6.1 WS-Notification service, you can expose a Java API for XML-based remote procedure calls (JAX-RPC) WS-Notification service by using the same technology that is provided in WebSphere Application Server V6.1. One reason for choosing this type of WS-Notification service is the ability to apply existing JAX-RPC handlers to the service. Another reason might be that you want to use the SOAP/JMS binding, which is *not* supported by the Version 7.0 service.

The Version 7.0 WS-Notification service exposes a JAX-WS 2.1 WS-Notification service. This is the recommended type of service for new deployments for the following reasons:

► It allows you to apply JAX-WS logical and protocol handlers.

► It allows you to apply policy sets for easy configuration of quality of service (QoS) features.

The applicability of policy sets is a compelling option that comes with WebSphere Application Server V7. With policy sets, you can easily add advanced QoS features, such as reliable message exchange, by using WS-ReliableMessaging, or security, by using WS-Security features.

WS-Notification service points

A WS-Notification service point defines access to a WS-Notification service on a given bus member through a specified Web service binding (for example, SOAP over HTTP). Applications connect to a WS-Notification service by the bus member that is associated with a service point.

You can define any number of service points for a given WS-Notification service. Each service point defined for the same WS-Notification service represents an alternative entry point to the service. Event notifications that are published to a particular WS-Notification service point are received by all applications that are connected to any service point of the same WS-Notification service (subject to subscription on the correct topic). This happens regardless of the particular service point to which they are connected.

There are two main cases for which you may want to create more than one service point for a given WS-Notification service:

► To expose one WS-Notification service on multiple bus members (servers or clusters)

► To expose a WS-Notification service on a single bus member (server or cluster) by using multiple bindings or by using different security configurations

Each WS-Notification service point encapsulates three SIB inbound ports, one corresponding to each of the three inbound services that belong to the parent WS-Notification service.

Relationships with service integration bus Web services

Figure 9-2 illustrates how WS-Notification service and service points relate to SIB Web services.
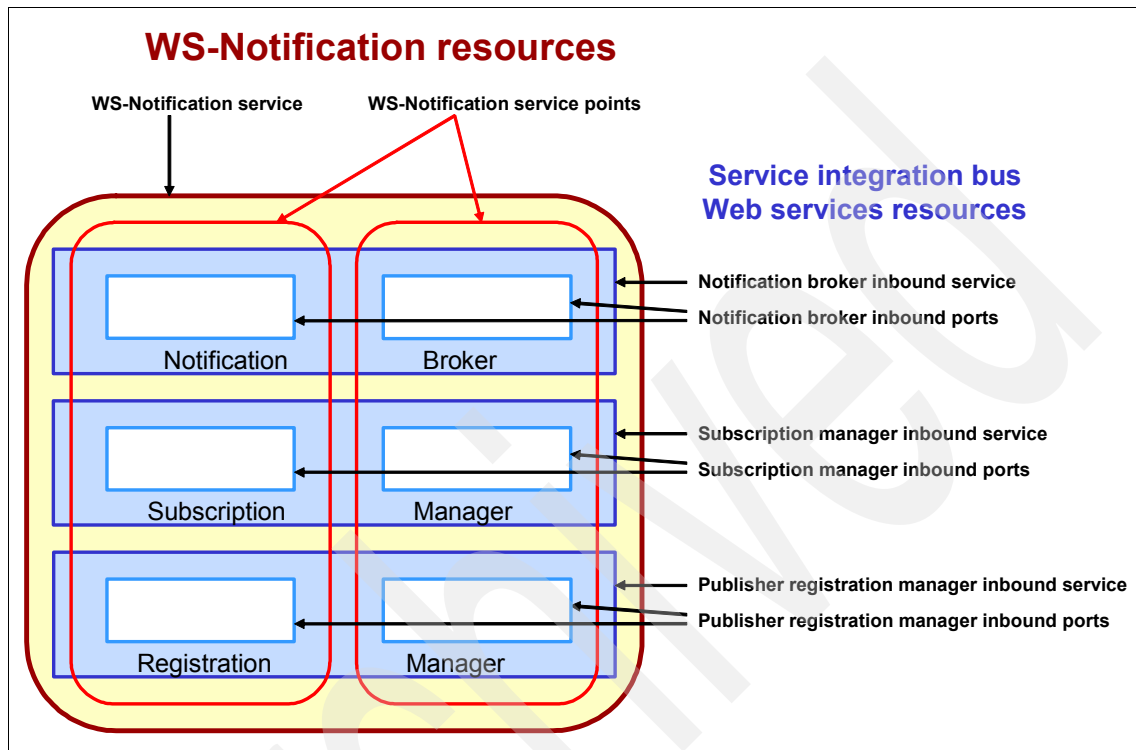
Figure 9-2   Relationships between WS-Notification and SIB Web services resources

We explain Figure 9-2 as follows:

► A WS-Notification service contains one or more service points (two in Figure 9-2) and refers to three inbound services. Each inbound service is related to port types from the WS-Notification specifications.

► Each WS-Notification service point refers to three inbound ports, each one belonging to an inbound service that is referred to the parent WS-Notification service.

► Each of the three inbound services relates to the same individual WS-Notification service.

► Each inbound port relates to the same individual WS-Notification service point and relates to one inbound service.

Topic namespaces and other topic-related features

Similar terminologies are used by the WS-Notification specifications and WebSphere Application Server messaging in relation to the handling of topics. In the following sections we list the terms and provide brief definitions.

For full definitions and further information, see the "WS-Notification terminology" topic in the WebSphere Application Server Information Center at the following address:

`http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/topic/com.ibm.websphere.pmc.nd.doc/ref/rjwsn_terms.html`

Topic-related terms

The following terms are related to the handling of topics:

Topic namespace
A WS-Notification term for a hierarchical collection (tree) of topics, referenced by a namespace Uniform Resource Identifier (URI). In WebSphere Application Server V7, topic namespaces are broken down into two patterns:

Permanent topic namespace
A static association between a WS-Notification topic namespace URI and a WebSphere Application Server topic space.

Dynamic topic namespace
Used in response to a request from a WS-Notification application for a topic namespace that has not been defined as a permanent topic namespace.

Topic space
A WebSphere Application Server term for a hierarchy of topics that is used for publish/subscribe messaging.

Topic
In WebSphere Application Server, a discriminator within a topic space. In WS-Notification, a discriminator within a topic namespace.

Topic expression
A WS-Notification term for the means by which you refer to a topic. A topic expression contains a dialect component. WebSphere Application Server V7 supports the three standard dialects defined by WS-Topics, which are simple, concrete, and full topic expressions. For more information about these dialects, see "WS-Topics" on page 401.

An additional term that might be encountered if you configure Java Message Service (JMS) resources or applications is *JMS topic*. JMS topic is an

administrative object that encapsulates the name of a topic and a topic space on an SIB. JMS applications can publish or subscribe to JMS topics.

Overview of WS-Notification in WebSphere Application Server

Figure 9-3 shows an outline of WS-Notification applications interacting with a WS-Notification service.
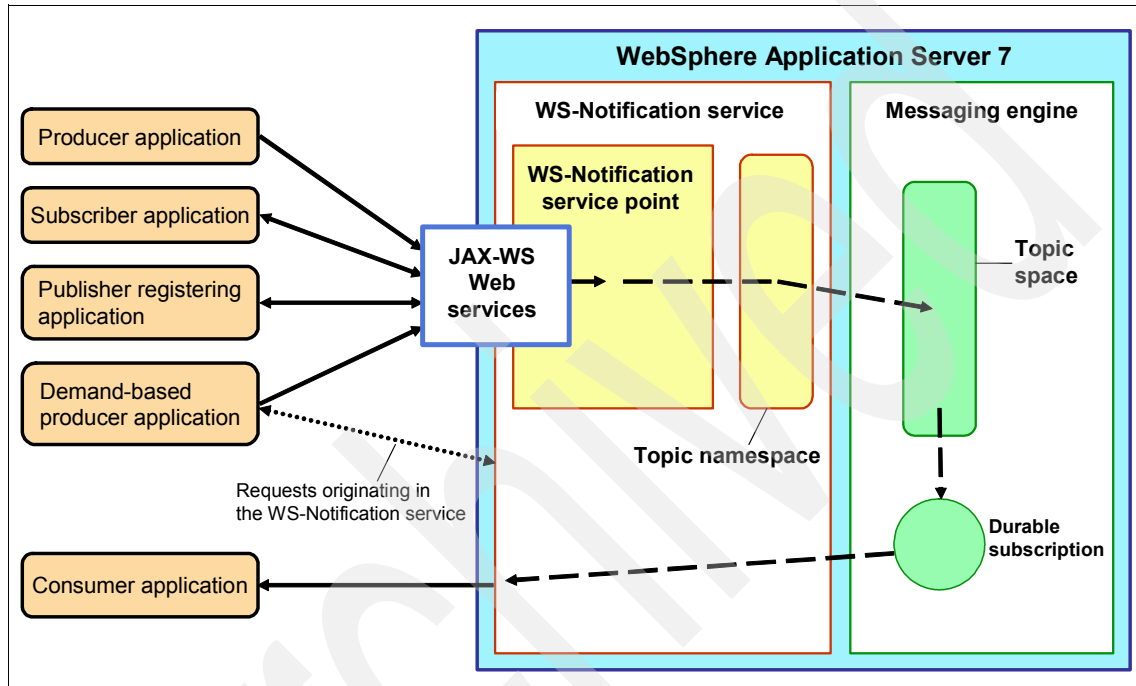
Figure 9-3   WS-Notification applications interacting with a WS-Notification service

In a Version 7.0 WS-Notification service, WS-Notification applications interact with the three inbound services (notification broker, subscription manager, and publisher registration manager) through JAX-WS Web services. These Web services are part of the WS-Notification service point that is implemented as a Java Platform, Enterprise Edition (Java EE), application.

WS-Notification message publishing is destined for a topic namespace in the WS-Notification service. This topic namespace in turn refers to an actual topic space on the active messaging engine of the SIB.

The following applications, as shown in Figure 9-3 on page 407, typically interact with a WS-Notification service in WebSphere Application Server V7:

▶ Producer applications

The applications that publish WS-Notification messages.

▶ Subscriber applications

The applications that subscribe on behalf of the consumer applications. Alternatively, subscription functionality can be part of the consumer applications.

▶ Publisher registering applications

The applications that register the (demand based) producer applications. Alternatively, this functionality can be part of the publisher applications.

▶ Demand-based producer applications

The applications that publish WS-Notifications messages and are subscribed to by the broker service.

▶ Consumer applications

The applications that consume WS-Notification messages.

9.2.2 Configuring a WS-Notification broker application

A WS-Notification broker application is represented in WebSphere Application Server V7 by a combination of a WS-Notification service and related WS-Notification service points. To configure a WS-Notification broker, it is necessary to configure these resources.

In 9.3, "Developing WS-Notification applications" on page 416, we introduce a daily weather WS-Notification development example. In this section we demonstrate how to create the WS-Notification broker application resources that are used for the example. The resources are created by using the WebSphere administrative console. However, it is possible to create all of these resources by using the `wsadmin` command-line tool if you prefer. See 9.2.3, "WS-Notification wsadmin commands" on page 415, for a list of WS-Notification `wsadmin` commands.

> **Using a script to get up and running quickly with WS-Notification:** To
> save time or to try WS-Notification, consult the "Using a script to get up and
> running quickly with WS-Notification" topic in the WebSphere Application
> Server Information Center at the following address:
>
> http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/topic/com.ibm.
> websphere.pmc.nd.doc/tasks/tjwsn_task_sysa0.html
>
> This topic provides a Jython script that automatically creates all the necessary
> resources that comprise a basic WS-Notification broker application. To run the
> script, run the `wsadmin` tool as follows:
>
> wsadmin -f wsnQuickStart.py

Creating a service integration bus

A WS-Notification service must be attached to a bus. For the Weather example,
we created an SIB named *weatherWSNBus* with a single application server
instance member. We restarted the application server to ensure that the bus was
active.

Creating WS-Notification resources by using the administrative console

You can create the core WS-Notification resources by using the administrative
console in WebSphere Application Server V7. The administrative console has a
wizard with which you can create a WS-Notification service, one or more
WS-Notification service points, and one or more permanent topic namespaces.

To create a WS-Notification service, a single service point, and a single
permanent topic namespace, create these resources for the weather
WS-Notification examples:

1. From the WebSphere administrative console, in the left pane, select **Service
   integration** → **WS-Notification** → **Services**. In the right pane, click **New**.

2. On the New WS-Notification service page (Figure 9-4):

a. For name, type `weatherWSNService`.

Figure 9-4   New WS-Notification service page

b. For service integration bus name, select **weatherWSNBus**.

Reminder: A WS-Notification service is associated with only one bus, but multiple WS-Notification services can be associated with the same bus.

c. Select the following options as appropriate for your WS-Notification service:

- Select the **Enable dynamic topic namespaces?** option if you want messages to be published to topics that belong to namespaces other than those that are defined by permanent topic namespaces. This option specifies whether dynamic topic namespaces are supported by the service. The default setting is to enable dynamic topic namespaces.

- Select the **Requires registration** option if you want to prevent the processing of messages that are received by non-registered publishers. This option specifies whether publishers (NotificationProducers) must register with the WS-Notification service before they can publish messages to it. The default setting is *registration not required*.

d. Click **Next**.

3. When prompted to choose between a Version 7.0 and Version 6.1 WS-Notification type service, select **Version 7.0** and click **Next**.

4. On the Configure handler and Web service policy settings page, accept the suggested default values and click **Next**.

5. On the Create WS-Notification service points page, create a service point and click **Next**.

6. For the service point name, type `weatherWSNServicePoint` and click **Next**.

7. On the Define transport settings page, select either a SOAP 1.1/HTTP binding or a SOAP 1.2/HTTP binding. If you choose SOAP 1.1, leave the default values as suggested. Then click **Next**.

8. Back on the Create WS-Notification service points page (Figure 9-5), on which you now see the service point, for Create another instance?, select **No** and then click **Next**.

Figure 9-5   Create WS-Notification service points page

9. On the Create permanent topic namespaces page, for Create a new instance?, select **Yes** and click **Next**.

10. On the Configure namespace and select service integration bus topic space page (Figure 9-5 on page 412):

    a. For namespace, type `http://weather`.

    b. For service integration bus topic space, select **Create a new topic space** to create a new topic namespace on the SIB that is dedicated to the WS-Notification messages.

    c. For the topic space name, type `weatherTS`. Click **Next**.

Figure 9-6   Configure namespace and select SIB topic space page

11. On the next page, for Create another instance?, select **No** and click **Next**.

12. On the Summary page, click **Finish**.

13. When prompted by the administrative console to save or discard the configuration changes, save all changes to the master configuration.

After you finish the WS-Notification configuration, WebSphere Application Server generates, among other output, the enterprise application *WSN_weatherWSNService_weatherWSNServicePoint*. This application contains Web service endpoints that correspond to the three inbound notification services, which are:

► Notification broker
► Subscription manager
► Publisher registration manager

Ensure that this enterprise application is started. Otherwise, none of the WS-Notification resources are accessible by using the three inbound Web services. As with any other enterprise application, you can start the WS-Notification service point application by using the administrative console. In the administrative console, select **Applications → Application Types → WebSphere enterprise applications**.

Administering WS-Notification resources

By using the administrative console, select **Service integration → WS-Notification** to see the WS-Notification services that you created. Figure 9-7 shows the one that we created.

Figure 9-7   Created WS-Notification services page

You can either create a new service or select the one that you created and delete it. If you click the WS-Notification service name, which is weatherWSNService in Figure 9-7, on the page that opens, you can add additional service points and topic namespaces, as well as perform miscellaneous administrative tasks.

9.2.3 WS-Notification wsadmin commands

The `wsadmin` program provides several `AdminTask` commands that you can use to administer WS-Notification resources in WebSphere Application Server V7. Table 9-1 provides an overview of these commands.

Table 9-1   The wsadmin commands for administering WS-Notification resources

Task description	Command
WS-Notification services	
Create a WS-Notification service.	`createWSNService`
Delete a WS-Notification service.	`deleteWSNService`
List WS-Notification services.	`listWSNServices`
Show properties of a WS-Notification service.	`showWSNService`
WS-Notification service points	
Create a WS-Notification service point.	`createWSNServicePoint`
Delete a WS-Notification service point.	`deleteWSNServicePoint`
List WS-Notification service points.	`listWSNServicePoints`
Show properties of a WS-Notification service point.	`showWSNServicePoint`
Permanent topic namespaces	
Create a permanent topic namespace.	`createWSNTopicNamespace`
Delete a permanent WS-Notification topic namespace.	`deleteWSNTopicNamespace`
List WS-Notification topic namespaces.	`listWSNTopicNamespaces`
Show properties of a WS-Notification topic namespace.	`showWSNTopicNamespace`
Administered subscribers	
Create a WS-Notification administered subscriber.	`createWSNAdministeredSubscriber`
Delete a WS-Notification administered subscriber.	`deleteWSNAdministeredSubscriber`
List WS-Notification administered subscribers.	`listWSNAdministeredSubscribers`

Task description	Command
Show properties of WS-Notification administered subscriber.	showWSNAdministeredSubscriber
Topic namespace documents	
Add a topic namespace document to a topic namespace.	createWSNTopicDocument
Remove a topic namespace document from a topic namespace.	deleteWSNTopicDocument
List topic namespace documents.	listWSNTopicDocuments
Show the contents of the topic namespace document.	showWSNTopicDocument
Related service integration bus Web service resources	
Get a reference to an inbound service associated with a WS-Notification service.	getWSN_SIBWSInboundService
Get a reference to an inbound port associated with a WS-Notification service point.	getWSN_SIBWSInboundPort

9.3 Developing WS-Notification applications

In this section we explain how to develop WS-Notification applications based on the JAX-WS programming model. The development steps and code examples used here are based on a weather example, but they can be easily adapted to any other business domain.

The sample applications were developed using Rational Application Developer for WebSphere Software 7.5. However, Rational Application Developer is not a requirement for developing WS-Notification applications for WebSphere Application Server. You can easily develop the same applications by using the following tools that ship with WebSphere Application Server:

► Rational Application Developer Assembly and Deployment

► WebSphere Application Server command-line tools such as the Java Developer Kit (JDK) 6 binaries `javac`, `wsimport`, and `wsgen`

In 9.3.1, "Introduction to the weather applications" on page 417, we introduce the three applications from both a user's point of view and a class design point of view.

9.3.1 Introduction to the weather applications

The three sample applications are Java EE 5 Web applications that use a few HTTP servlets. In this section, we introduce each of these applications by using simple Uniform Modelling Language (UML) class diagrams.

The basic theme of the applications is that the producer application can generate a daily weather WS-Notification message to the daily-weather topic. The pull consumer can create and remove a pull point in WebSphere Application Server and use that pull point to subscribe to the daily-weather topic. The push

consumer exposes a Web service that similarly can be subscribed to the daily-weather topic.

From a user perspective, there is no difference between the two consumer applications. As you will see, there are only small differences between the classes that are involved in them.

WS-Notification resource requirements

Before you can deploy and run the three WS-Notification applications in WebSphere Application Server V7, create the WS-Notification resources listed in Table 9-2.

Table 9-2   WS-Notification names and values required by the sample

Resource	Value
Service integration bus name	weatherWSNBus
WS-Notification service name	weatherWSNService
WS-Notification service point name	weatherWSNServicePoint
Permanent topic namespace URI	http://weather
Service integration bus topic space name	weatherTS

We explain how to create these WS-Notification resources in "Creating WS-Notification resources by using the administrative console" on page 409.

Deploying the applications

The three applications are available in an ordinary Rational Application Developer project interchange file. After you import them into Rational Application Developer, the resources shown in Figure 9-8 are listed in the workspace, as viewed from the Enterprise Explorer view in the Java EE perspective.
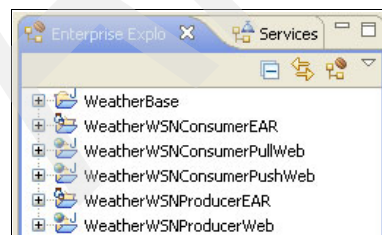
Figure 9-8   WS-Notification weather example projects

The project interchange contains the following projects:

- WeatherBase

  Contains the core weather classes used by the applications

- WeatherWSNProducerEAR

  An enterprise application project that deploys with a single Web application project, WeatherWSNProducerWeb, which is a Web module that contains the producer application code

- WeatherWSNConsumerEAR

  An enterprise application project that deploys with the following two consumer projects:

  – WeatherWSNConsumerPushWeb

    A Web module that contains the push consumer application code

  – WeatherWSNConsumerPullWeb

    A Web module that contains the pull consumer application code

You can deploy the two enterprise application projects, WeatherWSNConsumerEAR and WeatherWSNProducerEAR, to the Rational Application Developer 7.5 and WebSphere Application Server V7 test environment by using the normal Rational Application Developer deployment functionality. Alternatively, you can export the two enterprise application projects to enterprise archive (EAR) files and deploy them to WebSphere Application Server V7 by using the administrative console or `wsadmin` tools.

Using the sample applications

After the applications are deployed to WebSphere Application Server V7, you can try using them. In this section we use the applications for the following tasks:

- To subscribe a consumer application to the daily-weather topic
- To publish a WS-Notification message by using the publisher application
- To view the published WS-Notifications messages by using a consumer application

Subscribing to and from the daily-weather topic

To subscribe or unsubscribe the pull consumer to the daily-weather topic, access the SubscriptionManagerServlet by pointing your browser to the following URL:

`http://localhost:9080/WeatherWSNConsumerPushWeb/SubscriptionManagerServlet`

The servlet renders a simple HTML page that shows the action that it just performed (Figure 9-9).

Figure 9-9   SubscriptionManagerServlet

In this case, the pull consumer is subscribed to the daily-weather example. Similarly, to subscribe or unsubscribe the push consumer, point your browser to the following URL:

```
http://localhost:9080/WeatherWSNConsumerPullWeb/SubscriptionManagerServlet
```

Publishing a message to the daily-weather topic

To publish a daily weather message to the WS-Notification service in WebSphere Application Server, access the PublisherServlet by pointing your browser to the following URL:

```
http://localhost:9080/WeatherWSNProducerWeb/ProducerServlet
```

The ProducerServlet renders a simple HTML page (Figure 9-10) that shows the daily weather information that it published to the daily-weather topic.

Figure 9-10   ProducerServlet

Showing information received from the daily-weather topic

To see the WS-Notification message that was just published to the pull consumer (while the subscription is active), access the ConsumerServlet by pointing your browser to the following URL:

```
http://localhost:9080/WeatherWSNConsumerPullWeb/ConsumerServlet
```

The ConsumerServlet renders a simplistic HTML page (Figure 9-11) that shows the daily weather received so far.

Figure 9-11   ConsumerServlet

In a similar way, you can use the push consumer's ConsumerServlet by pointing your browser to the following URL:

```
http://localhost:9080/WeatherWSNConsumerPushWeb/ConsumerServlet
```

Producer application

The producer application is implemented in the WeatherWSNProducerWeb project. Figure 9-12 shows the classes that are involved in developing the producer application.
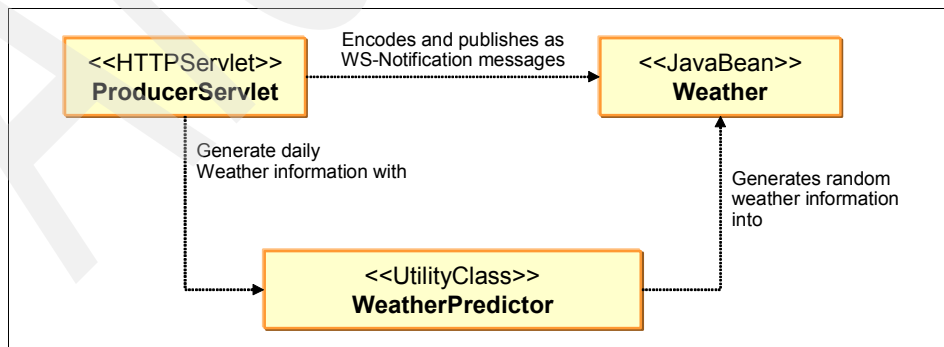
Figure 9-12   Producer application classes

The producer application is the simplest one because it contains only a single class that was developed for the ProducerServlet. The ProducerServlet is an ordinary HTTP servlet that uses the WeatherPredictor class to generate random values for a Weather object. On the contrary, the Weather object is serialized to a comma-delimited text format and published by using WS-Notification to the NotificationBroker Web service in WebSphere Application Server.

Push consumer application

The push consumer application is implemented in the WeatherWSNConsumerPushWeb project. Figure 9-13 shows the classes that are involved in developing the push consumer application.
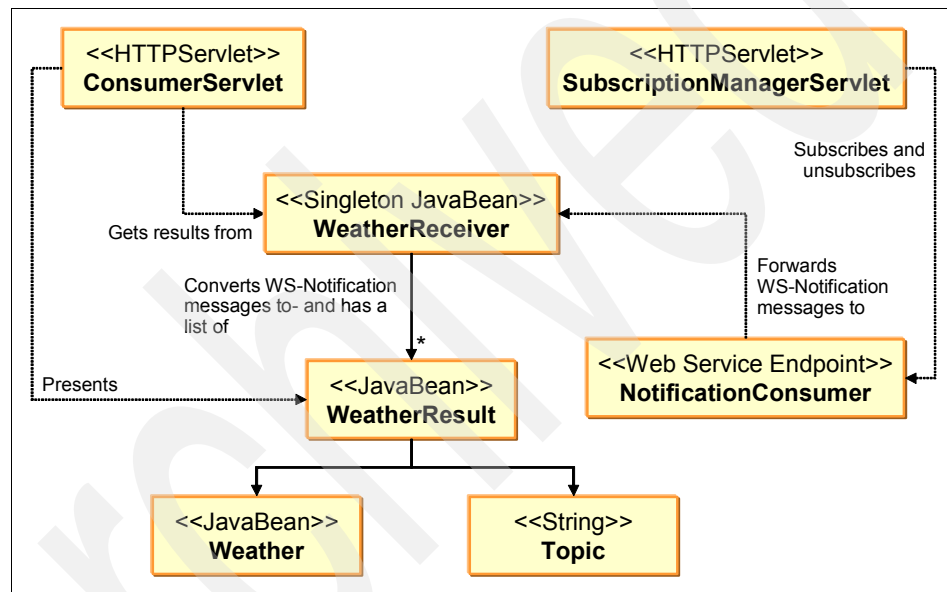
Figure 9-13   Push consumer application classes

The main class of interest in the push consumer application is the Web service endpoint implementation, NotificationConsumer. The implementation class's full name is WeatherNotificationConsumerSOAPImpl. This class receives WS-Notification messages that are sent by the WS-Notification service in WebSphere Application Server. It immediately gives the messages to the WeatherReceiver. WeatherReceiver is a singleton class that, for each WS-Notification message, decodes the comma-delimited weather information into a Weather object and stores a WeatherResult object in an internal ArrayList. Each WeatherResult object contains the decoded Weather object together with the publication topic name, which in this example is always daily-weather.

The application also comes with two HTTP servlets, which are ConsumerServlet and SubscriptionManagerServlet. The *ConsumerServlet* generates a simplistic HTML page that shows the daily weather information received so far. The *SubscriptionManagerServlet* contains functionality to switch between subscription and unsubscription of the Web service as a push consumer of the daily-weather topic. The subscribe/unsubscribe behavior of the SubscriptionManagerServlet alternates for every reload in the browser. However, regardless of the action taken, the servlet generates a simplistic HTML page that shows the action that was just performed.

Pull consumer application

The pull consumer application is implemented in the WeatherWSNConsumerPullWeb project. Figure 9-14 shows the classes that are involved in developing the pull consumer application.
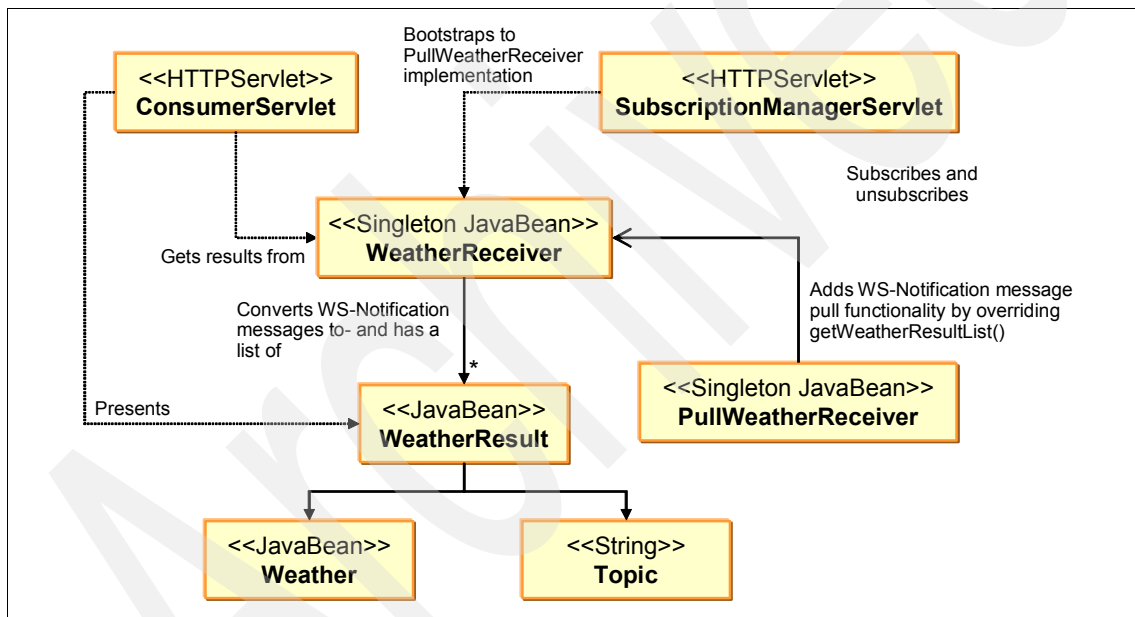
Figure 9-14   Pull consumer application classes

In the pull consumer application, you find copies of the same classes that are used in the push consumer application. These copies are of the WeatherReceiver, WeatherResult, and ConsumerServlet classes. However, contrary to the push consumer application, the pull consumer application does not include a Web service class. Instead, it includes a subclass of the WeatherReceiver class called *PullWeatherReceiver*.

The PullWeatherReceiver class is used as the WeatherReceiver singleton implementation. This is bootstrapped by a static initializer block in the pull application's SubscriptionManagerServlet). Whenever the ConsumerServlet in the pull consumer application is about to render the daily weather information, it invokes the getWeatherResultList() method on the PullWeatherReceiver class. This method contains the functionality that pulls WS-Notification messages from the WS-Notification pull point in WebSphere Application Server.

The SubscriptionManagerServlet in this application is similar to the one that is used in the push consumer application, but is not identical. The minor difference is that, prior to subscription to the daily-weather topic, the servlet first creates a pull point. Similarly, the servlet destroys the pull point when it is about to unsubscribe from the daily-weather topic.

9.3.2 Developing a producer

The simplest form of a producer application is one that publishes to a broker rather than directly to a consumer, but does not participate in demand-based publishing. In this scenario, the application is not required to expose a Web service endpoint and can be implemented as a Web service client application. For demand-based publishing or producers who will accept subscriptions directly from consumers, a Web service endpoint that implements the WS-BaseNotification-defined NotificationProducer port type must be exposed.

In this section we explain how to create a simple producer application that sends messages to the WS-Notification service (broker) in WebSphere Application Server. We explain the development techniques in the context of the daily weather example.

Developing a WS-Notification producer entails the following tasks:

1. Export the WSDL documents from WebSphere Application Server. The NotificationBroker port type is described in one of the WSDL documents that you can download from WebSphere Application Server by using the administrative console.

2. Create the application module. For the daily weather example, we implement a Java EE Web project.

3. Generate the JAX-WS Web service client code, which is used by the application code to publish messages to the WS-Notification broker service.

4. Implement the producer. In the weather example, we implemented a servlet client that posts daily weather information to the daily-weather topic.

> **Prerequisites**: The machine on which you develop the producer application
> must have access to the Internet so that the WS-Notification WSDL and
> schema documents can be resolved by the tooling.
>
> You must have configured a WS-Notification service to which you will publish
> messages. See "Creating WS-Notification resources by using the
> administrative console" on page 409.

Exporting WSDL documents from WebSphere Application Server

The WS-Notification Notify operation is used to receive messages at a
NotificationConsumer or NotificationBroker endpoint. In the application server
WS-Notification services, this operation is exposed by the notification broker
inbound service and related inbound ports. Therefore, we must obtain the WSDL
document that describes this service to develop the producer application.

To extract the NotificationBroker WSDL documents of the weather
WS-Notification service:

1. From the WebSphere administrative console, select **Service integration** →
   WS Notification → **Services.**

2. Click **weatherWSNService** to open the configuration page.

3. Click **WS-Notification service points** in the Additional Properties section.

4. Click **weatherWSNServicePoint** to open the configuration page.

5. Click **Publish WSDL files to zip** in the Additional Properties section.

6. Click the **WSN_weatherWSNService_weatherWSNServicePoint.ear
   _WSDLFiles.zip** link to download the WSDL ZIP archive to the operating
   system file system.

7. Extract the `WSN_weatherWSNService_weatherWSNServicePoint.ear/`
   `NBModule.war/WEB-INF/wsdl/NotificationBroker.wsdl` broker WSDL
   document from the ZIP archive file.

   The ZIP archive contains the WSDL documents from all three inbound
   services, which are the notification broker inbound service (`NBModule.war`),
   subscription manager inbound service (`SMModule.war`), and publisher
   registration manager inbound service (`PRMModule.war`).

8. Save the archive file because you will need it later to extract the
   SubscriptionManager WSDL document when developing the consumer
   applications.

Creating the application client module

After you download the WSDL files, create the application project. For the daily weather example, we created a dynamic Web project called *WeatherWSNProducerWeb*.

Generating the JAX-WS Web service client code

When you have the application module structure in place, you are ready to generate the Web service client code that can interact with the WS-Notification broker service in WebSphere Application Server. For the weather example, we generated the JAX-WS client classes into the WeatherWSNProducerWeb project by using the Web service client wizard in Rational Application Developer.

To generate the JAX-WS client classes by using the Web service wizard in Rational Application Developer:

1. Import the `NotificationBroker.wsdl` WSDL file into your project. For the weather example, we created the `WeatherWSNProducerWeb/WebContent/wsdlstart` folder and imported the document into that folder.

2. Right-click the **NotificationBroker.wsdl** file and select **Web services** → **Generate Client**.

3. In the wizard that opens, which is configured by default to generate the client classes into the project of the WSDL file, click **Next**.

4. On the WebSphere JAX-WS Web Service Client Configuration page:

    a. Select **Generate portable client**, which ensures that the WSDL document is copied to the client.

    b. Specify either **JAX-WS** or **JAXB binding files**, which is required for the next page.

    c. For the JAX-WS version, leave the suggested value of 2.1. Click **Next**.

5. On the Custom Binding Declarations page, click **Add** and add the `${rad.home}/runtimes/base_v7/util/ ibm-wsn-jaxb.xml` custom JAXB binding file from the WebSphere Application Server utility folder. This custom binding file ensures that you can take advantage of a few useful IBM helper classes when working with the generated client code.

> **More information**: For details see the IBM developerWorks article "WS-Notification in WebSphere Application Server V7: Part 1: Writing JAX-WS applications for WS-Notification" at the following address:
>
> `http://www.ibm.com/developerworks/websphere/techjournal/0811_partridge/0811_partridge.html`

6. Click **Finish** to generate the client code.

You have now generated the code that publishes messages to WebSphere Application Server by using the SOAP/HTTP protocol. If you look inside your client project, you see the following generated code:

▶ A package named com.ibm.websphere.wsn.notification_broker

This package contains the client-side representation of the broker Web service, which includes the following service class and endpoint interface:

– WeatherWSNServiceweatherWSNServicePointNB

This is the client-side service class that is annotated with WebServiceClient to indicate that this is a JAX-WS client. This name is specific to the daily weather example.

– NotificationBroker

This is the endpoint interface that is implemented by the dynamic proxy client and the one that we use to publish the WS-Notification messages.

▶ Several subpackages of the `org.oasis_open.docs.wsn.b_2` package

These subpackages contain miscellaneous types that are used by the NotificationBroker Web service.

Implementing the producer

With the project structure in place and generated Web service client code, you are ready to develop the Java code that publishes messages to the WS-Notification broker Web service in WebSphere Application Server. In the weather example, we created an ordinary HTTP Servlet, itso.servlet.ProducerServlet, to serve as the WS-Notification producer.

Producing a WS-Notification message is simple and only requires you to specify the actual message contents and topic to which you will publish the message. In the following steps we continue to use the weather example.

To add WS-Notification publishing functionality to a client:

1. Create an instance of the broker Web service port (service endpoint interface, or SEI). In a managed client, you can let WebSphere Application Server inject it into the servlet class as follows:

```
@WebServiceRef(WeatherWSNServiceweatherWSNServicePointNB. class)
private NotificationBroker brokerPort;
```

2. Prepare the contents of the message that you want to send and wrap it up in a SOAP with Attachments API for Java (SAAJ) 1.3 SOAPElement and store that in an OASIS type Message object.

   In the weather example, we perform the following tasks:

   a. Generate a random weather object by using the WeatherPredictor class:

   ```
   Weather weather = new Weather(Calendar.getInstance());
   WeatherPredictor.calculateWeatherValues(weather);
   ```

   b. Serialize its contents to a string:

   ```
   String weatherString = serializeWeatherObject(weather);
   ```

   c. Wrap the contents in a SOAPElement:

   ```
   SOAPFactory soapFactory = SOAPFactory.newInstance();
   SOAPElement dataElement =soapFactory.createElement("weatherobj");
   dataElement.addTextNode(weatherString);
   ```

   d. Store the SOAPElement in a Message object:

   ```
   Message message = new Message();
       message.setAny(dataElement);
   ```

3. Prepare the topic expression that identifies the topic to which the message is published. At this point, the IBM-specific helper class conveniently and easily allows you to configure the prefix mapping that you use in the expression configuration (highlighted in bold in the following expression). The following expression ensures that the daily weather message is published to the daily-weather topic in the permanent topic namespace `http://weather`:

```
TopicExpressionType topic = new TopicExpressionType();
    topic.setExpression("tns:daily-weather");
   topic.setDialect(TopicExpressionType.
DIALECT_SIMPLE_TOPIC_EXPRESSION);
       topic.addPrefixMapping("tns", "http://weather");
```

4. Assemble the message object and the topic expression object in a NotificationMessageHolderType object:

```
    NotificationMessageHolderType holder = new
NotificationMessageHolder Type();
    holder.setTopic(topic);
    holder.setMessage(message);
```

5. Publish the message by invoking the NotificationBroker.nofify() operation, which accepts a notify message that keeps a list of the holder objects:

```
    Notify notify = new Notify();
    notify.getNotificationMessage().add(holder);
    brokerPort.notify(notify);
```

Example 9-1 shows the daily weather ProducerServlet in its entirety.

Example 9-1   ProducerServlet

```
package itso.servlet;

import itso.objects.Weather;
import itso.utils.WeatherPredictor;

import java.io.IOException;
import java.io.PrintWriter;
import java.util.Calendar;

import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.xml.soap.SOAPElement;
import javax.xml.soap.SOAPException;
import javax.xml.soap.SOAPFactory;
import javax.xml.ws.WebServiceRef;

import org.oasis_open.docs.wsn.b_2.NotificationMessageHolderType;
import org.oasis_open.docs.wsn.b_2.Notify;
import
org.oasis_open.docs.wsn.b_2.NotificationMessageHolderType.Message;

import com.ibm.websphere.sib.wsn.jaxb.base.TopicExpressionType;
import com.ibm.websphere.wsn.notification_broker.NotificationBroker;
import
com.ibm.websphere.wsn.notification_broker.WeatherWSNServiceweatherWSNSe
rvicePointNB;

public class ProducerServlet extends HttpServlet {

    @WebServiceRef(WeatherWSNServiceweatherWSNServicePointNB.class)
    private NotificationBroker brokerPort;

    protected void doGet(HttpServletRequest req, HttpServletResponse
resp)
            throws IOException {

        // Create a random Weather object using WeatherPredictor
        Weather weather = new Weather(Calendar.getInstance());
        WeatherPredictor.calculateWeatherValues(weather);
```

```java
            // Encode Weather object, add it to a SOAPElement and
            // store the element in a OASIS type Message wrapper
            String weatherString = serializeWeatherObject(weather);
            SOAPElement dataElement = createSOAPElement(weatherString);
            Message message = new Message();
            message.setAny(dataElement);

            // Define topic
            TopicExpressionType topic = new TopicExpressionType();
            topic.setExpression("tns:daily-weather");

    topic.setDialect(TopicExpressionType.DIALECT_SIMPLE_TOPIC_EXPRESSION);
            topic.addPrefixMapping("tns", "http://weather");

            // Assemble topic expression + message in a holder object
            NotificationMessageHolderType holder = new
    NotificationMessageHolderType();
            holder.setTopic(topic);
            holder.setMessage(message);

            // Publish the message to the Broker Web service
            Notify notify = new Notify();
            notify.getNotificationMessage().add(holder);
            brokerPort.notify(notify);

            // Generate the HTML result page
            writeHtmlResponse(resp.getWriter(), weather);
        }

        private static SOAPElement createSOAPElement(String data) {
            try {
                SOAPFactory soapFactory = SOAPFactory.newInstance();
                SOAPElement element = soapFactory.createElement("weatherobj");
                element.addTextNode(data);
                return element;
            } catch (SOAPException e) {
                throw new RuntimeException(e);
            }
        }

        private static String serializeWeatherObject(Weather weather) {
            StringBuilder builder = new StringBuilder();
            builder.append(weather.getCondition()).append(",");
            builder.append(weather.getTemperatureCelsius()).append(",");
            builder.append(weather.getWindDirection()).append(",");
```

```
        builder.append(weather.getWindSpeed()).append(",");
        builder.append(weather.getDate().getTimeInMillis());
        return builder.toString();
    }

    private static void writeHtmlResponse(PrintWriter out, Weather
weather) {
        String title = ProducerServlet.class.getSimpleName();
        String weatherStr = weather.toString();
        out.printf("<html><head><title>%s</title></head><body>", title);
        out.printf("<h1>%s</h1>", title);
        out.printf("Published daily weather:<br>");
        out.printf("<small>%s</small>", weatherStr);
        out.printf("</body></html>");
    }
}
```

The code lines related to WS-Notification are highlighted in bold. The actual
WS-Notification business logic is placed in the servlet's doGet method. The
doGet method delegates functionality, such as weather object serialization and
so on, to a few simplistic helper methods.

9.3.3 Developing a push consumer

A push consumer is required to implement the NotificationConsumer port type
defined in the WS-BaseNotification specification and expose this as a Web
service. The WS-Notification broker service in WebSphere Application Server
then sends published messages to this Web service, provided that the messages
are destined for the appropriate topic subscription. The topic subscription can be
implemented as part of the consumer application, but is not required.

In this section we explain how to create a simple push consumer application that
receives messages from the WS-Notification service (broker) in WebSphere
Application Server. The consumer application includes functionality to subscribe
and unsubscribe from a WS-Notification topic. We explain the development
techniques in context of the daily weather example introduced earlier in this
chapter.

Developing a WS-Notification push consumer with subscribe/unsubscribe functionality entails the following tasks:

1. Export WSDL documents from WebSphere Application Server.

   We are interested in two of the inbound services, which are the notification broker service (for subscription) and the subscription manager service (for unsubscription).

2. Create the application module.

   In the daily weather example, we expose a JAX-WS 2.1-based Web service (the push consumer) in a Java EE Web module.

3. Import the WSDL documents to the application project and generate the JAX-WS Web service client code.

   You must do this task for the notification broker service and the subscription manager service. The generated code artifacts are used by the application code to subscribe and unsubscribe the push consumer.

4. Design a WSDL document for the push consumer.

   The WSDL document is simple. It imports the consumer contract from a standard WS-BaseNotification WSDL document, adding only a binding section and a service section.

5. Implement the push consumer Web service.

   In the weather example, we generated a skeleton implementation from the WSDL document by using the Rational Application Developer Web service wizard. Then we added the business logic to implement the notify operation.

6. Implement the subscription management code.

   In the weather example, we implement a servlet that subscribes the Web service to and unsubscribes it from the daily-weather topic, which is the same topic to which the publisher application posts messages.

Exporting WSDL documents from WebSphere Application Server

The instructions for this task are the same as those in "Exporting WSDL documents from WebSphere Application Server" on page 425. In addition to the `NotificationBroker.wsdl` file, ensure that you also extract the `SubscriptionManager.wsdl` file, which is in the `WSN_weatherWSNService_weatherWSNServicePoint.ear/SMModule.war/WEB-INF/wsdl/` directory of the WSDL bundle.

Creating the application module

To write a WS-Notification push consumer application, you must expose a Web service to which WebSphere Application Server sends notification messages.

For the weather example, we created a dynamic Web project called WeatherWSNConsumerPushWeb.

Generating the JAX-WS Web service client code

The Java code that subscribes and unsubscribes the push consumer endpoint uses two Web service endpoints from the WS-Notification service:

▶ NotificationBroker

We use the subscribe operation on this port type to subscribe the push consumer for the topic.

▶ SubscriptionManager

We use the unsubscribe operation on this port type to unsubscribe the push consumer from the topic.

To send messages to these Web services, generate the JAX-WS Web service client classes for both of the endpoints. Follow the procedure in "Generating the JAX-WS Web service client code" on page 426. However, this time, run it twice, once for the `NotificationBroker.wsdl` file and once for the `SubscriptionManager.wsdl` file.

In the weather example, we copied the two WSDL documents, `NotificationBroker.wsdl` and `SubscriptionManager.wsdl`, to the `WebContent/wsdlstart` folder of the WeatherWSNConsumerPushWeb module. We then executed the Rational Application Developer Web service client wizard twice to generate Web service client stubs into the module.

Designing a WSDL document for the push consumer

The Web service that you expose (the push consumer) must adhere to the interface that is defined by the WS-BaseNotification specification. By defining your Web service to this interface, you enable the possibility for WebSphere Application Server to send WS-Notification messages to it.

To design a consumer WSDL document:

1. Create a new WSDL document.

2. Add an import declaration that imports the interface definition from a WSDL document that is available online at the following address:

   http://docs.oasis-open.org

3. Add a SOAP/HTTP binding (document/literal style) for the NotificationConsumer port type.

4. Add a service section to define the endpoint location.

You can write the WSDL file manually or by using the Web service tools in Rational Application Developer. For the weather example, we used Rational Application Developer.

To create a consumer WSDL file by using Rational Application Developer:

1. Create a destination folder for the WSDL file. In the weather example, we used the `WebContent/wsdlstart` folder in the WeatherWSNConsumerPushWeb project.

2. Right-click the destination folder and select **New → Other**. Expand **Web services** and select **WSDL**. Click **Next**.

3. Enter a file name, for example, `WeatherNotificationConsumer.wsdl`. Click **Next**.

4. In the next panel:

   a. Enter the target namespace, for example:

      `http://weather.itso/WeatherNotificationConsumer/`

   b. You can change the prefix value, but the common convention is to use the default of `tns`.

   c. Select **Create WSDL Skeleton**.

   d. For protocol, select **SOAP**.

   e. For SOAP binding, select **document literal**.

   f. Click **Finish**.

5. Ensure that the new WSDL document is opened by using the WSDL Editor in Rational Application Developer (Figure 9-15). The elements of the WSDL file are denoted in italics.
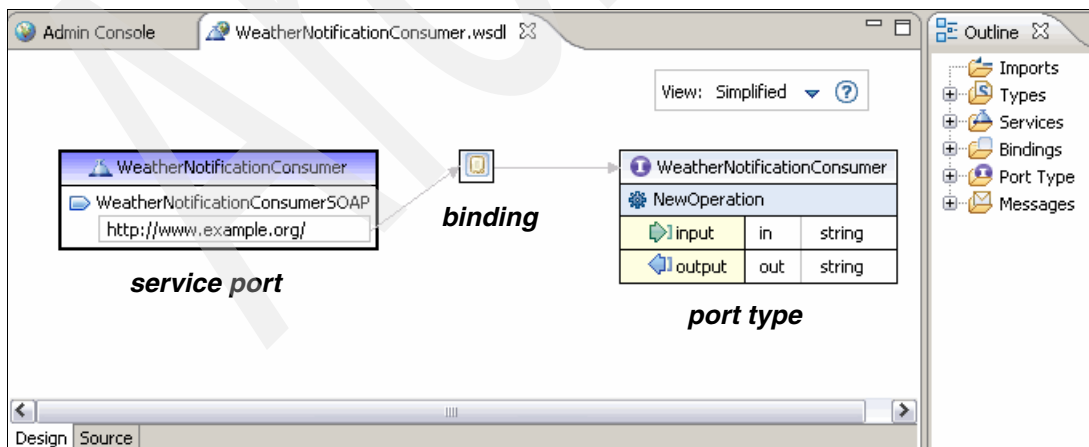
Figure 9-15   New WSDL file

6. In the outline view associated with the new WSDL document, right-click the **Imports** area and select **Add Import**.

7. In the WSDL editor, select the **Source** tab. Edit the import statement in the WSDL source:

   ```
   <wsdl:import namespace="http://docs.oasis-open.org/wsn/bw-2"

   location="http://docs.oasis-open.org/wsn/bw-2.wsdl"></wsdl:import>
   ```

8. In the outline view, right-click the import to open the **Properties** view. For the new import, type a prefix of bw2.

9. Return to the **Design** tab of the WSDL Editor. All the port types defined by WS-BaseNotification are now listed.

10. The next step is to update the references between the service port, binding, and port type. Update the properties of each of these as follows:

   a. Select the generated binding (see Figure 9-15 on page 434), right-click, and select **Set Port Type** → **Existing Port Type**. Select **NotificationConsumer**. Click **OK**.

   b. Select the binding again. In the Properties view click **Generate Binding Content**.

   c. In the Specify Binding Details panel, select **Overwrite existing binding information**.

   d. Specify SOAP for the protocol.

   e. Specify document literal for the SOAP binding option.

   f. Click **Finish**.

11. Use the Outline view to remove the local port type, WeatherNotificationConsumer (Figure 9-16). A default port type and messages are defined if you created a skeleton WSDL.
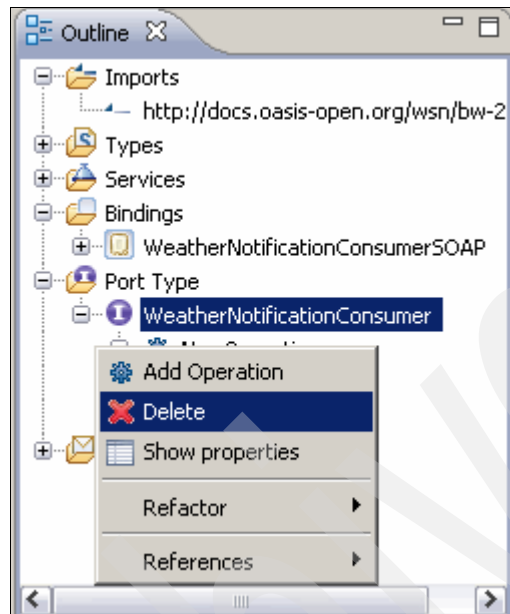
Figure 9-16   Remove the local port type

12. Use the Outline view to remove any type elements.

13. Edit select the port and use the Properties view to set the endpoint address location to match your application and port.
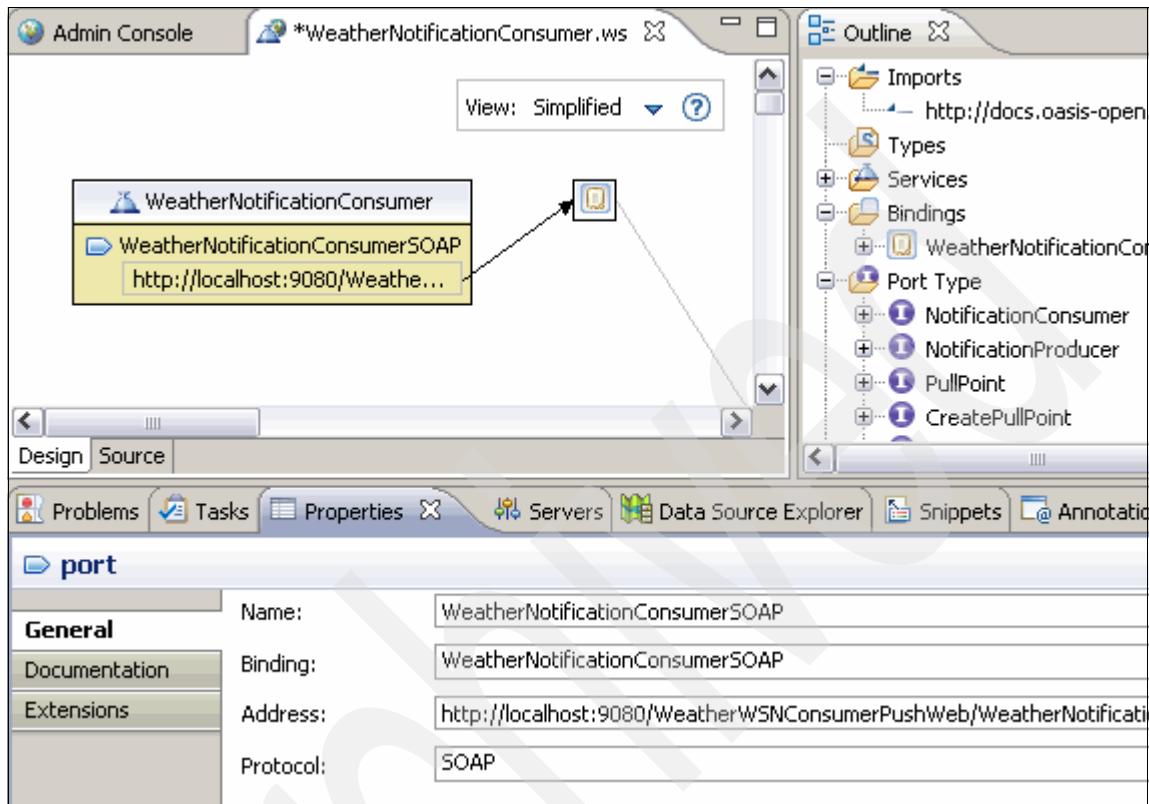
Figure 9-17   Update the port address

The WSDL document for the NotificationProducer port type is now finished. Example 9-2 shows the WSDL source for the weather push consumer.

Example 9-2   WSDL file for the push consumer

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions name="WeatherNotificationConsumer"
    targetNamespace="http://weather.itso/WeatherNotificationConsumer/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:tns="http://weather.itso/WeatherNotificationConsumer/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:bw2="http://docs.oasis-open.org/wsn/bw-2">

    <wsdl:import namespace="http://docs.oasis-open.org/wsn/bw-2"
        location="http://docs.oasis-open.org/wsn/bw-2.wsdl"></wsdl:import>
```

```
        <wsdl:binding name="WeatherNotificationConsumerSOAP"
type="bw2:NotificationConsumer">

            <soap:binding style="document"
                transport="http://schemas.xmlsoap.org/soap/http" />

            <wsdl:operation name="Notify">
                <soap:operation

soapAction="http://weather.itso/WeatherNotificationConsumer/Notify" />
                <wsdl:input>
                    <soap:body use="literal" />
                </wsdl:input>
            </wsdl:operation>

        </wsdl:binding>

        <wsdl:service name="WeatherNotificationConsumer">

            <wsdl:port binding="tns:WeatherNotificationConsumerSOAP"
                name="WeatherNotificationConsumerSOAP">
                <soap:address

location="http://localhost:9080/WeatherWSNConsumerPushWeb/WeatherNotificationCo
nsumerSOAP" />
            </wsdl:port>

        </wsdl:service>

</wsdl:definitions>
```

The WSDL import is highlighted in bold. As you can see, the resulting WSDL file
is simple. It merely specifies binding details and the endpoint where the binding
is exposed.

Implementing the push consumer Web service

With the push consumer Web service's WSDL document in place, you only need
to generate the Web service implementation bean and put in the business logic
that handles incoming WS-Notification messages.

Again you use Web service tools to generate the Web service implementation
classes. You can either use the command-line `wsimport` tool or the top-down
Web service wizard in Rational Application Developer to generate the Web
service. For the weather example, we used Rational Application Developer.

To generate a Web service using Rational Application Developer:

1. Right-click the WSDL document that you designed in "Designing a WSDL document for the push consumer" on page 433 (`WeatherNotificationConsumer.wsdl` in the weather example) and select **Web services → Generate Java bean skeleton**.

2. Select **Top down Java bean Web service** and **Deploy** on the slider widget. Ensure that the project configuration targets the Web module that should host the Web service (WeatherWSNConsumerPushWeb in the weather example). Click **Next**.

3. On the WebSphere JAX-WS Top Down Web service Configuration page, select **Customize JAX-WS or JAXB binding files**. Click **Next**.

4. On the Custom Binding Declarations page, click **Add** and add the `${rad.home}/runtimes/base_v7/util/ ibm-wsn-jaxb.xml` JAXB binding file. Adding this file enables IBM-specific helper classes in the same manner as for the Web service client generation.

5. Click **Finish** to generate the Web service code.

When the Wizard is done, the destination project contains a package with the service endpoint interface and the endpoint implementation bean. In the weather example, the following interface and bean are generated:

► itso.weather.weathernotificationconsumer.NotificationConsumer

   This is the service endpoint interface. It defines one method, which is the notify method.

► itso.weather.weathernotificationconsumer.WeatherNotificationConsumerSOAPImpl

   This is the endpoint implementation bean that implements the notify method.

Finally, implement the notify method in the Web service implementation. To process WS-Notification messages received by the push consumer's notify method:

1. Extract all the pushed messages from the Web service method input parameter (a notification object):

```
List<NotificationMessageHolderType> messages =
notify.getNotificationMessage();
```

2. Process each message:

```
for (NotificationMessageHolderType message : messages) {
// Determine the topic to which the message was published:
String topic = message.getTopic().getStringExpression();

// Get the actual message contents:
Element messageElement = (Element) message.getMessage().getAny();
```

```
                String messageContent = messageElement.getTextContent();
    }
```

The messageElement is a DOM Element, which means that, if you receive an XML document, you have all the Java XML processing facilities at your disposal.

Example 9-3 shows the endpoint implementation in the weather example.

Example 9-3   WeatherNotificationConsumerSOAPImpl

```
package itso.weather.weathernotificationconsumer;

import itso.receiver.WeatherReceiver;

import java.util.List;

import org.oasis_open.docs.wsn.b_2.NotificationMessageHolderType;
import org.oasis_open.docs.wsn.b_2.Notify;

@javax.jws.WebService(endpointInterface =
"itso.weather.weathernotificationconsumer.NotificationConsumer",
targetNamespace = "http://weather.itso/WeatherNotificationConsumer/",
serviceName = "WeatherNotificationConsumer", portName =
"WeatherNotificationConsumerSOAP")
public class WeatherNotificationConsumerSOAPImpl {

    public void notify(Notify notify) {

        // Get a reference to the published messages
        List<NotificationMessageHolderType> messages = notify
                .getNotificationMessage();

        // Let WeatherReceiver convert messages to Weather objects
        WeatherReceiver.getInstance().addWeatherObjects(messages);
    }
}
```

This implementation extracts the messages in the form of NotificationMessageHolderType objects and lets the WeatherReceiver singleton do the work that extracts the daily weather information and transforms that into weather objects.

Example 9-4 shows the WeatherReceiver class in its entirety.

Example 9-4   WeatherReceiver class

```
package itso.receiver;

import itso.objects.Weather;

import java.util.ArrayList;
import java.util.Calendar;
import java.util.List;
import java.util.Scanner;

import org.oasis_open.docs.wsn.b_2.NotificationMessageHolderType;
import org.w3c.dom.Element;

public class WeatherReceiver {

    private static WeatherReceiver instance;

    private List<WeatherResult> weatherList = new
ArrayList<WeatherResult>();

    public List<WeatherResult> getWeatherResultList() {
        return weatherList;
    }

    public void addWeatherObjects(List<NotificationMessageHolderType>
messages) {

        for (NotificationMessageHolderType message : messages) {

            // Extract topic
            String topic = message.getTopic().getStringExpression();

            // Extract message content
            Element messageElement = (Element)
message.getMessage().getAny();
            String serializedWeather = messageElement.getTextContent();

            // Reconstruct Weather object
            Scanner scanner = new Scanner(serializedWeather);
            scanner.useDelimiter(",");

            Weather weather = new Weather();
```

```
            weather.setCondition(scanner.next());
            weather.setTemperatureCelsius(scanner.nextInt());
            weather.setWindDirection(scanner.next());
            weather.setWindSpeed(scanner.nextInt());
            Calendar cal = Calendar.getInstance();
            cal.setTimeInMillis(scanner.nextLong());
            weather.setDate(cal);
            weatherList.add(0, new WeatherResult(topic, weather));
        }
    }

    public static WeatherReceiver getInstance() {
        if (instance == null) {
            instance = new WeatherReceiver();
        }
        return instance;
    }

    public static void setInstance(WeatherReceiver instance) {
        WeatherReceiver.instance = instance;
    }

}
```

The lines highlighted in bold of the addWeatherObjects method show how to
extract the topic and message contents from the WS-Notification-related APIs.
For each message received by the Web service, the addWeatherObjects method
stores a WeatherResult object inside a list. The WeatherResult object
(Example 9-5) is a simple bean that correlates a Weather object (generated from
parsing the comma-delimited message content text string) with the topic to which
it was published.

Example 9-5   WeatherResult object

```
package itso.receiver;

import itso.objects.Weather;

public class WeatherResult {
    public final String topic;
    public final Weather weather;

    WeatherResult(String topic, Weather weather) {
        this.topic = topic;
        this.weather = weather;
```

```
    }
}
```

The list of WeatherResult objects is obtained by the ConsumerServlet
(Example 9-6) to generate an HTML page that shows the received weather.

Example 9-6   ConsumerServlet

```
package itso.servlet;

import itso.receiver.WeatherReceiver;
import itso.receiver.WeatherResult;

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

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class ConsumerServlet extends HttpServlet {

    protected void doGet(HttpServletRequest req, HttpServletResponse
resp)
            throws ServletException, IOException {

        // Get the Weather notifications from the WeatherReceiver
        WeatherReceiver receiver = WeatherReceiver.getInstance();
        List<WeatherResult> weatherList =
receiver.getWeatherResultList();

        // Build a textual representation of the list
        StringBuilder weatherStr = new StringBuilder();
        if (weatherList.isEmpty()) {
            weatherStr.append("No publications yet");
        } else {
            for (WeatherResult weather : weatherList) {
                weatherStr.append("[Topic=").append(weather.topic)
                        .append("]: ");
                weatherStr.append(weather.weather).append("<br>");
            }
        }
```

```
        // Generate HTML document
        String title = ConsumerServlet.class.getSimpleName();
        PrintWriter out = resp.getWriter();
        out.printf("<html><head><title>%s</title></head><body>", title);
        out.printf("<h1>%s</h1>", title);
        out.printf("Weather publications: <br>");
        out.printf("<small>%s</small>", weatherStr.toString());
        out.printf("</body></html>");
    }

}
```

The line highlighted in bold shows how the servlet obtains a list of the published WeatherResult objects.

Implementing the subscription management code

Before a consumer receives any WS-Notification messages from the broker service in WebSphere Application Server, it must be subscribed to a topic to which messages are sent. In the weather example, we implement a servlet component that contains the functionality to subscribe to and unsubscribe from the daily-weather topic.

To programmatically subscribe a push consumer by using the weather example:

1. Build an endpoint reference object that references the push consumer endpoint URL:

    ```
    W3CEndpointReference endpointReference = new
    W3CEndpointReferenceBuilder().address(PUSH_ENDPOINT_URL).build();
    ```

2. Specify to which topic the endpoint should subscribe:

    ```
    TopicExpressionType topic = new TopicExpressionType();
    topic.setExpression("tns:daily-weather");
    topic.setDialect(TopicExpressionType.DIALECT_SIMPLE_TOPIC_EXPRESSION
    );
    topic.addPrefixMapping("tns", "http://weather");
    FilterType filter = new FilterType();
    filter.addTopicExpression(topic);
    ```

    The tns:daily-weather expression indicates to the broker that we are interested in the daily-weather topic in the tns namespace. The expression dialect specifies that the topic subscription expression is simple. See 9.1.3, "WS-Topics" on page 401, for details about topic subscription expressions.

    The prefix mapping tells the broker that the daily-weather topic specified in the expression is part of the http://weather permanent topic namespace.

The FilterType object accumulates the topics in which we are interested. In this case there is only one.

3. Optional: Specify a subscription termination time. The following code ensures that the subscription is terminated one hour after subscription:

```
        // Create an XMLGregorianCalendar representing 1 hour from
now    DatatypeFactory factory = DatatypeFactory.newInstance();
XMLGregorianCalendar calendar = factory.newXMLGregorianCalendar();
calendar.add(factory.newDuration("1H"));

        // Create a JAXBElement to hold the termination time
        ObjectFactory oFactory = new ObjectFactory();
        JAXBElement<String> terminationTime =
oFactory.createSubscribeInitialTerminationTime(calendar.toString());
```

Note that the weather example does not specify a subscription termination time.

4. Prepare a subscription request input wrapper object with the topic subscription criterion and the consumer endpoint location:

```
        Subscribe subscribeRequest = new Subscribe();
        subscribeRequest.setConsumerReference(endpointReference);
        subscribeRequest.setFilter(filter);
subscribeRequest.setInitialTerminationTime(terminationTime);
```

5. Create a NotificationBroker dynamic proxy with the WS-Addressing feature enabled:

```
        NotificationBroker broker =
brokerService.getNotificationBrokerPort(
new AddressingFeature());
```

6. Send the subscription request to the NotificationBroker Web service and store the subscription reference available in the response:

```
        SubscribeResponse response =
broker.subscribe(subscribeRequest);
        subscriptionReference = response.getSubscriptionReference();
```

To unsubscribe the consumer:

1. Use the subscriptionReference (returned by the subscription response) to produce a SubscriptionManager dynamic proxy with the WS-Addressing feature enabled:

```
        SubscriptionManager subscriptionManagerPort =
subscriptionReference.getPort(SubscriptionManager.class, new
AddressingFeature());
```

2. Unsubscribe from the topic as follows by sending an empty unsubscribe message to the SubscriptionManager Web service. Because of the WS-Addressing feature enablement, the subscriptionReference already contains information about the endpoint consumer's location.

```
subscriptionManagerPort.unsubscribe(new Unsubscribe());
```

Example 9-7 shows the SubscriptionServlet that is used in the weather example to subscribe and unsubscribe the push consumer.

Example 9-7 SubscriptionServlet

```
package itso.servlet;

import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.xml.ws.WebServiceRef;
import javax.xml.ws.soap.AddressingFeature;
import javax.xml.ws.wsaddressing.W3CEndpointReference;
import javax.xml.ws.wsaddressing.W3CEndpointReferenceBuilder;

import org.oasis_open.docs.wsn.b_2.Subscribe;
import org.oasis_open.docs.wsn.b_2.SubscribeResponse;
import org.oasis_open.docs.wsn.b_2.Unsubscribe;

import com.ibm.websphere.sib.wsn.jaxb.base.FilterType;
import com.ibm.websphere.sib.wsn.jaxb.base.TopicExpressionType;
import com.ibm.websphere.wsn.notification_broker.NotificationBroker;
import
com.ibm.websphere.wsn.notification_broker.WeatherWSNServiceweatherWSNSe
rvicePointNB;
import com.ibm.websphere.wsn.subscription_manager.SubscriptionManager;

public class SubscriptionManagerServlet extends HttpServlet {

    private static final String PUSH_ENDPOINT_URL =
"http://localhost:9080/WeatherWSNConsumerPushWeb/WeatherNotificationCon
sumer";

    @WebServiceRef()
```

```
    private static WeatherWSNServiceweatherWSNServicePointNB
brokerService;

    private static W3CEndpointReference subscriptionReference;

    protected void doGet(HttpServletRequest req, HttpServletResponse
resp)
            throws ServletException, IOException {

        boolean isSubscribed = (subscriptionReference != null);

        if (isSubscribed) {
            unsubscribe();
        } else {
            subscribe();
        }

        String title = SubscriptionManagerServlet.class.getSimpleName();
        PrintWriter out = resp.getWriter();
        out.printf("<html><head><title>%s</title></head><body>", title);
        out.printf("<h1>%s</h1>", title);
        String message = (isSubscribed) ? "unsubscribed from" :
"subscribed to";
        out.printf("The consumer %s topic <i>daily-weather</i>",
message);
        out.printf("</body></html>");
    }

    private static void unsubscribe() {

        // Get a SubscriptionManager from the subscription reference
        SubscriptionManager subscriptionManagerPort =
subscriptionReference
                .getPort(SubscriptionManager.class, new
AddressingFeature());

        // Unsubscribe by sending an empty Unsubscribe message
        try {
            subscriptionManagerPort.unsubscribe(new Unsubscribe());
        } catch (Exception e) {
            throw new RuntimeException(e);
        }

        // Enable subscription functionality again
        subscriptionReference = null;
```

```
    }

    private static void subscribe() {

        // Configure the receiver consumer endpoint
        W3CEndpointReference endpointReference = new
W3CEndpointReferenceBuilder()
            .address(PUSH_ENDPOINT_URL).build();

        // Subscribe to the random-weather topic
        TopicExpressionType topic = new TopicExpressionType();
        topic.setExpression("tns:daily-weather");

topic.setDialect(TopicExpressionType.DIALECT_SIMPLE_TOPIC_EXPRESSION);
        topic.addPrefixMapping("tns", "http://weather");
        FilterType filter = new FilterType();
        filter.addTopicExpression(topic);

        // Create subscription information input message
        Subscribe subscribeRequest = new Subscribe();
        subscribeRequest.setConsumerReference(endpointReference);
        subscribeRequest.setFilter(filter);

        // Send subscription request
        NotificationBroker broker = brokerService
            .getNotificationBrokerPort(new AddressingFeature());
        try {
            SubscribeResponse response =
broker.subscribe(subscribeRequest);
            subscriptionReference = response.getSubscriptionReference();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

By using the subscriptionReference variable, the doGet method checks whether
the consumer is already subscribed. If it is not, then it executes the subscribe
method, which subscribes the Web service endpoint as a WS-Notification
consumer to the daily-weather topic.

Upon successful subscription, the subscribe method stores a
subscriptionReference. This way, the next time that the doGet method is
executed, the servlet effectively calls the unsubscribe method instead.

The unsubscribe method similarly uses the techniques to unsubscribe the Web service again. After the doGet method has either subscribed or unsubscribed the push consumer, it renders a simple HTML page indicating which action it took.

9.3.4 Developing a pull consumer

A pull consumer is not required to implement any Web service endpoints. Instead, it uses the WS-Notification broker service to create a pull point and then configures a topic subscription for that pull point. The WS-Notification broker service in WebSphere Application Server then sends published messages to the pull point, provided that the messages are destined for the appropriate topic subscription. At its own frequency, the pull consumer application can then choose to connect to the pull point and fetch the published messages. As is the case with the push consumer, the topic subscription can optionally be implemented as part of the consumer application.

In this section we explain how to create a simple pull consumer application that receives messages from a pull point in WebSphere Application Server. The consumer application includes functionality to create and destroy the pull point as well as functionality to subscribe and unsubscribe from a WS-Notification topic. We explain the development techniques in the context of the daily weather example.

Developing a WS-Notification pull consumer with subscribe/unsubscribe functionality entails the following tasks:

1. Export WSDL documents from WebSphere Application Server.

   We are interested in two of the inbound services, which are the notification broker service (for subscription) and the subscription manager service (for unsubscription).

2. Create the application module.

   In the daily weather example, we developed the pull consumer in a Java EE Web module.

3. Import the WSDL documents to the application project and generate the JAX-WS Web service client code.

   We must generate clients for the notification broker service and the subscription manager service. The generated code artifacts are used by the application code to create or remove the pullpoint and subscriber unsubscribe the pull consumer.

4. Implement the pull consumer.

   Contrary to the push consumer example, this does not need to be in a Web service. In the weather example, the pull functionality is implemented in a simple JavaBean class.

5. Implement the subscription management code.

   In the weather example, we implement a servlet that subscribes the pull point to, and unsubscribes it from, the daily-weather topic (the same topic to which the publisher application posts messages).

Exporting WSDL documents from WebSphere Application Server

You perform the same step when developing the push consumer application. See "Exporting WSDL documents from WebSphere Application Server" on page 425.

Creating the application module

For the weather example, we used Rational Application Developer for WebSphere Software 7.5 to create a dynamic Web project called WeatherWSNConsumerPullWeb.

Generating the JAX-WS Web service client code

This step is identical to the one performed in developing the push consumer application. See "Generating the JAX-WS Web service client code" on page 426.

Implementing the pull consumer

Where a push consumer implementation (using JAX-WS) is a Web service implementation bean, the pull consumer is an ordinary Java application component. You can implement it as a standalone application, an EJB component, a servlet component, or another component that you choose.

To fetch messages from the pull point:

1. Use the pull point reference to obtain a NotificationBroker Web service dynamic proxy that, by using WS-Addressing headers, contains information about the pull point location:

```
NotificationBroker broker =
pullPointReference.getPort(NotificationBroker.class,
          new AddressingFeature());
```

2. Pull all messages from the endpoint:

```
                    GetMessages request = new GetMessages();
             GetMessagesResponse response =
broker.getMessages(request);
             List<NotificationMessageHolderType> messages =
                response.getNotificationMessage();
```

Instead of pulling all messages at once, you can specify the maximum number of messages to pull by using the setMaximumNumber method on the GetMessages request object.

In the push example, we implemented a ConsumerServlet that uses the WeatherReceiver class to obtain the published weather result messages. Both of these classes are unchanged in the pull consumer example. (They have been copied, unmodifed, to the WeatherWSNConsumerPullWeb Web module.) However, in the pull consumer, we added the PullWeatherReceiver class, which extends the WeatherReceiver such that, when the ConsumerServlet invokes the getWeatherResultList method, it synchronously pulls messages from the WS-Notification pull point.

Example 9-8 shows the PullWeatherReceiver class in its entirety.

Example 9-8   PullWeatherReceiver class

```
package itso.receiver;

import java.util.List;

import org.oasis_open.docs.wsn.b_2.GetMessages;
import org.oasis_open.docs.wsn.b_2.GetMessagesResponse;
import org.oasis_open.docs.wsn.b_2.NotificationMessageHolderType;

import com.ibm.websphere.wsn.notification_broker.NotificationBroker;

public class PullWeatherReceiver extends WeatherReceiver {

    public NotificationBroker broker;

    @Override
    public List<WeatherResult> getWeatherResultList() {
        if (broker != null) {

            // Fetch published messages
            List<NotificationMessageHolderType> messages = null;
            try {
                GetMessages request = new GetMessages();
```

```
                GetMessagesResponse response = broker.getMessages(request);
                messages = response.getNotificationMessage();
            } catch (Exception e) {
                throw new RuntimeException(e);
            }

            // Let WeatherReceiver convert messages to Weather objects
            super.addWeatherObjects(messages);
        }

        return super.getWeatherResultList();
    }
}
```

The code related to WS-Notification message pull functionality is highlighted in
bold. The NotificationBroker variable is injected by the SubscriptionManager
servlet after successful creation of the pull point and topic subscription. It is set to
null when the pull point is removed and the topic is unsubscribed from. The
getWeatherResultList method ensures that messages are synchronously pulled
and added to the super class list prior to the list being returned to the
ConsumerServlet.

Implementing the subscription management code

Before the pull consumer can receive any WS-Notification messages from the
broker service in WebSphere Application Server, you must perform the following
tasks:

1. Create a pull point to which messages will be sent.

   The pull point is the place where the pull consumer fetches its messages.

2. Create a subscription for the pull point.

   Without a subscription to a topic, the pull consumer's pull point will not receive
   any messages.

To programmatically create a pull point and subscribe the pull consumer to the
daily-weather topic by using the weather example:

1. Obtain a JAX-WS Service object. In the weather example, we use a servlet
   component so that WebSphere Application Server can inject it for us:

   ```
   @WebServiceRef()
   private static WeatherWSNServiceweatherWSNServicePointNB
   brokerService;
   ```

2. Create a NotificationBroker with WS-Addressing support:

```
        NotificationBroker broker =
brokerService.getNotificationBrokerPort(new AddressingFeature());
```

3. Create a pull point in WebSphere Application Server by using the NotificationBroker dynamic proxy:

```
CreatePullPoint request = new CreatePullPoint();
CreatePullPointResponse response = broker.createPullPoint(request);
W3CEndpointReference pullPointReference = response.getPullPoint();
```

4. Specify to which topic the endpoint should subscribe (identical to the push consumer's approach):

```
        TopicExpressionType topic = new TopicExpressionType();
        topic.setExpression("tns:daily-weather");
topic.setDialect(TopicExpressionType.DIALECT_SIMPLE_TOPIC_EXPRESSION
);
        topic.addPrefixMapping("tns", "http://weather");
        FilterType filter = new FilterType();
        filter.addTopicExpression(topic);
```

5. Optional: Specify a subscription termination time. The following code ensures that the subscription is terminated one hour after subscription:

```
        // Create an XMLGregorianCalendar representing 1 hour from
now     DatatypeFactory factory = DatatypeFactory.newInstance();
XMLGregorianCalendar calendar = factory.newXMLGregorianCalendar();
calendar.add(factory.newDuration("1H"));

        // Create a JAXBElement to hold the termination time
        ObjectFactory oFactory = new ObjectFactory();
        JAXBElement<String> terminationTime =
oFactory.createSubscribeInitialTerminationTime(calendar.toString());
```

> **Subscription termination in the weather example**: The weather example does not specify a subscription termination time.

6. Prepare a subscription request input wrapper object with the topic subscription criterion and the pull point consumer endpoint location:

```
        Subscribe subscribeRequest = new Subscribe();
        subscribeRequest.setConsumerReference(pullPointReference);
        subscribeRequest.setFilter(filter);
subscribeRequest.setInitialTerminationTime(terminationTime);
```

7. Send the subscription request to the NotificationBroker Web service and store the subscription reference available in the response:

```
    SubscribeResponse response =
broker.subscribe(subscribeRequest);
    subscriptionReference = response.getSubscriptionReference();
```

You have now ensured that messages targeted for the topic are sent to the pull point that you created.

8. Use the pullPointReference to create a new NotificationBroker dynamic proxy that you can use to pull messages from the pull point that you created.

```
NotificationBroker newBroker = pullPointReference.getPort(
NotificationBroker.class, new AddressingFeature());
```

This new dynamic proxy knows, by means of WS-Addressing, from which pull point messages can be pulled. In the weather example, we stored this broker reference inside the PullWeatherReceiver so that the getWeatherResultList method can pull messages from it.

To unsubscribe the consumer and destroy the pull point:

1. Use the subscriptionReference (returned by the subscription response) to produce a SubscriptionManager dynamic proxy with the WS-Addressing feature enabled:

```
    SubscriptionManager subscriptionManagerPort =
subscriptionReference.getPort(SubscriptionManager.class, new
AddressingFeature());
```

2. Unsubscribe from the topic by sending an empty Unsubscribe message to the SubscriptionManager Web service as follows. Because the WS-Addressing feature is enabled, the subscriptionReference already contains information about the endpoint consumer's location:

```
subscriptionManagerPort.unsubscribe(new Unsubscribe());
```

3. To remove the pull point from WebSphere Application Server, invoke the destroyPullPoint method on the NotificationBroker:

```
broker.destroyPullPoint(new DestroyPullPoint());
```

Example 9-9 shows the SubscriptionManager servlet from the WeatherWSNConsumerPullWeb project.

Example 9-9   SubscriptionManager servlet (pull consumer)

```
package itso.servlet;

import itso.receiver.PullWeatherReceiver;
import itso.receiver.WeatherReceiver;
```

```
import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.xml.ws.WebServiceRef;
import javax.xml.ws.soap.AddressingFeature;
import javax.xml.ws.wsaddressing.W3CEndpointReference;

import org.oasis_open.docs.wsn.b_2.CreatePullPoint;
import org.oasis_open.docs.wsn.b_2.CreatePullPointResponse;
import org.oasis_open.docs.wsn.b_2.DestroyPullPoint;
import org.oasis_open.docs.wsn.b_2.Subscribe;
import org.oasis_open.docs.wsn.b_2.SubscribeResponse;
import org.oasis_open.docs.wsn.b_2.Unsubscribe;

import com.ibm.websphere.sib.wsn.jaxb.base.FilterType;
import com.ibm.websphere.sib.wsn.jaxb.base.TopicExpressionType;
import com.ibm.websphere.wsn.notification_broker.NotificationBroker;
import
com.ibm.websphere.wsn.notification_broker.WeatherWSNServiceweatherWSNSe
rvicePointNB;
import com.ibm.websphere.wsn.subscription_manager.SubscriptionManager;

public class SubscriptionManagerServlet extends HttpServlet {

    @WebServiceRef()
    private static WeatherWSNServiceweatherWSNServicePointNB
brokerService;

    private static W3CEndpointReference subscriptionReference;

    protected void doGet(HttpServletRequest req, HttpServletResponse
resp)
            throws ServletException, IOException {

        boolean isSubscribed = (subscriptionReference != null);

        if (isSubscribed) {
            unsubscribe();
        } else {
            subscribe();
```

```
        }

        String title = SubscriptionManagerServlet.class.getSimpleName();
        PrintWriter out = resp.getWriter();
        out.printf("<html><head><title>%s</title></head><body>", title);
        out.printf("<h1>%s</h1>", title);
        String message = (isSubscribed) ? "unsubscribed from" :
"subscribed to";
        out.printf("The consumer %s topic <i>daily-weather</i>",
message);
        out.printf("</body></html>");
    }

    private static void unsubscribe() {

        // Get a SubscriptionManager from the subscription reference
        SubscriptionManager subscriptionManagerPort =
subscriptionReference
            .getPort(SubscriptionManager.class, new
AddressingFeature());

        // Unsubscribe by sending an empty Unsubscribe message
        try {
            subscriptionManagerPort.unsubscribe(new Unsubscribe());
        } catch (Exception e) {
            throw new RuntimeException(e);
        }

        // Destroy the pull point and broker reference
        try {
            WeatherReceiver receiver = WeatherReceiver.getInstance();
            PullWeatherReceiver pullReceiver = (PullWeatherReceiver)
receiver;

            pullReceiver.broker.destroyPullPoint(new DestroyPullPoint());
            pullReceiver.broker = null;
        } catch (Exception e) {
            throw new RuntimeException(e);
        }

        // Enable subscription functionality again
        subscriptionReference = null;
    }

    private static void subscribe() {
```

```
        // Create the dynamic proxy client for the NotificationBroker
        NotificationBroker broker = brokerService
            .getNotificationBrokerPort(new AddressingFeature());

        // Create a pull point
        W3CEndpointReference pullPointReference = null;
        try {
            CreatePullPoint request = new CreatePullPoint();
            CreatePullPointResponse response =
broker.createPullPoint(request);
            pullPointReference = response.getPullPoint();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }

        // Subscribe to the random-weather topic
        TopicExpressionType topic = new TopicExpressionType();
        topic.setExpression("tns:daily-weather");

topic.setDialect(TopicExpressionType.DIALECT_SIMPLE_TOPIC_EXPRESSION);
        topic.addPrefixMapping("tns", "http://weather");
        FilterType filter = new FilterType();
        filter.addTopicExpression(topic);

        // Create subscription information input message
        Subscribe subscribeRequest = new Subscribe();
        subscribeRequest.setConsumerReference(pullPointReference);
        subscribeRequest.setFilter(filter);

        // Send subscription request
        try {
            SubscribeResponse response =
broker.subscribe(subscribeRequest);
            subscriptionReference = response.getSubscriptionReference();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }

        // Store the Broker dynamic proxy in the PullWeatherReceiver
        // "singleton"
        broker = pullPointReference.getPort(NotificationBroker.class,
            new AddressingFeature());
        WeatherReceiver receiver = WeatherReceiver.getInstance();
        ((PullWeatherReceiver) receiver).broker = broker;
```

```
    }

    static {
        // Bootstrap the WeatherReciver "singleton" such that
        // it returns the receiver that can pull messages
        WeatherReceiver.setInstance(new PullWeatherReceiver());
    }
}
```

The structure of the SubscriptionManager servlet in Example 9-9 on page 454 is similar to the one in the push example. That is, the doGet method switches between subscription and unsubscription behavior. However, there is a slight difference if you look inside the subscribe and unsubscribe methods. These methods have been modified to also create and destroy the pull point respectively. The static block at the end of the servlet ensures that, when the SubscriptionManager servlet is loaded by the Java virtual machine (JVM™), it bootstraps the WeatherReceiver singleton so that it uses the PullWeatherReceiver subclass. This action ensures that the ConsumerServlet triggers WS-Notification message pulling whenever it is reloaded in the browser.

9.4 WS-Notification runtime administration

By using the administrative console of WebSphere Application Server, you can manage runtime entities such as subscriptions, pull points, and publisher registrations. With the administrative console, you can also manage WS-Notification messages on the underlying SIB.

To access the runtime management features of the WS-Notification service:

1. Expand **Service integration** → **WS-Notification** → **Services** and select a WS-Notification service, for example, **weatherWSNService**.

2. Click the **Runtime** tab (Figure 9-18). On the Runtime page, you can select any of the additional properties to see publisher registrations, pull points, or subscriptions.
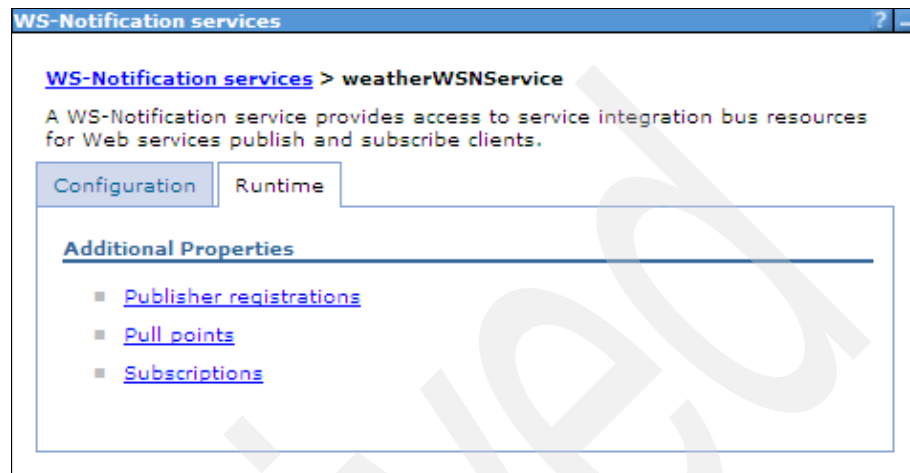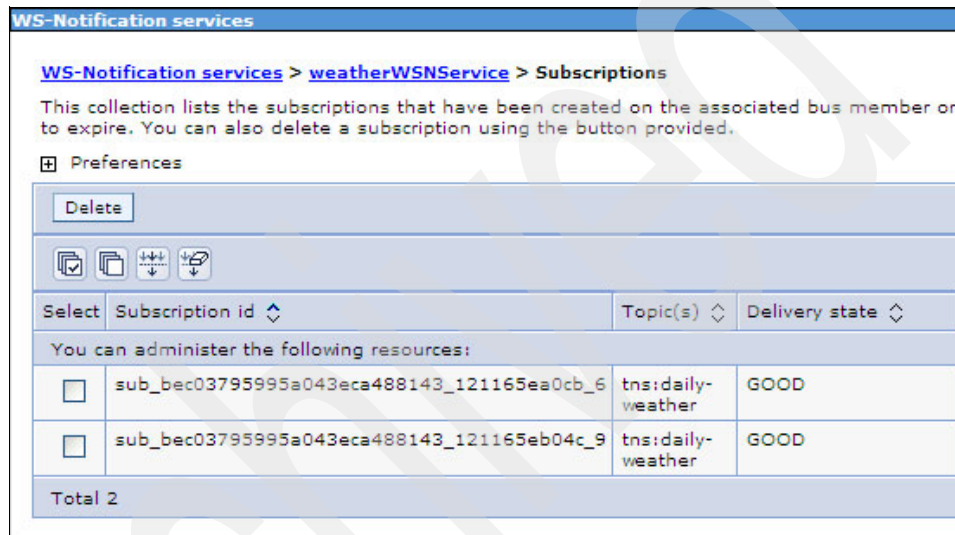

Figure 9-18   WS-Notification service runtime view

> **Tip:** Use the administrative console to monitor the subscriptions and pull points and remove old information. This is especially helpful while you are developing and testing the applications. At each change, the applications are redeployed, and old subscriptions and pull points remain in the server.

9.4.1 Administering subscriptions

In the weather WS-Notification example, after both consumers are subscribed to the daily-weather topic, click the **Subscriptions** link (from the WS-Notification services Runtime tab) to open the Subscriptions page (Figure 9-19). This page shows an overview of all consumer subscriptions. Additional columns are not shown in Figure 9-19. You can select any of the subscription lines and delete them if you need to, which might be necessary if your applications fail to unsubscribe themselves.

Figure 9-19   Managing subscriptions

Each subscription line specifies the following information:

► Subscription ID

  A unique internal ID.

► Topics

  Indicates which topics the consumer is subscribed to.

► Delivery state

  Indicates successfullness of message delivery.

► Consumer EPR

  The Web service endpoint address of the consumer. This can be a pull point Web service address.

- ► Creation time

  Indicates the date of subscription.

- ► Termination time

  Indicates when the subscription expires. If the client application has not specified it in the subscription request, then the value shown is *None*.

- ► Pull type

  Indicates whether the consumer is a pull consumer. In the weather example, we have one of each.

- ► Service integration bus subscriptions

  Provides a link to the underlying SIB topic space runtime page.

9.4.2 Administering pull points

In the weather WS-Notification example, after the pull consumer subscribes to the daily-weather topic, clicking the **Pull points** link (from the WS-Notification services Runtime tab) opens the Pull points page (Figure 9-20).

Figure 9-20   Managing pull points

From this page, you see an overview of all consumer pull points and can delete them if necessary. Each pull point line specifies the following information:

Pull point id   A unique internal ID
Creation time   Indicates at which time the pull point was created
Termination time   Indicates at which time the pull point will be removed

9.4.3 Administering messages

A feature that you might find useful is the ability to inspect published WS-Notification messages that have not yet been delivered to registered consumers. Published WS-Notification messages are stored by WebSphere Application Server on the SIB until the following actions occur:

▶ Messages are delivered to all subscribed push consumers (Web service endpoints).

▶ Messages are consumed by the subscribed pull consumers.

If you click any of the SIB subscriptions links in the Subscriptions runtime view, you see the runtime view of the underlying SIB topic space, as shown in Figure 9-21.
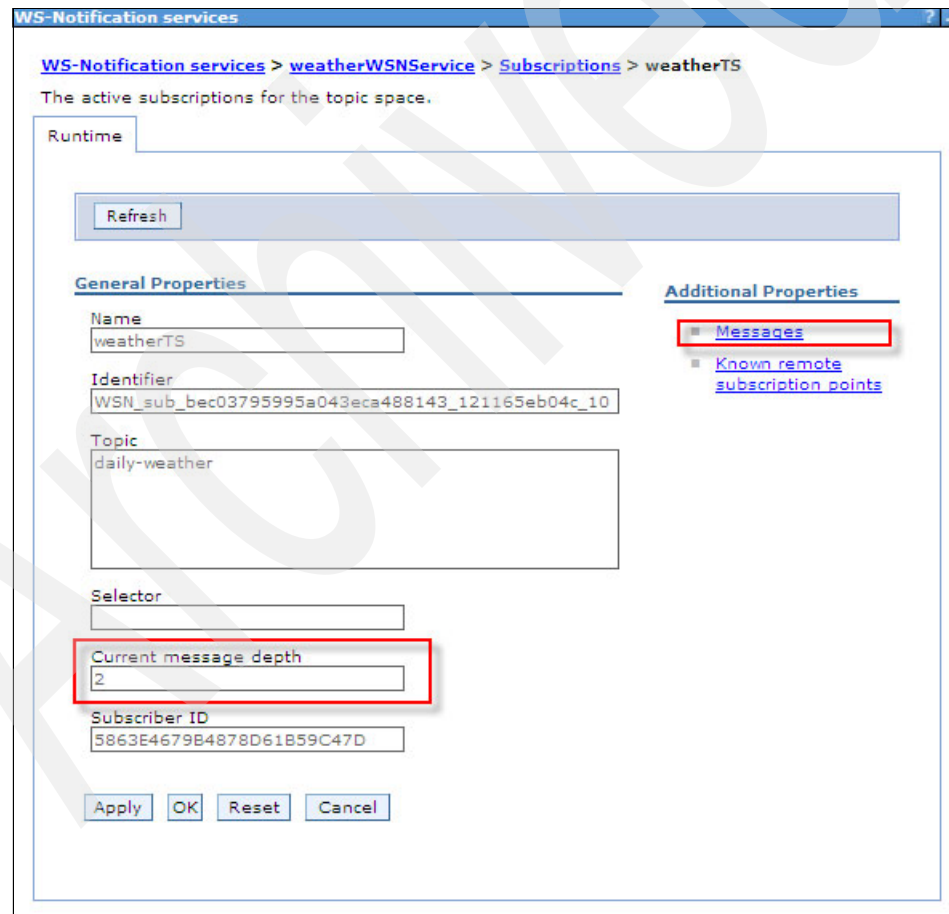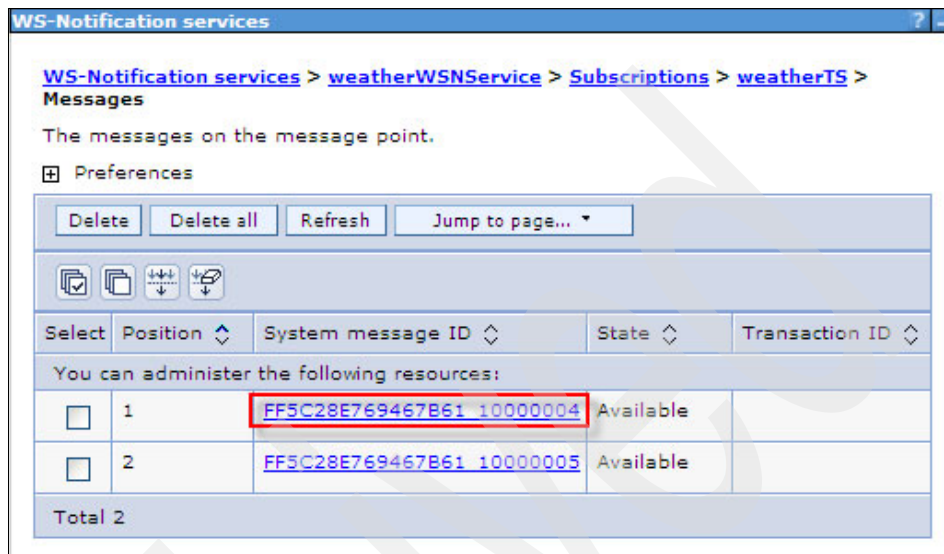
Figure 9-21   Message depth of undelivered messages

The Current message depth section shows the number of messages that are yet to be consumed. If you click the **Messages** link under Additional Properties, you see a list of the unconsumed messages (Figure 9-22).
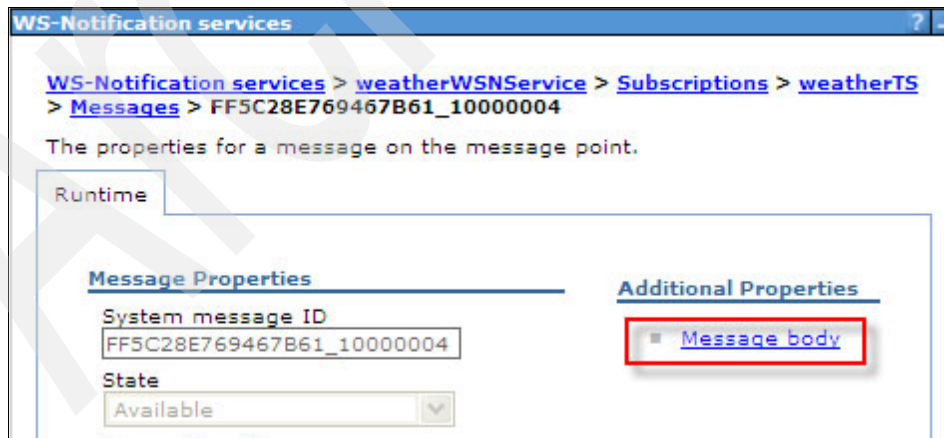
Figure 9-22   Unconsumed messages

The administrative console gives you the ability to delete either selected messages or all of them if necessary. If you click one of the links, you see the page shown in Figure 9-23, where you can inspect the message details.

Figure 9-23   Message details

Although not shown in Figure 9-23 on page 463, this page contains many interesting properties about the message, including details about how long the message has been on the messaging engine, the message length, and so on.

If you click the **Message body** link under Additional Properties on the message details page, you see the actual contents. The message contents in Figure 9-24 show the serialized version of one of the published daily weather messages.
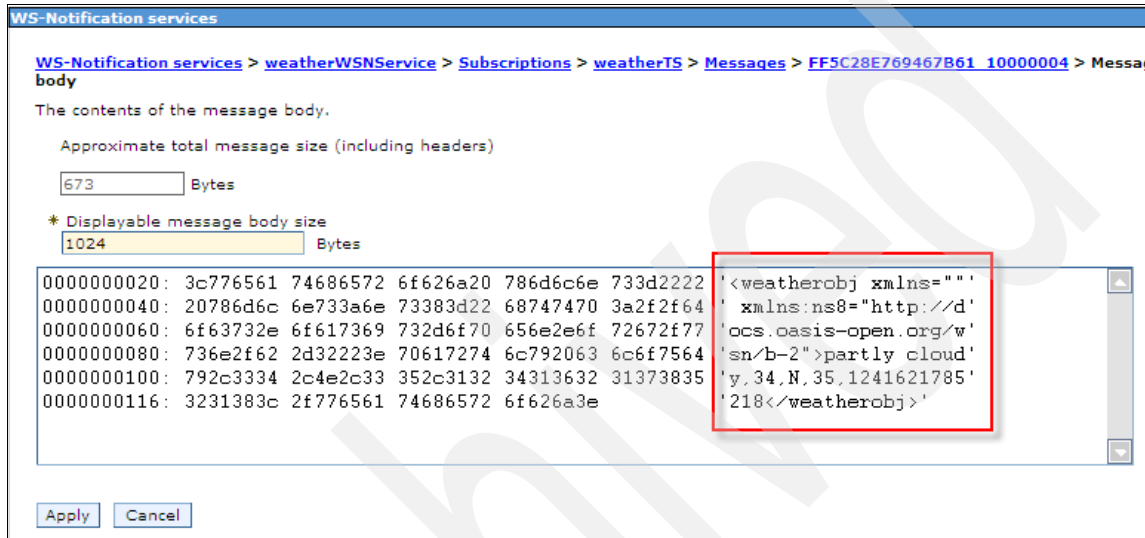
Figure 9-24   Inspecting the message content

9.5 Advanced features and options

WS-Notification offers several advanced features.

9.5.1 Using policy sets with WS-Notification services

You can apply policy sets to Version 7.0 WS-Notification services. This policy set configuration is a compelling feature because it allows you to easily configure qualities of service behavior such as reliability or security.

The configuration of policy sets for WS-Notification is two-fold:

► Service provider policy set configuration

   This configuration refers to the policy set configuration of Web services that are exposed by the WS-Notification service points. Version 7.0 WS-Notification service points (NotificationBrokers, SubscriptionManagers, and PublisherRegistrationManagers) are implemented as JAX-WS applications. The policy sets for these service points are configured by using the policy set administrative infrastructure for service providers.

► Service client policy set configuration

   This configuration refers to the policy set configuration of the Web service clients for the WS-Notification service. Each WS-Notification service has two Web service clients, which are OutboundNotificationService and OutboundRemotePublisherService. The policy sets for these two services are configured by using the policy set administrative infrastructure for service clients.

For more information about applying policy sets to WS-Notification, see the "WS-Notification: Overview" topic in the WebSphere Application Server V7 Information Center at the following address:

http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/index.jsp?topic=/
com.ibm.websphere.pmc.nd.doc/concepts/cjwsn_overview.html

9.5.2 Implementing demand-based publishers

The WS-Notification development examples in this chapter demonstrate how to create a simple WS-Notification publisher application. However, as explained in 9.1.2, "WS-BrokeredNotification" on page 399, another publisher variant exists, which is the demand-based publisher.

The demand-based publisher is more resource aware in that it ensures that the producer does not publish messages to topics to which no consumers subscribe. However, the demand-based publisher is slightly more complex to implement than the simple publisher because it requires development of a Web service endpoint that keeps track of the current demand for publishing.

9.5.3 Using handlers with WS-Notification services

Depending on the WS-Notification service type that you have, you can associate either JAX-WS handlers (Version 7.0) or JAX-RPC handlers (Version 6.1). WebSphere Application Server allows handlers to be associated with inbound requests (and related responses) to a WS-Notification service and outbound

requests (and related responses) from the service. Handlers for inbound requests are configured on the inbound ports that belong to a WS-Notification service point. Handlers for outbound requests are configured on the WS-Notification service itself.

9.5.4 JMS producers and consumers

WS-Notification services in WebSphere Application Server V7 use the service integration bus (SIB). In particular, they publish messages to SIB topic spaces. Topic spaces can by published and subscribed to by JMS applications by using JMS topics. Therefore, it is possible for a JMS application to receive messages that are published by a notification producer and for a JMS application to publish messages that will be received by a notification consumer.

For more information about sharing notifications with other bus clients, see the "Sharing event notification messages with other bus client applications" topic in the WebSphere Application Server V7 Information Center at the following address:

`http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/index.jsp?topic=/com.ibm.websphere.pmc.nd.doc/ref/rjwsn_ex_sysa2.html`

9.5.5 Administered subscribers

An administered subscriber provides a mechanism for a WS-Notification service point to subscribe to an external notification producer at server startup time. The WS-Notification service point thereby plays a consumer role to the external producer. One possible use of such a configuration is to have a connection between WS-Notification services on different buses. Messages published to a particular topic on WS-Notification service A are received by a consumer that is subscribed to the same topic on WS-Notification service B.

Creating an administered subscriber
To create an administered subscriber:

1. From the WebSphere administrative console, select **Service integration** → **WS-Notification** → **Services**. Select the service that contains the WS-Notification service point to which you want to add an administered subscriber.

2. Under Additional Properties, select **WS-Notification service points** and then select the service point to which you want to add an administered subscriber.

3. Under Additional Properties, select **Administered subscribers** and then click **New**.

4. Complete the general properties for the notification producer endpoint that you want to be subscribed to at server startup.

5. Click **OK** and save your changes.

9.5.6 Topic namespace documents

Topic namespace documents define the hierarchical structure of the topics in a topic namespace. They also allow the user to define restrictions for messages received on a topic or the ability to use subtopics. Example 9-10 shows a simple topic namespace document.

Example 9-10   Topic namespace document

```
<?xml version="1.0" encoding="UTF-8"?>
<t1:TopicNamespace name="RedbookTopicNamespace"
targetNamespace="http://example.redbook"
    xmlns:tns="http://example.redbook"
    xmlns:abc="http://example2.redbook"
    xmlns:xyz="http://example3.redbook"
    xmlns:t1="http://docs.oasis-open.org/wsn/t-1">
    <t1:Topic name="myTopic1">
        <t1:Topic name="subTopic1a" messageTypes="abc:SomeData"/>
        <t1:Topic name="subTopic1b"/>
    </t1:Topic>
    <t1:Topic name="myTopic2">
        <t1:Topic name="aSubTopic" messageTypes="xyz:Information"/>
        <t1:Topic name="anotherSubTopic" final="true"/>
    </t1:Topic>
</t1:TopicNamespace>
```

In Example 9-10, note the following points of interest:

▶ The attribute final

A value of true for this attribute indicates that messages cannot be published to subtopics of this topic. The default value, if not specified, is false.

▶ The attribute messageTypes

This attribute defines a list of qualified names. All messages published to this topic must have a global element definition that matches one of the qualified names listed.

For a full list of attributes and elements that can be used in topic namespace documents, see the WS-Topics specification.

In WebSphere Application Server V7, topic namespace documents can be applied to permanent topic namespaces. To apply a document, first make sure that the target namespace for the topic namespace document matches the namespace URI for the permanent topic namespace to which you want to apply it. Then follow these steps:

1. From the WebSphere administrative console, select **Service integration** → **WS-Notification** → **Services**. Select the service to which the permanent topic namespace belongs.

2. Under the Additional Properties list, click **Permanent topic namespaces**. From the list, locate the permanent topic namespace to which you want to apply a topic namespace document. The value in the Topic namespace documents column indicates the number of documents applied or shows the value *None*. Click the value.

3. When you see a list of topic namespace documents that are applied, click **New**. Enter the URL location of the namespace document and optionally a description. Click **OK**.

4. Save your changes.

9.5.7 Raw notification message format

The WS-BaseNotification specification defines the concept of the raw notification format. A *raw notification* is one where the notification message contents are not sent in the form described by the notify operation. Rather, the contents of the message form the entire body of the SOAP message (Figure 9-25).

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" …>
  <soapenv:Header>
  …
  </soapenv:Header>
  <soapenv:Body>
    <b2:Notify xmlns:b2="http://docs.oasis-open.org/wsn/b-2">
      <b2:NotificationMessage>
        <b2:Topic>…</b2:Topic>
        <b2:ProducerReference> … </b2:ProducerReference>
        <b2:Message>

          <abc:SomeData xmlns:abc="http://somenamespace">
          …
          </abc:SomeDate>                                    ← Message content

        </b2:Message>
      </b2:NotificationMessage>
    </b2:Notify>
  </soapenv:Body>
</soapenv:Envelope>
```

Notify (default) format message

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" …>
  <soapenv:Header>
  …
  </soapenv:Header>
  <soapenv:Body>

    <abc:SomeData xmlns:abc="http://somenamespace">
    …
    </abc:SomeDate>                                    ← Message content

  </soapenv:Body>
</soapenv:Envelope>
```

Raw format message
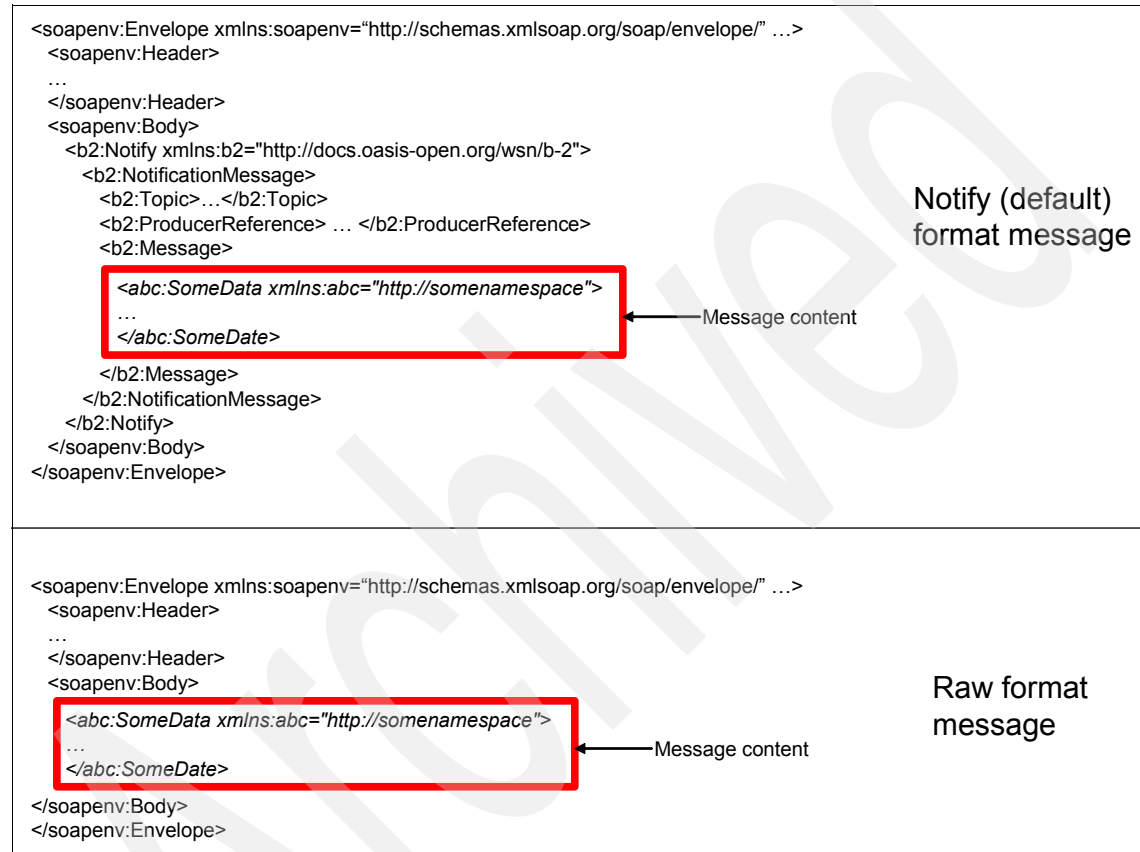
Figure 9-25   Notify and raw message formats

Raw message support in WebSphere Application Server V7

WS-Notification services in WebSphere Application Server V7 support the use of the raw notification format for messages sent from the service but *not* for messages received by the service. Therefore, a consumer can subscribe to the WS-Notification service, requesting that messages be sent to it by using the raw notification format, but a producer cannot publish messages in the raw format to the WS-Notification service. For an example that shows how to specify the use of raw messages in a subscription, see the "Example: Subscribing a WS-Notification consumer" topic in the WebSphere Application Server V7 Information Center at the following address:

http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/topic/com.ibm.
websphere.pmc.nd.doc/ref/rjwsn_ex_sub.html

10

WS-SecureConversation

In this chapter we discuss support for the Web services Secure Conversation (WS-SecureConversation) specification in WebSphere Application Server V7. We review the basic concepts in WS-Security and present an overview of Web Services Trust (WS-Trust). We also explore the motivation and mechanism of WS-SecureConversation and introduce WS-SecureConversation scenarios. In addition, we provide two examples for applying WS-SecureConversation to a Web services application by using WebSphere Application Server and Rational Application Developer.

This chapter contains the following topics:

10.1 WS-Security review

WS-SecureConversation is built on top of the Web services Security (WS-Security) and WS-Trust models to provide secure communication across one or more messages. We first review the WS-Security concepts.

10.1.1 Message-level security versus transport-level security

Web services are secured by using the transport-level security and message-level security mechanisms.

Traditionally, WS-Security has used transport-level security to secure point-to-point communications. HTTPS is used to maintain the security context between messaging endpoints. This approach works well for securing remote procedure call-based Web services. In cases where network intermediaries and multiple sessions connect the client with the server, trust relationships are associated with the point-to-point communications. The trust relationship is between the requester and the intermediary, as well as between the intermediary and the Web service (Figure 10-1).
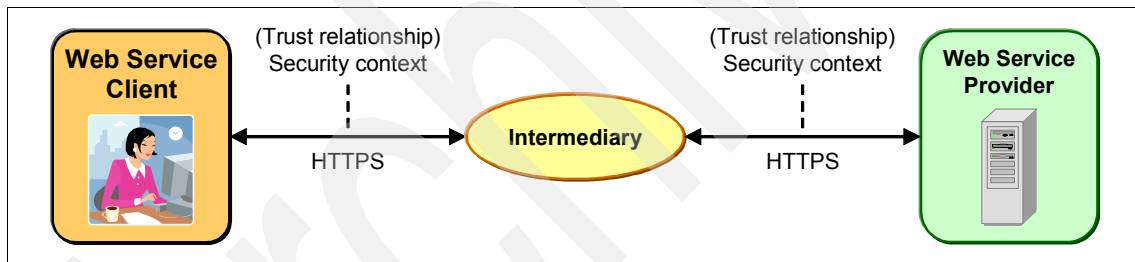
Figure 10-1   Transport-level security: trust relationship between the requester and intermediary

While this is acceptable in some scenarios, it might not be acceptable in others. With intermediaries, the entire message must be decrypted to access the routing information, which might break the overall security context. Also, with HTTPS, there is no option to apply security selectively on certain parts of the message.

With message-level protection, security is encapsulated in the SOAP message. Message-level security focuses on securing the entire end-to-end communication within a single security context. This type of security is done through a combination of message integrity, confidentiality, and security tokens to verify messages.

The trust relationship is established between the requester and the target Web service, even when an intermediary is involved in the message flow, as shown in Figure 10-2. Sometimes intermediaries must work with parts of the message. Parts of the message can be left clear, and other parts secured with different keys for different recipients. The WS-Security specification provides message-level security.
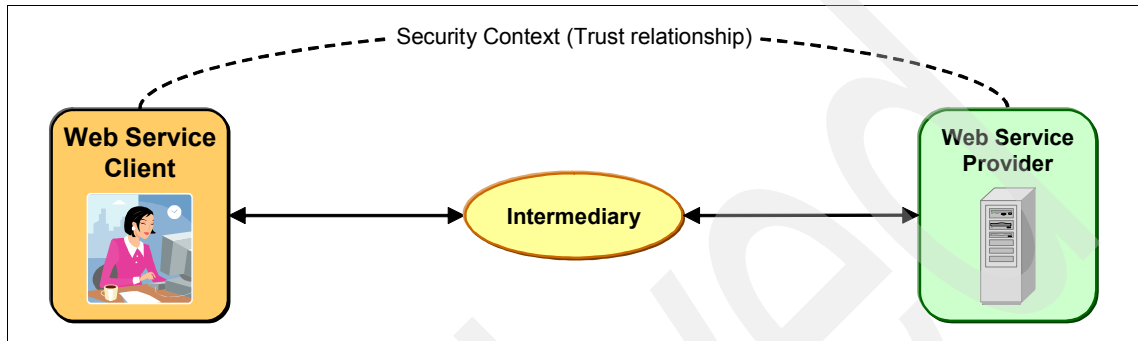
Figure 10-2   Transport-level security: trust relationship between the requester and target Web service

10.1.2 Major issues addressed by WS-Security

WS-Security addresses three issues with securing SOAP message exchanges:

► Authentication
► Message integrity
► Message confidentiality

Authentication
Authentication ensures that parties within a business transaction are really who they claim to be. Therefore, proof of identity is required. This proof can be claimed in the following ways:

► Presenting a user ID and a password, which is referred to as a *username token* in WS-Security domain

► Using an X.509 certificate issued by a trusted Certificate Authority (a more complex method)

The certificate contains identity credentials and has a pair of private and public keys associated with it. The proof of identity presented by a party includes the certificate and a separate piece of information that is digitally signed by using the certificate's private key. By validating the signed information using the public key associated with the party's certificate, the receiver can authenticate the sender as being the owner of the certificate, thereby validating their identity.

Two WS-Security specifications, the *Username Token Profile 1.0/1.1* and the *X.509 Certificate Token Profile 1.0/1.1*, describe how to use these authentication mechanisms with WS-Security.

Message integrity

To validate that a message has not been tampered with or corrupted during its transmission over the Internet, the message can be digitally signed by using security keys. The sender uses the private key of the their X.509 certificate to digitally sign the SOAP request. The receiver uses the sender's public key to check the signature and identity of the signer. The receiver signs the response with its private key, and the sender can validate that the response has not been tampered with or corrupted by using the receiver's public key to check the signature and identity of the responder.

The *WS-Security: SOAP Message Security 1.0/1.1* specification describes enhancements to SOAP messaging to provide message integrity.

Message confidentiality

To keep the message safe from eavesdropping, encryption technology is used to scramble the information in Web services requests and responses. The encryption ensures that no one accesses the data in transit, in memory, or after it has been persisted, unless they have the private key of the recipient.

The *WS-Security: SOAP Message Security 1.0/1.1* specification describes enhancements to SOAP messaging to provide message confidentiality.

10.1.3 Digital signature and XML encryption

WS-Security uses a digital signature to provide message integrity, XML encryption, and security tokens to authenticate the client. In this section, we look at digital signature and XML encryption technologies.

Digital signature

A digital signature provides message integrity as follows:

1. The sender creates a hash of the SOAP message to be sent by using a certain algorithm. A hash function takes the SOAP message as input and produces a fixed length string as output, sometimes called a *message digest*. The hash value is a concise representation of the longer message or document from which it was computed.

2. The sender encrypts the hash data (message digest) by using the *sender's private key* and attaches the result to the SOAP message. The result is the signature of the message.

3. When the recipient receives the message, the receiver decrypts the signature by using the *sender's public key*. The result is the value of the hash the sender put in the signature. Then the receiver runs the same hash algorithm to calculate the hash value (message digest) of the received message. If these two hash values match, the recipient is confident that the message has not been tampered with. If the original message changes during transmission, these two hash values will not match.

Figure 10-3 illustrates the digital signature process.
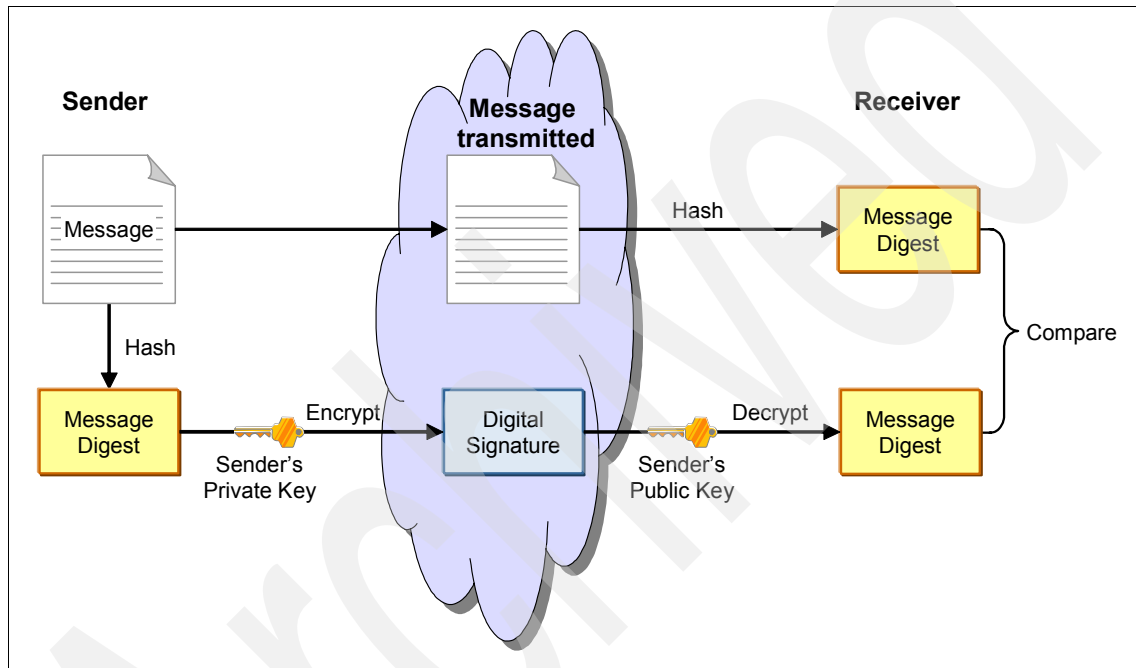
Figure 10-3   Digital signature process

XML encryption

There are two popular algorithms in modern encryption methods. One algorithm is the symmetric key algorithm, and the other one is the asymmetic key algorithm.

In a *symmetric key algorithm*, the sender and receiver must have a shared key set up in advance. The shared key must be kept in secret from all other parties. The sender encrypts the message by using the shared key, and the receiver must use the same key to decrypt the message.

In an asymmetric key algorithm, a certificate has a pair of keys, a public key and a private key. The *private key* is kept secret and the *public key* can be widely

distributed. The keys are related mathematically, but they cannot be derived from each other. A message encrypted with the public key can be decrypted only with the corresponding private key and vice versa.

A symmetric key algorithm is more efficient than an asymmetric key algorithm. However, it requires management of shared keys between the parties and has the inherent security risks of the keys being exposed to people unauthorized to know the key. Asymmetric key algorithms do not suffer from this weakness, but the algorithms are slow. To benefit from the best of both algorithms, XML encryption usually uses a two-phase process with both symmetric and asymmetric algorithms:

1. The sender generates a symmetric key, which is used for only one communication session. Therefore, it is referred to as the *session key*. The sender uses this key to encrypt the message.

   The sender encrypts the session key by using the *public key of the message recipient* and attaches the encrypted key to the message. By encrypting the session key instead of the entire SOAP message by using the asymmetric key algorithm, XML encryption provides a more efficient way to encrypt the SOAP message.

2. The recipient receives the message. It uses *its own private key* to decrypt the session key and then to decrypt the message. Because the session key is used only once, even if someone managed to discover it, they might only be able to decrypt one message.

Figure 10-4 illustrates the XML encryption process.
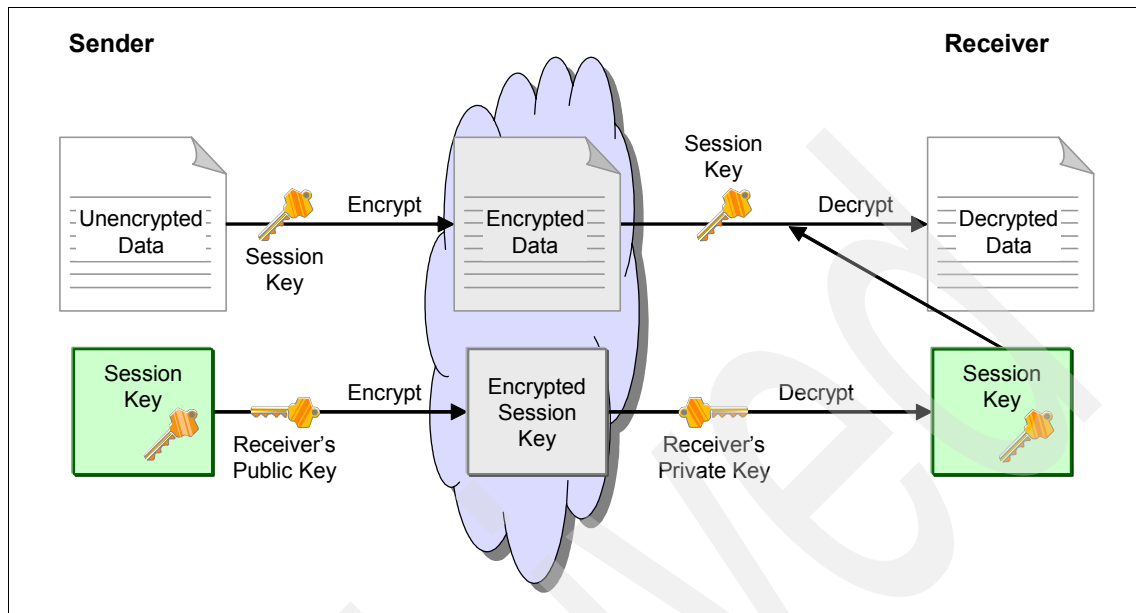
Figure 10-4   XML encryption process

XML encryption provides both a *relatively* efficient solution and one that is easy to manage by using a hybrid of symmetric and asymmetric algorithms. Note the word *relatively.* In scenarios that involve long duration, multi-message conversations between the Web services, the computationally expensive asymmetric key algorithm is repeatedly used to encrypt the session key. Ideally, you should be able to perform a simple negotiation that defines conversation-specific keys and then use the conversation-specific keys to encrypt the subsequent message exchanges. WS-SecureConversation uses this technique, and we discuss the mechanism in 10.3, "Overview of WS-SecureConversation" on page 482.

10.1.4 WS-Security support in WebSphere Application Server V7

WebSphere Application Server V7 provides full support for the following Organization for the Advancement of Structured Information Standards (OASIS) specifications and WS-I profiles:

► OASIS: WS-Security: SOAP Message Security 1.1 (WS-Security 2004)
► OASIS: WS-Security: UsernameToken Profile 1.1
► OASIS: WS-Security X.509 Certificate Token Profile 1.1
► WS-I Basic Security Profile (WS-I BSP) 1.0

WebSphere Application Server V7 also supports the following specifications:

- ► OASIS: WS-Trust Version 1.3.
- ► OASIS: WS-SecureConversation Version 1.3
- ► OASIS: Kerberos Token Profile Version 1.1
- ► OASIS: WS-SecurityPolicy Version 1.2

10.2 WS-Trust

WS-Security describes enhancements to SOAP messaging to provide quality of protection through message integrity, message confidentiality, and single message authentication. An important class of WS-Security is to define mechanisms for signing and encrypting SOAP messages by using security tokens.

Security tokens are a collection of claims that are used to prove the identity of a client. They contain an identifier for the client and a proof of the client's identity, such as a password. They can also include information, such as a signature, to indicate that the issuer certifies the claims in the credential. While WS-Security specifies mechanisms to securely exchange messages by using security tokens, it does not address how security tokens are issued and exchanged.

The WS-Trust specification builds on the WS-Security specification. The goal of WS-Trust is to enable applications to construct trusted SOAP message exchanges. This trust is represented through the exchange and brokering of security tokens. WS-Trust provides a protocol agnostic way to issue, renew, and validate these security tokens. It defines ways to establish, access the presence of, and broker trust relationships. It is designed to support the creation of multiple security token formats to accommodate a variety of authentication and authorization mechanisms.

10.2.1 Security Token Service

The key concept in WS-Trust is a security token service. A *s*ecurity token service is a distinguished Web service that issues, exchanges, and validates security tokens. WS-Trust allows Web services to set up and agree on which security servers they trust and to rely on these servers. To communicate trust, a security token service requires proof, such as a signature, to verify knowledge of a security token or set of security tokens.

The security token service has broad applicability in that it can be used to issue security tokens that make a wide range of assertions. It can also rely on a separate security token service to issue a security token with its own trust

statement. In many cases, security token service is used to issue the same assertions but in different formats. For example, a security token service might issue an X.509 certificate, asserting that the key holder is Alice, and it might do this based on a Kerberos token issued by a Kerberos key distribution center (KDC). The process, illustrated in Figure 10-5, forms the basis of trust brokering by issuing a range of security tokens that can be used to broker trust relationship between different trust domains.
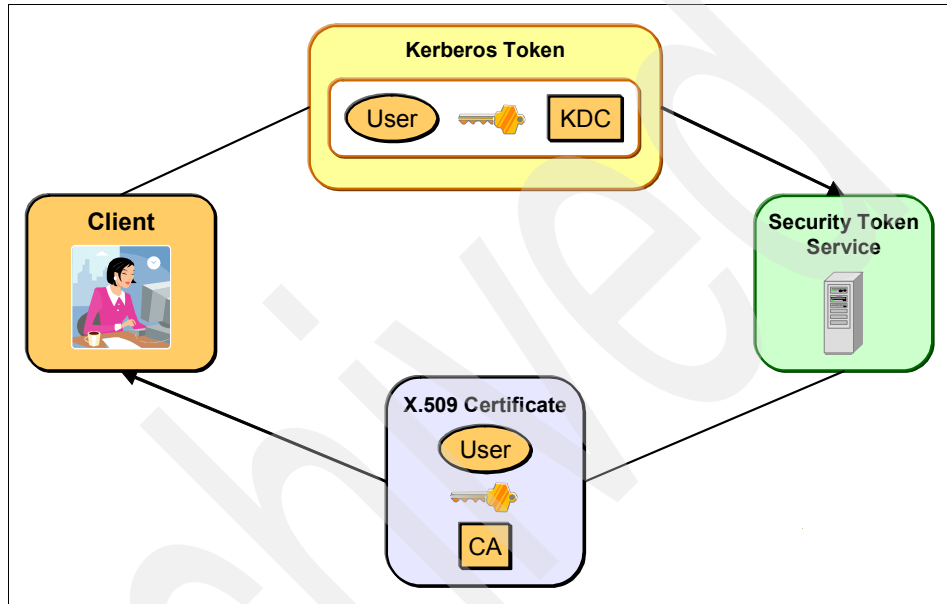
Figure 10-5   Issuing security tokens in different formats by the security token service

10.2.2 WS-Trust model

The Web service security model defined in WS-Trust is based on a process in which a Web service requires that an incoming message prove a set of claims (for example, name, key, permission, capability, and so on) before it is authenticated as a trustworthy consumer of the Web service. If the requester does not have the necessary tokens to prove required claims to a service, it contacts the security token service and requests the tokens with the proper claims. In turn, the security token service can require its own set of claims for authenticating and authorizing the request for security tokens. In this case, the security token service establishes two separate trust relationships. One is with the Web service and the other is with the Web service client.

Figure 10-6 illustrates the WS-Trust model.
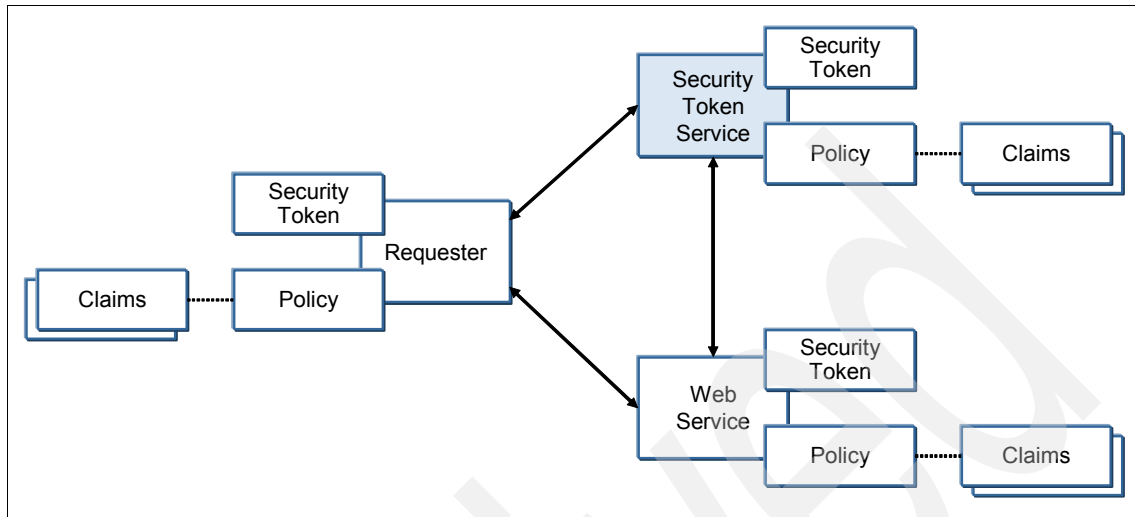
Figure 10-6   WS-Trust model

10.2.3 Security token service framework

The security token service defines a framework for token issuance. A requester sends a request. If the policy permits and the recipient's requirements are met, then the requester receives a security token response. This process uses the <wst:RequestSecurityToken> element to send the request and the <wst:RequestSecurityTokenResponse> element to receive the new or renewed security token.

Requesting a security token

The <wst:RequestSecurityToken> element is used to request a security token. This element is signed by the requester, by using tokens contained or referenced in the request that are relevant to the request. Four possible requests are sent to the Security Token Service:

► Issue a new token.
► Renew token.
► Validate a token.
► Cancel a token.

Example 10-1 shows the syntax for this element.

Example 10-1   The <wst:RequestSecurityToken> element

```
<wst:RequestSecurityToken Context="...">
<wst:TokenType>...</wst:TokenType>
<wst:RequestType>...</wst:RequestType>
...
</wst:RequestSecurityToken>
```

Returning a security token

The <wst:RequestSecurityTokenResponse> element is used to return a security token or response. The security token is used in subsequent SOAP messages and is referred to based on the mechanisms defined by WS-Security. Example 10-2 shows the syntax for this element.

Example 10-2   The <wst:RequestSecurityTokenResponse> element

```
<wst:RequestSecurityTokenResponse Context="...">
<wst:TokenType>...</wst:TokenType>
<wst:RequestedSecurityToken>...</wst:RequestedSecurityToken>
...
</wst:RequestSecurityTokenResponse>
```

WS-Trust in WebSphere Application Server V7

WebSphere Application Server V7 supports the WS-Trust 2005 Submission Draft specification (Version 1.1). However, WebSphere Application Server does not provide a full security token service that implements all the contents of the WS-Trust draft specification.

WebSphere Application Server V7 also supports the approved version WS-Trust 1.3 specification, which is dated March 2007. The security context token provider supports the OASIS Version 1.3 specifications for WS-Trust and WS-SecureConversation. A configuration option allows support for the two different levels of the WS-Trust standard to coexist on the same server. This option provides interoperability between systems and products that support different specification levels.

A setting is also provided to specifically disable support for the WS-Trust 2005 Submission Draft specification (Version 1.1) for the security context token provider. For more details see the "Web services Trust standard" topic in the WebSphere Application Server Information Center at the following address:

http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/index.jsp?topic=/
com.ibm.websphere.express.doc/info/exp/ae/cwbs_wstruststd.html

10.3 Overview of WS-SecureConversation

The WS-Security Version 1.1 standard from OASIS defines how to digitally sign and encrypt a SOAP message to provide message-level protection. The standard also defines how to attach and reference a security token for digital signature and encryption. However, it does not provide session-based protection when a long series of related messages are exchanged.

The WS-SecureConversation standard is a building block. It is used in conjunction with the other Web service and application-specific protocols, including WS-Security and WS-Trust, to accommodate a wide variety of security models and technologies. WS-SecureConversation is built on top of the WS-Security and WS-Trust models to provide secure communication across one or more messages. The WS-SecureConversation specification explains how to establish a security context token between two parties. It uses the WS-Trust specification to issue and exchange security context tokens.

10.3.1 Motivation

Two major forces drive WS-SecureConversation. One force is from the performance perspective and the other force is from the security requirements of WS-ReliableMessaging. For details about WS-ReliableMessaging, see *Web Services Feature Pack for WebSphere Application Server V6.1*, SG24-7618.

Improving Web services performance

Some Web service scenarios only involve the short, sporadic exchange of a few messages. WS-Security readily supports this model. Other scenarios involve long-duration, multimessage conversations between the Web services. WS-Security also supports this model, but the solution is not optimal.

WS-Security has two suboptimal usages in these scenarios:

► Repeated use of computationally expensive cryptographic operations such as public key validation

► Sending and receiving many messages by using the same cryptographic keys, providing more information that allows brute force attacks to *break the code*

For these reasons, protocols such as HTTPS use public keys to perform a simple negotiation that defines conversation-specific keys. This key exchange allows more efficient security implementations and decreases the amount of information encrypted with a specific set of keys.

WS-SecureConversation provides similar support for WS-Security. Participants often use WS-Security with public keys to start a *conversation* or *session*. They use WS-SecureConversation to agree on session-specific keys for signing and encrypting information.

Protecting the sequence of reliable messaging

With reliable messaging in Web services, applications can send and receive messages simply, reliably, and efficiently even in the face of application, platform, or network failure. Reliable messaging uses a message sequence to reliably deliver a set of messages.

The WS-Security policy secures the Web services application messages, but it does not secure the sequence of the messages and thus not the WS-ReliableMessaging message sequence numbers. This approach can expose the recipient to *sequence spoofing*. The reliable messaging policy requires the reliable messaging headers to be signed to overcome sequence spoofing. If you want to use secure conversation and reliable messaging policies in the same policy set, the secure conversation bindings must be configured to require that the reliable messaging headers are signed.

Sequence spoofing is a class of threats in which the attacker uses knowledge of the identifier for a particular sequence to forge sequence life-cycle or traffic messages. Imagine two valid clients, each with a sequence. Both are authorized at the service level, but one of the clients is in reality a hacker who wants to attack the other sequence. If the hacker can guess the sequence identifier, this person can create a fake TerminateSequence message that references the target sequence and sends this message to the appropriate RM destination.

Figure 10-7 illustrates this scenario.
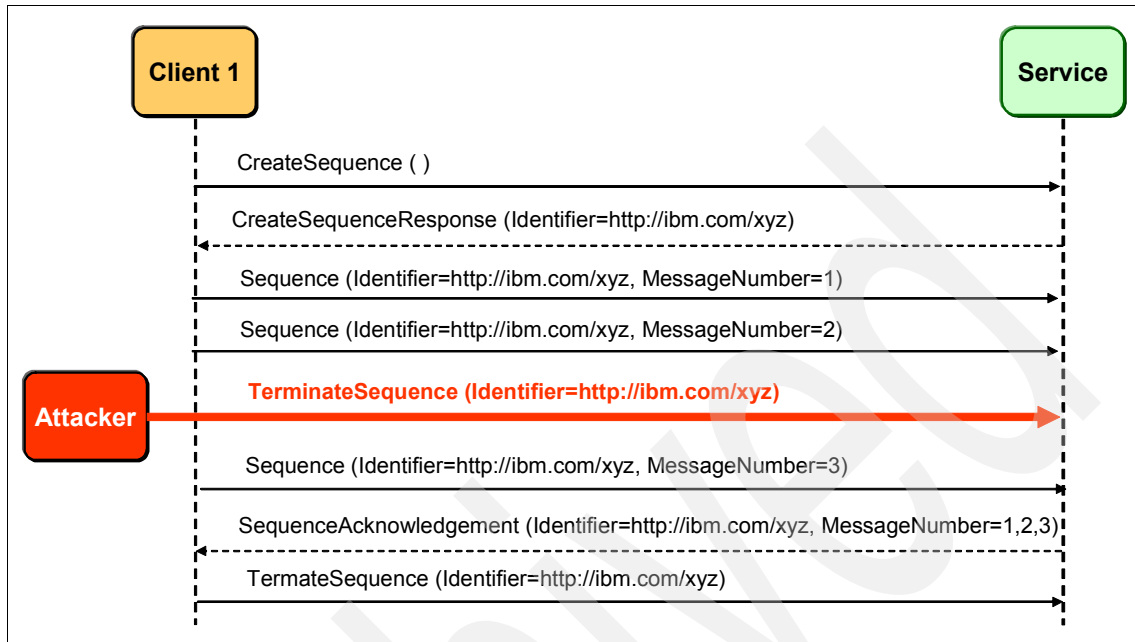
Figure 10-7   Sequence attack

WS-SecureConversation provides a mechanism for protecting sequences. We explain the mechanism in 10.3.4, "Secure conversation with reliable messaging scenario" on page 495.

10.3.2 Key concepts

There are two key concepts in WS-SecureConversation, which we explain in the following sections:

► Security context token
► Derived key

Security context token

A *security context* is an abstract concept that refers to an established authentication state and negotiated keys that might have additional security-related information. Parties that want to exchange multiple messages establish a security context in which to exchange multiple messages. A security context is shared among the communicating parties for the lifetime of a communications session. A security context is a way to provide session-based security, rather than establishing new keys for every message.

A *security context token* is a type of security token that represents a security context that is shared by the two communicating parties, which are the Web service and the consumer of that Web service. A security context token typically contains keys that are used as the basis of providing WS-Security-related services, such as XML encryption and digital signature.

In the WS-SecureConversation specification, a security context is represented by the <wsc:SecurityContextToken> security token. The following Uniform Resource Identifier (URI) represents the security context token type that is required to establish a secure conversation:

```
thttp://docs.oasis-open.org/ws-sx/ws-secureconversation/200512/sct
```

To request a security context token, a *request security token* (RST) is sent to the service endpoint to which you are setting up a secure conversation. The request is transparently rerouted to the trust service. The trust service processes the RST and responds with a *request security token response* (RSTR). This response is returned to the requester as though it were generated by the endpoint service.

Example 10-3 shows an RST request to issue a security token.

Example 10-3   Security context token request

```
<wst:RequestSecurityToken
   xmlns:wst="http://docs.oasis-open.org/ws-sx/ws-trust/200512"
Context="http://www.ibm.com/login/">
   <wst:TokenType>
      http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512/sct
   </wst:TokenType>
   <wst:RequestType>
      http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue
   </wst:RequestType>
   <wsp:AppliesTo xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy">
      <wsa:EndpointReference xmlns:wsa="http://www.w3.org/2005/08/addressing">
         <wsa:Address>
            http://localhost:80/WSSampleSei/EchoService
               </wsa:Address>
      </wsa:EndpointReference>
   </wsp:AppliesTo>
   <wst:Entropy>
      <wst:BinarySecret
         Type="http://docs.oasis-open.org/ws-sx/ws-trust/200512/Nonce">
         zb//KsawV6DmfC8kB6vNOQ==
            </wst:BinarySecret>
   </wst:Entropy>
```

```
    <wst:KeySize>128</wst:KeySize>
</wst:RequestSecurityToken>
```

Example 10-4 shows an RSTR response to issue a security token.

Example 10-4   RST response

```
<wst:RequestedSecurityToken>
    <wsc:SecurityContextToken
        xmlns:wsc="http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512"

xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility
-1.0.xsd"
        wsu:Id="uuid:FFA51A32EB818FB6EA1222986227363">
        <wsc:Identifier>
            uuid:FFA51A32EB818FB6EA1222986227346
                    </wsc:Identifier>
        <wsc:Instance>
            uuid:FFA51A32EB818FB6EA1222986227345
                    </wsc:Instance>
    </wsc:SecurityContextToken>
</wst:RequestedSecurityToken>
<wsp:AppliesTo xmlns:wsp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
    <wsa:EndpointReference xmlns:wsa="http://www.w3.org/2005/08/addressing">
        <wsa:Address>
            http://localhost:80/WSSampleSei/EchoService
                    </wsa:Address>
    </wsa:EndpointReference>
</wsp:AppliesTo>
<wst:RequestedProofToken>
    <wst:ComputedKey>
        http://docs.oasis-open.org/ws-sx/ws-trust/200512/CK/PSHA1
                </wst:ComputedKey>
</wst:RequestedProofToken>
```

Derived key

A security context token implies or contains a shared secret. This secret can be used to sign and encrypt messages. However, it is considered bad practice to sign and encrypt messages with the same key because certain attacks are more likely to succeed in this case. Signing and encrypting multiple messages by using the same key in multimessage conversations is also considered bad practice because it provides too much data to attackers to analyze. Therefore, use derived keys to sign and encrypt messages that are associated only with the security context.

WS-SecureConversation provides a secured session for long-running message exchanges and usage of the symmetric cryptographic algorithm with coordinated derived keys.

By using a common secret in the security context token, parties can define different key derivations to use. For example, four keys can be derived so that two parties can sign and encrypt using separate keys, as illustrated in Figure 10-8.
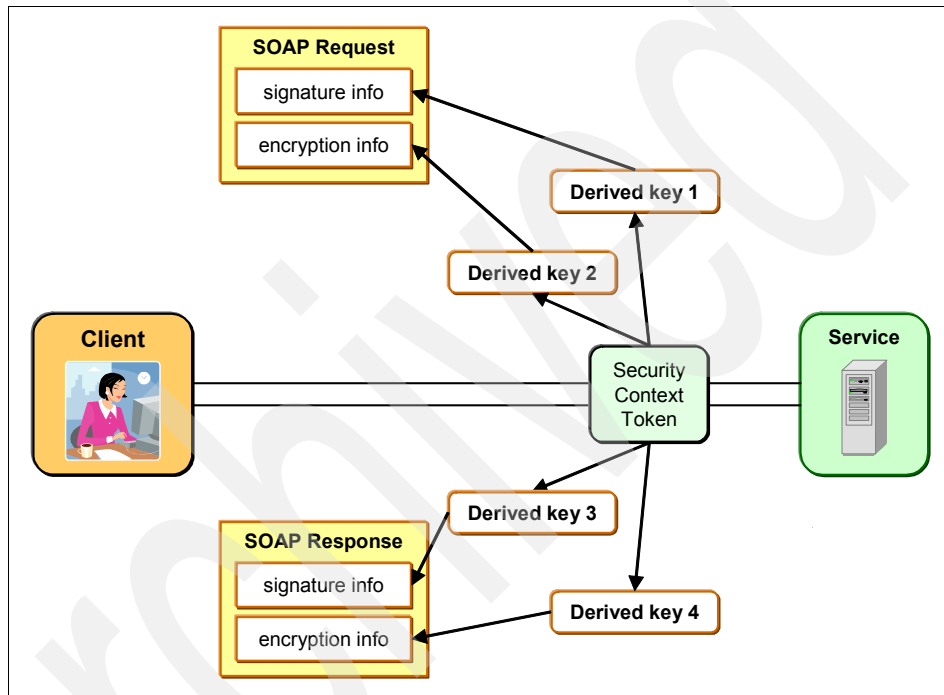
Figure 10-8   Using derived keys to sign and encrypt SOAP messages

To keep the keys fresh, subsequent derivations can be used in multimessage conversations. WS-SecureConversation introduces the <wsc:DerivedKeyToken> token as a mechanism for indicating which derivation is being used within a given message.

With the derived key, you can then use symmetric cryptography to sign and encrypt the message. A symmetric cryptographic algorithm is less CPU intensive than the asymmetric cryptography. It also provides better performance and throughput when compared with the asymmetric cryptographic algorithms.

10.3.3 Secure conversation scenario

In this section, we describe the overall flow on the message exchanges in WS-SecureConversation. During the secure conversation session, the initiator establishes the security context token by using the WS-Trust protocol for session-based security with the recipient. Then derived keys from the security context token are used to sign and encrypt the SOAP message to provide message-level protection.

Trust service

The security token service that is provided by WebSphere Application Server is called the trust service. The WebSphere Application Server trust service uses the secure messaging mechanisms of WS-Trust to define additional extensions for the issuance, exchange, and validation of security tokens.

Message overflow

WebSphere Application Server supports the ability of an endpoint to issue a security context token for WS-SecureConversation, and thereby provides a secure session between the initiator and recipient of SOAP messages.

Figure 10-9 illustrates the flow that is required to establish a secured context and to use session-based security.
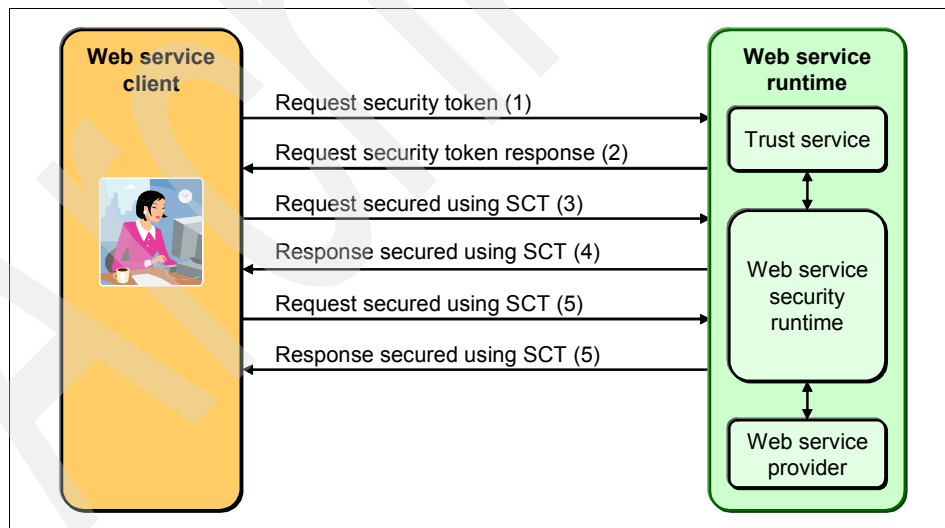

Figure 10-9   Message overflow in a secure conversation

To use secure conversation, the following steps are involved:

1. The client sends an RST for a security context token to an application endpoint. The RST is encrypted and signed by using WS-Security information that is defined in the bootstrap security policy. The bootstrap policy is used by the initiator to acquire a security token from the security token services.

2. The RST is processed by the trust service, and if the request is trusted based on the bootstrap policy, the trust service returns the security context token by using the RSTR. The RSTR is also signed and encrypted to ensure that the established security context token is not compromised. The client verifies whether the RSTR can be trusted, based on the bootstrap policy.

3. If the RSTR is trusted, the client secures (signs and encrypts) the subsequent application messages by using the derived keys. The derived keys are derived from a secret contained in the security context token that is obtained from the initial RST and RSTR messages that are exchanged between the initiator and the recipient.

4. The target Web service uses the derived key to verify and decrypt the message based on the application security policy. Then the target Web service uses the derived key to sign and encrypt the response based on the application security policy. Finally, the client uses the derived key to verify and decrypt the message based on the application security policy. The first message exchange is finished.

5. For subsequent message exchanges, steps 3 and 4 are repeated until the communications session is finished.

In-depth look at the stages of WS-SecureConversation

To help understand WS-SecureConversation better, in this section we look in-depth at the two stages of WS-SecureConversation. We examine how a security context token is established and the usage of derived keys.

Establishing a security context token

Figure 10-10 shows how the messages are exchanged between the initiator and the recipient to establish the security context token in WebSphere Application Server.
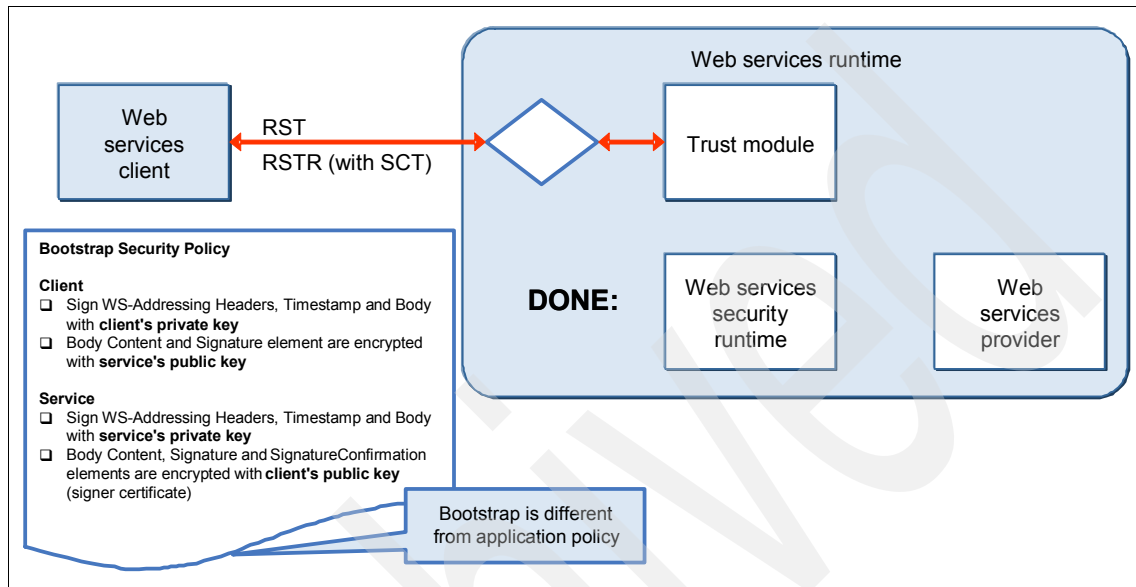
Figure 10-10   Establishing a security context token between the initiator and recipient

The Web services client sends an RST for a security context token to an application endpoint. It uses its private key to sign the WS-Addressing header, timestamp, and body. The body content and signature elements are encrypted by using the trust service's public key.

The trust service that processed the request is trusted based on the bootstrap policy. After the request is validated, the trust service returns the security context token by using RSTR. The trust service uses its private key to sign the WS-Addressing header, timestamp, and body. The body content and signature elements are encrypted by using the client's public key. This process uses the asymmetric cryptography algorithm.

Note: The bootstrap policy, which is used to secure the RST and validate the RSTR request, is different from the application security policy.

Using a derived key

After the security context token is established, the application messages are secured by using a derived key based on the security context token, as illustrated in Figure 10-11.
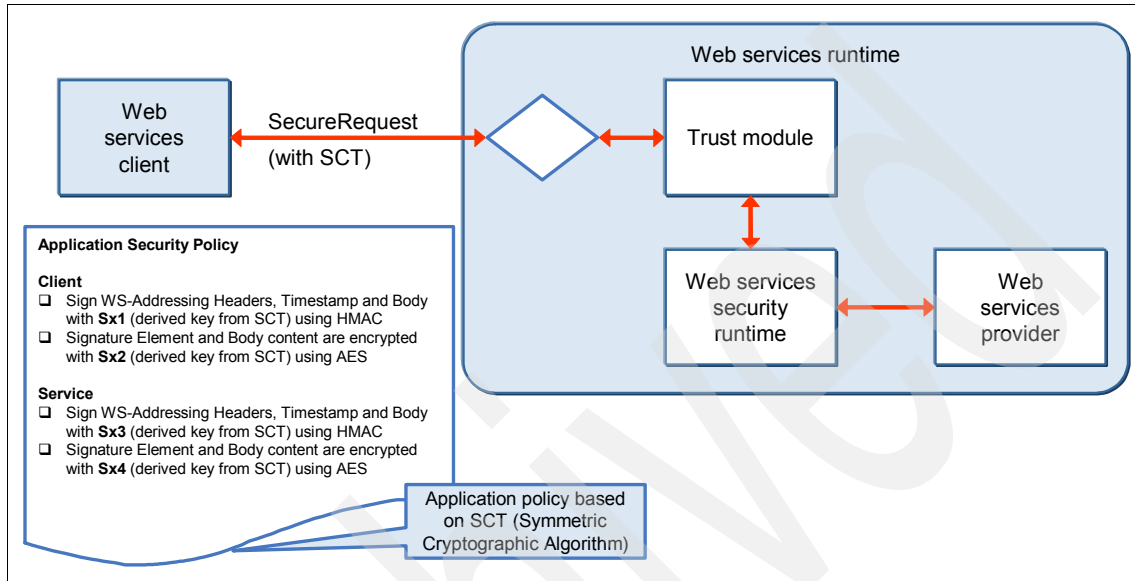

Figure 10-11   Using a derived key to secure the application messages

The derived keys are used to secure the application messages by signing and encrypting the application messages. The security context token contains a Universally Unique Identifier (UUID), which is used as the identification of a shared secret. The token UUID can be used in the SOAP message to identify the security context token for the message exchanges. The secret must be kept in memory by the session participants (in this case, the initiator and the recipient) and protected. Compromising the secret undermines the secure conversation between the participants.

With the derived key, the client and the service can use the symmetric cryptography algorithm to communicate, which is more efficient than the asymmetric cryptography algorithm.

Exploring the trust service configuration

In Chapter 6, "Policy sets" on page 261, we explain how to use a policy set to apply a group of policies to your Web services. The trust service provided by WebSphere Application Server is a security token service that can issue, cancel, renew, and validate security tokens. Because the trust service is a Web service,

you can also configure the security policies against the trust service. For the trust service, you must use a special class of policy sets, known as *system policy sets*. The main difference between a system policy set and an application policy set is that a system policy set is not available for application resources.

To view the system policy sets in the administrative console, expand **Services** → **Policy sets** → **System policy sets**. The default system policy sets that ship with WebSphere Application Server are displayed (Figure 10-12).
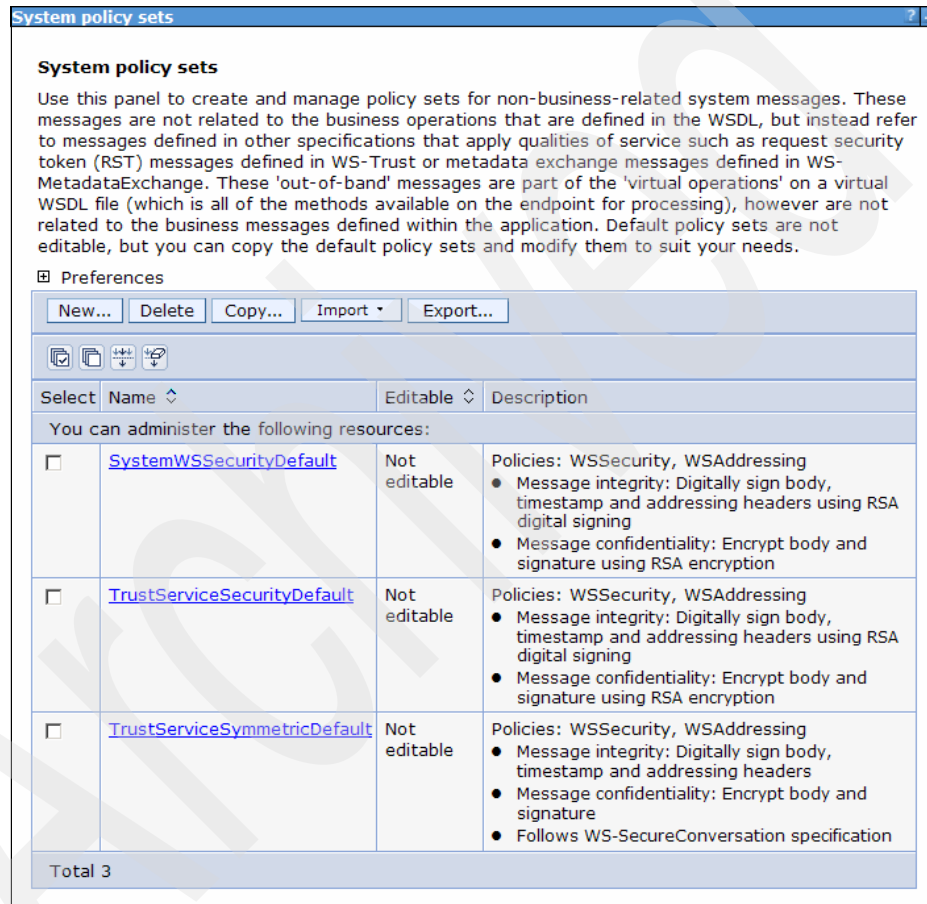
Figure 10-12   System policy set

WebSphere Application Server provides three system policy sets for the security trust service:

► SystemWSSecurityDefault
► TrustServiceSecurityDefault
► TrustServiceSymmetricDefault

Figure 10-12 on page 492 describes each of these default policy sets.

You can create your own custom system policy set and apply it to the trust service instead of using the default system policy set. For example, WebSphere is configured to use the security token reference by default. Windows Communication Foundation (WCF) is configured to expect a key identifier (KeyID) reference for signed messages by default. If your application is running on WebSphere and it needs to interoperate with WCF by using secure conversation, you might have to create a custom system policy set to use a key identifier.

To begin exploring the trust service configuration, click **Services** → **Trust service** → **Token providers** → **Security Context Token** to open the Security context token information (Figure 10-13).

Figure 10-13   Security context token provider configuration page

To access the page to configure trust service endpoint targets (Figure 10-14), click **Services** → **Trust service** → **Targets**.

Figure 10-14   Trust service endpoint targets configuration page

The trust service manages tokens on behalf of service endpoints. A token provider is either explicitly or implicitly associated with each service endpoint. A specific token can be explicitly assigned to be issued when access to an endpoint is requested. Otherwise, the trust service default token is issued.

To open the trust service attachments page (Figure 10-15), click **Services** → **Trust service** → **Trust service attachments**.

Figure 10-15   Trust service attachments page

Each new endpoint that is specified initially has four operations:

►   Issue.
►   Renew.
►   Cancel.
►   Validate.

By default, all endpoints inherit the policy set and binding that are attached to the respective trust service operation under Trust Service Defaults. However, you can explicitly attach a different policy set.

10.3.4 Secure conversation with reliable messaging scenario

Reliable messaging uses a message sequence to deliver a set of messages. WS-Security secures the Web services application messages, but it does not secure the sequence. This exposes the danger for a possible sequence attack.

In the secure conversation with reliable messaging scenario, the security context token is used to secure the reliable messaging sequence. Figure 10-16 shows the message flows that are required to establish a security context token to secure reliable messaging.
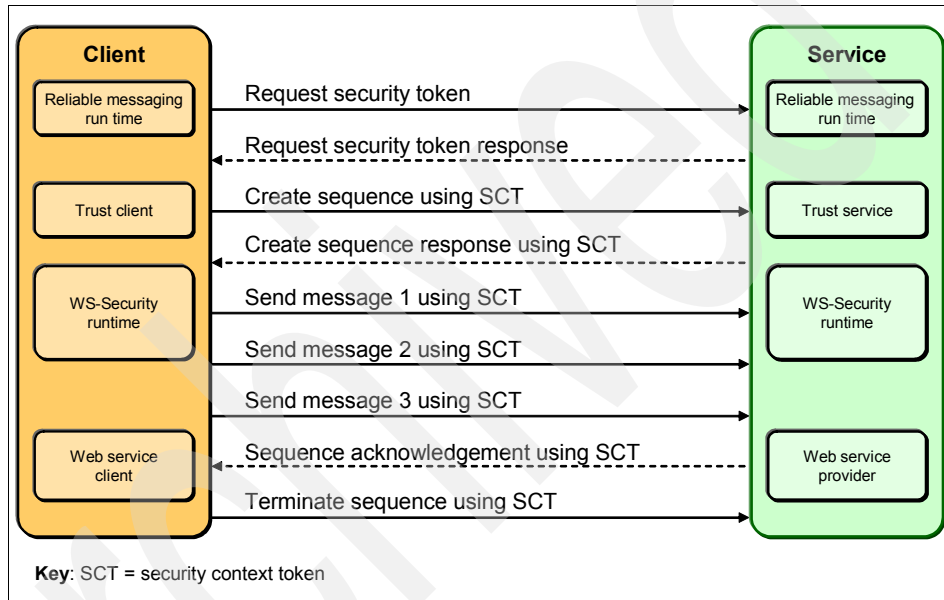
Figure 10-16   Message overflow for secure conversation with reliable messaging

To use secure conversation with reliable messaging, the following steps are involved:

1. The WS-ReliableMessaging run time calls WS-Security APIs to get the UUID of the security context token (SCT) for the session. If a security context token is already established, the UUID of the existing security context token is returned to WS-ReliableMessaging. If no security context token is established, the WS-Security run time initiates a call to the recipient to establish the security context token.

2. After the WS-ReliableMessaging run time acquires the UUID of the security context token, the WS-ReliableMessaging run time scopes the CreateSequence message to the security context token by using the security

token reference argument in the CreateSequence message and responds with the CreateSequenceResponse message.

3. The exchange of the application messages is similar to the WS-SecureConversation scenario. The messages are secured by the security context token.

4. The WS-ReliableMessaging run time responds with the CreateSequenceResponse message.

5. The WS-ReliableMessaging run time sends a SequenceAcknowledgement message to acknowledge that the message is properly delivered and the message is secured by the security context token.

6. The WS-ReliableMessaging run time sends a TerminateSequence message to terminate the sequence, and the message is secured by the security context token.

10.4 Secure conversation example

In this section we provide two examples of how to apply secure conversation to Web services applications by using Rational Application Developer. In the first example, we apply WS-SecureConversation to the Web service. In the second example, we apply the WS-I Reliable Secure Profile (RSP) to our Web services. WS-I RSP is an interoperability profile to deal with secure, reliable messaging capabilities for Web services. We also monitor the SOAP traffic to see the message flows of secure conversation.

10.4.1 Applying secure conversation to Web services

The first example is to apply secure conversation to the WeatherJavaBean Web services. WebSphere Application Server V7 ships with the Username SecureConversation policy set, which is ready for immediate use. For our example, we do not need the username token for message authentication. To see the SOAP traffic, we create a custom policy set to include the HTTPTransport policy. We use TCP/IP Monitor (TCPMon), which ships with WebSphere Application Server, to monitor the SOAP traffic.

> **TCP/IP Monitor**: The TCP/IP Monitor that ships with Rational Application Developer is not capable of handling WS-SecureConversation-related traffic, which includes multiple messages and SOAP envelopes in requests and responses. For this reason, we use TCPMon that ships with WebSphere Application Server as the monitoring tool.
>
> To use the TCPMon to monitor the WS-SecureConversation-related traffic, we must also add the HTTPTransport policy. We add this policy because the proxy capability provided by HTTPTransport policy offers a convenient mechanism to configure TCPMon to capture both the application request/response and the trust service request/response.

Preparing for the example

> **Downloadable material**: The examples in this chapter use the WeatherJavaBean application. This application is included in the download material for this book in the `WeatherBase/WeatherWebService.zip` archive.
>
> The project interchange file contains the following projects:
>
> ► WeatherBase: contains the core weather classes used by the applications (See 3.1.1, "The WeatherForecast application packages" on page 148.)
>
> ► WeatherJavaBeanServer: the Web service provider application
>
> ► WeatherJavaBeanWebClient: the Web service client application
>
> For information about downloading the material, see Appendix A, "Additional material" on page 537.
>
> For information about importing the application into your workspace, installing it on the server, and testing it, see "Using the WeatherJavaBean application" on page 543

Creating the custom policy set and general binding

In this section we create a custom policy set to include the SecureConversation policy and HTTPTransport policy. We also create a general binding to direct the SOAP request to a proxy server running on port 9088. Then we export the custom policy set and general binding to the local directory to import them into the Rational Application Developer development environment.

To create the custom policy set:

1. Log in to the administrative console.

2. In the left pane, select **Services** → **Policy Sets** → **Application Policy Sets**.

3. Select **Username SecureConversation** (Figure 10-17) and click **Copy** at the top of the page.

| ☑ | Username SecureConversation | Not editable | Policies: WSSecurity, WSAddressing
● Message integrity: Digitally sign body, timestamp, signature confirmation, addressing headers and Username token
● Message confidentiality: Encrypt body, signature, signature confirmation and Username token
● Message authentication: Using Username token
● Follows WS-SecureConversation specification |

Figure 10-17   Copying Username SecureConversation

4. On the Application policy sets page (Figure 10-18), in the Name field, type `ITSO SecureConversation` and click **OK**.

Figure 10-18   Naming the custom policy set

5. Click **ITSO SecureConversation.**

6. Click **Add** and select **HTTP transport** (Figure 10-19). Then click **Apply**.

Figure 10-19   Adding the HTTP transport policy

7. Click **WS-Security** → **Main policy** → **Request token policies**. The user name token authentication is not needed.

8. Select **token_auth**. Click **Delete** and then click **Save**.

9. Navigate back to the list of application policy sets. Select **ITSO SecureConversation Policy Set** and click **Export**.

10. Click **ITSO SecureConversation.zip** and save it to your local directory.

You have now created the custom policy set. To create the general client binding:

1. In the administrative console, expand **Services** → **Policy sets** → **General client policy set bindings**.

2. Select **Client sample** and then click **Copy**.

3. For name, type `ITSO Secure Conversation client binding`. Then click **OK**.

4. Click **ITSO Secure Conversation client binding**, then click **HTTP transport**.

5. On the HTTP transport page (Figure 10-20), for host, type `localhost`. For port, type 9088. Click **OK**, then click **Save**.

Figure 10-20   HTTP transport configuration

We choose port 9088 because usually this port is not used by any process. You can choose any other port as long as it is not used by any process. After you configure the proxy for outbound service requests, the SOAP request from the client is directed to the proxy server running at `http://localhost:9088`.

TCPMon is also started in proxy mode and listens to port 9088. It directs the SOAP request to the Web service provider. Using the proxy capability of the HTTPTransport policy is a convenient mechanism to configure TCPMon to capture both the application request/response and the trust service request/response.

6. Select **ITSO Secure Conversation client binding** and click **Export**. Click **ITSO Secure Conversation client binding.zip** and click **Save** to save the file to your local drive.

7. Restart WebSphere Application Server.

Applying a policy set to a Web service and client

In this section we import the ITSO SecureConversation policy set into the Rational Application Developer development environment. Then we apply the ITSO SecureConversation policy set to the Web service and client.

To import the policy set:

1. In Rational Application Developer, click **File** → **Import** → **Web services** → **WebSphere Policy Sets**. Click **Next**.

2. Click **Browse** and select **ITSO SecureConversation.zip**, which you created in "Creating the custom policy set and general binding" on page 497. Then click **Finish**.

3. From the main menu, click **File** → **Import** → **Web services** → **WebSphere Named Bindings**. Click **Next**.

4. Click **Browse** and select **ITSO Secure Conversation client binding.zip**, which you created in "Creating the custom policy set and general binding" on page 497. Then click **Finish**.

5. Verify that the policy set and the general binding imported successfully:

   a. Click **File** → **Preferences**.

   b. In the left pane of the Preferences window (Figure 10-21), select **Service Policies**. In the right pane you should see that the ITSO SecureConversation policy set and ITSO Secure Conversation client binding are imported.
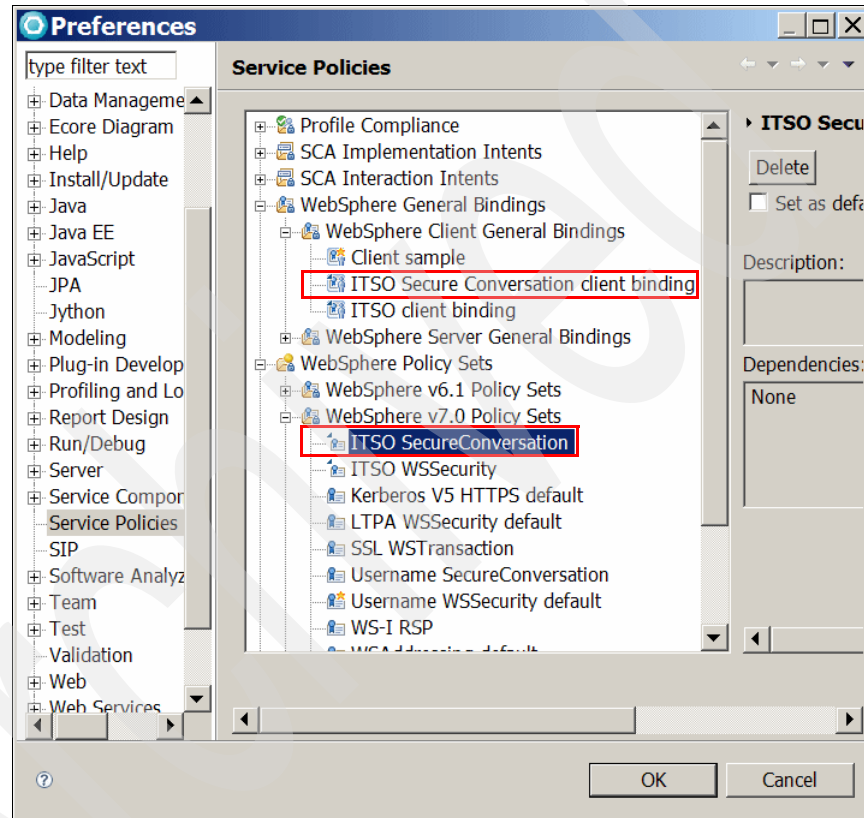
Figure 10-21   Importing the ITSO SecureConversation policy set and binding

6. In the Services view, expand **JAX-WS** → **Services**. Right-click **WeatherJavaBeanService** and select **Manage policy set Attachment**.

7. In the Add Policy Set Attachment to Service window, select the **WeatherJavaBeanServer** application from the drop-down list if it is not already selected. Click **Add** under the table inside the Application group.

8. In the End Point Definition Dialog window (Figure 10-22), for policy set select **ITSO SecureConversation**, and for binding leave **Provider sample**. Click **OK**.
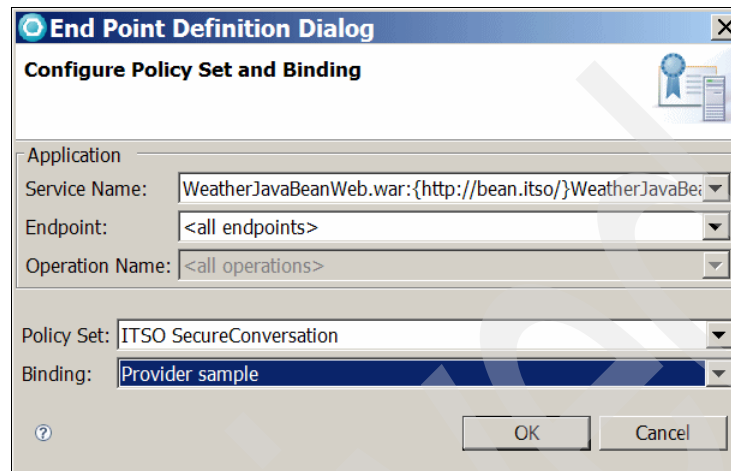
Figure 10-22   Applying the ITSOSecureConversation policy set to the service

9. In the message window that opens, click **Ignore**.

10. Click **Finish**. The ITSO SecureConversation policy set is now applied to your service.

To apply this policy set to your client:

1. In the Services view, expand the **JAX-WS** → **Clients**. Right-click **WeatherJavaBeanService** and select **Manage policy set Attachment**.

2. On the Add Policy Set Attachment to Web service Client page, click **Add**.

3. In the End Point Definition Dialog window (Figure 10-23), for policy set, select
   ITSO SecureConversation. For binding, type `ITSO Secure Conversation
   client binding` and click **OK**.
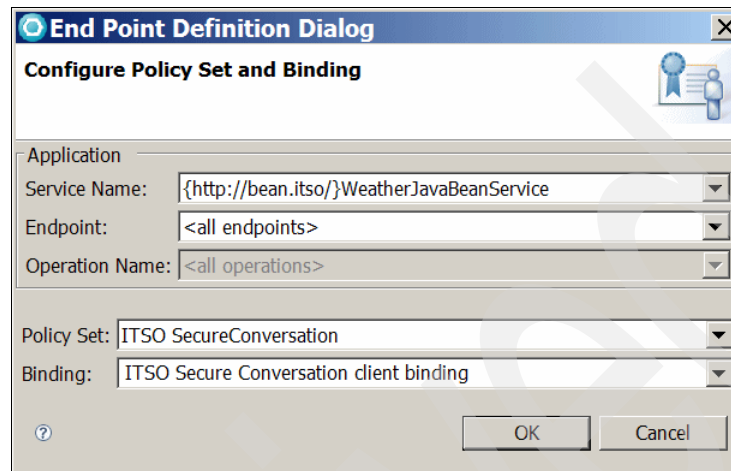
Figure 10-23   Attaching the ITSO SecureConversation policy set to the client

4. In the message window that opens, click **Ignore**.

5. Click **Finish**.

Monitoring SOAP traffic in WS-SecureConversation

As explained earlier, use TCPMon that ships with WebSphere Application Server
to monitor the SOAP traffic, because the TCP/IP Monitor that ships with Rational
Application Developer is not capable of handling
WS-SecureConversation-related traffic.

To monitor the SOAP traffic using TCPMon:

1. Start TCPMon:

   a. Open a command prompt and navigate to the `<%WAS_HOME%>\bin` folder.

   b. Type the commands shown in Example 10-5 to start TCPMon.

Example 10-5   Launching TCPMon

```
C:\Documents and Settings\Administrator>cd \ibm\was\nd\bin
C:\IBM\WAS\ND\bin>setupcmdline
...
C:\IBM\WAS\ND\bin>set
classpath=C:\IBM\WAS\ND\runtimes\com.ibm.ws.webservices.thinclient_7.0.0.jar;%classpath%
```

```
C:\IBM\WAS\ND\bin>java –Djava.ext.dirs=%WAS_EXT_DIRS%
com.ibm.ws.webservices.engine.utils.tcpmon
```

> **Tip:** A simple way to set up TCPMon is to download it from the Apache
> Software Foundation Web site at the following address:
>
> http://ws.apache.org/commons/tcpmon/download.cgi
>
> When you reach this page, select **Binary Distribution**. Download and
> extract the file. Then double-click **tcpmon.bat** in the
> `tcpmon-1.0-bin\build` folder to start TCPMon.
>
> Notice that the TCPMon version from Apache has more functions than the
> version that ships with WebSphere Application Server.

2. In the TCPMonitor window (Figure 10-24), for Listen Port #, type 9088. Select
   Proxy and click **Add**. Port 9088 is now ready for the SOAP traffic.
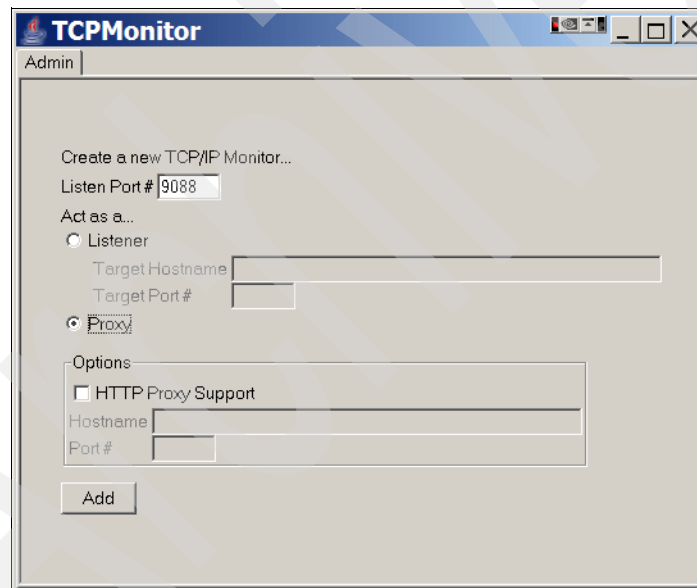
Figure 10-24   TCPMonitor window

3. Open a Web browser and run the sample JSP client. For example, if your
   server is running at port 9080, then type the following URL:

   ```
   http://localhost:9080/WeatherJavaBeanWebClient/sampleWeatherJavaBean
   PortProxy/TestClient.jsp
   ```

4. Click the **getDayForecast** method and enter a value for arg0. You can copy and paste the example value that is given by the JSP client. For example, type 2009-04-10T16:22:19 and click **Invoke**.

   The weather information is shown in the JSP page. The SOAP traffic is displayed in TCPMon, as shown in Figure 10-25.
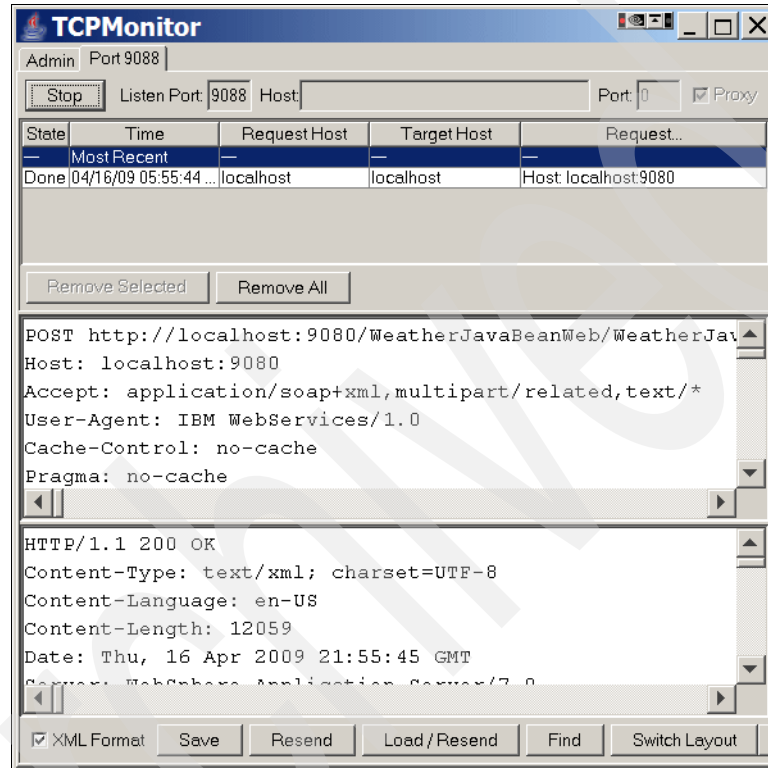
Figure 10-25   SOAP traffic in TCPMon

The SOAP request is shown in the upper pane, and the SOAP response is shown in the lower pane. The request pane has two SOAP requests, and the response pane has two SOAP responses, which reflects the nature of secure conversation. As discussed in 10.3.3, "Secure conversation scenario" on page 488, the secure conversation has two stages:

–   In the first stage, the Web service client sends an RST for a security context token to an application endpoint.

   The RST is encrypted and signed by using WS-Security information that is defined in the bootstrap security policy. The request is transparently rerouted to the trust service. The trust service processes the RST and responds with an RSTR. This response is returned to the requester as

though it were generated by the endpoint service. Because the security context token is encrypted, you might not be able to find anything interesting in the first request/response pair in TCPMon.

– In the second stage, after the security context token is established, the application messages are secured by using a derived key based on the security context token.

In taking a closer look at the second request/response pair, Example 10-6 shows the interesting part of the second SOAP request. Two derived key tokens are used to secure the SOAP message. One is for signing and one is for encrypting.

Example 10-6   SOAP request snippet

```
<wsc:DerivedKeyToken
    xmlns:wsc="http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512"

xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
    wsu:Id="w_29">
    <wsse:SecurityTokenReference>
        <wsse:Reference URI="uuid:3AE92E133DC70A8B1C1239918947523"

ValueType="http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512/sct"></wsse:Reference>
    </wsse:SecurityTokenReference>
    <wsc:Length>16</wsc:Length>
    <wsc:Nonce>hC3DuVY1fHLOsskwSh9Clg==</wsc:Nonce>
</wsc:DerivedKeyToken>
<wsc:DerivedKeyToken
    xmlns:wsc="http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512"

xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
    wsu:Id="w_26">
    <wsse:SecurityTokenReference>
        <wsse:Reference URI="uuid:3AE92E133DC70A8B1C1239918947523"

ValueType="http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512/sct"></wsse:Reference>
    </wsse:SecurityTokenReference>
    <wsc:Length>20</wsc:Length>
    <wsc:Nonce>IUnoIWG9fLDpRlWVRj488Q==</wsc:Nonce>
</wsc:DerivedKeyToken>
```

5. In the JSP client, click **Invoke** again.

The second invocation is much faster than the first one. With the security context token, the client and the service use the symmetric cryptography algorithm to communicate, which is more efficient than the asymmetric cryptography algorithm.

By looking at TCPMon, you can see that the subsequent message invocations use the security context token to encrypt and sign the messages.

10.4.2 Apply secure conversation and reliable messaging

The second example is to apply secure conversation and reliable messaging to Web services. WebSphere Application Server V7 ships with the WS-I RSP policy set that can be used immediately.

This policy set provides the following features:

► Reliable message delivery to the intended receiver by enabling WS-ReliableMessaging

► Message integrity by digital signature that includes signing the body, timestamp, WS-Addressing headers, and WS-ReliableMessaging headers by using the WS-SecureConversationand WS-Security specifications

► Confidentiality by encryption, which includes encrypting the body, signature, and signature confirmation elements, by using the WS-SecureConversation and WS-Security specifications

To see the SOAP traffic, you must create a custom policy set to include the HTTPTransport policy. We use TCPMon that ships with WebSphere Application Server to monitor the SOAP traffic. The steps to apply Secure Conversation and reliable messaging are very similar to those in 10.4.1, "Applying secure conversation to Web services" on page 496. We simply outline the steps here:

1. Remove the policy set that you applied to the WeatherJavaBean sample.

2. Create a custom policy set named `ITSO RSP` to include the WS-I RSP policy and HTTPTransport policy. You can create a copy of the WS-I RSP Policy set and then add the HTTPTransport policy.

3. Create a general binding named `ITSO RSP client binding` and update the HTTP transport to use 9088 as the port.

4. Import the ITSO RSP policy set and the ITSO RSP client binding into the Rational Application Developer development environment.

5. Apply the ITSO RSP policy set to the Web service and client.

6. Monitor the SOAP traffic by using TCPMon.

Next, study the SOAP traffic that is shown in TCPMon:

1. The first SOAP request/response pair shows that the Web service client sends an RST, and the trust service processes the RST and responds with an RSTR. This process is signed and encrypted by WS-Security.

2. In the second SOAP request, after the reliable messaging run time acquires the UUID of the security context token, scopes the CreateSequence message to the security context token by using the security token reference argument.

   `<wsrm:UsesSequenceSTR soap:mustUnderstand='1'/>` forces the reliable messaging destination to ensure that the sequence is secured by using the supplied security token reference. Example 10-7 shows the SOAP request.

Example 10-7   Second SOAP request snippet

```
<c:DerivedKeyToken u:Id="w_31">
   <s:SecurityTokenReference>
      <s:Reference URI="uuid:832FF02B2C7A42A4581239995679380"
         ValueType="http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512/sct" />
   </s:SecurityTokenReference>
   <c:Length>16</c:Length>
   <c:Nonce>q4t6xFhYMaGGeRO7rGSaaQ==</c:Nonce>
</c:DerivedKeyToken>
<c:DerivedKeyToken u:Id="w_28">
   <s:SecurityTokenReference>
      <s:Reference URI="uuid:832FF02B2C7A42A4581239995679380"
         ValueType="http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512/sct" />
   </s:SecurityTokenReference>
   <c:Length>20</c:Length>
   <c:Nonce>UeRO3dHc5rRrKMqhvXflVQ==</c:Nonce>
</c:DerivedKeyToken>
... ...
<wsrm:UsesSequenceSTR xmlns:wsrm="http://docs.oasis-open.org/ws-rx/wsrm/200702"

xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
   soapenv:mustUnderstand="1" wsu:Id="w_26"></wsrm:UsesSequenceSTR>
... ...
<wsa:Action xmlns:wsa="http://www.w3.org/2005/08/addressing"

xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
   wsu:Id="w_25">http://docs.oasis-open.org/ws-rx/wsrm/200702/CreateSequence</wsa:Action>
```

3. The reliable messaging destination responds with the CreateSequenceResponse message. Because the SOAP body is encrypted and signed, you do not see the returned sequence identifier. Example 10-8 shows the SOAP message.

Example 10-8   Second response snippet

```
<wsa:Action

xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
wsu:Id="w_24">http://docs.oasis-open.org/ws-rx/wsrm/200702/CreateSequenceResponse</wsa:Action>

<soapenv:Body

xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
    wsu:Id="w_21">
    <e:EncryptedData xmlns:d="http://www.w3.org/2000/09/xmldsig#"
        xmlns:e="http://www.w3.org/2001/04/xmlenc#"

xmlns:s="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd"
        Id="w_28" Type="http://www.w3.org/2001/04/xmlenc#Content">
        <e:CipherData>
            <e:CipherValue>Sequence identifier is encrypted here</e:CipherValue>
        </e:CipherData>
    </e:EncryptedData>
</soapenv:Body>
</soapenv:Envelope>
```

4. The client sends the message to the service secured by the security context token, as shown in Example 10-9.

Example 10-9   Third request snippet

```
<e:EncryptedData Id="w_30"
    Type="http://www.w3.org/2001/04/xmlenc#Element">
    <e:CipherData> signature is encrypted here<e:CipherValue>
    </e:CipherData>
</e:EncryptedData>
<wsrm:Sequence xmlns:wsrm="http://docs.oasis-open.org/ws-rx/wsrm/200702"

xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility
-1.0.xsd"
    soapenv:mustUnderstand="1" wsu:Id="w_26">
    <wsrm:Identifier>urn:uuid:6383BD25ABF40B4B691239998749544
    </wsrm:Identifier>
    <wsrm:MessageNumber>1</wsrm:MessageNumber>
</wsrm:Sequence>
```

The goal of the RSP policy is to ensure the integrity of the reliable messaging elements. This is accomplished by digitally signing all the WS-ReliableMessaging header elements such as <wsrm:Sequence> and <wsrm:SequenceAcknowledgement>. It is difficult to tell that this has happened from looking at the messages that are sent over the wire because the <wsse:Signature> element has been encrypted. However, one artifact of the signing operation that you can see is the addition of the wsu:Id attribute that is added to the WS-RM header elements. Both the <wsrm:Sequence> elements and the two child elements, <wsrm:Identifier> and <wsrm:MessageNumber>, are signed.

5. The reliable messaging run time sends a SequenceAcknowledgement message to acknowledge that the message is properly delivered and secured by the security context token, as shown in Example 10-10.

Example 10-10   Third response snippet

```
<wsrm:SequenceAcknowledgement
   xmlns:wsrm="http://docs.oasis-open.org/ws-rx/wsrm/200702"

xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility
-1.0.xsd"
   soapenv:mustUnderstand="1" wsu:Id="w_25">
   <wsrm:Identifier>urn:uuid:832FF02B2C7A42A4581239995689759
   </wsrm:Identifier>
   <wsrm:AcknowledgementRange Lower="1" Upper="1"></wsrm:AcknowledgementRange>
</wsrm:SequenceAcknowledgement>
```

Again, you can see that all the WS-ReliableMessaging header elements are digitally signed. The SOAP body is encrypted and signed by the security context token. By signing the WS-ReliableMessaging header and the message body, the sequence of WS-ReliableMessaging is secured.

10.5 More information

You can find the WS-Trust specification on the Web at the following address:

http://docs.oasis-open.org/ws-sx/ws-trust/200512/ws-trust-1.3-os.html

For the WS-SecureConversation specification, go to the following address:

http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512/ws-secure
conversation-1.3-os.html

In addition, see the WebSphere Application Server V7 Information Center at the following address, which offers a tremendous amount of detail about WS-SecureConversation:

http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/index.jsp?topic=/
com.ibm.websphere.express.doc/info/exp/ae/cwbs_wssecureconvstd.html

11

Leading practices for Web services

This chapter introduces approaches that you should consider in order to deliver quality of service for your Web services development, design, and architecture. It contains leading practices to for leveraging your Web services components. These leading practices have been collected from various published documents and advice from leading IBM specialists.

This chapter contains the following topics:

11.1 Web services design best practices

This section discusses basic considerations for designing a Web services solution. It includes high-level guidelines that apply to any development effort, and then discusses technology selection options.

You should also consider design patterns when planning for Web services. For more information see *Patterns: Service-Oriented Architecture and Web Services*, SG24-6303.

11.1.1 Basics of Web services planning

The first step is to perform the basics of design planning. Understand what you have in place today, what your goals are, and what technology you want to use in order to position your applications for future growth. The following list provides a high-level view of the planning tasks that you should perform:

1. Review the standards used in the development and design phase.

   Web services can be based on a variety of Java programming models. Start by identifying how your existing Web services are designed as well as the APIs, standards, and specifications that were part of the design.

2. Identify your goals.

   Consider what you want to accomplish by using Web services. Identify applications and business logic that you want to make available as a service. Consider which existing services you want to migrate to newer technology.

3. Determine how Web services fit into your current topology, applications, and programming model.

   Determine how your current Web services process requests on the server and how the clients manage and use the Web service. Keep these factors in mind when planning for new or migrated Web services.

4. Design your Web services for non-functional requirements to fit your e-business solution.

   In other words, design your Web services for reliability, availability, manageability, and security. For example, you may want your Web services to process a transaction in a reasonable amount of time at all hours of the day and provide users with optimal security, such as authentication mechanism.

   Have a dialogue with your security organization on what business functions should be exposed over the Internet and what precautions should be taken to protect them.

Factor in the performance implications of using a Web service over an EJB call in tightly coupled, high-volume internal applications. Web services might not be the answer in this situation.

5. Decide what development and implementation tools to use.

You can use a variety of manual development and implementation tasks. Whether you have an existing Web service to implement or you want to develop your own from a JavaBeans implementation or from an Enterprise JavaBeans (EJB) module, you can choose different tasks respective to your resources. You can also use assembly tools to complete development and implementation tasks.

6. Select the runtime environment.

You should choose the application server runtime topology that best applies for your architecture based on your functional and non-functional requirements.

11.1.2 When is the use of a Web service an appropriate choice

A critical part of the design phase for an application is determining how to make the function provided by the application available. Exposing the function as a Web service is not always the correct choice. This section discusses when, and when not, to use a Web service.

Do not use Web services between the layers of an application

Try not to use XML-based Web services between the layers of a logical application. Web services function best where they complement other J2EE technologies, not replace them. For instance, one way to use Web services is when connecting an application client running out on the Internet to business logic written in EJBs inside an application server. Here you get a nice, clean separation of communication between the controller and domain layers of your application. This is the same place where you would use EJBs and so, if you consider Web services as another object distribution mechanism, then you can see why this would be appropriate. SOAP over HTTP can work in places where RMI over IIOP cannot, and so this allows the XML-based Web services to complement the existing EJBs architecture.

However, where people often go wrong with this is when they assume that if this works between one pair of layers, it will work well between another. For instance, a common anti-pattern is a design where a persistence layer is wrapped inside an XML API and then placed in a process separate from the business logic that must invoke the persistence layer. In versions of this design, we have seen people serializing Java objects into XML, sending them over the network, deserializing them again, performing database queries with the objects (which

were sent in as an argument), converting the database result set to XML, and then sending the result set back across the network only to be converted into Java objects and finally operated on.

There are several major problems with such an approach:

► Persistent objects should *always* remain local to the business object that operates on them. The overhead of serialization and deserialization is something that you want to avoid whenever possible.

► In EJBs with RMI-IIOP you have the option (although you are not required to) of including persistence operations in an outer transaction scope if you use entity beans or session beans with mapper objects. If you introduce a layer of Web services between the persistent objects and the business objects operating on them, then you lose that ability.

In general, XML Web services are not appropriate for fine-grained interactions, even more so than RMI-IIOP. For instance, do not use a Web service between the view layer and the controller layer of an application. The overhead of the parsing/XML generation and the garbage generation overhead kills the performance of your overall application.

When to use Web services between application servers
Be very careful when using Web services between application servers. In many ways, interoperability between systems is the main reason for applying Web services. Therefore, if you are connecting to a system written using Microsoft .NET, the use of Web services is almost a given. Even though you could use other mechanisms like WebSphere Application Server's COM support, the best solution for interoperability going forward for both the Microsoft and IBM platforms is probably Web services.

Sometimes it makes sense to use Web services when connecting disparate Java application servers from different vendors, but this is a less common occurrence. It is possible, for instance, to connect to EJBs written in WebSphere from a JBoss® or WebLogic server by using the WebSphere Thin Application Client for WebSphere Application Server V7. This would be a much better performing solution than one using HTTP over SOAP-based Web services.

A more common occurrence is when you want asynchronous invocation of business logic written either in another application server or in an existing enterprise information system. In this case, sending XML over JMS makes a lot of sense, and if you wrap your document-oriented XML in a SOAP envelope then you can take advantage of the header structure of SOAP and even possibly gain some out-of-the box features like WS-Security support.

11.1.3 JAX-WS versus JAX RPC

JAX-WS is the next-generation Web services programming model extending the foundation provided by the JAX-RPC programming model. Using the strategic JAX-WS programming model, the development of Web services is simplified through the support of a standards-based annotations model. Although the JAX-RPC programming model is still supported by WebSphere Application Server V7, you should take advantage of the easy-to-implement JAX-WS programming model to develop new Web services applications. Give some thought also to re-writing existing JAX-RPC applications to take advantage of the features of the JAX-WS programming model.

11.1.4 When to use JavaBeans or EJB as provider implementation

You must choose to use either JavaBeans or EJB components for your Web service provider's SEI. This decision is no different from when you are architecting other Java EE applications. If your solution does not require support for transactions, security, and the management that are enabled through the use of EJB components and the EJB container, then JavaBeans can suffice and provide better performance.

If you are using EJB components and deploying them locally within the same JVM as the SOAP engine, then ensure that you deploy them such that they are called using pass by reference. By enabling pass by reference, the parameters of the method are not copied to the stack with every remote call, which can be expensive. Enabling pass by reference can improve performance up to 50%, when the SOAP engine (EJB client) and the Web service provider (EJB Server) are installed in the same application server instance and remote interfaces are used.

11.1.5 Considerations when using SOAP over JMS transport

You can use SOAP over Java Message Service (JMS) transport protocol as an alternative to SOAP over HTTP for communicating SOAP messages between clients and servers. When using SOAP/JMS, it is a best practice to use the industry standard SOAP/JMS protocol. The IBM proprietary SOAP/JMS protocol has been deprecated with this release. However, if your application must interoperate with previous versions of the product, use the proprietary protocol.

If you use the industry standard SOAP over JMS protocol, then use JMS bindings to specify an endpoint URL prefix that adheres to the JMS endpoint URI syntax that is associated with the standard (Example 11-1).

Example 11-1   JMS endpoint URI syntax sample

```
jms:jndi:jms/StockQuote_Q&jndiConnectionFactoryName=jms/StockQuote_CF
```

If you use the IBM proprietary SOAP over JMS protocol, use the JMS bindings to specify a JMS endpoint URL prefix that adheres to the IBM proprietary SOAP over JMS protocol (Example 11-2).

Example 11-2   JMS endpoint URI syntax sample

```
jms:/queue?destination=jms/StockQuote_Q&connectionFactory=jms/StockQuote_CF
```

In addition, It is important to mention that SOAP over JMS is not compliant to WS-I Basic Profiles 1.1, whereas SOAP over HTTP is.

More Information: For an example of using SOAP over JMS see 4.4.1, "Creating an EJB Web service" on page 198.

11.2 Leading practices for developing Web services

This section discusses leading practices to incorporate into the development cycle for Web services.

11.2.1 Common best practices

Basic common practices that you must always consider when developing your Web services are:

► Use simple data types.

Even though Web services were designed with interoperability in mind, it is a good practice to use simple data types where possible. By simple, we mean integers and strings. Compound types (similar to Struts in C++) and arrays of simple types are also considered simple data types. Anything that does not fall into this pattern should be used carefully. In particular, the Java collection classes and similarly complex data types should be avoided altogether because there might be no proper counterparts at the client side. While JAX-WS and JAXB have made it easier to map XSD types to Java types,

doing so will still require someone who understands those types on the other end of the pipe.

► Avoid nillable primitives.

Nillable primitive types (indicating that an element can be null) are allowed for Web services, but there are interoperability issues when using them. The best advice is not use them at all, and use dedicated flags to control the condition that a value does not exist.

► Avoid fine-grained Web services.

Web services use a simple, but powerful format to exchange data using the SOAP protocol: XML. While reading and structuring XML documents with a simple text editor eases the use of SOAP, the process of automatically creating and interpreting XML documents is more complex. Without careful design, you can end up in a situation where the complexity of dealing with the SOAP protocol has a higher performance cost than performing the actual computation.

Design course-grained Web services that perform more complex business logic. This allows the Web service to return more data in response to a single request, rather than having multiple requests to retrieve smaller portions of data. Working with coarser grained services also allows a single service to be reused, instead of creating multiple fine-grained services.

► Avoid Web services for intra-application communication.

Intra-application communication (that is, communication within an application) is generally not exposed to any third-party clients. Therefore, it is not necessary to allow for an interoperable interface in this case. However, try to take into consideration that this might change in the future.

► Use short attribute, property, and tag names.

Because each attribute, property, and tag name is transmitted verbatim, the length of a message is directly dependent on the length on the attribute and property names. The general guideline is that the shorter the attribute, property, and tag names are, the shorter the transmitted message and the faster the communication and processing.

► Avoid deep nesting of XML structures.

Because parsing of deeply nested XML structures increases the processing time, deeply nested compound data types should be avoided. This also increases comprehension time of the data type itself.

► Apply common sense (also known as being defensive).

If a standard or specification is not clear enough, try to implement your Web service such that it can handle any of the interpretations that you can think of.

► Use Web services caching as provided by the platform.

WebSphere Application Server provides an excellent caching framework that allows for caching of information at various levels. This framework allows you to cache Web service requests, and thus save processing time.

► Minimize parsing of XML data.

If a business function is to be exposed as an XML Web service that leverages SOAP for both internal consumption and for external consumption by business partners, intermediaries such as gateways or service agents should avoid or minimize the parsing of the SOAP Body element.

If a gateway component is used but no network transport or message manipulation is required (such as SOAP/HTTP to RMI/IIOP), then the gateway should not perform parsing of the SOAP body.

Service agents, however, often rely on business context information within the SOAP body, such as partner IDs, transaction correlators, message IDs, and authorization codes in order to provide their system with management capabilities. Using the business context, the service agents provide statistics on business events, enforce business policies, and route requests to meet quality of service commitments. In this case, do no more parsing than necessary.

11.2.2 JAX-WS best practices

This section provides some of the leading practices to be applied during the development phase for a Web services application based on the new version of JAX-WS 2.1 supported by JSR-224 specification.

Use existing samples that implement best practices

When developing a Web service application, it is useful to use existing samples known to illustrate best practices as a starting point.

WebSphere Application Server provides Web services sample applications. These are stored in the following directory:

`WAS_HOME/profiles/server_name/samples/src/WebServicesSamples`

IBM developerWorks also has a Web services page that contains links to Web services information, including community groups, learning resources, and articles on Web services. You can find this Web page at:

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

Considerations for the bottom-up approach

We recommended that you generate your Web service Java code from a well-defined WSDL mapping as dictated by the top-down technique. However, sometimes it makes sense to use the bottom-up approach to annotate your Java beans and, thereby, customize your WSDL file.

The following are leading practices for using the bottom-up approach.

Use JAX-WS annotations for Java to WSDL mapping

Use the following annotations to perform Java to WSDL mapping customizations:

► Use the operationName property of the @WebMethod to customize the WSDL operation name.

► Use the @WebParam annotation to customize the mapping of a parameter to a Web service message part and XML element.

Example 11-3 shows an example of using the @WebParam annotation to use these annotations.

Example 11-3   Using the WebMethod and WebParam annotations

```
@WebService
public class Calculator {
    @WebMethod(operationName="Add")
    public int add(@WebParam(name="param1")int a,
@WebParam(name="param2")int b) {

        …
    }
}
```

In addition, consider the following implications of using the bottom-up mechanism:

► Inheritance

  – All public methods on inherited classes with @WebService will be included as operations in the base service.

  – Use the @WebMethod(exclude=true) to remove methods from the service.

► Void operations

  The void operations must be marked as one way (@OneWay) to ensure one-way semantics. Otherwise, they will be treated as two-way operation with empty return.

Use wrapper styles for request and response objects

Two styles are available for defining request and response objects:

- ► Wrapper stye
- ► Non-wrapper style

Wrapper style is the default and that is what we recommend that you use in most circumstances.

With wrapper style, the @RequestWrapper annotation supplies the JAXB-generated request wrapper bean, the element name, and the namespace for serialization and deserialization with the request wrapper bean that is used at run time. When starting with a Java object, this element is used to resolve overloading conflicts in document literal mode. Only the className attribute is required in this case.

Example 11-4   Wrapper style sample

```
@WebService
public class Calculator {
   @RequestWrapper(className="sample.Add")
   @ResponseWrapper(className="sample.AddResponse")
   public int add(int a, int b) {
      ...
   }
}
```

For more information, see "Which style of WSDL should I use?" at:

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

Use JAX-WS annotations to manage life cycle

Use the @PostConstruct and @PreDestroy annotations (Example 11-5) to help you and your team understand what must be processed when your Web services are initialized and destroyed.

Example 11-5   Example of @PostConstruct and @PreDestroy annotations

```
@Stateless
@WebService
public class HelloServiceBean {
   private String message = "Hello, ";
   @WebMethod
      public String sayHello(String name) {
         return message + name + ".";
      }
   @PostConstruct
```

```
    void init() {
    //....
    }
@PreDestroy
    void release() {
        //....
    }
}
```

The @PostConstruct method is called by the container before the implementing
class begins responding to Web service clients. The @PreDestroy method is, on
the other hand, called by the container before the endpoint is removed from
operation.

Web service client static and dynamic APIs

The JAX-WS Web service client programming model supports both the Dispatch
client API and the Dynamic Proxy client API. The Dispatch client API is a
dynamic client programming model, whereas the static client programming
model for JAX-WS is the Dynamic Proxy client.

- ► Use the Dispatch client when you want to work at the XML message level or
  when you want to work without any generated artifacts at the JAX-WS level.

- ► Use the Dynamic Proxy client when you want to invoke a Web service based
  on a service endpoint interface.

Dynamic proxy client API

The following considerations are relevant for static API usage:

- ► Reuse proxy instances.

  Web services applications that use the static API do not need to access the
  WSDL artifact at run time. The binding itself is performed statically. Therefore,
  consider re-using proxy instances when possible. It is expensive to create
  them because of metadata processing and potential QoS initialization.

- ► Create or generate an SEI for usage on client and server.

  Create or generate a service endpoint interface for use on the client and
  server side. This approach allows for greater symmetry between client and
  server code.

Dispatch client API

The Dispatch API is intended for advanced XML developers who prefer using XML constructs at the java.lang.transform.Source or javax.xml.soap.SOAPMessage level. For convenience, the use of the Dispatch with JAXB data binding object is supported. Consider the following recommendations when using the dispatch API:

► Use the PAYLOAD mode unless sending SOAP headers.

   The Dispatch client can send data in either MESSAGE or PAYLOAD mode.

   – When using the javax.xml.ws.Service.Mode.MESSAGE mode, the Dispatch client is responsible for providing the entire SOAP envelope including the <soap:Envelope>, <soap:Header>, and <soap:Body> elements.

   – When using the javax.xml.ws.Service.Mode.PAYLOAD mode, the dispatch client is only responsible for providing the contents of the <soap:Body>. JAX-WS includes the payload in a <soap:Envelope> element. The PAYLOAD approach allows the run time to do less parsing.

   When using a SOAP protocol binding, a Web services client application should work with the contents of the SOAP Body (PAYLOAD mode) rather than the SOAP messages (MESSAGE mode) as a whole.

   Figure 11-1 shows an example of a SOAP envelope. In this figure:

   – When using PAYLOAD mode, the parameter is the <soap:body> content: <hello:greeting>.

   – When using MESSAGE mode, the parameter is the entire SOAP message (XML source).
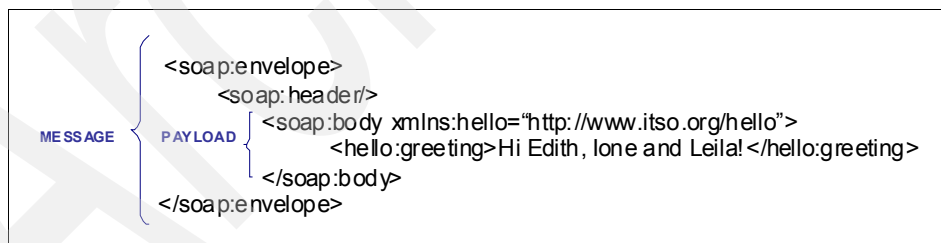

Figure 11-1   SOAP envelope

For further information see 2.1.4, "Web service clients" on page 89.

The following WebSphere Information Center article also contains additional relevant information about Dispatch clients:

http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/index.jsp?topic=/com
.ibm.websphere.nd.multiplatform.doc/info/ae/ae/cwbs_jaxwsclients.html

Invoke Web services asynchronously when appropriate

An asynchronous invocation of a Web service sends a request to the service endpoint and then immediately returns control to the client program without waiting for the response to return from the service. JAX-WS asynchronous Web service clients consume Web services using either the callback approach or the polling approach.

Consider using asynchronous calls for your requests when appropriate. Be sure to examine the length of the target service invocation, adjust client timeout, and handle errors appropriately in order to avoiding losing control of the flow of your application. The following list contains additional information about asynchronous calls that you must know:

► Asynchronous calls require WS-Addressing.

► Asynchronous communication can be enabled using the com.ibm.websphere.webservices.enable.async.mep property on the client. For example:

```
// Use Proxy Instance as BindingProvider
BindingProvider bp = (BindingProvider) port;
Map<String, Object> rc = bp.getRequestContext();
rc.put("com.ibm.websphere.webservices.enable.async.mep",
Boolean.TRUE);
// invoke the operation asynchronously
```

► For client asynchronous listeners that start on demand, the listening port can be secured using a policy set.

► The CallBack model uses Java 5 Executor for response threads. The default Executor has a configurable thread pool.

► With the polling model, the polling request is not sent automatically to the service. It must be controlled (sent) manually.

For more information see 2.1.4, "Web service clients" on page 89.

The "JAX-WS Asynchronous client" WebSphere Information Center article contains additional relevant information:

http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/index.jsp?topic=/
com.ibm.websphere.express.doc/info/exp/ae/twbs_jaxwsclientasync.html

WSDL binding customization

The JAX-WS specification defines standard XML-based customization for WSDL to Java mapping. These customizations, or binding declarations, can customize almost all WSDL components that can be mapped to Java, such as the service endpoint interface class, method name, parameter name, exception class, and

so on. The following list provides recommendations for the binding customizations:

- ► Using WSDL document/literal style

  Use document/literal wrapped WSDL style for the greatest level of interoperability.

  Although JAX-WS 2.1 supports both the RPC and the document style, it defaults to the document style. The default document style is the recommended style because it enforces strictly typed payloads. The entire payload content, using the document style, adheres to the XML schema types declared in the WSDL document's types section (either inlined schema or externally referenced schema).

  The primary purpose of the document/literal wrapped pattern is to make document/literal-generated SEI code look like RPC style code. Among other things, the pattern ensures that the SOAP messages always contain a top-level element that wraps the actual input/output.

  For request messages, the wrapped pattern ensures that the top-level element has the same name as the operation being invoked. For response messages the wrapper element is the operation name appended with *Response*.

  In contrast to the RPC style, the element is part of the well-defined XML schema declared in the WSDL documents types section (either inlined or external document).

- ► Using binding customizations for JAX-WS

  Consider using the binding customizations in order to resolve mapping differences. Certain industry schemas are not consumable by JAX-WS tooling without it. Example 11-6 shows an example of binding customizations (in bold).

Example 11-6   Binding customizations

```
<wsdl:portType name="StockQuoteUpdater">
   <wsdl:operation name="setLastTradePrice">
     <wsdl:input message="tns:setLastTradePrice"/>
     <wsdl:output message="tns:setLastTradePriceResponse"/>
     <jaxws:bindings>
       <jaxws:method name="updatePrice"/>
     </jaxws:bindings>
   </wsdl:operation>
   <jaxws:bindings>
     <jaxws:enableAsyncMapping>true</jaxws:enableAsyncMapping>
```

```
        </jaxws:bindings>
</wsdl:portType>
```

► JAX-WS and JAXB binding customizations

JAX-WS binding customization files can contain JAXB customizations. The customizations available include:

– Package to namespace mapping.
– Enable wrapper style.
– Enable async mapping.
– SEI name mapping.
– Method/parameter name mapping.

► Using separate files for bindings

Use separate files for bindings. Do not inline bindings in WSDL documents. Leave the WSDL implementation agnostic. Example 11-7 illustrates how to generate JAX-WS artifacts for sample.wsdl and uses the customization file sample.xml (jax-ws customization file) in the process.

Example 11-7   wsimport using separate files for WSDL customization

```
c:\> wsimport sample.wsdl –b sample.xml
```

The -b option specifies the external JAX-WS or JAXB binding files. You can specify multiple JAX-WS and JAXB binding files with this option. However, each file must be specified with its own -b option.

Additional information

The following WebSphere Information Center articles contain additional relevant information:

► Using separate files for bindings

http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/index.jsp?topic=/
com.ibm.websphere.nd.multiplatform.doc/info/ae/ae/twbs_jaxwsclientfrom
wsdl.html

► Generating JAX-WS artifacts for JAX-WS applications from WSDFL file

http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/index.jsp?topic=
/com.ibm.websphere.nd.multiplatform.doc/info/ae/ae/twbs_jaxwsfromwsdl.
html

► General sample bindings for JAX-WS applications

http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/index.jsp?topic
=/com.ibm.websphere.nd.doc/info/ae/ae/cwbs_defaultconfigjaxws.html

Command-line tools

WebSphere provides JAX-WS and JAXB command-line tools to help you to generate and import your Web services artifacts. The wsimport, wsgen, schemagen, and xjc command-line tools are located in the *WAS_HOME*\bin\ directory of your WebSphere's installation.

While the wsimport and wsgen tools are also provided by the JDK, WebSphere Application Server provides its own version of these tools. For the most part, artifacts generated by the tools that come with WebSphere and the JDK are the same. In general, artifacts generated by the JDK tools are portable across compliant runtime environments. However, it is best to use the WebSphere tools to achieve seamless integration within the WebSphere environment. Note that the wsimport, wsgen, schemagen, and xjc command-line tools are not supported on the z/OS® platform. This functionality is provided by the assembly tools provided with WebSphere Application Server running on the z/OS platform.

This following list provides leading practices for developing Web services using these tools:

► Generate beans from the WSDL schema, then writes the service class.

Use the JAXB tools to generate Java classes from an XML schema with the xjc schema compiler tool. This tool allows explicit control over changes to the schema and still makes data objects round-trip capable.

You can generate fully annotated Java classes from an XML schema file by using the JAXB schema compiler, xjc command-line tool. Use the xjc schema compiler tool to start with an XML schema definition (XSD) to create a set of JavaBeans that map to the elements and types defined in the XSD schema.

Once the mapping between XML schema and Java classes exists, XML instance documents can be converted to and from Java objects through the use of the JAXB binding runtime API. The resulting annotated Java classes contain all the necessary information that the JAXB run time requires to parse the XML for marshaling and unmarshaling. You can use the resulting JAXB classes within Java API for XML Web services (JAX-WS) applications or in your non-JAX-WS Java applications for processing XML data.

► Use the `wsimport` command for JAX-WS applications.

Use the top-down development approach whenever possible. The top-down approach provides the greatest amount of metadata to the run time.

The `wsimport` command-line tool processes an existing WSDL file and generates the required artifacts for developing JAX-WS Web service applications. Using `wsimport` provides strict conformance to WS-I BP 1.1. Operation wrappers are generated for the document/literal wrapped pattern.

When using `wsimport`, you must specify the –keep flag to generate the source.

► Use wsgen to generate the necessary artifacts for JAX-WS applications.

When using a bottom-up approach to develop JAX-WS Web services and you are starting from a service endpoint implementation, use the `wsgen` command-line tool to generate the required JAX-WS artifacts.

Additional information

For further information related to command-line tools see 4.1.1, "Web services development tools" on page 162.

The following WebSphere Information Center articles contain additional relevant information:

► "Using JAXB xjc tooling to generate JAXB classes from an XML schema file"

http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/index.jsp?topic
=/com.ibm.websphere.zseries.doc/info/zseries/ae/twbs_jaxbschema2java.
html

► "Using wsgen command for JAX-WS applications"

http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/index.jsp?topic
=/com.ibm.websphere.nd.iseries.doc/info/iseriesnd/ae/rwbs_wsgen.html

► "Command tools (wsimport and wsgen) for JAX-WS applications"

http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/topic/com.ibm.
websphere.nd.multiplatform.doc/info/ae/ae/rwbs_wsimport.html

Use MTOM for attachments larger than 30 k

For small binary data, it is more efficient to base64 encode the data. However, after a certain threshold it becomes more efficient to use MTOM as a serialization mechanism. In WebSphere, that threshold is typically around 30 k in attachment size. Therefore, anything larger than that size is typically more efficient to send via MTOM. Note that WebSphere streams from in-memory to disk at 100 k in order to efficiently handle large attachments.

The following examples show how to enable MTOM. Example 11-8 shows how to enable MTOM on the client side.

Example 11-8   Client code enabling MTOM

```
Dispatch d = … create a Dispatch instance
BindingProvider bp = d.getBindingProvider();
SOAPBinding soapBnd = (SOAPBinding) bp.getBinding();
soapBnd.setMTOMEnabled(true);
```

Example 11-9 shows how to enable MTOM on the server side.

Example 11-9   Annotated server-side code to enable MTOM

```
@WebService
@BindingType(SOAPBinding.SOAP11HTTP_MTOM_BINDING)
   public class MTOMService {
   …
   }
}
```

For further information see 2.1.6, "Handling binary content" on page 110.

Considerations when using SOAP 1.2

This section contains leading practices to consider when designing your Web services for SOAP 1.2:

► Configure the service provider implementation.

   The service provider implementation is configured on endpoint implementations using the @BindingType annotation (like MTOM). The interfaces model the PortType element, which remains protocol agnostic. However, without @BindingType, the service defaults to SOAP 1.1.

> **Note**: JAX-WS requires that run times *not* generate WSDL for endpoints that use SOAP 1.2. Therefore, a WSDL file cannot be accessed using a URL that ends with ?WSDL when the endpoints specify a SOAP 1.2 binding but do not provide an explicit WSDL file.

► Use an explicit WSDL file in the client implementation with SOAP 1.2.

The client dispatch implementation must specify SOAP 1.2 during its creation. A dynamic proxy client requires that WSDL be present to determine SOAP 1.2. Without WSDL, the proxy uses SOAP 1.1.

► Special considerations for wsgen with SOAP 1.2.

The wsgen generation using WSDL with SOAP 1.2 bindings requires the -extension flag to be used. Note that extension-generated artifacts are not guaranteed to be portable (or migratable in the future).

To generate WSDL with SOAP 1.2 bindings, two additional parameters are required on the `wsgen` command, as shown in Example 11-10:

– -extension
– –wsdl:Xsoap1.2

Example 11-10   wsgen with the -extension parameter

```
c:\> wsgen -classpath . example.Stock -wsdl:Xsoap1.2 –extension
```

Even with an @BindingType annotation, wsgen options are required. Example 11-11 shows an example of the error that you receive when you do not specify these parameters.

Example 11-11   Error

```
c:\> wsgen -classpath . example.Stock -wsdl
error: The -wsdl option cannot be used with SOAP1.2 bindings.
Try using "-wsdl:Xsoap1.2 -extension".
Class "example.Stock" binding:
"http://www.w3.org/2003/05/soap/bindings/HTTP/".
```

Additional information

For additional information see:

► "Generating the Web service interface (from a command line)" on page 185
► SOAP 1.2 specification

http://www.w3.org/TR/soap12

Resource injection

Use resource injection to access MessageContext and transport properties. Example 11-12 shows a simple example of resource injection.

Example 11-12   Resource injection example

```
@WebService
public class ProxyProvider implement Provider {
   @Resource WebServiceContext wsContext;
   public SOAPMessage invoke(SOAPMessage input) {
      MessageContext mc = wsContext.getMessageContext();    …
   }
}
```

For further information see "Accessing the context" on page 88.

Package and deployment

Server-side implementations can be deployed without WSDL. In the absence of WSDL, from the service implementation's perspective, the deployment configuration will be based on JAX-WS annotations, resolved by appending the ?WSDL parameter to the end of service endpoint URI (for example, <host>weather-app/weather?WSDL) and generating the WSDL descriptor by caching it on the application server.

However, from the client's perspective, once you generate client objects and interfaces using the command-line tools, you will notice that wsimport captures the location of your WSDL. If the WSDL resides within your file system, you will see the actual physical file location of the endpoint (for example, C:\itso-projects\weather-app\weather.wsdl). Therefore, you are better off changing the location to something that is relative to your project, thereby making it portable.

> **Note:** Rational Application Developer performs this task automatically for you. The relative path is defined at the Web services's creation time when running the wizard to generate the client code.

Example 11-13 shows you how to change your code manually in case you need to use **wsimport** to generate your code.

Example 11-13   Changing the WSDL location to a relative path

```
@WebService(wsdlLocation="WEB-INF/wsdl/weather.wsdl")
```

11.3 Leading practices for Web services performance

Web services are developed and deployed based on standards provided mainly by the Web Services for Java Enterprise Edition specification and JAX-WS. This topic explains performance considerations for Web services supported by these specifications.

11.3.1 Design for performance

When you develop or deploy a Web service, several artifacts are required, for instance, a WSDL file. The WSDL file describes the format and syntax of the Web service input and output SOAP messages. When a Web service is implemented in the WebSphere Application Server run time, the SOAP message is translated based on the Java EE request. The Java EE-based response is then translated back to a SOAP message.

The most critical performance consideration is the translation between the XML-based SOAP message and the Java object. Performance is high for a Web service implementation in WebSphere Application Server. However, application design, deployment, and tuning can be applied in order to improve such performance. The following are some basic considerations for achieving high-performance, which you should know once you start designing a Web services application:

► Reduce the Web services requests by using a few highly functional APIs, rather than several simple APIs.

► Design your WSDL file interface to limit the size and complexity of SOAP messages.

► Use the document/literal style argument when you generate the WSDL file.

► Leverage the caching capabilities offered for WebSphere Application Server.

11.3.2 Monitor the performance of your Web services

On-going performance management requires that you test your applications for performance and then monitor them to ensure that performance goals are being met. In order to monitor the performance of your Web services you can use the following tools:

► RAD V7.5 TCP/IP Monitor
► Tivoli® Performance Monitor

Rational TCP/IP Monitor

The first (and preferred) option for a developer to use to monitor Web services is the TCP/IP Monitor that is embedded in the Rational tools. This monitor allows you to track the SOAP request and response payload as well as the HTTP headers and SOAP/HTTP traffic.

The TCP/IP Monitor works like a proxy server, passing TCP/IP requests on to another server and directing the returned responses back to the originating client. The TCP/IP messages that are exchanged are displayed in a special view within Rational Application Developer.

> **Note:** For further information about monitoring Web services, see Chapter 18 in *Rational Application Developer V7.5 Programming Guide*, SG24-7672.

Performance Monitoring Infrastructure tool

You can monitor the performance of a Web service that is deployed to WebSphere Application Server by using the Performance Monitoring Infrastructure (PMI) tool. You can use the Performance Monitoring Infrastructure to measure the time required to process Web services requests. The PMI services are enabled through the WebSphere administrative console. The Tivoli Performance Monitor is available in the administrative console to allow you to monitor the results.

PMI provides detailed statistics that can help you gain clear insight into the runtime behavior and performance of Web services. Performance counters enable you to see key performance data for each individual Web service including:

► The number of requests dispatched to an implementation bean.

► The number of requests dispatched with successful replies.

► The average time in milliseconds to process full requests.

► The average time in milliseconds between receiving the request and dispatching it to the bean.

► The average time in milliseconds between the dispatch and receipt of a reply from the bean. This represents the time spent in business logic.

► The average time in milliseconds between the receipt of a reply from a bean to the return of a result to the client.

► The average size of the SOAP request.

► The average size of the SOAP reply.

Additional information can be found at:

► Monitoring the performance of Web services applications

```
http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/index.jsp?topic=
/com.ibm.websphere.base.iseries.doc/info/iseries/ae/tprf_prfstartadmin
.html
```

► Monitoring performance with Tivoli Performance Viewer

```
http://bidoc.torolab.ibm.com:9089/help/index.jsp?topic=/com.ibm.
websphere.nd.doc/info/ae/ae/twbs_performance.html
```

11.4 For more information

The following articles and IBM Redbooks publications were used as references to create this chapter:

► IBM WebSphere Developer Technical Journal: Web services Architectures and Best Practices

```
http://www.ibm.com/developerworks/websphere/techjournal/0310_brown/
brown.html
```

► This chapter is restricted to high-level and common best practices. For further information about Web services see Patterns *Patterns: Service-Oriented Architecture and Web Services*, SG24-6303

► *Web Services Handbook for WebSphere Application Server 6.1*, SG24-7257

► *WebSphere Version 6 Web Services Handbook Development and Deployment*, SG24-6461

► JAX-WS annotations

```
http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/index.jsp?topic
=/com.ibm.websphere.nd.doc/info/ae/ae/rwbs_jaxwsannotations.html
```

A

Additional material

This book includes additional material that can be downloaded from the Web. This appendix explains how to get this material and how to use it.

Locating the Web material

The Web material associated with this book is available in softcopy on the Internet from the IBM Redbooks publications Web server. Point your Web browser at:

ftp://www.redbooks.ibm.com/redbooks/SG247758

Alternatively, you can go to the IBM Redbooks publications Web site at:

ibm.com/redbooks

Select the **Additional materials** and open the directory that corresponds with the IBM Redbooks publication form number, SG247758.

Using the Web material

The Web material consists of files required to build the sample applications and project interchange files intended for import into a Rational Application Developer

V7.5 workspace. The material also includes the files required to build the sample Derby database. Most of the chapters that use this material assume that the WebSphere Application Server V7 integrated test environment in Rational Application Developer is used as a runtime. The exceptions to this are Chapter 4, "Developing Web services applications" on page 161 and Chapter 5, "Web services administration" on page 225. Both of these chapters use a WebSphere Application Server V7 installation as the runtime.

This appendix provides instructions for setting up the weather database, importing the project interchange file into a workspace, deploying the application to the integrated test environment in Rational Application Developer, and testing the application.

The additional Web material that accompanies this book includes the following files:

► The Chapter3 folder

This folder contains the material used in Chapter 3, "The WeatherForecast sample application" on page 147. You would typically only use one of these files.

– The `ch03_sample_app.zip` file contains the itso package files that are needed to create the WeatherForecast application. If you follow the instructions in this chapter, extract these files into a temporary directory for use in the example.

– The `ch03_PIF_TestApp.zip` file contains a project interchange file with the completed application. The file can be imported into Rational Application Developer. If you want to simply reference the application discussed in this chapter, rather than build it, import this file.

► The Chapter4 folder

This folder contains the material used in Chapter 4, "Developing Web services applications" on page 161.

`ch04_app_dev.zip` contains a `Files` folder that has the artifacts used in the development process outlined in this chapter. To follow the instructions in this chapter, extract these files into a temporary directory for use in the example.

Each finished example in this chapter is also available for download with this book as a project interchange file for Rational Application Developer:

– `ch04_PIF_wsdev_from_wsdl.zip` for 4.2.1, "Web services development from a WSDL file" on page 166

– `ch04_PIF_wsdev_from_javabean.zip` for 4.2.2, "Web services development from an existing Java bean" on page 183

- ch04_PIF_crt_managed_wsclient.zip for 4.3.1, "Creating a managed Web service client" on page 189

- ch04_PIF_crt_ws_thinclient.zip for 4.3.2, "Creating a Web service thin client" on page 194

- ch04_PIF_EJB_ws.zip for the whole of 4.4, "EJB Web services" on page 197

► The WeatherBase folder

This folder contains the WeatherWebService.zip project interchange file. The WeatherJavaBean application contained in this archive can be used as a starting point for the following chapters:

- Chapter 6, "Policy sets" on page 261
- Chapter 7, "WS-Policy and WS-MetadataExchange" on page 327
- Chapter 9, "WS-Notification" on page 397
- Chapter 10, "WS-SecureConversation" on page 471

Instructions for importing this application into your workspace can be found in "Using the WeatherJavaBean application" on page 543.

► The Chapter6 folder

This folder contains PolicySet.zip, which has the keystore files used for the policy set examples.

► The Chapter8 folder

This folder contains two project interchange files that contain the completed examples for the chapter:

- ws-at.zip
- ws-ba.zip

This folder also contains the DB2V8 folder with the DB2 command files to define and load the WEATHER database.

► The Chapter9 folder

This folder contains the following project interchange file that contains the completed examples for the chapter:

ws-notification project interchange.zip

► The Weather Derby Database folder

This folder contains a zip file of the Weather database:

weather_database.zip

Set up the WEATHER database (Derby)

The WEATHER database is implemented as a Derby database for all chapters with the exception of Chapter 8, "Web services transaction specifications" on page 361. The following procedure illustrates how to set up the WEATHER Derby database.

1. Extract the weather database files from `weather_db.zip` into the (root) *C:* directory. Figure A-1 shows a listing of these files.
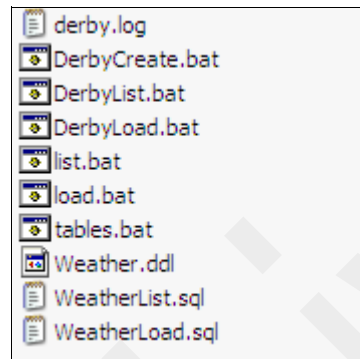

Figure A-1   Weather database files

2. Open the following files in a text editor:

   – `DerbyCreate.bat`
   – `DerbyLoad.bat`
   – `DerbyList.bat`

3. Modify the paths on the first three lines to match the installation directory of WebSphere Application Server:

   ```
   SET WAS_HOME=C:\WAS_HOME
   SET JAVA_HOME=C:\WAS_HOME\java
   SET DERBY_HOME=C:\WAS_HOME\derby
   ```

4. Run `DerbyCreate.bat` to create the database schema and tables. A `WEATHER` directory will be created containing the database itself. See Figure A-2.
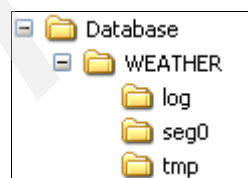

Figure A-2   The WEATHER directory

> **Note:** Keep the full path of the database directory handy. In the succeeding chapters, it will be used when configuring the data source for the Web service modules derived from the weather forecast application packages.
>
> The path is:
>
> `C:/Database/WEATHER`

5. Run **DerbyLoad.bat** to load the weather application data into the database.
6. Run **DerbyList.bat** to test the database and list all its contents. See Example A-1.

Example: A-1   Database listing

```
C:\weatherdb\Database>java -classpath
"C:\webspherev7\appserver\derby\lib\derby
jar;C:\webspherev7\appserver\derby\lib\derbytools.jar"
org.apache.derby.tools.i
 list.bat
ij version 10.3
ij> connect 'jdbc:derby:WEATHER';
ij> run 'WeatherList.sql';
ij> SELECT * from ITSO.SANJOSE;
WEATHERDA&|CONDITION              |WINDDIR          |TEMPERATURE|WINDSPEED
-------------------------------------------------------------------------
2006-01-07|stormy                |W                |6          |11
2006-07-07|rainy                 |NE               |33         |5
2006-09-17|sunny                 |SW               |23         |6
2006-09-18|partly cloudy         |W                |20         |9
2006-09-19|cloudy                |W                |17         |11
2006-09-28|sunny                 |WE               |30         |7

6 rows selected
ij> disconnect all;
ij> exit;
```

7. Create a JDBC provider for Derby and a data source for the weatherdb database. Create these in the EAR file (see "Configuring a JDBC data source in the EAR file" on page 156) or use the administrative console for the application server (see 5.4.1, "Configuring JDBC resources" on page 244).

   – The JNDI name is `jdbc/weather`.
   – The path to the database is `C:/Database/WEATHER`.

Set up the WEATHER database (DB2)

The WEATHER database is implemented as a DB2 database for Chapter 8, "Web services transaction specifications" on page 361. If you plan to import the applications used in that traffic and want to test them, you must install the DB2 database.

DB2 command files to define and load the `WEATHER` database are provided in `\Chapter8\DB2V8`:

► Execute the `DB2Create.bat` file to define the database and table.
► Execute the `DB2Load.bat` file to delete the existing data and add six records.
► Execute the `DB2List.bat` file to list the contents of the database.

These command files use the SQL statements provided in:

► `Weather.ddl`: database and table definition
► `WeatherLoad.sql`: SQL statements to load sample data
► `WeatherList.sql`: SQL statement to list the sample data

You must define a JDBC provider for DB2 and a data source for the DB2 database in your server run time. 5.4.1, "Configuring JDBC resources" on page 244, provides instructions for creating a JDBC provider

The JNDI name that the application in Chapter 8, "Web services transaction specifications" on page 361 uses is `jdbc/weather1`.

Importing project interchange files

The following procedure can be used to import project interchange files (PIFs) to a Rational Application Developer workspace:

1. In the workbench of the Rational Application Developer, select **File** →
   Import → **expand Other folder** → **Project Interchange™**. Click **Next**.
2. Click **Browse** to locate the project interchange file (*name*`.zip` file) and then click **Select All**.
3. Click **Finish**.

Using the WeatherJavaBean application

The WeatherJavaBean application can be used as a starting point for the following chapters:

► Chapter 6, "Policy sets" on page 261
► Chapter 7, "WS-Policy and WS-MetadataExchange" on page 327
► Chapter 9, "WS-Notification" on page 397
► Chapter 10, "WS-SecureConversation" on page 471

Importing the base Web services application

To import the base Weather Web services application, complete the following steps:

1. In the workbench of the Rational Application Developer, select **File** → **Import**. Expand **Other** and select **Project Interchange**. Click **Next**.

2. Click **Browse** to locate the `WeatherWebService.zip`, then click **Select All**.

3. Click **Finish**.

Deploying the enterprise applications to the server

The next step is to deploy the application to the server:

1. In the Servers view, double-click **WebSphere Application Server V7.0** to open the server editor.

2. Under the Publishing settings for WebSphere Application Server section, select **Run server with resources on server**.

3. Start the server.

4. In the Servers view, right-click **WebSphere Application Server V7.0** and select **Add and Remove Projects.** Select the following projects to add to the server:

   – WeatherJavaBeanServer
   – WeatherJavaBeanWebClient

   If the projects have already been installed, remove them first.

> **Note:** The Run server with resources on server option installs and copies the full application and its server-specific configuration from the workbench into the directories of the server.
>
> If you use the Run server with resources within the workspace option, you are unable to modify the settings inside the EAR file by using the WebSphere administrative console.
>
> In Chapter 6, "Policy sets" on page 261, the administrative console is used to apply the policy set and binding to the Web service application so the Run server with resources on Server option is used.

Testing the enterprise applications

Our Weather Web services application is built on top of an existing Java EE application, so we must make the base Java EE application work before we expose it as a Web service. To test the basic functionality of the applications, a simple servlet is included in the base code:

1. In the Enterprise Explorer expand **WeatherJavaBeanWeb** →
   Java Resources → **src** → **itso.test**. Right-click **WeatherServlet** and select
   Run As → **Run on Server**.

2. When prompted for Server Selection, ensure that the correct server is selected.

3. Click **Finish**.

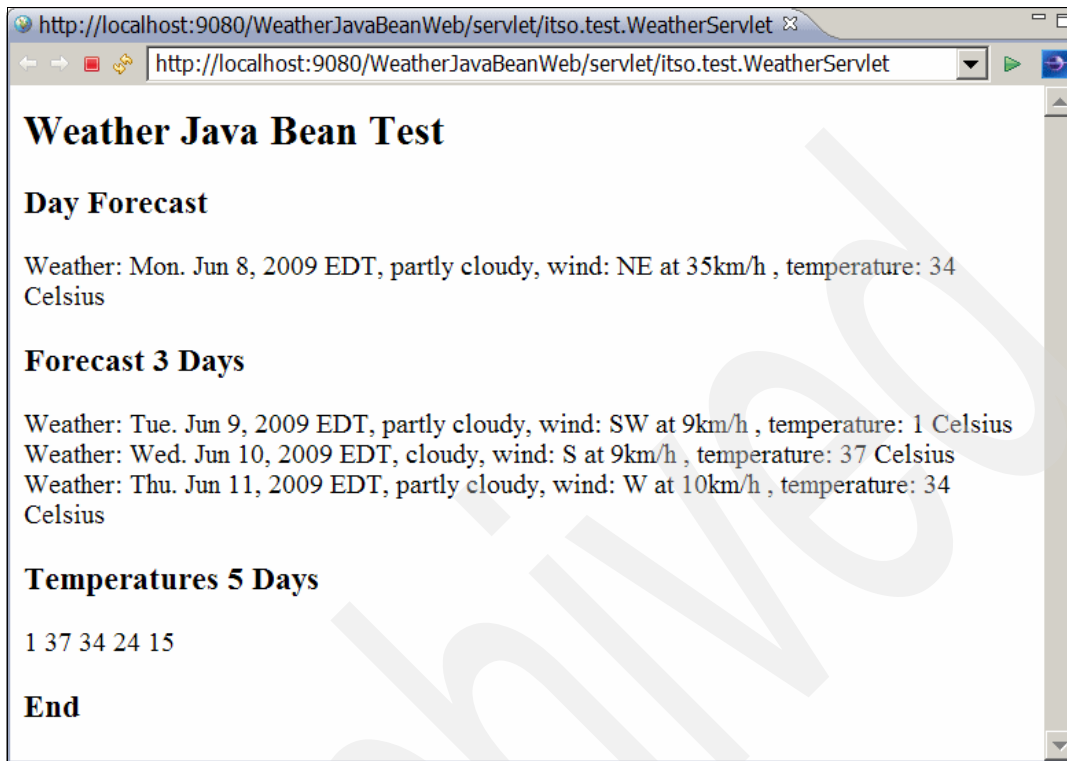The servlet should produce sample output in the Web browser (Figure A-3).

Figure A-3   Test the enterprise application

Testing the Weather Web service application

A sample JSP client is provided to test the weather Web service application. To test it, do the following:

1. In the Enterprise Explorer expand **WeatherJavaBeanWebClient** →**WebContent** → **sampleWeatherJavaBeanPortProxy**. Right-click **TestClient.jsp** and select **Run As** → **Run on Server**. The JSP page is displayed in the browser (Figure A-4).
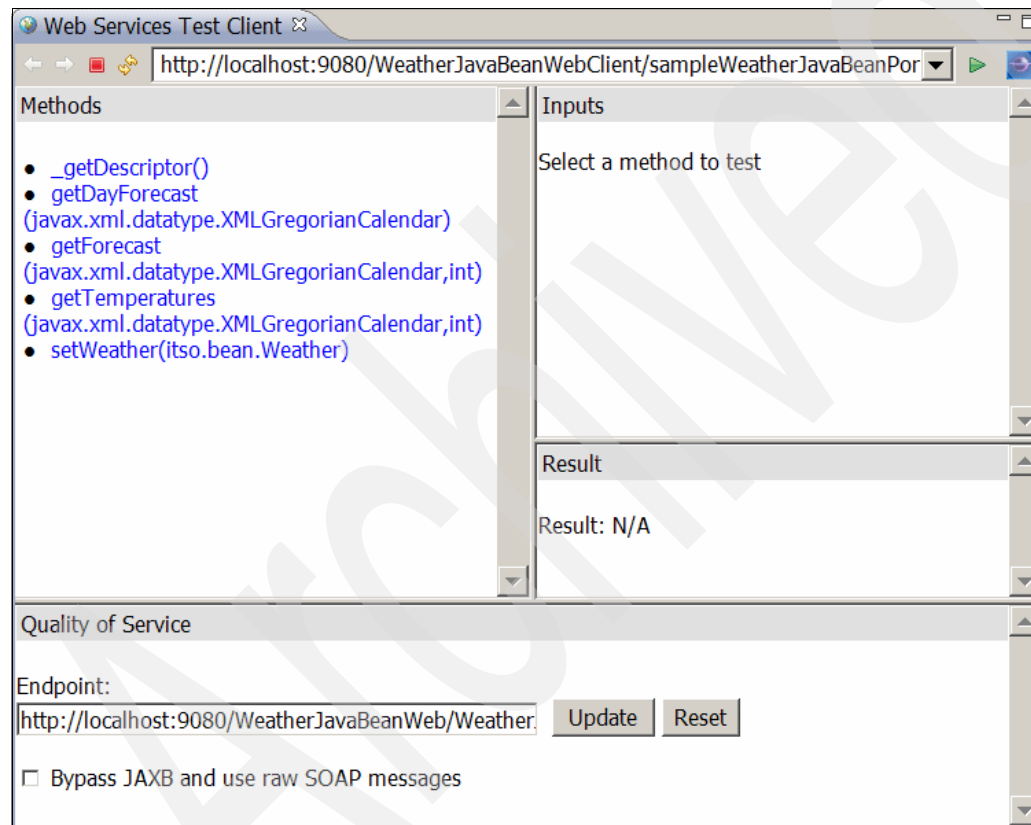
Figure A-4   Sample Web service JSP client

2. Take a close look at the Endpoint section. Note that the endpoint is set to:

```
http://localhost:9080/WeatherJavaBeanWebClient/sampleWeatherJavaBean
PortProxy/TestClient.jsp
```

If your WebSphere Application Server is not running on port 9080, you must update the endpoint. For example, if the server is running on port 9081, you must set the endpoint to the following URL and then click **Update**:

```
http://localhost:9081/WeatherJavaBeanWebClient/sampleWeatherJavaBean
PortProxy/TestClient.jsp
```

3. Click the **getDayForecast** method and enter a value for arg0. You can copy and paste the example value given by the JSP client, for example, 2009-04-10T16:22:19. Click **Invoke**. The result is shown in Example A-2.

Example: A-2   getDayForecast result

```
returnp:
  condition: partly cloudy
  date: 2009-06-08T00:00:00-04:00
  dbflag: true
  temperatureCelsius: 34
  windDirection: NE
  windSpeed: 35
```

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this book.

IBM Redbooks

For information about ordering these publications, see "How to get Redbooks" on page 553. Note that some of the documents referenced here may be available in softcopy only.

► *Patterns: Service-Oriented Architecture and Web Services*, SG24-6303

► *Best Practices for SOA Management*, REDP-4233

► *Web Services Feature Pack for WebSphere Application Server V6.1*, SG24-7618

► *Web Services Handbook for WebSphere Application Server 6.1*, SG24-7257

► *WebSphere Version 6 Web Services Handbook Development and Deployment*, SG24-6461

Online resources

These Web sites are also relevant as further information sources:

► WebSphere Application Server V7 Information Center

  `http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/index.jsp`

► IBM WebSphere Developer Technical Journal: Web services Architectures and Best Practices

  `http://www.ibm.com/developerworks/websphere/techjournal/0310_brown/brown.html`

► Achieving Web services interoperability between the WebSphere Web services Feature Pack and Windows Communication Foundation, Part 2: Configure and test WS-Security

  `http://www.ibm.com/developerworks/websphere/library/techarticles/0712_levay/0712_levay.html`

- ► Changes Between SOAP 1.1 and SOAP 1.2

  `http://www.w3.org/TR/2003/REC-soap12-part0-20030624/#L4697`

- ► Java Community Process home page

  `http://www.jcp.org`

- ► JSR-109 V1.2 specification

  `http://jcp.org/aboutJava/communityprocess/maintenance/jsr109/jsr-109`
  `-changelog-1_2-fcs.html`

- ► Index of OASIS documents

  `http://docs.oasis-open.org`

- ► OASIS Standards for security Web services

  `http://www.oasis-open.org/specs/#wssv1.0`

- ► *Publish-Subscribe Notification for Web services*

  `http://www-128.ibm.com/developerworks/webservices/library/specification`
  `/ws-pubsub/`

- ► SOAP 1.2 specification

  `http://www.w3.org/TR/2003/REC-soap12-part0-20030624`

- ► Using the WS-I Supply Chain Management application in WebSphere V6.1 Web services Feature Pack, Part 2: Apply WS-Security 1.0 to the JAX-WS SCM application

  `http://www.ibm.com/developerworks/websphere/library/techarticles/0801`
  `_zeitouni/0801_zeitouni.html`

- ► Which style of WSDL should I use?

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

- ► WS-Addressing specification

  `http://www.w3.org/TR/ws-addr-core/`

- ► WS- AtomicTransaction 1.1

  `http://docs.oasis-open.org/ws-tx/wstx-wsat-1.1-spec-os/wstx-wsat-1.1`
  `-spec-os.html`

- ► WS- AtomicTransaction 1.0

  `http://www.ibm.com/developerworks/library/specification/ws-tx/#atom`

- ► WS-BaseNotification

  `http://docs.oasis-open.org/wsn/wsn-ws_base_notification-1.3-spec-os.pdf`

- ► WS-BrokereredNotification

  `http://docs.oasis-open.org/wsn/wsn-ws_brokered_notification-1.3-spec-os.pdf`

- ► Web services Interoperability Organization Web site

  `http://www.ws-i.org/`

- ► WS-I Basic Profile Version 1.1

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

- ► WS-I Basic Profile Version 1.2

  `http://www.ws-i.org/Profiles/BasicProfile-1.2.html`

- ► WS-I Basic Profile Version 2.0

  `http://www.ws-i.org/Profiles/BasicProfile-2_0(WGD).html`

- ► WS-BusinessActivity 1.1 specification

  `http://docs.oasis-open.org/ws-tx/wstx-wsba-1.1-spec-os/wstx-wsba-1.1-spec-os.html`

- ► WS-BusinessActivity 1.0

  `http://www.ibm.com/developerworks/library/specification/ws-tx/#ba`

- ► WS-Coordination specification

  `http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-tx/WS-Coordination.pdf`

- ► WS-Coordination 1.1

  `http://docs.oasis-open.org/ws-tx/wstx-wscoor-1.1-spec-os/wstx-wscoor-1.1-spec-os.html`

- ► WS-Coordination

  `http://www.ibm.com/developerworks/library/specification/ws-tx/`

- ► WS-Coordination 1.0

  `http://www.ibm.com/developerworks/library/specification/ws-tx/#coor`

- ► WS-MetadataExchange specification

  `http://www.w3.org/TR/2009/WD-ws-metadata-exchange-20090317/`

- ► WS-MetadataExchange 1.2 specification draft

  `http://specs.xmlsoap.org/ws/2004/09/mex/WS-MetadataExchange.pdf`

- ► WS-Notification in WebSphere Application Server V7: Part 1: Writing JAX-WS applications for WS-Notification:

  http://www.ibm.com/developerworks/websphere/techjournal/0811_partridge/0811_partridge.html

- ► Learning about WS-Policy

  http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/topic/com.ibm.websphere.nd.doc/info/ae/ae/twbs_wsp_learning.html

- ► Web services Policy Working Group

  http://www.w3.org/2002/ws/policy/

- ► Web services Policy 1.5 - Framework

  http://www.w3.org/TR/ws-policy/

- ► WS-PolicyAttachment

  http://www.ibm.com/developerworks/library/specification/ws-polatt/

- ► WS-PolicyAssertions

  http://www.ibm.com/developerworks/webservices/library/specification/ws-polas/

- ► Web services Policy 1.2 - Attachment (WS-PolicyAttachment) specification

  http://www.w3.org/Submission/WS-PolicyAttachment/

- ► WS-I Reliable Secure Profile specification

  http://www.ws-i.org/deliverables/workinggroup.aspx?wg=reliablesecure

- ► WS-MetadataExchange specification at the following address:

  http://www.w3.org/TR/2009/WD-ws-metadata-exchange-20090317/

- ► Web services Resource Framework (WSRF) Primer V.2

  http://docs.oasis-open.org/wsrf/wsrf-primer-1.2-primer-cd-02.pdf

- ► Web services Resource Framework overview

  http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrf#overview

- ► WS-ReliableMessaging specification

  http://docs.oasis-open.org/ws-rx/wsrm/200702

- ► WS-ReliableMessaging artifacts (schema, WSDL)

  http://www.ibm.com/developerworks/webservices/library/specification/ws-rm/?S_TACT=105AGX04&S_CMP=LP

► Web services Reliable Messaging Policy Assertion

  http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-rm/ws-rmpolicy200502.pdf

► WS-SecureConversation specification

  http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512/ws-secureconversation-1.3-os.html

► Web services Secure Conversation

  http://www.ibm.com/developerworks/library/specification/ws-secon/

► WS-Security specification

  http://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf

► WS-SecurityPolicy 1.2 specification

  http://docs.oasis-open.org/ws-sx/ws-securitypolicy/v1.2/ws-securitypolicy.html

► Web services Security Policy Language

  https://www.ibm.com/developerworks/webservices/library/specification/ws-secpol/

► WS-Topics

  http://docs.oasis-open.org/wsn/wsn-ws_topics-1.3-spec-os.pdf

► WS-Transaction specification

  http://docs.oasis-open.org/ws-tx/wstx-wsat-1.1-spec-os/wstx-wsat-1.1-spec-os.html

► WS-Trust V1.3 specification

  http://docs.oasis-open.org/ws-sx/ws-trust/200512/ws-trust-1.3-os.html

► WS-Trust IBM developerWorks article, "Web services Trust Language"

  http://www.ibm.com/developerworks/library/specification/ws-trust/

How to get Redbooks

You can search for, view, or download Redbooks, Redpapers, Technotes, draft publications and Additional materials, as well as order hardcopy Redbooks publications, at this Web site:

ibm.com/redbooks

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services

IBM WebSphere Application Server V7.0 Web Services Guide

IBM ®

IBM WebSphere Application Server V7.0 Web Services Guide

Redbooks ®

Explore new technology

Develop Web services by example

Find leading practices

This IBM Redbooks publication describes how to implement Web services in WebSphere Application Server V7. It starts by describing the concepts of the major building blocks on which Web services rely and leading practices for Web services applications. It then illustrates how to use Rational Application Developer and the WebSphere tools to build and deploy a Web services application.

In addition to the fundamentals of Web services development, this book provides information about advanced topics, including WS-Policy, WS-MetadataExchange, Web services transactions, WS-Notification, and WS-SecureConversation.