WebSphere software

IBM

Web Services Feature Pack for WebSphere Application Server V6.1

Asynchronous

Reliable

Secure

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

Redbooks

ibm.com/redbooks

International Technical Support Organization

Web Services Feature Pack for WebSphere Application Server V6.1

July 2008

SG24-7618-00

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

First Edition (July 2008)

This edition applies to Version 6, Release 1, Modification 0.13 of Feature pack for Web services.

Contents

Notices

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

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

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

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

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

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

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

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

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

COPYRIGHT LICENSE:

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

Trademarks

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:

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

The following terms are trademarks of other companies:

EJB, J2EE, J2SE, Java, JMX, JRE, JSP, JVM, and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Expression, Internet Explorer, Microsoft, MSDN, SQL Server, Visual Studio, Windows Server, Windows Vista, Windows, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Intel Core, Intel, Intel logo, Intel Inside logo, and Intel Centrino logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States, other countries, or both.

Linux is a trademark of Linus Torvalds 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 is aimed at architects, designers, and programmers with a background in Web services, particularly with experience in the JAX-RPC support in WebSphere Application Server. Our intent is to help you judge when to step forward to use the Feature pack for Web services. We provide all the education you need to make the transition to the Feature pack for Web services from the Web services support supplied with WebSphere Application Server v6.1.

We give you lots of examples with supporting materials, many written as step-by-step tutorials, to help you become skilled in using the Feature pack for Web services:

- Installation on WebSphere Application Server and WebSphere Application Server for z/OS
- The differences between JAX-WS and JAX-RPC
- How to use the Policy sets feature to configure reliable messaging and security, including the complicated setup for Secure Conversation
- An extensive example showing many aspects of configuring interoperability between Windows Communication Foundation and WebSphere Application Server using reliable and secure qualities of service

We also review the benefits of the Feature pack for Web services. This section should be of particular interest to architects and designers. We think the best way to show how the Feature pack for Web services benefits the development of a Web services solution is to discuss business scenarios that can make use of the capabilities in the feature pack to particular advantage. From these scenarios we draw out the general facets and patterns that make the feature pack a good choice for the implementation of the scenarios. Other business scenarios that share these facets and patterns would also benefit from using the Feature pack for Web services.

For those with a background in Web services, we summarize the technical features of the feature pack and their advantages and provide a forward looking historical account of how the Feature pack for Web services came to be built, and where Web services go next, in version 7.0 of WebSphere Application Server.

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, Austin Center. Here we show the members of our team.

Henry, Andrew, Russell, Peter, David, Mark

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

Russell Butek is a Certified IT Specialist in the U.S.A. He has twenty-two years of experience in software engineering. He holds B.S. and M.S. degrees in Computer Science from the University of Wisconsin. His areas of expertise include object oriented programming, Java™, CORBA, Web services, and SOA. He has written extensively on Web services.

Henry Cui is a Software Developer working at the IBM Toronto lab. Henry has been in the IBM Rational® Application Developer service and support team for five years, where 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. Henry is a frequent contributor of developerWorks® articles and Web services technotes. He also co-authored the *Rational Application Developer V7 Programming Guide*, one of the IBM Redbooks. Henry holds a degree in Computer Science from York University.

Andrew Das is a Staff Software Engineer and an IBM Certified Web Services Developer. He is part of the WebSphere® Application Server Web Services Interoperability development team. His current responsibilities include evaluating Web Services standards and how they interoperate with other IBM products and other vendor products.

David Illsley is a Staff Software Engineer with WebSphere Application Server team in the IBM Hursley laboratory in the UK. He has five years of experience working with emerging Web services standards, including eighteen months as a member of the W3C WS-Addressing Working Group, and is a member of the Apache Web Services Project Management Committee. He holds a degree in Computer Science from the University of Edinburgh.

Mark Lewis is a Web services development lead in the UK. He has six years of experience developing Web services technology for WebSphere Application Server. He holds an honours degree in Software Engineering from City University, London.

Acknowledgements

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

Greg Truty, Senior Technical Staff Member and lead for WebSphere Web services architecture and development at IBM Austin lab, Texas, for being our technical sponsor, and a bottomless source of help and advice.

Nicholas Gallardo, Web services developer at the IBM Austin lab, Texas, for providing us with lots of technical help.

Lizet Ernand, Web services developer at the IBM Austin lab, Texas, for providing us with lots of technical help.

Bill Dodd, Software Developer, WebSphere Security Development, IBM Austin lab, for providing the insights into the Web services Secure Conversation.

Chunlong Liang, Software Developer, WebSphere Security Development, IBM Austin lab, for providing the help in the Web services security API.

Hyen-Vui (Henry) Chung, architect and team lead, WebSphere Web services security Architecture and Development, IBM Austin lab, for providing the general help in the Web services security area.

We would particularly like to acknowledge:

Salim Zeitouni, Advisory Software Engineer, and
Charles Le Vay, Senior Software Engineer, Web Services Interoperability Architect

For providing the example and helping us modify it for the tutorial in Chapter 11, “Interoperability” on page 403.

And also for helping set up and document the installation and configuration of the Feature pack for Web services on z/OS®, we thank:

Richard Conway, who is ITSO residency enablement for z/OS

Don Bagwell, Senior Technical Sales Specialist for z-Series based out of the IBM Washington Systems Center in Gaithesburg, Maryland

Linda Robinson, for turning sketches into pictures for us; and
Yvonne Lyon, for editing this book.

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 in one of the following ways:

- Use the online **Contact us** review Redbooks 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

Part 1

Introduction

This book is divided into three parts.

Part 1 is a complete summary of the contents of the Feature pack for Web services and discussion about how the it furthers the direction in which Web services is moving.

In Part 2 we look at the value the feature pack offers. This is important to understand in deciding whether to make use of the capabilities of the feature pack or not; JAX-WS is offered in addition to the JAX-RPC support in WebSphere Application Server v6.1, qualities of service are managed as policies as well as part of the programming model, and WS-Reliable Messaging overlaps in its objectives with the assurances offered by Message Oriented Middleware (MOM). An architect or designer deciding to use the capabilities of the feature pack needs to weigh the feature pack against alternatives.

In Part 3, which is the major content of the book, we provide detailed expositions of the capabilities of the Feature pack for Web services with plenty of examples showing how to use them. Finally, there is an extensive example of building a solution using reliable and secure services between WebSphere Application Server on distributed platforms and on z/OS, and with Web services on .Net 3.5 running in Windows Communication Foundation.

1

What is in the feature pack

The primary reason for the Web services feature pack (WS-FEP) is to support reliable, secure messaging. Reliable, secure messaging requires a number of new features. In this chapter we itemize these features and discuss each of them in turn.

1.1 The 10,000 foot view

The purpose of the feature pack is to support reliable messaging. Reliable messaging is a broad concept. It describes the idea that once a sender has dispatched a message from an application, the message will arrive, untampered with, at the receiver. In order to support reliable messaging, the feature pack needs new Web services functions over the existing version 6.1 of the WebSphere Application Server. Figure 1-1 gives you a high level view of the new functions in the feature pack.

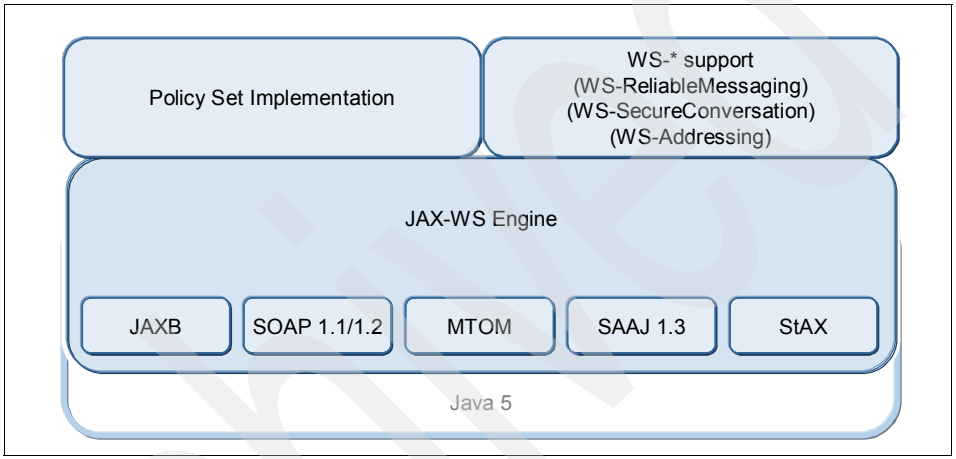


Figure 1-1 The big picture for the Feature pack for Web services

We summarize the content of the feature pack in this chapter, and then discuss the contents in more detail throughout the remainder of the book.

1.1.1 Rationale for the feature pack contents

Why are existing Web services not reliable? There are possibilities of temporary network, component, or machine failures to be overcome, and with the message flowing through servers and network connections that are not governed by the sender or receiver, it is desirable to ensure that the contents of the message have not been viewed or tampered with in transit between the client and server applications. Reliable messaging requires both a guarantee that a message is delivered, and a guarantee that the content of the message which is delivered has not been tampered with, has not been read, and has been sent from whomever it says has sent it.

In order to achieve reliable messaging, and to interoperably send reliable messages, the industry needed to define a profile of uses cases and resolve ambiguities in WS-* specifications. IBM developed the Reliable Asynchronous Messaging Profile (RAMP) in conjunction with members of the automobile industry. This is what the feature pack was built to support. Recently, the Web Services Interoperability organization (WS-I)[1] chartered a working group to develop a Reliable Secure Profile (RSP), but at the time that the feature pack was released, it was still too early for them to have produced a final profile. The RSP is the industry direction and the feature pack is a step in that direction.

The two requirements of both RAMP and RSP are:

- **Reliability:** We must deliver the message.
- **Security:** The receiver must receive *our* messages securely.

In addition, it is useful to support “fire and forget” when sending reliable messages. Fire and forget has always been available using a one-way interaction pattern. Asynchrony is also valuable for two-way interactions, so the sender does not have to block to wait either for delivery or for a response to our message.

For reliability and security, there are three specifications that WS-I’s RSP (and RAMP) are concerned with. In this chapter we discuss each in turn:

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

These are all OASIS specifications. OASIS is the Organization for the Advancement of Structured Information Standards.

To support a two-way asynchronous model for reliable messaging, we have to support asynchrony between sending requests and receiving responses. This is a programming requirement. Because the WebSphere Application Server is a Java environment, the feature pack has to implement the asynchronous client programming model that is part of the new Java API for XML-based Web services (JAX-WS) 2.0 specification, developed by the Java Community Process (JCP).

The JAX-WS 2.0 specification also builds on a number of other new specifications not included in the WebSphere Application Server v6.1 Web services implementation. So in the discussion of JAX-WS we also discuss these specifications, which are also implemented by the feature pack:

- Java Architecture for XML Binding (JAXB) 2.0

[1] WS-I builds profiles to enable interoperability. Profiles clear up ambiguities and inconsistencies between the varied Web services specifications.

- Message Transmission Optimization Mechanism (MTOM) and XML-binary Optimized Packaging (XOP)
- SOAP 1.2
- SOAP with Attachments API for Java (SAAJ) 1.3
- Streaming API for XML (StAX) 1.0

In addition, and in support of the RSP qualities of service (QoS), the feature pack also introduces the notion of runtime-configurable policy sets.

All of this new functionality relies on Java annotations, and requires the Java 5 runtime environment provided by WebSphere Application Server v6.1.

1.2 Reliable and secure messaging

Reliable messaging is a requirement of many enterprise systems. Because more and more enterprise applications are being built as Web services, the industry has to make Web services communications reliable. WS-ReliableMessaging was written to address this, and other, needs.

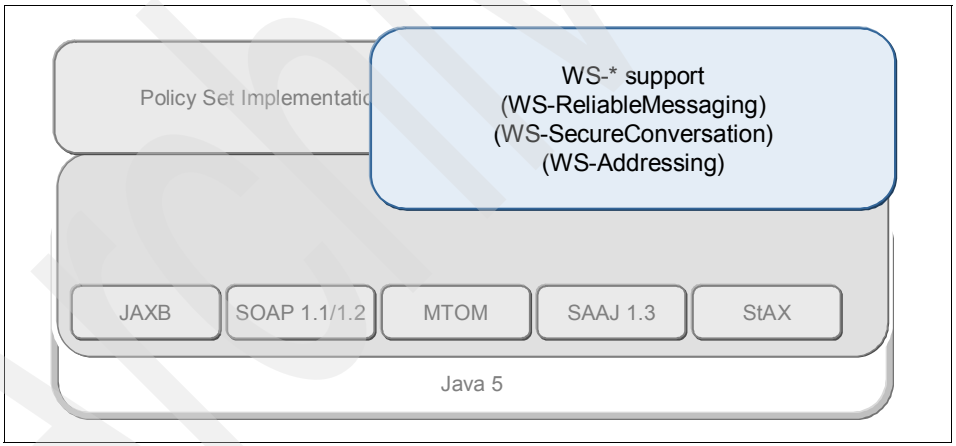


Figure 1-2 The reliable messaging focus of the big picture

The WS-ReliableMessaging specification strongly recommends securing the contents of messages. Because this is a Web service specification, using WS-Security to secure the messages is suggested. The WS-ReliableMessaging specification goes even further: Because reliable messaging indicates an exchange of a sequence of messages — in other words, a conversation — the WS-ReliableMessaging specification recommends using WS-SecureConversation (WS-SecConv) to secure WS-ReliableMessaging messages.

Finally, because identity is an important aspect of reliable messaging — the receiver must know where an incoming message originates and must send acknowledgement and other messages back to the originator, WS-ReliableMessaging makes use of WS-Addressing constructs. Next we discuss each of these specifications, as well as some functions that the feature pack provides beyond what the specifications require.

1.2.1 WS-ReliableMessaging

The abstract of WS-ReliableMessaging v1.1 states:

> *"This specification (WS-ReliableMessaging) describes a protocol that allows messages to be delivered reliably between distributed applications in the presence of software component, system, or network failures. The protocol is described in this specification in a transport-independent manner allowing it to be implemented using different network technologies. To support interoperable Web services, a SOAP binding is defined within this specification."* [2]

Besides the obvious statement that this specification allows messages to be delivered reliably, there are two items of note in this abstract:

- **Transport-independence:** WS-ReliableMessaging states that it is transport-independent. This is an important aspect; however, because the Feature pack for Web services only supports SOAP/HTTP, that is all that we will consider in this book.
- **Protocol:** WS-ReliableMessaging describes a communications protocol to ensure against network failures.

> **Note:** The protocol is an important aspect of reliable messaging, but it is not the complete picture. It addresses network failures, but it does not fully address reliable messaging in the face of component or system failures. See "Persistence" on page 10 for a discussion of additional necessary function.

Figure 1-3 shows the example of a reliable message exchange taken from the WS-ReliableMessaging specification. In brief, endpoint A creates a sequence which will contain a series of messages. Endpoint B receives the sequence of messages, acknowledging that they have been received. If B does not

[2] You can read about WS-ReliableMessaging at: http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-rm/ws-reliablemessaging200502.pdf

acknowledge all messages, A resends the unacknowledged ones. When the sequence is complete, A terminates it.

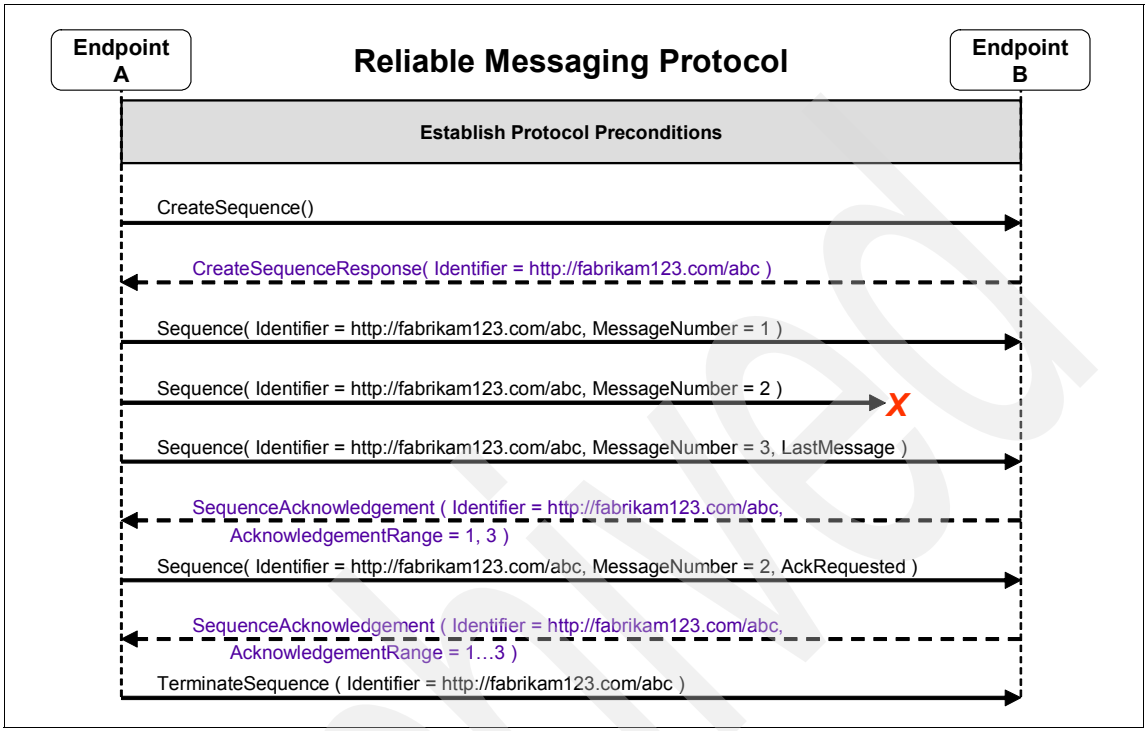


Figure 1-3 Example message exchange (from WS-ReliableMessaging)

When messages are synchronous, WS-ReliableMessaging information is sent in the headers of the normal request and response messages. 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 what Figure 1-3 shows, acks can 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 WS-Addressing. For more details, see 10.1.3, "How WS-ReliableMessaging works" on page 358.

1.2.2 WS-SecureConversation

Security is expensive in terms of performance. A major part of that expense is the cost of encryption and decryption, especially if asymmetric algorithms are used. When you have a session-based protocol, if your security model is aware of the session, it can decrease some of the performance costs by using

asymmetric encryption only to set up security, and then using symmetric keys thereon. This is one of the features of HTTPS.

But HTTPS is a transport-level security mechanism. WS-ReliableMessaging requires a SOAP protocol security mechanism. WS-Security is a SOAP protocol security mechanism, but it does not know about sessions, so it will repeatedly invoke computationally expensive cryptographic operations that are unnecessary in a “conversation.” WS-SecureConversation is aware of session information, and can follow the HTTPS pattern of using an asymmetic algorithm to set up security, and then use symmetric algorithms thereon. So it is an improvement upon WS-Security. For more details, see 9.3, “WS-SecureConversation” on page 322.

1.2.3 WS-Addressing

In a moderately complex system, a receiver of a message must know something about the sender. With asynchronous request/response communications, the receiver must at least know the address of the sender so that it can send responses back disconnected from the request channel. The purpose of WS-Addressing is to provide an interoperable way of communicating this information.

WS-Addressing defines a number of SOAP headers. The most important ones for asynchronous communications are:

- ReplyTo: The address of the sender - responses go to this address
- MessageID: This is the UUID of the message.
- RelatesTo: This header exists on response messages and contains the UUID of the related request message.

WS-Addressing is much richer than this, but for the needs of asynchronous messages, these headers are the most relevant.

1.2.4 Going beyond the specifications

WS-ReliableMessaging and its related specifications go a long way to providing reliable, interoperable, messaging. But they do not go all the way. In this section we describe aspects of the feature pack that go beyond the specifications.

Which version of the specification?

Web services reliable messaging is an evolving set of standards. Refer to Chapter 2, “History and roadmap” on page 23 for more discussion.

In brief:

- First there was RAMP (Reliable Asynchronous Message Profile).
- RAMP incorporates WS-ReliableMessaging 1.0.
- Now WS-ReliableMessaging 1.1 exists.
- WS-I's Reliable Secure Profile (RSP) is in working draft status. Version 1 revision 11 was published in October 2007.

The Web services feature pack was designed to support RAMP, but it evolved as the specification world evolved. It maintains a policy set[3] for each of the existing specifications/profiles: RAMP, WS-ReliableMessaging 1.0, WS-ReliableMessaging 1.1. In time, a future release of WebSphere Application Server will support RSP, but not before the OASIS WS-SecureConversation specification is finalized. The Feature pack for Web services will be superseded by the RSP support in WebSphere Application Server.

Note that if you want interoperability with .NET, you should use the WS-ReliableMessaging 1.0 policy set.

Persistence

The abstract of WS-ReliableMessaging hints at persistence when it mentions reliability in the face of failures of software components and systems. You cannot ensure this level of reliability merely with a wire protocol. You need some level of message persistence.

The feature pack allows for three levels of persistence[4]:

- Unmanaged non-persistent

  This is the level expected by WS-ReliableMessaging. It provides an in-memory message store so that messages can be resent. The message store does not survive server failures.

- Managed non-persistent

  This level uses a messaging engine to manage sequence state. In a clustered environment, messages are recovered after individual server failures, but not if the messaging engine itself fails.

- Managed persistent

  Persistent stores of messages are managed on both the sender and receiver.

The default is unmanaged non-persistent.

[3] See the Policy Set section, 1.4.

[4] For more details, see the InfoCenter at http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/index.jsp?topic=/com.ibm.websphere.wsfep.multiplatform.doc/info/ae/ae/twbs_wsspsptrmd.html

Programmatic APIs

The WS-* specifications essentially describe SOAP communications. They do not describe how a programmer should code to these specifications.

The Feature pack for Web services provides programmatic APIs for WS-ReliableMessaging, WS-SecureConversation, and WS-Addressing. Normally, applications should not have to deal with qualities of service, but there are cases when it is necessary.

1.2.5 Reliable messaging versus messaging and queueing

Note that Web services reliable messaging is not a messaging and queuing system like WebSphere MQ, nor is it a messaging service, like JMS. The most obvious distinction is that Web services reliable messaging provides qualities of service that are applied to application services, whereas WebSphere MQ and JMS provide distributed messaging and queuing software layers on top of which applications are built. Another way of putting this is to say that the goal of Web services reliable messaging is to make service requests traverse the Internet reliably, whereas the goal of WebSphere MQ and JMS is to provide a platform for building reliable, loosely coupled, message driven, distributed applications that can be deployed to different kinds of networks, including the Internet.

A messaging and queuing layer adds important operational and administrative capabilities to a solution which contribute to its manageability, scalability, robustness and flexibility. But the messaging and queuing layer also adds to the cost and footprint of the solution in terms of setting up the queuing infrastructure. This could be prohibitive for an Internet scale application requiring reliable messaging for millions of clients. On the other hand, if high standards of service are required without adding to the cost by developing additional infrastructure to manage a Web services solution, then building applications using a messaging and queueing layer is a better choice.

1.3 JAX-WS

The primary reason for introducing JAX-WS in the feature pack is because it provides an asynchronous client model, which is necessary for some reliable messaging scenarios. But it also makes sense to include JAX-WS in the feature pack simply because it is the latest version of the specification for mapping between Web services and Java. See Figure 1-4.

You can think of JAX-WS 2.0 as JAX-RPC 2.0. Early in its development it was actually known as JAX-RPC 2.0[5]. The Web services Java programming model has evolved from JAX-RPC 1.0 through JAX-RPC 1.1 to JAX-WS 2.0. The name was changed because the term “WS” (Web services) is more indicative of what the specification is than “RPC” (Remote Procedure Call).

Besides the asynchronous API, the main features of JAX-WS are:

- An improved programming model
- New Java 5 features: annotations, future, generics
- Inclusion of a number of new specifications: JAXB 2.0, SAAJ 1.3, SOAP 1.2, MTOM, StAX 1.0

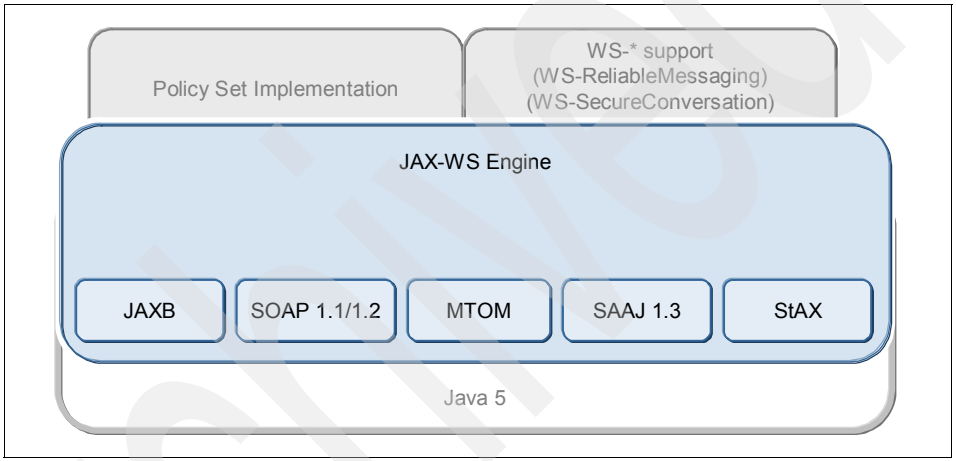


Figure 1-4 The JAX-WS focus of the big picture

We outline the features here. Chapter 7, “JAX-WS programming model” on page 175 goes into details.

1.3.1 The programming model

If you are familiar with the JAX-RPC programming model, JAX-WS is not much different. It still has interfaces. You implement the interface on the server side. You call a proxy which implements the interface on the client side. The main differences are: a new asynchronous model, which we cover in 1.3.2, “Asynchronous client APIs” on page 13, an improved client-side dynamic model, support for MTOM, a new server-side dynamic model, and an improved handler model.

[5] Now JAX-RPC is sometimes called JAX-WS 1.0.

Client-side dynamic model

JAX-RPC defines a Call object as its dynamic programming interface. You use the Call object to set parameters and context information on an operation, and then to invoke the operation. It works for basic examples but breaks down with anything complex.

JAX-WS replaces the Call object with what it calls the Dispatch object, which is more robust because it is simpler and more flexible. The client programmer views the message from either an XML or an object (JAXB) point of view. The programmer constructs either the payload of the message (the contents of the SOAP body) or the entire message; then invokes the operation.

Server-side dynamic model

The client-side API is the Dispatch object. The corresponding server-side API is the Provider object. Like the client-side programmer, the server-side programmer views messages from an XML point of view, manipulating either the full SOAP message or just the payload.

Handler model

The purpose of the handler framework is to allow interception of a message at various points in its transmission. With outgoing messages, handlers are invoked before a message is sent out to the wire. With incoming messages, handlers are invoked before the receiving application receives the message.

The JAX-WS handler framework improves on the JAX-RPC framework in that it provides two levels of handlers: logical and protocol. A logical handler operates on message context and payloads. A protocol handler operates on protocol-specific information. For instance, a SOAP handler operates on the SOAP message and SOAP context information.

1.3.2 Asynchronous client APIs

JAX-WS provides APIs for both synchronous and asynchronous invocations. A Java client application can invoke a Web service asynchronously and receive responses either via a polling API or a callback API. This means that a client can do useful work while waiting for a reply using a request/reply message exchange pattern.

In addition to providing an asynchronous client API, the feature pack also introduces asynchronous message exchanges (so-called asynchronous on the wire) which means the reply does not have to be transmitted using the same transport session as the request. This makes only a minor difference to the client programing model (to turn on asynchronous on the wire), but changes the performance and reliability of the system.

1.3.3 Java 5 features

The Java language is evolving. Internet Java 5 has a number of new features over what is available in Java 1.4. Because JAX-RPC was written during the era of Java 1.4 it was defined in the context of Java 1.4. Now that Java 5 exists, it only makes sense that JAX-WS is defined within the context of Java 5. The primary new features which JAX-WS includes are annotations, futures, and generics.

Annotations

Java 5 introduces the concept of annotations. An annotation is a non-programmatic decoration on a Java construct.

There are Web services annotations that provide mapping details. For instance, a Java method signature could contain an annotation describing the WSDL operation with which it is associated. Because mapping details can be placed on Java artifacts, helper classes and separate mapping files are not necessary as they are for JAX-RPC.

Future

A future represents the result of an asynchronous computation. It is a placeholder for a result that does not exist at the time of creation, but will exist at some point in the future. It is only natural that the definition of JAX-WS's asynchronous APIs include the Java Future type.

Generics

Java Generics allow the definition of a class that contains some data, but where the type of that data is immaterial to the logic of the class. Collections are a prime example where generics are very useful. A java.util.Vector is essentially an array of java.lang.Object. However, a Vector almost always contains types that are subclasses of java.lang.Object. With generics you can define exactly what the Vector contains. For example, if you build a Vector that will only contain objects of type java.lang.String, you can define a string vector:

```
Vector<String> stringVector = new Vector<String>;
```

This new Vector<String> class will contain strongly typed methods. Instead of methods with signatures such as:

```
public void add(java.lang.Object);
public boolean contains(java.lang.Object);
```

The strongly typed Vector<String> class will contain strongly typed methods:

```
public void add(java.lang.String);
public boolean contains(java.lang.String);
```

JAX-WS makes use of generics. The Java 5 Future class itself is a generic class which can be declared to be of a specific type which, in the case of JAX-WS, is the response type.

The polling version of an asynchronous operation invocation returns a Response object. Response is defined with a specific response type in the generated interface. For example:

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

Handlers can be defined as generics. For instance, the following coding contains a MessageContext:

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

However, the following coding contains a LogicalMessageContext, which is a subclass of MessageContext:

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

1.3.4 New specifications

Web services have evolved in the years since JAX-RPC was written. JAXB was not complete at the time, so it was not included in JAX-RPC, but it is included in JAX-WS. SOAP and SAAJ have gone through new versions. MTOM and StAX did not even exist when JAX-RPC was written. Because Web services have evolved, it is only reasonable that JAX-WS — the latest Web services to the Java mapping specification — will also evolve to incorporate these new specifications.

JAXB 2.0

JAXB is a mapping between XML types and Java types. JAX-WS is a mapping between WSDL and Java. Note the distinction. JAX-WS maps WSDL messages, portTypes, bindings, and services. Anything that appears in the types section — in other words, XML schema types — is not mapped by JAX-WS. JAX-WS defers that mapping to JAXB.

Compare that to JAX-RPC. JAX-RPC incorporates both a WSDL mapping and an XML mapping. The authors of JAX-RPC originally wanted to include JAXB, but the time lines of the two specifications did not mesh well, so the authors chose to produce an XML mapping that covered 90+% of XML. This mapping has fairly adequately survived the test of time, but it was consciously incomplete. This incompleteness has now been remedied.

MTOM/XOP

MTOM is a specification for sending binary data, often called attachments. Before MTOM there was no universally interoperable way to transmit attachments. SOAP with Attachments (Sw/A) came closest, but Microsoft® never agreed to support that specification.

JAX-WS dictates that a compliant Web service engine must support MTOM as well as Sw/A.

SOAP 1.2

A WSDL description of a Web service describes whether the service speaks SOAP 1.1 or SOAP 1.2. JAX-WS supports SOAP 1.2 as well as SOAP 1.1. This opens up improved interoperability because the feature pack runtime can now talk with providers who speak either version of SOAP.

There are very few practical differences between SOAP 1.1 and SOAP 1.2. For details of the differences, see the “Changes Between SOAP 1.1 and SOAP 1.2” section of the SOAP specification[6]. Briefly, those changes are as follows:

- SOAP is no longer an acronym — it was misleading (in other words, the specification acknowledges that SOAP really is not Simple!).
- SOAP 1.2 has been rewritten in terms of XML infosets.
- SOAPAction is optional.
- The SOAP encoding and RPC definitions have been cleaned up.
- SOAP 1.2 adds a few new attributes and more crisply defines some existing ones.
- SOAP 1.2 adds a few new fault codes.
- SOAP 1.2 is not incorporated into the Web Services Interoperability organization Basic Profiles 1.*. It is planned for BP 2.0 (see Figure 1-5).

[6] See http://www.w3.org/TR/soap12-part0/#L4697

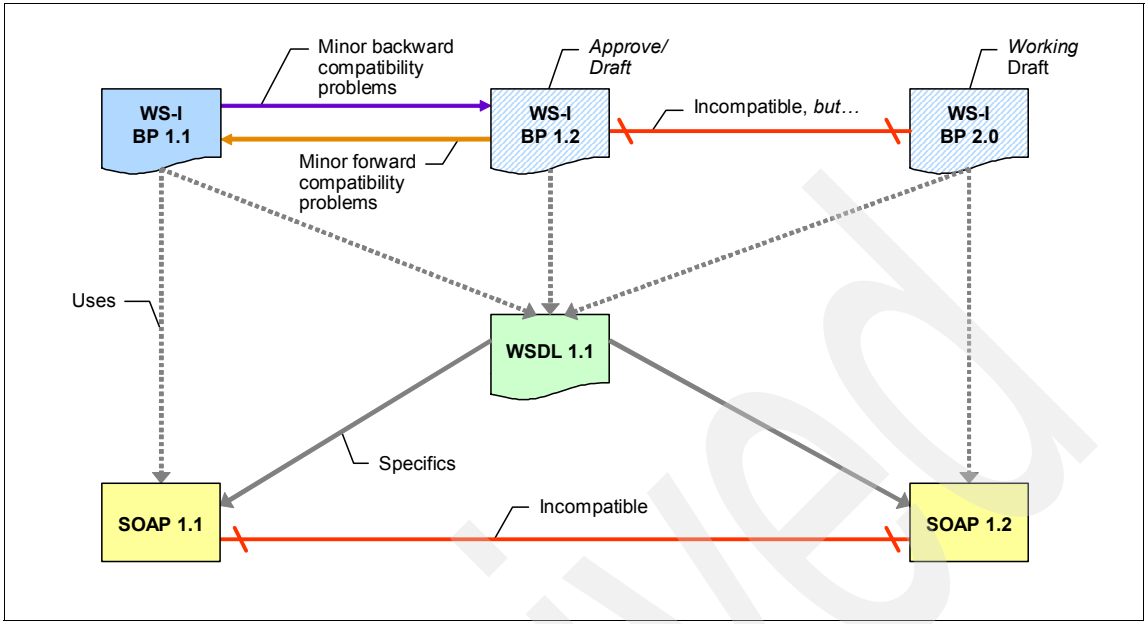


Figure 1-5 Relationship between WS-I profiles, SOAP and WSDL specifications

SAAJ 1.3

SAAJ (SOAP with Attachments API for Java) is an API to create and process SOAP messages rather than writing the XML yourself. SAAJ 1.3[7] is a maintenance release over SAAJ 1.2, which means that there are few changes. If you wrote code using the SAAJ 1.2 APIs, they should still work on an SAAJ 1.3 engine. See the “Change History” section of the specification for details. In short:

- The core classes — Node, SOAPElement, etc. — now extend the equivalent interfaces in the org.w3c.dom package - Node, Element, etc.
- SAAJ 1.3 supports SOAP 1.2.
- SAAJ 1.3 APIs now accept the more universal QName type as well as the SAAJ-specific Name type.
- A number of ease-of-use extensions have been added.

[7] You can find the SAAJ Web pages at `https://saaj.dev.java.net/`

StAX 1.0

StAX is the Streaming API for XML. Before StAX there were two rather opposite ways of parsing XML: tree based parsing (such as DOM) and event-based parsing (such as SAX). Tree based parsing has the disadvantage that the full XML tree must be loaded into memory, but once it is there, access is quick and easy. Event-based parsing does not require the full XML tree to be loaded, and it is often faster, but it is a more complex interface to an XML document.

StAX provides a middle ground between these two approaches. It is closer to event-based parsing in that it maintains a position in the XML document, so the full document does not have to be loaded into memory ("streaming parser"). But unlike event-based parsing, which pushes information to an application and is only used to read a document, StAX allows an application to pull information from the parser, which simplifies the API and gives parsing control to the application (Figure 1-6). It also allows an application to write data to the document.

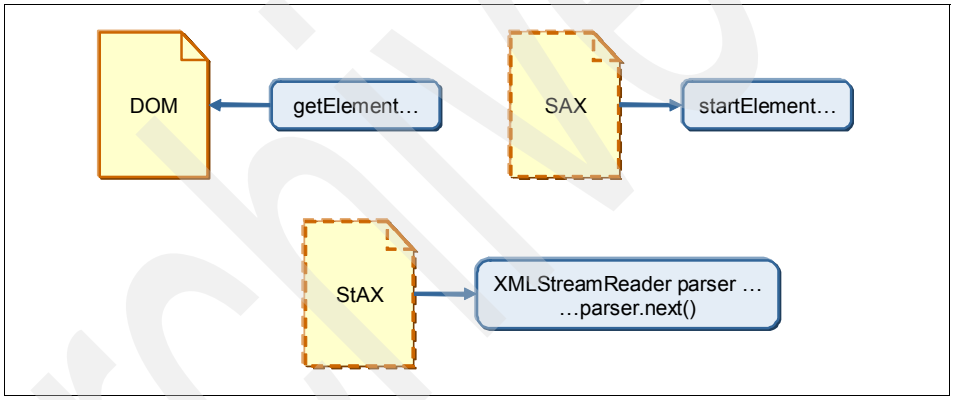


Figure 1-6 Schematic comparison of XML parsers

The feature pack's Web services runtime uses a StAX parser to manipulate SOAP messages, but the StAX API is also available to programmers. This is particularly useful if you are coding a dynamic Web service application that needs to work at the XML level.

1.3.5 JAX-WS summary

We have discussed the reason JAX-WS is important to the feature pack — it provides an asynchronous programming model and makes Web services programming simpler and more flexible. To see more details on JAX-WS, refer to Chapter 7, "JAX-WS programming model" on page 175.

1.4 Policy sets

The feature pack introduces policy sets as a mechanism to simplify the configuration of qualities of service, including the reliable messaging and secure conversation qualities of service. For example, to make a Web service reliable, you associate the reliable messaging policy set with the service.

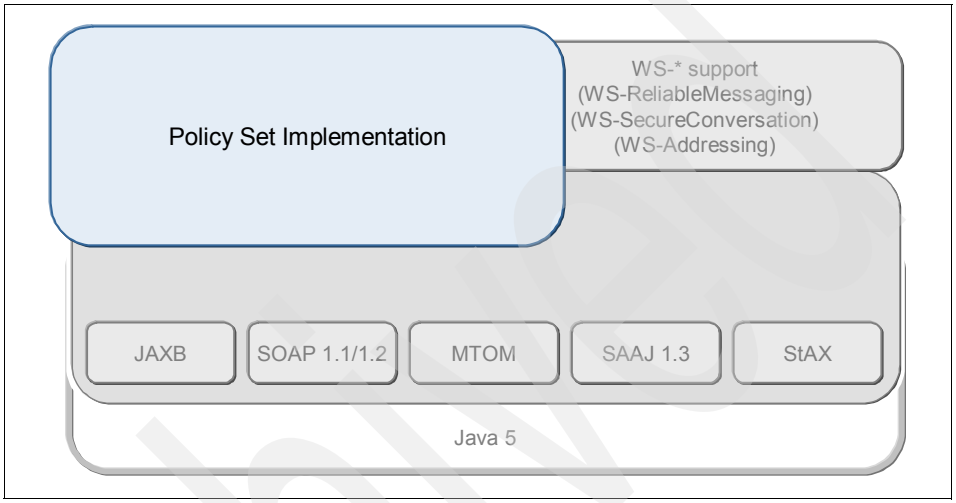


Figure 1-7 The policy set focus of the big picture

1.4.1 Details

A policy is a configuration of a quality of service. A policy set is a collection of policies. A policy set, when bound to an application, defines the QoS on that application. Figure 1-8 illustrates the concepts. Note how different bindings can be applied to the policy set to vary the quality of service.

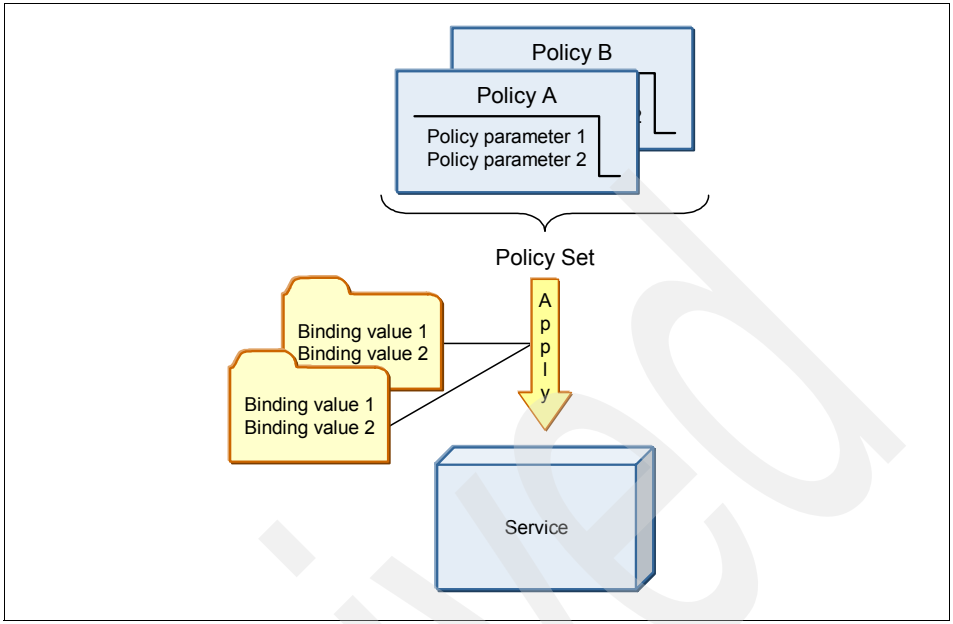


Figure 1-8 Policy sets

You can think of a policy set as a template QoS. When you apply a policy set to an application, you create a binding that contains the application-specific variables for that policy set. For instance, the WS-Security policy defines, among other things, the parts of a message that are encrypted and signed, but not how they are encrypted and signed; when you create the binding for that policy on an application, you specify the particular certificates used to encrypt and sign those parts of the message. Each policy set has a default binding, but in some cases, such as WS-Security, the default will likely not be appropriate for your particular use.

When you combine policies in a Policy set, for example, combining policies for WS-SecureConversation and WS-ReliableMessaging, you have the opportunity to customize the policies. For instance, you can specify which parts of the WS-ReliableMessaging message have to be secured (see 11.7.3, “Policy set for secure conversation client” on page 447 for an example).

You can apply policy set QoSs to an application at development time, just as you do today with deployment descriptors. A big advantage of a policy set is that you can also choose to apply it to an application operationally via the administrator console.

WebSphere Application Server provides an extensive list of policy sets for your use. If these policy sets do not meet your needs, you can also build your own. The list of pre-packaged policy sets includes these:

- The WS-Addressing default policy set consists of the WS-Addressing policy.
- The SecureConversation policy set consists of WS-Addressing and WS-Security policies.
- The RAMP default policy set consists of WS-Addressing, WS-ReliableMessaging, and WS-Security policies.

1.4.2 Issues

Policy sets are new, and there are some things to be aware of and gaps to be filled in future releases.

Bindings are part of the EAR

Although policy sets need not be bound to an application development time, they are still attached to an EAR file. When you version an application with dynamically configured policy sets, you should *update* it in the Administrator console; do not delete the old EAR and install the new EAR — if you do, you would lose your policy set and binding configuration.

What about WS-Policy?

A policy set contains policies. But these policies are not WS-Policy compliant. They follow an early release of WS-Policy, so the direction is clearly that they will follow WS-Policy in some future release. But for the feature pack, WS-Policy came out too late. The feature pack was first released in mid-2007 and WS-Policy only became a W3C recommendation in September of 2007. It would be convenient to provide standardized policy sets with services to configure clients, regardless of the client platform — something for the future, we hope.

Importing and exporting policy sets

Policy sets can be imported and exported. Policy set bindings, however, cannot yet be imported or exported, though that feature is part of future plans. In the interim, you can manually copy policy set and binding files. You must manually copy these files if you are building a thin client that requires a policy set's QoS.

1.4.3 Policy set summary

The ability to configure your qualities of service long after the development phase of a project is a very good feature. It moves QoS out of development and into the administrators' hands. Developers can still bind policy sets at development time if that is the way your environment operates, but if you prefer to have your administrators be responsible for QoSs, then policy sets enable them to do their jobs more easily. Refer to Chapter 8, "Policy sets" on page 247 for more details.

1.5 Summary

In this chapter we have discussed the reason for the feature pack and, at a high level, the new features that have gone into the feature pack. In the next chapters, we discuss some business problems that the feature pack functionality helps to solve. Even deeper into the book, we go into details of each of the new features.

2

History and roadmap

The relatively short history of Web services consists of many products from a multitude of vendors with claims of standards compliance, interoperability, and completeness. Users have found that the realities fall short of the ideals, and that interoperability is frequently a problem. In this chapter we place the Feature pack for Web services in context in the history and roadmap of Web services.

2.1 Standards and profiles

The Feature pack for Web services introduces the third Web services engine for WebSphere Application Server, bringing along with it new programming models and qualities of service reflecting the progress in the world of Web services in recent years. Figure 2-1 is a history of specifications, standards, and profiles.

In many respects, the three generations of Web services engines in WebSphere Application Server reflect three stages in the history of Web services: SOAP, WS-*, and Web Services Interoperability organization profiles. Profiles bring together a number of existing standards, defining how they work together, and introducing restrictions necessary to improve interoperability. It is also key that, at this point in time, many specifications — documents defining behavior without support of a standards body — have made their way through the standards process, resulting in new, standardized versions.

This has a parallel to the underlying Apache Web services implementation used for each generation. Initially, IBM built SOAP4J and contributed it to Apache (which became known as Apache SOAP), and WebSphere version 4.0 included it as a simple SOAP engine. When JAX-RPC 1.0 came out, WebSphere v5.0.2 was built using a JAX-RPC based programming model on top of Apache Axis 1.x. This runtime did include support for some basic WS-* specifications. Finally IBM contributed the JAX-WS support on top of Apache Axis 2, and includes it in Feature pack for Web services. The Feature pack for Web services support includes an implementation of JAX-WS, core SOAP functionality, and WS-Addressing, WS-ReliableMessaging, and WS-SecureConversation.

The Feature pack for Web services supports the IBM Reliable Asynchronous Messaging Profile (RAMP), bringing together the final versions of WS-ReliableMessaging, WS-Addressing, and a draft version of WS-SecureConversation. This was the basis of the design for the Feature pack for Web services, and allows for secure, reliable, and asynchronous Web services message exchange.

In the time since the Feature pack for Web services was released, further progress has been made with a number of standards and profiles. The Web Services Interoperability organization (WS-I) will soon release the Reliable Secure Profile, which is similar to RAMP and additionally includes WS-SecureConversation version 1.3:

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

Beyond wire level message interoperability, WS-Policy version 1.5 has been released, which will allow Web services configuration to be stored in an interoperable format, allowing configuration to be shared between platforms.

Specification	
Standard	
Profile	
Existing Specifications/Standards/Profiles	
May 2000	SOAP 1.1
June 2003	SOAP 1.2
March 2004	WS-Security 1.0
April 2004	WS-I Basic Profile 1.0
August 2004	WS-Addressing "Submission"
September 2004	WS-Policy 1.2
February 2005	WS-Reliable Messaging 1.0
February 2005	WS-SecureConversation 1.0
August 2005	WS-Addressing 1.0
February 2006	WS-Security 1.1
April 2006	WS-I Basic Profile 1.1
February 2007	WS-Reliable Messaging 1.1
March 2007	WS-SecureConversation 1.3
March 2007	WS-I Basic Security Profile 1.0
September 2007	WS-Policy 1.5
Predicted Specifications/Standards/Profiles	
June 2008	WS-I Basic Profile 1.2
	WS-I Reliable Secure Profile 1.0
December 2008	WS-I Basic Profile 2.0

Figure 2-1 History of Specifications, Standards, and Profiles (not exhaustive)

2.1.1 Current profiles

Profiles bring together existing standards, introducing clarifications and restrictions to the standards they are based on. Each profile reflects implementation and real-world experience in the use of a standard. During the development of a standard, it is sensible to leave implementation options open if the “right” answer is not known. With real-world experience, these answers become clear, and it is a substantial help for interoperability when the open options are reduced to the single best option.

WS-I Basic Profile 1.0/1.1

The best known profiles, the WS-I Basic Profiles v1.0 and 1.1 are based around SOAP, WSDL, XML Schema, and UDDI. The relative age of these profiles means that there are a large number of compliant tools and runtimes available, and a large number of services deployed that are already compliant. This includes services generated by Rational Application Developer v6 and deployed on WebSphere Application Server v6.1. Support for these profiles, as the basis of much of the Interoperability in the last few years, is a required part of JAX-WS 2.0, and hence is supported by the Feature pack for Web services.

WS-I Basic Security Profile 1.0

The Basic Security Profile is a follow-on to the Basic Profile, focusing on improving interoperability of secured Web services. This includes transport level security, message level encryption and signing, and username authentication. This profile is supported by the Feature pack for Web services. Some of the standards included in and constrained by the profile are: HTTPS, WS-Security 1.0, Basic Profile 1.1, XML-Signature Syntax and Processing, XML Encryption Syntax and Processing, and Web Services Security: UsernameToken Profile 1.0.

Reliable Asynchronous Messaging Profile 1.0

In 2005, in advance of standardization of a number of specifications, IBM, Ford Motor Company and DaimlerChrysler released the Reliable Asynchronous Messaging Profile (RAMP). This profile includes WS-Addressing, WS-ReliableMessaging, and WS-SecureConversation, and was driven by specific Business to Business scenarios encountered by the contributors.

This profile drove the decision making process during the development of the Feature pack for Web services, determining what was included.

2.1.2 Future profiles

The effort to define further profiles continues, building on new and undated standards.

WS-I Reliable Secure Profile 1.0

In early 2006, RAMP 1.0 was submitted to the WS-I as the basis of a new working group to further develop the profile in that forum. The WS-I opted to split up RAMP, directing the WS-I Basic Profile Working Group to develop a Basic Profile 1.2 including WS-Addressing. It also set up the Reliable Secure Working Group, chartered with developing the Reliable Secure Profile (RSP) 1.0. The RSP will build on the WS-I Basic Profile 1.2 and WS-I Basic Profile 2.0, and includes WS-ReliableMessaging and WS-SecureConversation, and will satisfy the same scenarios that RAMP was developed for.

WS-I Basic Profile 1.2

The first of two follow on versions of the WS-I Basic Profile, version 1.2 adds WS-Addressing, MTOM for SOAP 1.1, and errata from Basic Profile 1.1 into the mix. This essentially profiles the “A” from RAMP, and by adding WS-Addressing to the Basic Profile allows compliant Asynchronous services. At the time of writing, the WS-I Basic Profile 1.2 is not yet complete and is currently expected to be complete in the first half of 2008.

WS-I Basic Profile 2.0

The second follow on version of the Basic Profile is based on 1.2, and includes an overhaul to be based on SOAP 1.2 and MTOM/XOP for SOAP 1.2. This profile is at an early stage in development and is currently expected to complete sometime in 2008.

The relationships between these profiles and immediate WS-* standards and specifications the profiles refer to is shown in Figure 2-2.

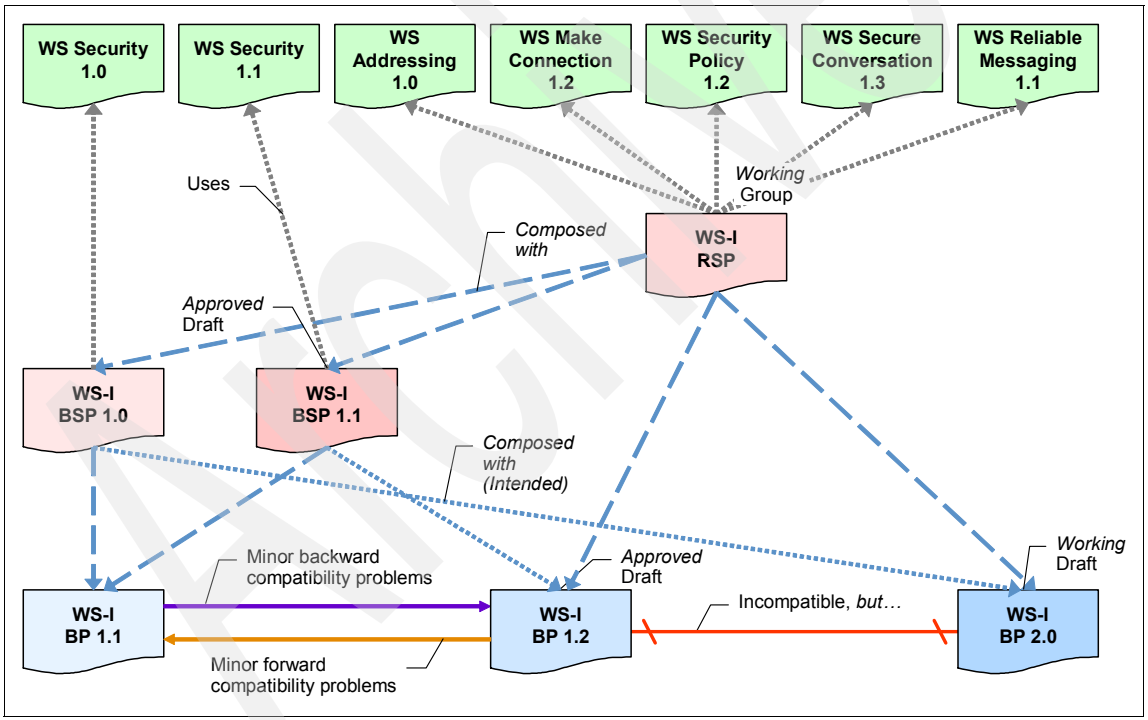


Figure 2-2 Relationships between WS-I profiles and some WS- specifications*

2.2 Programming models

WebSphere Application Server with the Feature pack for Web services includes the deprecated Apache SOAP API, JAX-RPC 1.1, and JAX-WS 2.0 APIs. The changes in the programming models provided reflect the changing standards and direction in the Web services space. Figure 2-3 shows the programming models used in the Feature pack for Web services and what implementation is being used.

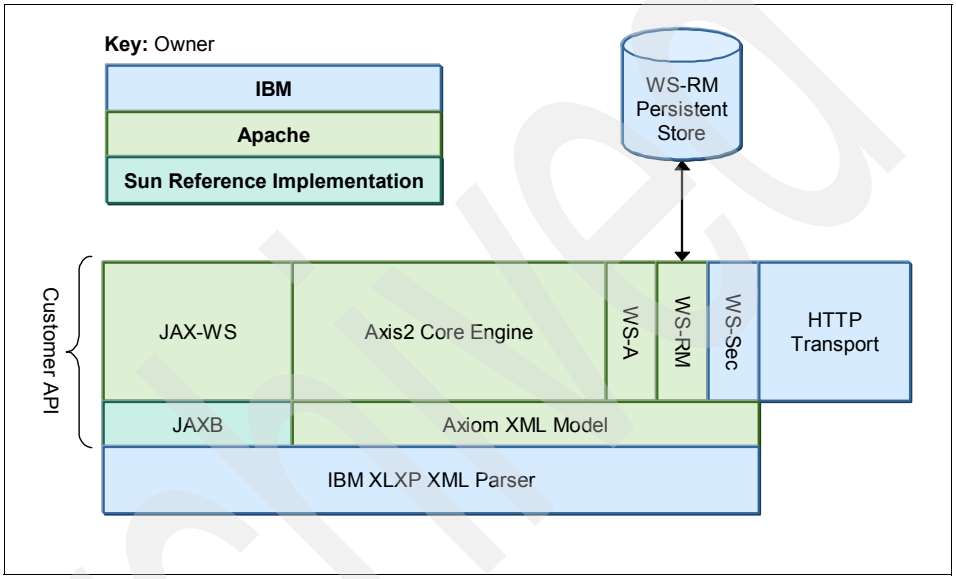


Figure 2-3 Open and proprietary components in the implementation

2.2.1 JAX-RPC

As a result of the early interest in Web services, the Java Community Process developed the Java API for XML Remote Procedure Call (JAX-RPC), building on early experience with SOAP 1.1 and WSDL 1.1. This became a popular API for implementing Java based Web services, and was included in J2EE™ 1.3, and subsequently in J2EE 1.4 and J2EE 5.

JAX-RPC 1.1 is retained as part of J2EE 5, and hence it remains a fully supported, non-deprecated part of WebSphere Application Server. If at some point the Java Community Process removes or deprecates JAX-RPC from J2EE, the standard WebSphere Application Server deprecation policy would apply. It is possible to use JAX-RPC clients with JAX-WS services and JAX-WS clients with JAX-RPC services as long as any qualities of service are supported by both, and that SOAP 1.1 is used.

2.2.2 JAX-WS

After the success of JAX-RPC 1.1, the Java Community Process expert group decided to create a new API reflecting the progress that had been made with new standards, and the fact that Web services were now being used in a document centric rather than RPC style. This effort was originally known as JAX-RPC 2.0, but reflecting the non-RPC uses it was renamed “Java API for XML Web services 2.0”.

JAX-WS 2.0 introduces support for SOAP 1.2, MTOM, document centric message exchanges, and the WSDL XML/HTTP binding for cases where SOAP is not required. It also exploits additions to Java 5 to ease development and deployment. This includes use of generics for holder types, java.util.concurrent classes for the asynchronous clients, and annotations to complement or replace XML deployment descriptors.

The Feature pack for Web services introduces the JAX-WS 2.0 APIs to WebSphere Application Server, in advance of J2EE 5.

Looking beyond the Feature pack for Web services, the JAX-WS APIs are also included in the Java SE 6 release. The implementation in Java SE does not include many of the advanced quality of service implementations provided by the Feature pack for Web services. The JAX-WS APIs continue to evolve, with JAX-WS 2.1 now complete and included in recent Java SE6 release, and hence will be available in future application servers based on Java SE 6. The JAX-WS 2.1 revision adds standardized WS-Addressing APIs and JAXB 2.1.

2.3 Products and interoperability

There is a tension between innovation and standards, exemplified by releasing policy sets in the Feature pack for Web services ahead of the standardization of WS-Policy and specifications, such as WS-ReliableMessaging, that use WS-Policy to make policy assertions. The result is that policy sets included in the Feature pack for Web services provide a convenient, declarative configuration model for configuring a Web service (Figure 2-4). However, we shall have to wait for incorporation of WS-Policy 1.5, which is now a W3C recommendation, for interoperability with other products and automating the configuration of clients (Figure 2-5).

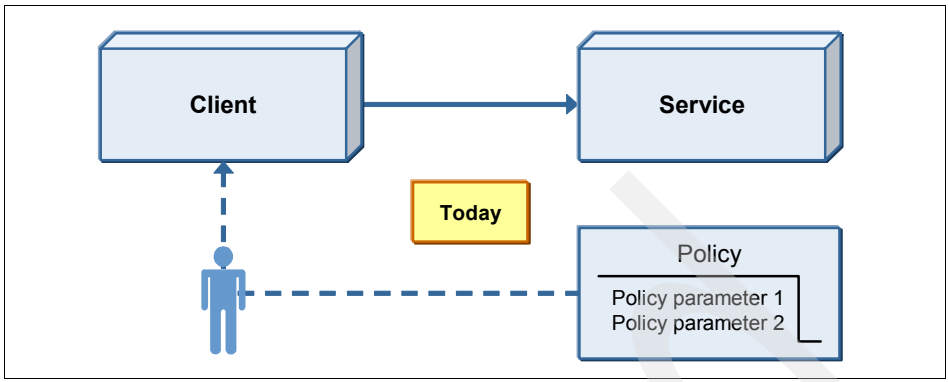


Figure 2-4 Manual configuration of Client QoS

Looking forward, policy sets provides the groundwork for a fully WS-Policy compliant configuration model that will allow for configuration to be shared interoperably across IBM and other vendor products. WS-Policy will also allow for more dynamically configured clients, whereby a server can describe a choice of qualities of service in its WS-Policy and the client can make a choice and configure itself appropriately.

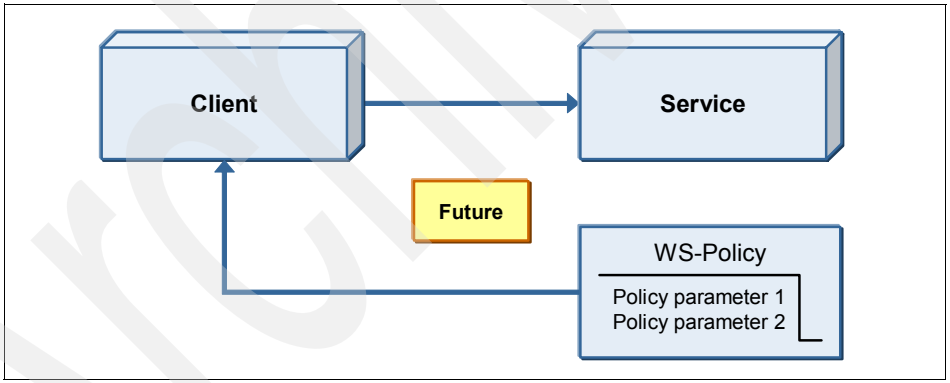


Figure 2-5 Moving towards automated client QoS configuration

The Feature pack for Web services embeds Apache Axis2 as its Web services engine and JAX-WS implementation as shown in Figure 2-3 on page 28. IBM is also using the core Axis2 engine and some of the quality of service implementations in other products, including WebSphere Message Broker v6.1. This will aid interoperability between IBM Web services enabled products using the common component, and aid interoperability between all IBM products and non-IBM products as the interoperability work done for WebSphere Application Server will be inherited by other products using the component. It will also aid interoperability between IBM products and Apache Axis2.

2.3.1 EJB Support[1]

You cannot implement a JAX-WS service as a session bean using the Feature pack for Web services. The “top-down” Web services wizard offers a Java bean implementation skeleton, but not an EJB™ skeleton. Likewise, the “bottom-up” wizard only offers a Web service wrapper for a Java bean, and not for an EJB. You have two alternatives to create a Web service to call an EJB:

- JAX-RPC
- Create a JAX-WS Java bean proxy for the EJB

Released in late 2007, the Feature Pack for EJB 3.0 for WebSphere Application Server v6.1 provides access to the EJB 3.0 programming model on WebSphere Application Server V6.1. It is possible to install both that feature pack and the Feature pack for Web services on the same WebSphere Application Server, however, it is not possible to write an EJB 3.0 bean exposed directly as a JAX-WS Web service. But it is possible to proxy the EJB with a redirector Java bean Web service.

In order to do this, you should generate WSDL and XSD files from the EJB 3.0 implementation bean by annotating it with **`@WebService`** and running **`wsgen`** to create the files. You then run the Web services wizard to generate a Java bean skeleton from the WSDL. The generated bean has all the same methods and signatures as the EJB 3.0 bean. It is a simple matter to implement the Java bean by redirecting the method calls to the EJB 3.0 bean (Figure 2-6).

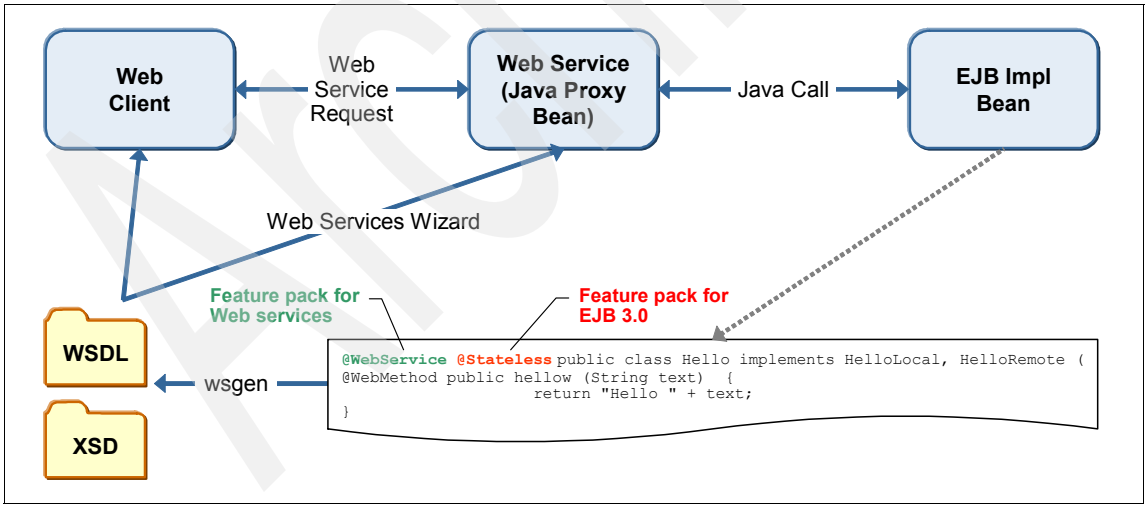


Figure 2-6 Emulating Web services with EJB 3

[1] The restrictions documented in this section are expected to be resolved in WebSphere Application Server v7.0.

Randy Schnier describes how to do this at:

http://www-128.ibm.com/developerworks/forums/message.jspa?messageID=14051200#14051200

2.3.2 WebSphere Process Server and WebSphere ESB

Within the WebSphere Application Server based product stack, WebSphere Enterprise Service Bus 6.1.0.1 and WebSphere Process Server 6.1.0.1 should tolerate installation of the Feature pack for Web services. Toleration means that EJBs developed using functions in the Feature pack for Web services are deployable and run just as if they are hosted by WebSphere Application Server. At the time of writing, installing the feature pack on WebSphere Enterprise Service Bus or WebSphere Process Server is not advised until some fixes planned for WebSphere Application Server v6.1.0 Fix Pack 15 are shipped. (Nor is advised to install WebSphere Enterprise Service Bus or WebSphere Process Server on top of WebSphere Application Server Network Deployment with the feature pack installed).

2.3.3 Service Component Architecture

Currently, plans for exploiting the capabilities of the Feature pack for Web services through the Service Component Architecture (SCA) Web services bindings have not been published.

2.3.4 Web Services Interoperability organization profiles

As the focus in the standards world is migrating away from the base technologies to profiles and collections of technologies, so are the scenarios used for interoperability testing. Current scenarios are focusing on profiles such as RAMP and RSP, and business scenarios that combine different capabilities in different areas of the scenarios. As interoperability improves at the technology level, the interoperability focus will continue to move to higher levels, towards real-world production scenarios.

2.3.5 Windows Communication Foundation (WCF)

At the time the Feature pack for Web services was released in June 2007, the most up-to-date version of Microsoft WCF was 3.0. This is the version that all the pre-release WCF interoperability work was carried out with, and which most of the interoperability advice in this book is related to. However, the interoperability work continues, with work ongoing with WCF 3.5 now that it is available.

Part 2

Benefits of the feature pack

We have called this part of the book “Benefits of the feature pack” because we wanted to answer a number of questions: “Should I install and use the feature pack or not?”, “Should I use WS-ReliableMessaging or WebSphere MQ for building reliable distributed applications?”, and then, when developing Web services, “Should I use JAX-RPC Web services or JAX-WS?”.

There is no use in leaving these questions to be answered by a developer as an “implementation detail.” These questions must be thought through by the Enterprise Architect, or lead designer for a project, and a practical policy for installing and using the Feature pack for Web services agreed to by the organization.

The capabilities provided in the feature pack are component parts of a solution. But they change the odds of survival in the IT ecosystem, making possible different kinds of solutions, and ways of building and managing solutions. To help you think about how the Feature pack for Web services can change the way you architect solutions, we have looked at the capabilities of the Feature pack for Web services in three ways, in the following three chapters.

In Chapter 3, “Business scenarios” on page 35, we have described four business scenarios that could benefit from use of the feature pack:

1. Price comparison Web site
2. Insurance underwriter
3. Outsourced development
4. Check clearing

These are four examples of solutions that benefit from the feature pack; you will surely think of others.

For each solution, we have identified one or two common patterns of usage of the feature pack to implement the scenario. These are described in Chapter 4, “Facets and patterns” on page 55. The patterns are:

1. Fan-out with responses
2. Roaming client
3. Externalization of application QoS
4. Robust fire-and-forget
5. In-band transfer of binary data

You might be able to think of other patterns of usage for the capabilities in the Feature pack for Web services and identify business scenarios that benefit from your patterns. We do not pretend that our list is exhaustive, but we hope it is suggestive.

Finally, in a more traditional approach to identifying the benefits of a software component, in Chapter 5, “Technical advantages” on page 67, we list the capabilities of the Feature pack for Web services and describe the possible benefits of each capability.

3

Business scenarios

In this chapter we outline several business scenarios. Each of the scenarios motivates one or more of the capabilities introduced by the Feature pack for Web services in a real-world context, and describes business benefits delivered by using these capabilities.

We discuss the following scenarios:

3.1 Scenario matrix

The following chapter “Facets and patterns” describes what might loosely be called design patterns that exemplify how to use the functions in the feature pack. Looking ahead to the next chapter, Table 3-1 indicates which patterns are employed in each of the business scenarios.

Table 3-1 Scenario matrix

Pattern / Scenario	4.1, “Fan-out with responses”	4.2, “Roaming client”	4.3, “Externalization of application QoS”	4.4, “Robust fire-and-forget”	4.5, “In-band transfer of binary data”
Price comparison	✓				
Insurance underwriter		✓			
Outsourced development			✓		
Check clearing				✓	✓

3.2 Price comparison Web site

Numerous Web sites offer price comparison services. These types of services have a very broad applicability, covering products ranging from general retail goods to travel, insurance or energy suppliers. Regardless of the product or service being searched, the fundamental system design is similar.

This scenario demonstrates application responsiveness benefits of using an asynchronous client. This provides a better user experience and improved customer satisfaction.

3.2.1 Business requirements

Figure 3-1 illustrates the components of this scenario. A customer, using a Web browser, accesses the comparison Web-site. The comparison service talks to back-end supplier systems to retrieve price information relevant to the customer query. Results are then displayed to the customer as a Web page.

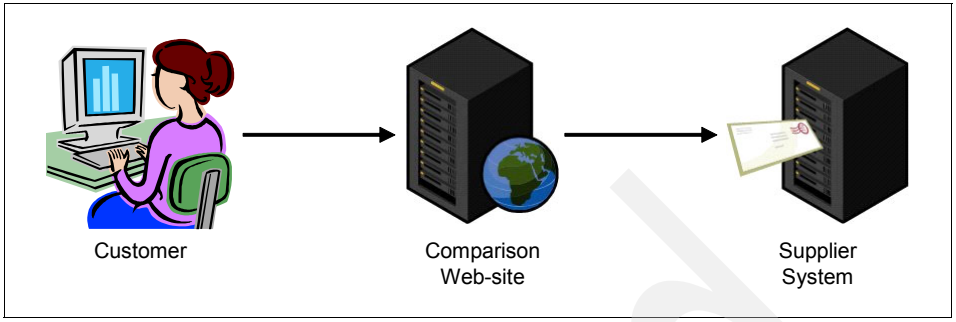


Figure 3-1 Price comparison overview

The key quality of service for this scenario is responsiveness for the customer, who is likely to simply give up and not use the comparison service if they have to wait a long time for responses. If a response from a supplier takes too long or fails for some reason, we can ignore this and report partial results to the customer. If we are able to return results to the client “piece-meal” then we can report back the results to the customer as they arrive. If we can only return results to the client when we receive a request from the client, then we can temporarily store late results, and update the client next time we receive a request.

There is no requirement for service invocations to the suppliers to be ordered. In fact, calling each supplier in turn and waiting for a response before calling the next supplier is counter-productive; it increases the time taken to call to all the suppliers and lowers responsiveness for the customer. As the number of suppliers increases, the performance penalty of sequential calls increases.

There are no security or reliability requirements for the communication between the customer and the comparison service, or for the communication between the comparison service and the suppliers.

3.2.2 System overview

As illustrated in Figure 3-2, the basic design is for a client to send a request to the price comparison service. The comparison service then requests prices from a number of suppliers. The suppliers return results to the comparison service, and the comparison server returns results to the client.

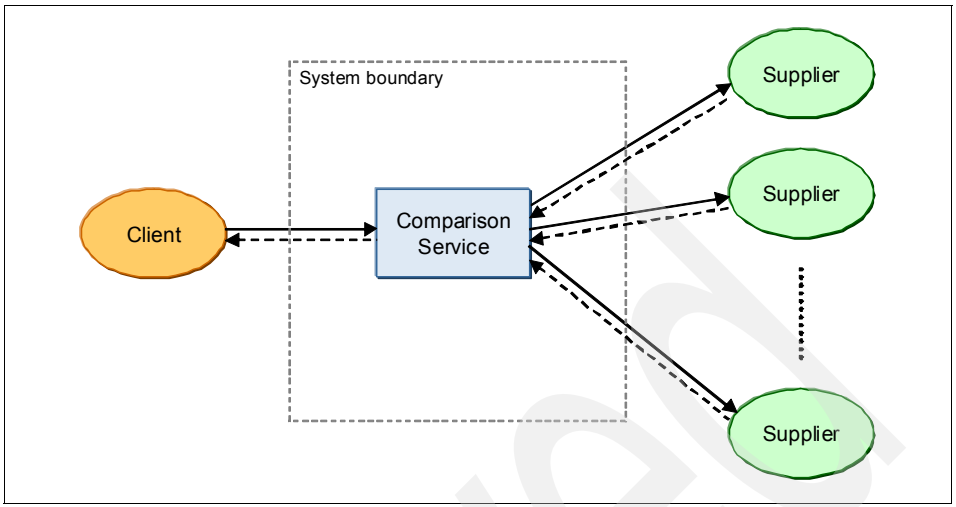


Figure 3-2 Price comparison system

The scope of the system encompasses just the comparison service. The client and supplier services are both third parties that the implementer of the comparison service would not have direct control over. We assume that the suppliers have provided us with Web Services Interoperability organization Basic Profile 1.0 or 1.1 compliant services.

In the most obvious example of a price comparison Web site, the client is a Web browser. The client could just as easily be a Web service client, or the comparison service could provide a variety of access methods to cater for a range of client types.

For the communication between the comparison service and back-end suppliers, Web services would be a natural choice to due to their standards-based nature, which facilitates loose coupling between the implementation languages and platforms.

3.2.3 Solution

The overall solution is composed of two parts:

- Communication between the client and the comparison service.
- Communication between the comparison service and the supplier services.

The interesting part of this scenario from a feature pack perspective is the communication between the comparison service and suppliers.

Client to comparison service communication

There are several methods of communication between the client and the comparison service that can be appropriate. The method chosen will depend on the client audience for the comparison service. If required, several methods of client access to the comparison service could be provided rather than a single access method.

The most likely method of client access is Web-based, where the comparison service is provided using a Web-page. In this case the client communication is a simple HTTP request, such as a form submission. The comparison service would be programmed to wait some amount of time, or until all the suppliers have responded, and then return all the available responses to the client. The comparison service could be programmed to cache the suppliers' responses, and if the client requests the same comparison again, the comparison service might now have some late arriving responses.

Some sites might choose a richer Web-browsing experience by making use of AJAX, in which case the client communications can be in the form of requests to update the client with information from the suppliers rather than returning a full form of information on each request.

In both cases the HTTP request matches the client quality of service requirements as it expects a timely response and does not guarantee reliability.

Comparison service to supplier communication

The suppliers have provided a request/response Web service interface for price query. The requirement for good response times to the client suggest that the supplier services are called in parallel.

Existing solution

In the JAX-RPC client programming model, request/response invocations imply sequential calling of each of the supplier services, waiting for the response to each request before calling the next supplier.

Feature pack for Web services solution

With the JAX-WS programming model introduced in the Feature pack for Web services, we make the requests to the suppliers directly, one after the other, and then wait for all of responses, or until we decide to time out and return the responses we have received to the customer. The pattern for this solution is described in 4.1.1, "Fan-out programming model" on page 56.

3.3 Insurance underwriter

This scenario describes an insurance underwriter service. This service allows agents, running an insurance agent application on their local laptop, to set up an insurance policy with the underwriter on behalf of a customer. The agents might be working at remote offices or might be in the field connected to the Internet using mobile phone networks. This is illustrated in Figure 3-3.

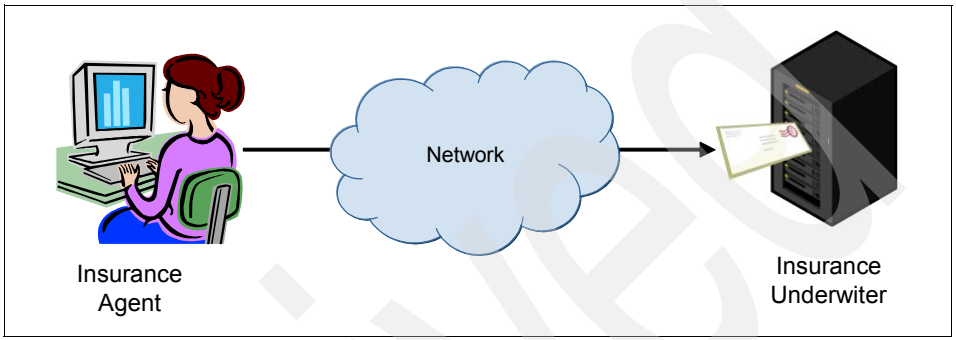


Figure 3-3 Insurance underwriter overview

The main feature of this scenario that we are going to focus on here is the importance of not submitting duplicate requests to set up an insurance policy, and transmitting the requests securely across the unmanaged Internet. Duplicate requests are typically an issue when availability of the network connection between client and service cannot be guaranteed. In our case, the requesting application can be left in doubt whether, after the agent pressed the submit button, the failure to receive an acknowledgement from the server means that the request never reached the insurance underwriter, or that the acknowledgement was never returned to the insurance agent. Figure 3-4 illustrates this.

Examples of other related scenarios include:

- An inventory barcode scanner wirelessly transmitting warehouse stock information to a central system
- A purchasing application placing an order with a retailer's ordering system across the Internet

The unavailability of the network might be due to many different kinds of reasons, from lack of connection, server maintenance periods, congestion, and equipment or software failure.

There are many points of failure between submitting the request, and receiving the response, and all of the failures are eventually diagnosable either by the client, or by the server if they re-synchronize after the failure. Figure 3-4 identifies six main points of failure within the indoubt window when calling a remote service. If the failure is at points 1 to 3, then the insurance policy is not created. If the failure is at points 4 to 6 then the policy is created, but the client does not know this. Without synchronization, the best a client can do is to narrow the indoubt window to between 3 and 6, which still leaves the client indoubt as to whether the policy was created or not.

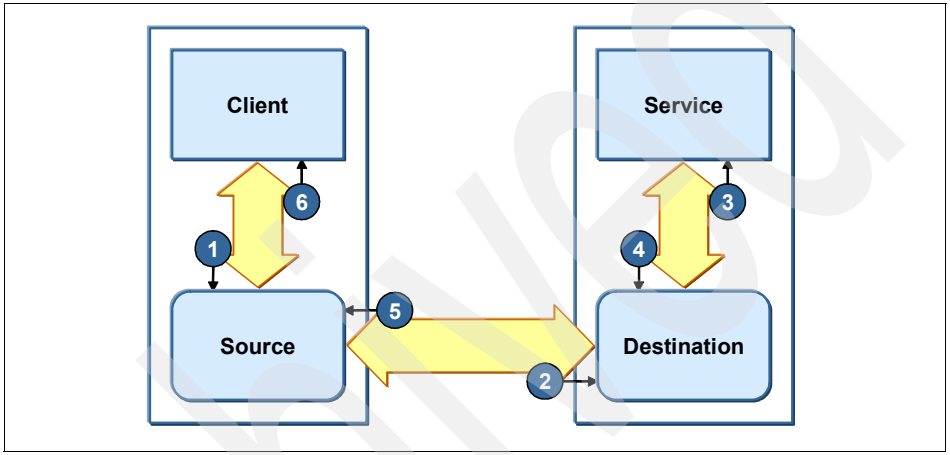


Figure 3-4 Six points of failure in remote service call

Resynchronization code is expensive to develop, and introduces technical administrative tasks. In many cases, manual error recovery at the business level, perhaps prompted by not receiving a confirmatory e-mail, or by comparing logs of new policies created recorded at the insurance agent and at the insurance underwriter, is a good enough solution. Also, it is cheaper than adding code and technical administrative procedures to the client and server applications to automate the re-synchronization.

An alternative is to extend robust solutions, such as WebSphere MQ or distributed transaction processing, out to proprietary clients running on the insurance agents systems. While these solutions are excellent, they do introduce financial and administrative issues that might not suit the business.

The feature pack offers a third way: using reliable messaging to overcome the unreliability of the network connection without requiring the insurance agent to install proprietary client software.

The feature pack offers three qualities of service for reliable messaging:

1. Unmanaged non-persistent
2. Managed non-persistent
3. Managed persistent

Therefore, you have an 80/20 type of trade-off between just making the network part of the interaction robust (unmanaged non-persistent), and making the solution robust up to the request and response interfaces in the client application.

This scenario demonstrates benefits of reliability and simplification of application logic provided by the Feature pack for Web services.

3.3.1 Business requirements

The insurance underwriter service sets up an insurance policy on behalf of a customer. It is important that duplicate requests are not received by the underwriter service.

The network connection between the agent and the underwriter service might be unreliable at times; particularly if the agent is connected from a remote site using the mobile phone network. Intermittent service interruption in the network connection must be tolerated.

Security requirements dictate that policy information can only be submitted by authorized agents and that data must be transmitted securely across the public network. Specifically, any data transmitted on the public network must be encrypted.

The application footprint and system management overhead on the agent laptops must be kept to a minimum to reduce the risk of problems and to keep costs down.

3.3.2 System overview

Insurance agents run an insurance application on their laptops to capture customer policy information and obtain quotes for different policies. Once quotes are obtained and the customer accepts the policy, the agent application submits policy details to the central insurance underwriter service to create the new policy.

The insurance underwriter service is a request/response service. It receives policy information from the agent and returns confirmation details, including the customer's new policy number, as shown in Figure 3-5.

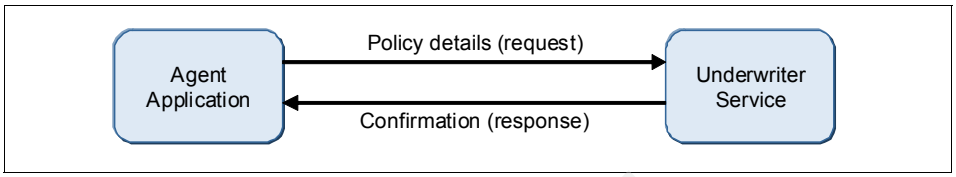


Figure 3-5 Underwriter service invocation

The network connection might suffer intermittent service outages, for example, if an agent temporarily loses their mobile phone signal. For traditional request/response service invocations using SOAP over HTTP, this presents a problem when once-only delivery of the message is required. This is the case with this scenario because the service invocation has side effects and so it must not receive duplicate requests.

The problem is caused by the uncertainty introduced by the use of HTTP in failure conditions. Because HTTP does not guarantee once-only delivery of messages, it is not possible for the client to accurately infer the point at which a request/response message failed. Figure 3-6 and Figure 3-7 illustrate the two possible points at which network failure might occur during a request/response invocation (corresponding to points 2 and 5 in Figure 3-4 on page 41).

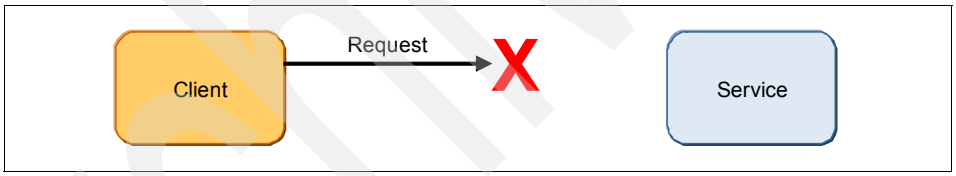


Figure 3-6 SOAP over HTTP request failure

On a request failure, the request message is never processed by the service. This leaves client and service in a consistent state because neither of them has successfully completed the service invocation. Error-handling on the part of the client might be required to re-submit the failed message, or it might be sufficient for the user to manually re-submit the request for an interactive application.

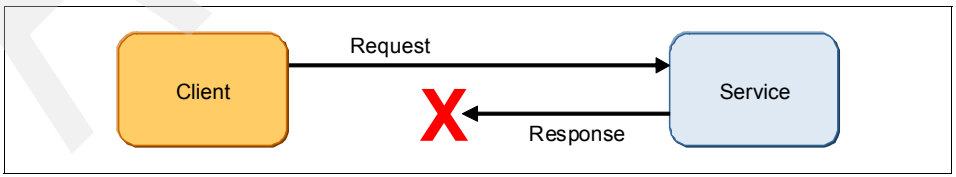


Figure 3-7 SOAP over HTTP response failure

On a response failure, the service has successfully received and processed the request message. Any side effects that occur as part of this processing have been carried out, for example, setting up the customer insurance policy. However, the client simply sees an error and might assume (incorrectly) that the request has not reached the service. The client and service are left in an inconsistent state.

When once-only delivery of a request is required using plain HTTP, both client and service must implement custom logic to ensure that:

- Requests already received by the service are not re-processed if received again later.
- Duplicate requests are answered by the original response message to ensure that the client gets a valid response to service invocations.

This logic can be cumbersome to implement, and couples the client and service in an “ad hoc” protocol that will have to be maintained whenever new clients or services are created. Therefore interoperability is significantly reduced in these cases.

3.3.3 Solution

The overall solution has two parts:

- Securing the communication between the insurance agent and the insurance underwriter
- Ensuring that the request to create an insurance policy creation is made once and once only for each insurance policy

Existing solution

Virtual Private Network (VPN) software is often used by remote workers to connect securely to a corporate network across the Internet. This type of secure connection might be sufficient to meet the requirement that data is transmitted securely across the public network. However, loss of network connections often interrupts a VPN session, which disrupts work for the agent.

VPN connections are sometimes difficult to set up through different ISPs, and connection parameters might have to be changed. VPN software also adds an additional component that must be managed, although it might suit some scenarios. These considerations all add to the cost of creating a robust VPN solution for mobile client facing workers.

Use of HTTPS and key-based client authentication satisfies the need for only authorized agents to submit policy information and for that information to be encrypted.

Reliable transports such as JMS deal with issues of unreliable network connectivity. In some cases this might not meet interoperability requirements because it requires a compatible JMS provider at both client and service ends, and the SOAP over JMS protocol is not fully standardized. In other cases, such as when the overall system is operated with a single business, this can be a suitable solution. For this scenario, the additional footprint and management overhead of a messaging infrastructure on the client machines and transmission across the public Internet make this choice less desirable.

It is likely that many systems rely on insurance agents ringing the help desk or scrutinizing end of day reports to reconcile any applications that did not return a successful response.

Feature pack for Web services solution

The key characteristic of WS-ReliableMessaging, when applied to request/response message flows, is once-only delivery of messages. This solves the requirements both for toleration of the unreliable network and for once-only message delivery. Because these requirements are met by the WS-ReliableMessaging infrastructure, the need for manual intervention is reduced — perhaps considerably if the insurance agent is working over unreliable networks.

Reliable messaging can be used in conjunction with HTTPS to provide a secure transport. Using RAMP and thus WS-Security to authenticate and secure the requests is an alternative to using HTTPS.

The Feature pack for Web services provides three quality of service levels for reliable messaging. These are:

- Unmanaged non-persistent
- Managed non-persistent
- Managed persistent

The Unmanaged non-persistent quality of service is the most likely to be employed in this scenario, because it imposes the least overhead on setting up and administering the client. For an interactive mobile application in which there is a person in attendance throughout the interaction, it is reasonable to treat problems such as the client or server going down during an interaction as something that will be fixed manually.

Referring to the “Six points of failure in remote service call” in Figure 3-4 on page 41, the frequency of failures 2 and 5 will be reduced by using the unmanaged non-persistent quality of service. Failures at other points are more likely to be returned to the client to diagnose, leaving fewer indoubt situations to be resolved manually.

The design pattern for this scenario is explained in 4.2, “Roaming client” on page 57.

3.4 Outsourced development

This scenario considers the case where the organization that develops a Web service application is different from the organization that manages the deployed application and administers network policy. This separation might simply be because the application is developed by a development team and then managed by an operations team within the same business, or it might be that the application is written by a different company, such as a business partner.

This scenario demonstrates cost savings and the ability to more rapidly implement operational changes by using Feature pack for Web services.

3.4.1 Business requirements

There is a cost to the business of managing the application after development is complete. The more work, resources, or time required to manage the application after development, the higher the overall cost to the business. There is a clear business requirement to minimize these costs for tasks such as:

- Changing the application to use different service endpoints
- Changing network security policies
- Changing reliability characteristics for Web service invocations

There is a particularly significant cost involved with any changes that require re-developing or re-packaging the application, because these are tasks that are typically carried out by the development team rather than the operations team. In the case of outsourced development, if might not be practical to re-develop portions of the application at a later date to implement operational changes.

3.4.2 System overview

The high-level development and deployment lifecycle for Web service applications often looks similar to Figure 3-8.

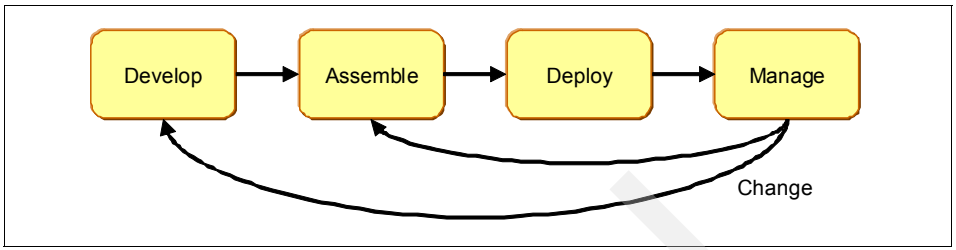


Figure 3-8 Inefficient application lifecycle

To realize cost benefits to the business, the lifecycle needs to look similar to Figure 3-9 in as many cases as possible. In other words, operational tasks such as changing security policies should be management tasks not requiring any re-development, re-packaging or re-deployment of the application.

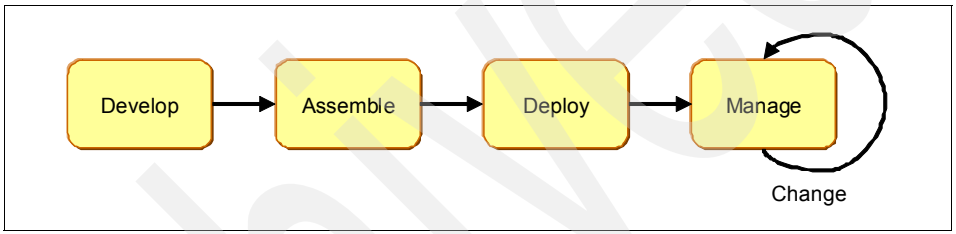


Figure 3-9 Efficient application lifecycle

3.4.3 Solution

Reduction of post-development costs for application management is addressed in two areas:

- Application development practices
- Administration capabilities

Development practices

Good application design dictates that to avoid re-developing or re-packaging application code, configuration should be externalized. JAX-WS and JAX-RPC both provide APIs for programmatically configuring variable values such as endpoint addresses, which are then read in from configuration files or from a Web services registry such as WebSphere Services Registry and Repository. This allows application management to be carried out by the operations team without requiring any development changes.

Administration capabilities

A major change introduced in the Feature pack for Web services is the use of policy sets for managing quality-of-service configurations such as security and reliability. The use of policy sets to configure quality of service attributes, rather than having this behavior defined by application code or by deployment descriptors packaged within the application EAR, helps to separate application management and development roles.

Policy sets provide cost and efficiency benefits in two important areas:

- Policy sets are reusable. A single policy set applies to multiple Web services. This saves work for the administrator and allows a uniform policy, such as a new corporate security policy, to be quickly applied to appropriate applications.
- Policy sets are managed using the WebSphere Application Server administration console. Changes are made without requiring re-development, re-packaging, or re-deployment of applications.

In 4.3, “Externalization of application QoS”, we describe the pattern for externalizing qualities of service.

3.5 Check clearing

Checks are paid in by customers and businesses at a many locations. Depending on local banking practices, this includes branches of the issuing bank, branches of other banks and post offices. Information on these checks, and images of the checks, must be transferred from the location or organization where they are paid in to the clearing system of the issuing bank.

Let us look at how this problem is solved today by check clearing systems, and then consider how reliable and secure Web services can be part of alternative solutions for scenarios similar to check clearing, but with different cost and governance characteristics that can open up new business opportunities.

Check clearing, and other funds transfer systems, are usually managed by a banking consortia, operating through a jointly owned company. The consortia solve the problem of defining and deploying a secure and reliable clearing system between its members. Traditionally the consortia have written or mandated proprietary software for the consortia to use, as well as specifying isolated networks to connect participants in the clearing system. The result is a highly reliable, available and secure clearing system, but with high entry costs for participants.

Reliable and secure Web services, by providing a standard and interoperable way of connecting clients and services reliably and securely, makes possible the definition of a standards based reliable and secure service based on protocols, rather than providing specialized software, hardware, and adapters to interface to a proprietary clearing system.

Availability remains an issue, if the service is to run across the Internet — typically the ISP becomes the weakest link in the network — and the alternative of using a Value Added Network (VAN) increases the costs significantly. An option, which has been implemented for a banking ATM network, is to use redundant network connections and different ISPs.

Some other examples of exchanging important electronic information in a secure manner between participants in a federated governance structure might be:

- Exchanging health records
- Electronic invoices
- Commodity transactions

3.5.1 Business requirements

This scenario defines fairly detailed business requirements for the solution. These can be broken down into non-functional and functional requirements.

Non-functional requirements

Due to the high-value nature of the data involved, key business requirements for this scenario are non-functional quality-of-service requirements; specifically security and reliability:

- Data transferred across the network must be encrypted to prevent confidential information from being stolen.
- Data transferred across the network must be digitally signed to ensure that it is not tampered with.
- Network communications must be authenticated to ensure that only trusted parties send and receive data.
- Each check must be processed exactly once. A check must not be processed more than once, and no check information can be lost.

This is a high volume business-to-business process with no human interaction, so there are no specific responsiveness requirements for the end-to-end check processing. What is important is that the overall throughput is sufficient to handle check volumes, and, if things go wrong, then recovery has to be automated, because the check volumes are so high as to preclude manual sorting of successful from unsuccessful clearings and resubmission of the failures.

It is desirable for this sensitive business-to-business communication to be carried out over public networks. This would represent a substantial cost and operational management saving over using leased lines and VAN technology to satisfy security and reliability requirements.

Functional requirements

Because the systems at each end of the communication will be automated “batch” systems, there are some functional requirements:

- The client-side system must be able to proceed with processing of checks without having to wait for the clearing system to receive or process the check information.
- Additional transactional processing must be carried out for each check, both in the client-side system and in the clearing system. This work should be carried out exactly once for each check that is communicated between the systems. In other words, the processing must be done as one transactional unit of work along with the associated send or receive of check information.

Check information communicated to the clearing system must include a scanned image of the check for audit purposes and to capture the customer signature.

3.5.2 System overview

As illustrated in Figure 3-10, the high-level design consists of a check processing system that communicates with a clearing service. The check processing system is a client to the clearing service.

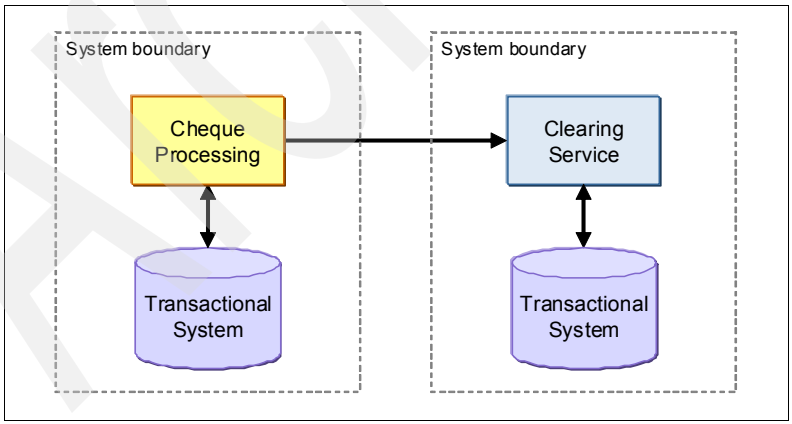


Figure 3-10 Check clearing system overview

It is important to note that a check processing system communicates with many different clearing service implementations. Likewise, a clearing service receives check information from many different check processing systems. Web services are a natural choice for communication between the check processing system and clearing service because these systems are potentially owned by different businesses and can be using very different software and hardware platforms.

Both the check processing and clearing service systems interact transactionally with local systems as part of the check handling process. For the check processing system, this transactional work involves logging check details for audit purposes. For the check clearing service, this transactional work includes the transfer of funds between accounts.

3.5.3 Solution

The business requirements are quite complex and the constraints that they impose dictate much of the solution. Functional requirements have the clearest impact on system design, so we start with them. We then address non-functional quality-of-service requirements.

Functional requirements

The requirements for the check processing and clearing services to be able to continue working independently requires that they are decoupled from each other. The client simply needs to deliver the check information to the clearing service. No response information is required. Therefore the communication between the check processing client and the clearing service is a one-way request.

The use of one-way requests aids decoupling of client and service because the client does not have to wait for a response message. One-way requests also facilitate more robust transaction message exchanges because the send and receive, at the client and service ends of the communication respectively, are done within the scope of a transaction. The flow of local processing and service invocation within the scope of a single transaction for the check processing application is illustrated in Figure 3-11.

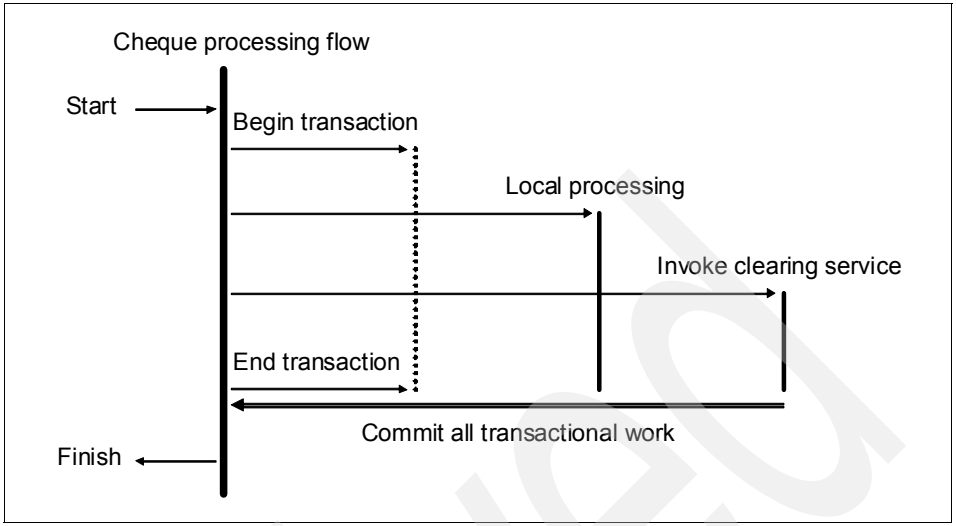


Figure 3-11 Check processing interaction diagram

The requirement for all data transferred across the network to be encrypted and signed suggests that the check image has to be included in the SOAP message rather than being transferred by some out-of-band mechanism.

Two standards exist for attachment of binary data, such as the check image, to SOAP messages: SOAP with attachments and MTOM. MTOM provides two advantages over the SOAP with attachments standard that are relevant to this scenario:

- MTOM is interoperable with Microsoft platforms, whereas SOAP with attachments is not supported by Microsoft platforms.
- MTOM allows the attachment to be included in the computation of the digital signature for the SOAP message, whereas the SOAP with attachments implementation in WebSphere Application Server treats the attachment as a separate entity to the SOAP message body for signing purposes.

The business requirement for all check information to be digitally signed requires the use of MTOM rather than SOAP with attachments because the check image attachment also comprises part of the check information.

Non-functional requirements

Quality-of-service requirements for this scenario focus on security and reliability. They are extremely important to this scenario and must be addressed carefully.

The use of one-way WS-ReliableMessaging for communication between client and service is a good match for the business requirements for reliability. Due to the high value of the data being transferred, no data loss during transmission is acceptable. To ensure that no data loss occurs in the event of a client or server failure during message transmission, a managed persistent WS-ReliableMessaging configuration must be used at both ends of the communication. This will ensure that messages are persisted to disk and provide robustness in the event of a server failure.

WS-SecureConversation is required in addition to WS-ReliableMessaging for security purposes. Without WS-SecureConversation, it might be possible for authorized clients to view data in another client's reliable message flow.

The use of reliable and secure Web services protocols over standard HTTP makes it potentially viable for this sensitive business-to-business communication to be carried out over public networks. Refer to 4.4, "Robust fire-and-forget" and 4.5, "In-band transfer of binary data" for more details on the patterns used.

Note: The WS-ReliableMessaging standard speaks permissively of the QoS applied to the endpoints, only requiring that the QoS of the endpoints has no affect of the WS-ReliableMessaging protocol.

> *"The protocol can also be implemented with a range of robustness characteristics ranging from in-memory persistence that is scoped to a single process lifetime, to replicated durable storage that is recoverable in all but the most extreme circumstances. It is expected that the endpoints will implement as many or as few of these reliability characteristics as necessary for the correct operation of the application using the protocol. Regardless of which of the reliability features is enabled, the wire protocol does not change."*
>
> OASIS Reliable Messaging Model, found at:
>
> http://docs.oasis-open.org/ws-rx/wsrm/200608/wsrm-1.1-spec-cd-04.html

You have to be sure of the endpoint capabilities provided by different suppliers and how they manage persistent endpoints. See Chapter 10, "Reliable messaging" on page 355 for more discussion.

4

Facets and patterns

In this chapter we describe several design patterns for technologies delivered in the Feature pack for Web services. These patterns reflect facets of the business scenarios but are approached from a technical rather than a business perspective.

4.1 Fan-out with responses

The fan-out with responses pattern describes an implementation that calls many services at one time as shown in Figure 4-1, and then collects the responses.

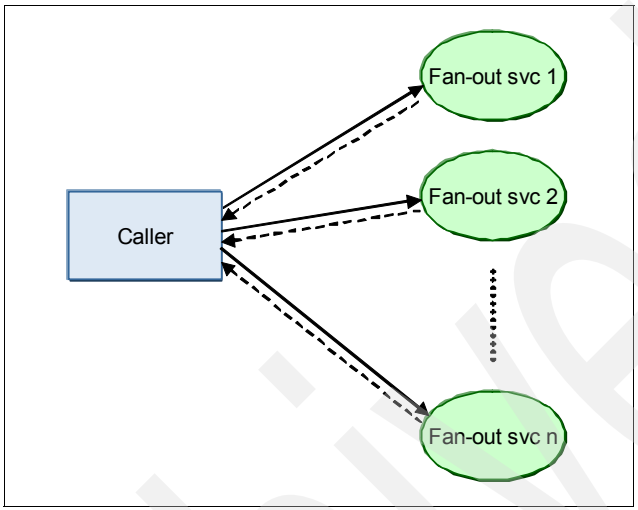


Figure 4-1 Fan-out pattern

Characteristics of the fan-out with responses pattern are as follows:

- The calls are typically asynchronous.
- There is no relationship between the services (in other words, the input to one does not rely on the output of another).
- Often the services share the same interface.

This pattern is used in the price comparison scenario (“Price comparison Web site” on page 36). For performance reasons, that scenario has an asynchronous requirement, which this pattern supports. The scenario dictated that there is no invocation order, which fits within the no-relationship characteristic. The scenario did not dictate whether all of the fan-out interfaces are the same, but whether or not, they are immaterial to the pattern.

4.1.1 Fan-out programming model

The programming model for the fan-out pattern is to invoke all the services, and then handle the responses in any order, typically with a time-out, so that responses returned after the time-out are ignored, or handled in some other fashion, such as being written to a log file (Figure 4-2).

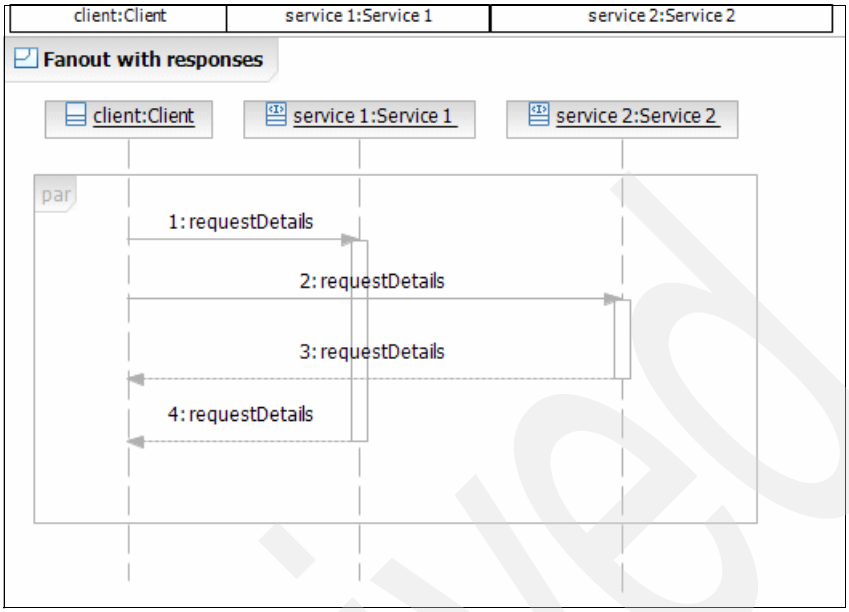


Figure 4-2 Fan-out with responses programming model

JAX-WS provides two-ways to program asynchronous clients: polling and callback (see 7.5, “The asynchronous client API” on page 196). There is not much to choose from, between the two methods, in this case; the callback method is perhaps marginally simpler to program in Java.

The client program has a time-out to return to *its* client, and within this time-out period, either polls to see if it can collect all the responses sooner, or processes each callback as it arrives. The client program keeps a table of all the potential responses, and if the table is filled before the overall time-out, it can respond to its client sooner.

4.2 Roaming client

The insurance underwriter scenario demonstrates a common pattern for using WS-ReliableMessaging, which we have called Roaming client. This is a situation where you have a mobile worker at the end of an unreliable network connection, for example, GPRS or public WiFi in an airport or coffee shop. The main characteristics of this situation are as follows:

- Service invocation is not idempotent — that is, invoking the service a second time on the same endpoint produces a different result.

- You want to minimize custom retry and duplication detection logic.
- Failures in the network are common, but failures in the endpoints are uncommon.
- Endpoint failures can be dealt with administratively, or out-of-band.
- A lightweight client is required.
- Multiple client vendors are possible.
- An untrusted network is used.

In this scenario you can use a WCF client or a JAX-WS thin client with WS-ReliableMessaging (see Figure 4-3 on page 59). These clients insulate your application code from intermittent networking problems and you do not have to write your own custom retry or duplicate detection.

Once the message is sent from the application code, the Feature pack for Web services stores the message in memory and using the WS-ReliableMessaging protocol, resends the message to the service until it receives an acknowledgement that the service received the message. The service, using information built in to the WS-ReliableMessaging protocol, examines incoming messages to determine if they are duplicates, and if a message has been previously processed, it is discarded. In this way, the WS-ReliableMessaging implementation ensures that for each message received from the client, the service is invoked exactly once.

To receive an acknowledgement that the service has been executed and to know what the outcome is, the client uses a request-response operation. The response is treated just as reliably as the request. In normal operation, the responses are “piggy-backed” as replies in the same TCP/IP transport session used for the request.

But what if the session is lost? How is a new session established to get the response back? If the session is reestablished by the server, there can be firewall issues; the firewalls at the client and server might be configured only to accept outgoing session creates from the client and incoming session creates at the server. WS-ReliableMessaging overcomes this with a protocol that enables the client to reestablish the session expressly to receive an outstanding reply (see 10.2.3, “WS-ReliableMessaging 1.0 Sync two-way message flow” on page 365).

Lightweight clients store the WS-ReliableMessaging state in memory and hence do not protect against endpoint failure. Endpoint failure is a relatively uncommon case while a worker is interacting with the service. If a failure does occur, the low volume of transactions makes it likely to be more economic to deal with it via a telephone support service, based on data logged at the client prior to the transaction.

The untrusted network can be dealt with by the use of WS-SecureConversation associated with the WS-ReliableMessaging sequence. This can be used to provide one or more of confidentiality, integrity, and authentication.

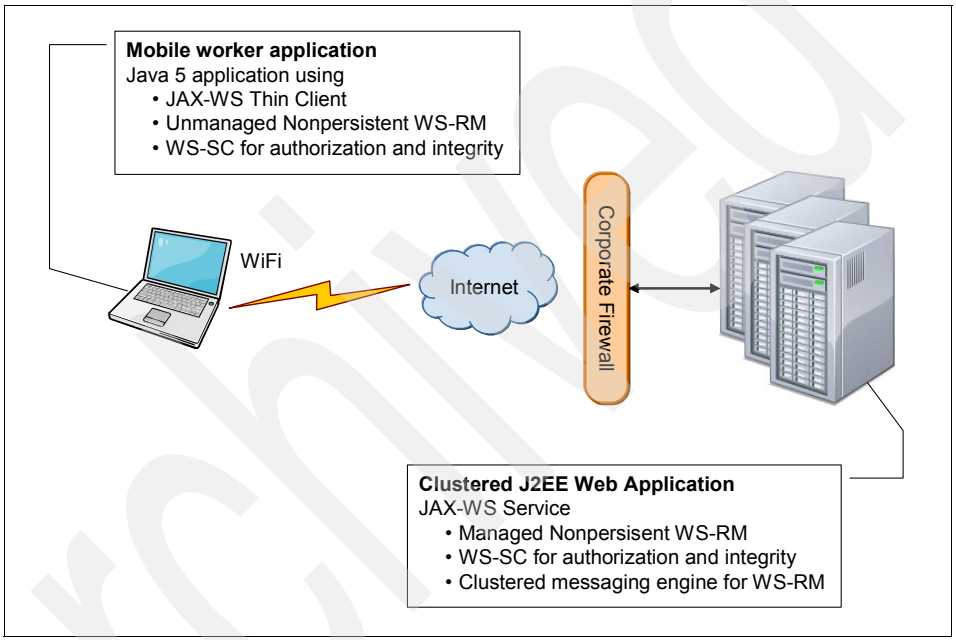


Figure 4-3 Roaming client pattern

4.3 Externalization of application QoS

The outsourcing scenario demonstrates a pattern for using Policy sets. A Policy set is a grouping of policy or configuration data supporting qualities of service associated with Web Services, such as security or reliable messaging. Policy sets decouple qualities of service from applications. Typically different Policy sets are defined by experts in Web services quality of service, and then the appropriate Policy set is associated with a Web service using the Administrator Console. When an application or service is defined, the required quality of service is matched against the reusable Policy sets that are defined, or a new Policy set is developed by an expert (Figure 4-4).

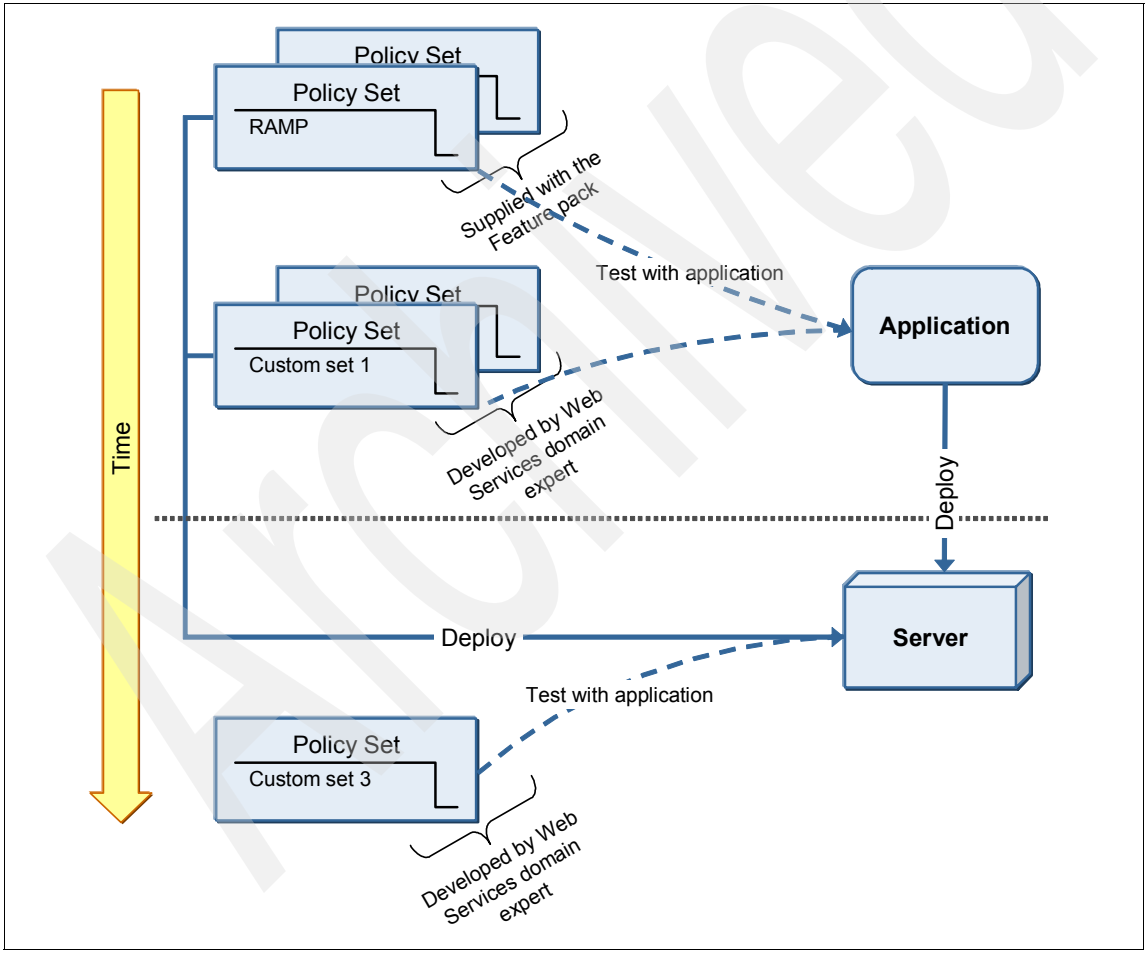


Figure 4-4 Developing and using Policy sets

The developer then only has to be concerned with implementing the application to work with the chosen Policy sets, and not with how implement the qualities of service themselves

The use of Policy sets reduces development and maintenance cost by promoting reuse and supporting a separation of concerns between functional application code and the implementation of different qualities of service. For example, after your organization puts the application developed by a business partner into production, you might want to change the Web services security policies, or add reliable messaging policies.

Using Policy sets, you can re-configure your qualities of service after the development phase of a project has finished. Changing the quality of service does not require any re-development, repackaging, and redeployment of the application. If you develop new Policy sets after development of the application, you will have to review whether the application design supports the new qualities of service and test to determine if the application works correctly with the new Policy set.

4.4 Robust fire-and-forget

The check clearing scenario demonstrates another common WS-ReliableMessaging pattern, that of using it to provide reliable asynchronous one-way messaging. This is common for batch processing where you want to do some work and then pass the results on to the next stage in a process on a different machine without having to worry that the results might get lost in transition from one process to the next.

The main motivating characteristic for using this pattern with WS-ReliableMessaging is vendor interoperability. A similar model is common using messaging oriented middleware, however, products like WebSphere MQ require code from a single vendor to be running at both ends of the communication link to achieve maximum reliability.

This pattern can be implemented with differing levels of reliability, availability, and transactionality, depending on the requirements laid down by your specific business scenario. Qualities of service depend on vendor implementations — all the WS-ReliableMessaging standard requires is robustness of the message transport — the reliability and availability of the endpoints is left up to the vendor to implement.

A transactional, highly available WS-ReliableMessaging configuration provides the highest level of robustness. In this scenario, the messages are sent as part of a transaction, as illustrated in Figure 4-5. When the transaction commits in transaction scope **1**, the message is written to the persistent WS-ReliableMessaging store, from where it is later extracted and sent in transaction scope **2**.

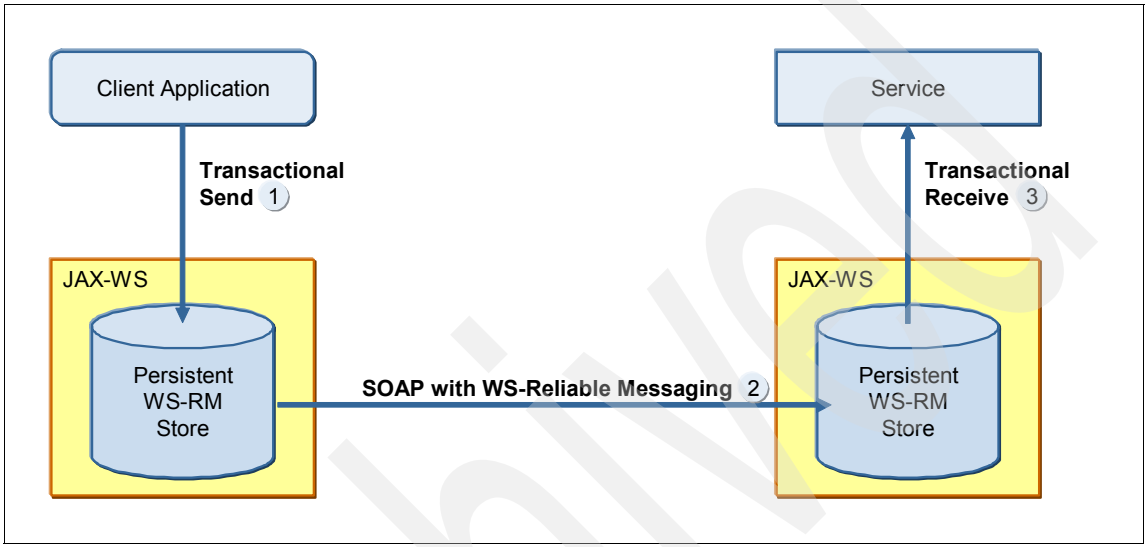


Figure 4-5 Three transactional model for sending data reliably

When the message is received, as part of transaction scope **2**, it is placed in the persistent store, and then, under transaction scope **3**, the message is extracted and endpoint invoked. If transaction **3** gets rolled back the message is returned to the persistent store and the endpoint is invoked again under a new transaction **3**. This is the “Three Transaction” pattern familiar to users of WebSphere MQ, which decouples client and server application processing from the availability of the software at the other end of the message exchange.

For the Feature pack for Web services, high availability is supported through the use of WebSphere Application Server clustering. This allows the persistent store to fail-over among the cluster members, and for any cluster member to process a message belonging to a given WS-ReliableMessaging sequence.

Given the asynchronous and persistent nature of the processing, it might be that high availability is not actually a requirement, because temporary failure is tolerated by storing messages in the persistent store.

The persistent quality of service allows the system to tolerate endpoint failure without losing messages. This is the most likely usage of this pattern.

4.5 In-band transfer of binary data

Transferring binary data is a facet of communication. If you have to send binary data, there are a number of technical considerations:

- Should the binary data be sent in-band or out-of-band?

  In other words, do you have to send the data or merely a reference to the data? The size of attachment is the main consideration — there are other reasons for deciding on in-band or out-of-band — but a decision to use out-of-band transmission is seldom made for only small quantities of data.

- If in-band, should the data be sent in the SOAP envelope or as an attachment?

  Interoperability has, in the past, encouraged use of data embedded in the SOAP envelope because of the lack of general consensus on how to handle attachments. Choosing to embed binary data imposes an unavoidable performance burden, so much that it is worth solving the interoperability problems of sending attachments on a case by case basis.

- If attached, which specification should you use, MTOM (interoperable with Microsoft .Net) or Soap with Attachments (Sw/A) (interoperable with some older Web services applications)?

This section expands on these considerations and provides advice about how to address them.

4.5.1 Out-of-band transmission

A SOAP message is part of a shared communication channel. You must decide whether it is sensible to load the communication channel with binary data or whether it is sufficient for the receiver merely to know that the binary data exists. Your decision depends on the size of the binary data, the bandwidth of the channel, the responsiveness requirements placed by all users on the channel, the need to attach the binary data to the SOAP message itself, and whether alternative channels more suitable for the transmission of large amounts of data exist.

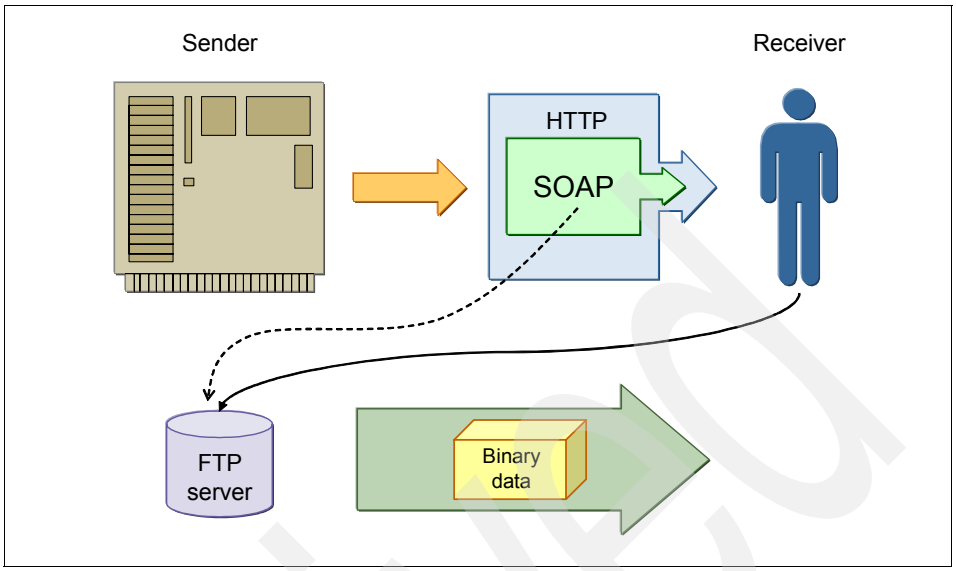


Figure 4-6 Send binary data out-of-band

For instance, a hospital might have a PatientQuery Web service. X-rays are part of patient data, but an X-ray image is typically quite large. The X-ray itself is probably not necessary in the query. The service returns a reference to it — a URL or an FTP address, for instance — which the caller of the service goes to if and when they actually have to view the X-ray.

4.5.2 Send data within the SOAP envelope

You could simply send binary data as xsd:base64Binary or xsd:hexBinary data within the SOAP message body as shown in Figure 4-7.

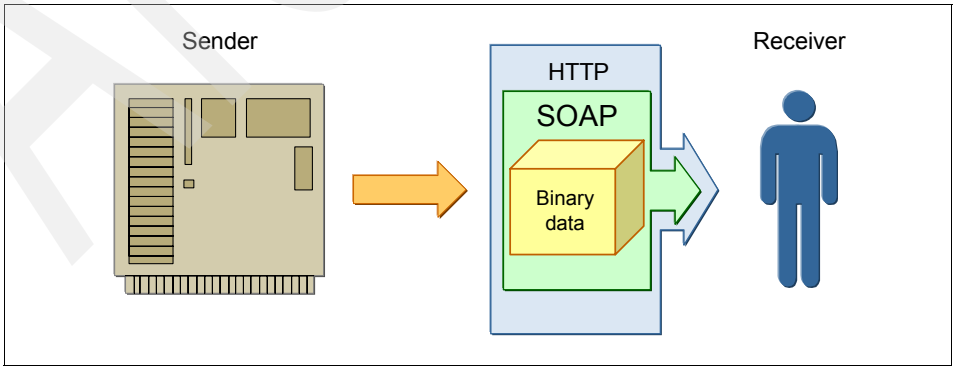


Figure 4-7 Send binary data within the SOAP envelope

These are some advantages to this approach:

- The data is part of an XML instance, parseable by any XML parser, and therefore completely interoperable.
- As part of the SOAP message, it is secured when the message as a whole is secured.

But there are disadvantages to this approach:

- There is a performance hit because the sender must encode the data and the receiver must decode it.
- The data has a bigger footprint because the process of encoding it to a string format — which is what xsd:base64Binary and xsd:hexBinary are — increases the size by approximately a third. The bigger the binary data, the bigger the impact.
- When mapped to Java, these binary XML formats become byte arrays, so the application program will have to know how to manipulate the data as byte arrays.

4.5.3 Send data as attachments

If you decide that neither of the first two options is appropriate, you can send the binary data as an attachment as shown in Figure 4-8.

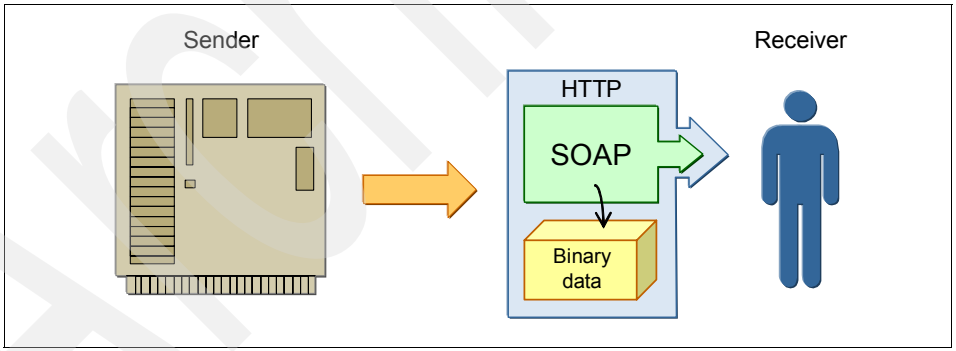


Figure 4-8 Send binary data outside the SOAP envelope as an attachment

As an attachment, the data is not encoded, it is sent in its native binary form, so performance is improved and footprint costs are reduced. And in either JAX-RPC or JAX-WS/JAXB, a few binary data types are mapped to more easy-to-use Java types. For instance, image/gif is mapped to java.awt.Image.

There are a number of ways to send an attachment. Which one should you choose? The two most popular are SOAP with Attachments (Sw/A) and Message Transmission Optimization Mechanism (MTOM). Sw/A has existed for a few years. It is mostly interoperable with one large exception: Microsoft does not support Sw/A. If you must interoperate with Microsoft products, Sw/A is not an option.

The new MTOM attachment specification is being adopted by everyone. And the IBM security story around MTOM is better than it is around Sw/A. But MTOM is still new, which means that you will find pockets of products that do not yet support it.

4.5.4 In-band transfer of binary data facet in the business scenarios

This facet is visible in the check clearing scenario. The scanned image of the check is binary data. The data is not unreasonably large, so sending it in the communication channel is acceptable. The data itself is an inherent part of the check clearing process, and needs to be sent as reliably as the other contents of the message. Performance is a consideration, so sending it inline is less than optimal for the scenario. The message must be as widely interoperable as possible, and security is important. Given these requirements, the scenario chose MTOM as the solution for this facet.

5

Technical advantages

In this chapter we discuss benefits of the Feature pack for Web services from a technical perspective. The first part of the chapter describes benefits with reference to the specific technologies introduced by the Feature pack for Web services. The second part of the chapter highlights some aspects of capability and interoperability to be aware of in order to use the new capabilities in the Feature pack for Web services wisely.

5.1 Benefits

The technical benefits of the feature pack discussed in this section are:

- Interoperability with Windows Communication Foundation
- Reliability
- Asynchronous programming model
- Portability
- Ease of use
- Performance

5.1.1 Interoperability with Windows Communication Foundation

Prior to the Feature pack for Web services, there were two areas where interoperability between WebSphere Application Server v6.1 and Windows Communication Foundation presented issues:

- SOAP version
- Binary attachments

Both of these areas have been addressed with new capabilities introduced in Feature pack for Web services.

SOAP 1.2

The default for Microsoft Web Service Extensions 3.0 (WSE) and Windows Communication Foundation (WCF) is to generate services that use the SOAP 1.2 specification level for SOAP messages. WebSphere Application Server v6.1 supports SOAP 1.1 only. Because SOAP 1.2 messages are not compatible with SOAP 1.1 services, there is an interoperability issue with Microsoft clients generated using the default settings.

The Feature pack for Web services introduces support for SOAP 1.2 in addition to retaining SOAP 1.1. The SOAP specification level support is configured on a service-by-service basis and is declared as a different SOAP namespace in the service WSDL file (see 7.12, “SOAP 1.2” on page 243).

MTOM

The mechanism for attaching binary data to a SOAP message used in WebSphere Application Server prior to the Feature pack for Web services is SOAP with Attachments. This standard is not supported by Windows Communication Foundation.

The Feature pack for Web services adds support for the MTOM standard for binary data attachments, which is the standard adopted in Windows Communication Foundation (see 7.8, “MTOM” on page 213).

5.1.2 Reliability

Web service invocations using SOAP over HTTP suffer some inherent limitations in reliability and uncertainty of message delivery in failure conditions, due to the nature of the HTTP transport. Reliability concerns related to HTTP can be addressed by using more reliable transports, such as SOAP over JMS.

Within a business, the use of SOAP over JMS might be a good choice. However, SOAP over HTTP is the only SOAP binding that is fully standardized and provides the best interoperability between vendors.

Due to its simplicity and prevalence, HTTP is an easy protocol to work with across public networks and wherever firewalls might be involved. WS-ReliableMessaging is introduced in the Feature pack for Web services to provide an HTTP-based reliable transport. Reliable messaging should not be viewed as a direct Web services equivalent to traditional reliable message and queuing infrastructures such as WebSphere MQ, but it does answer reliability concerns of Web service invocations using SOAP over HTTP.

Simplification of application logic

Business applications often impose requirements such as the need for service invocations to be re-tried on a failure, and for once-only message delivery. Dealing with reliability requirements in application code increases development costs and increases maintenance costs due to the complex nature of reliability code. Application-level reliability mechanisms are also, by their nature non-standard and increase coupling between clients and services, because both client and service implementations must share the same protocol.

The use of WS-ReliableMessaging provides reliability in a standard manner within the Web service infrastructure. This simplifies application logic, thus lowering costs, and improves interoperability.

Asynchronous transactionality

It is possible to pass a transaction context with a Web service request using WS-AtomicTransaction. In this case, work done at both the client and service end of a Web service invocation are carried out within the same transaction as a single unit of work.

By its nature, managing transactional work using WS-AtomicTransaction is *synchronous*. Both the client and service are enrolled in the same transaction. This suits some scenarios, but for many other scenarios, a distributed transaction model introduces many forms of coupling that are undesirable. Particularly in the Internet world, with its low manageability, high error rate, and with many transactions taking place between different businesses or individuals, the distributed transaction model is too complex to deploy.

As an alternative to synchronous transactionality, reliable messaging allows you to build asynchronous transactionality. The design of asynchronous transactions is very different from synchronous ones, but they have been proven effective in the messaging and queueing world for certain kinds of problems.

Reliable loosely coupled transactional applications are built by building processes with interlocking transaction scopes, using reliable messaging to pass the results of one process on to the next. Conflicts between multiple processes have to be resolved manually — typically by redoing the later arriving transactions again. One-way messaging best suits this loosely coupled environment. Two-way asynchronous messaging is an alternative if there are restrictions about which end of an interaction pair is able to initiate communication. WS-ReliableMessaging supports both one-way and two-way messaging models enabling the response message to be either pushed from the server or pulled from the client, depending on the existence of the underlying transport session and the firewall rules at both ends.

WS-ReliableMessaging allows reliable, once-only transmission of one-way messages and it also allows the message send and receive, at the client and service ends of the invocation respectively, to be carried out within the scope of a their own transactions. This allows a client to send a message as part of a one transactional unit of work, and for a service to receive a message as part of another transactional unit of work.

Security

WS-ReliableMessaging flows consist of multiple messages flowing between a client and a service. In order to secure messages within these message flows from being viewed by other authorized clients, it is necessary to associate messages with a security context.

WS-SecureConversation provides a standard, interoperable mechanism for associating all the messages in a message flow with a single security context. The composition of WS-SecureConversation and WS-ReliableMessaging provides a necessary level security to the overall message flow when dealing with sensitive data.

5.1.3 Asynchronous programming model

Traditional synchronous service invocation requires the client to block while it awaits a response to a service invocation. Asynchronous invocation of a request/response service is desirable because it allows the client to continue working while it awaits the response. This is particularly true in cases where the response might take a significant amount of time, perhaps because of volumes of data transferred, network speed, or the amount of processing required by the service before a response can be sent.

The JAX-RPC programming model used in WebSphere Application Server prior to the Feature pack for Web services does not provide any mechanism for a client to invoke a service and then asynchronously receive the response to that service invocation.

The JAX-WS programming model introduced by the Feature pack for Web services provides an explicit API for asynchronous service invocations. Two styles of asynchronous service invocation are supported by the Feature pack for Web services: callback and polling.

The callback style is easy to implement and is most useful for a single service invocation. When there are multiple responses, the services are invoked in parallel, and then the responses are collected at a time to suit the application by polling, or as soon as each one arrives by using the callback style. The callback style is typically easier to program — the polling style is most suitable when the application has a definite point in its logic when it is prepared to handle each response.

5.1.4 Portability

The Feature pack for Web services delivers improvements in Web service application portability between vendors. These improvements are realized by facets of the JAX-WS programming model.

Vendor-neutral Web service implementation

The JAX-RPC model and tooling produces numerous artifacts when developing a Web service client and service implementation. These include portable interfaces and stub implementation classes. The stub class implementations are proprietary to the vendor platform for which they are generated.

In order to deploy a JAX-RPC service on a different vendor platform, it is necessary to re-generate the stub implementation classes using tooling, prior to deployment of the application.

The JAX-WS programming model utilizes annotations in the service code to define behavior. The tooling generates only interfaces. Stub implementation classes are not generated. Instead, the runtime environment uses information contained in the annotations to implement the service[1].

The removal of tooling-generated stub implementation classes from packaged JAX-WS applications improves the ease of portability between vendor runtimes of Web services.

Improved type mappings

In some areas, the JAX-RPC specification defined mappings between XML and Java types are incomplete or ambiguous. In these cases, portability and interoperability issues can arise between vendor implementations of the specification.

The JAX-WS specification does not define mappings between XML and Java types. Instead, it makes use of the Java Architecture for XML Binding (JAXB) 2.0 specification. This specification provides a more complete and robust mapping between Java and XML than the JAX-RPC specification.

The use of JAXB for type mappings in the Feature pack for Web services provides improved portability and Interoperability for Web service implementations.

5.1.5 Ease of use

The Feature pack for Web services provides several ease-of-use improvements. The two most significant of these improvements are:

- The use of Policy sets for configuration and management of quality of service
- The simplifications to handler and service development provided by the JAX-WS programming model

These improvements deliver benefits in cost and time savings throughout the development and production cycle.

[1] A developer could “walk-through” the code generated for JAX-RPC for WebSphere Application Server v6.1, and understand the run-time path from the point that the service request was passed to the .EAR. This is neither necessary, or possible, with the JAX-WS implementation for the Feature pack for Web services; the behavior is built into the runtime Web services engine.

Management

Prior to the Feature pack for Web services, many quality of service features were configurable only through development tooling or programmatically within the application code. Changes to the configuration necessitated repackaging and redeploying the application, even re-developing it. There were no facilities for applying a standard configuration to a set of services.

The Policy set features of the Feature pack for Web services provide a mechanism for defining reusable policies for numerous quality of service features, including:

- WS-Addressing
- WS-Security
- WS-Transactions
- WS-ReliableMessaging
- WS-SecureConversation

Development

The Feature pack for Web services provides ease of use and productivity benefits in two areas of Web service application development:

1. Handler development.
2. Developing service implementation code.

JAX-WS handlers

The JAX-WS programming model defines a new style of handler for applying business logic to messages sent or received by Web service clients and services. The new handler model is similar to the JAX-RPC handler model, but it provides significant ease of use enhancements for message access and manipulation.

The JAX-RPC handler model provides access to message elements only as Document Object Model (DOM) object representations of the raw XML data. This requires knowledge of XML and the DOM APIs for working directly with XML data.

The JAX-WS handler model makes use of JAXB to map in a standard way between raw XML data and Java objects. Even in the handler, SOAP message data is accessed as Java objects rather than as raw XML data. Java object representations are a much more natural way for Java developers to work with data, and this matches the way that data is represented within service implementation code.

Annotations

The JAX-RPC programming model did not define a means of identifying a piece of code as a service or for specifying runtime behavior of that service. Instead, the JSR109 deployment model specified mechanisms for describing Java code as services using deployment descriptors.

Deployment descriptors are complex and typically require tooling to generate. This requires skills outside the core Java development skill set.

The JAX-WS programming defines a standard use of Java annotations within the code itself to identify methods as services, and to define the behavior of services. The use of annotations provides two distinct advantages:

- Ease of use for Java developers, because service configuration can be achieved purely in Java code use existing development skills
- Portability of service implementation, because JAX-WS defines a standard set of annotations that are portable across different vendor implementations

5.1.6 Performance

The Feature pack for Web services improves on the performance of the earlier Web service runtime in a variety of areas.

Secure Conversation

In part due to the use of asymmetric encryption keys on every secure service invocation, WS-Security imposes a significant performance overhead to the normal transmission of SOAP messages.

With WS-SecureConversation, a shared, symmetric private key is negotiated for the lifetime of a conversation flow of messages, not just for a single message. In flows where volumes of messages flow between two endpoints, this provides a performance benefit over the use of WS-Security alone.

Scenarios where multiple secure messages can flow between endpoints occur when a client makes regular calls to the same service implementation. In these cases, the use of WS-SecureConversation provides performance improvements compared to plain WS-Security.

StAX parser

The JAX-WS Web service runtime utilized by the Feature pack for Web services utilizes Streaming API for XML (StAX) parser technology. This provides performance benefits, particularly in areas such as memory usage, over other parser technology, such as DOM.

Memory footprint

The implementation of the JAX-WS Web service runtime by the Feature pack for Web services has been optimized to provide more efficient memory utilization than the JAX-RPC runtime in WebSphere Application Server v6.1. This results in a smaller memory footprint for equivalent Web service applications.

Message processing

The JAX-WS Web service runtime delivers higher service invocation and message processing performance than the JAX-RPC runtime.

5.2 Capability and interoperability

There are some issues related to capability and interoperability that are important to consider when developing new services and Web service applications using the Feature pack for Web services.

5.2.1 SOAP over JMS

SOAP over JMS is not supported by the new features delivered in the Feature pack for Web services. The JAX-RPC runtime in WebSphere Application Server v6.1 continues to be fully supported alongside the JAX-WS runtime in the Feature pack for Web services, and SOAP over JMS is fully supported in the JAX-RPC runtime.

It is not currently possible with the Feature pack for Web services to use new capabilities, such as WS-ReliableMessaging, WS-SecureConversation, MTOM, SOAP 1.2, or any other features of the JAX-WS programming model with the SOAP over JMS transport.

5.2.2 RPC-encoded

The JAX-WS runtime does not support the use of RPC-encoded style SOAP messages. This binding and encoding style is prohibited by the WS-I.org Basic Profile 1.0, so it is not recommended for use in new Web service development. However, this style remains supported in the JAX-RPC runtime for backwards compatibility with legacy services.

5.2.3 Import/export of policy bindings

The Feature pack for Web services does not provide capability to import or export Policy set bindings for services. This capability is planned to be introduced in WebSphere Application Server v7.0 to allow services to be distributed along with policy information defining quality of service attributes of the service.

5.2.4 Interoperability

New features introduced by the Feature pack for Web services are applicable only to services making use of the new JAX-WS programming model and runtime. New features specific to the Feature pack for Web services are not interoperable with services developed using the JAX-RPC programming model and runtime. These features are:

- WS-ReliableMessaging
- WS-SecureConversation
- MTOM
- SOAP 1.2

It is important to note that JAX-WS clients are interoperable with JAX-RPC services, and JAX-RPC clients are interoperable with JAX-WS services. You can be sure that services that comply with the WS-I profiles supported by both JAX-RPC and JAX-WS implementations are fully interoperable. Clearly JAX-WS services that support newly supported specifications are not going to be compatible with JAX-RPC clients, and likewise there are capabilities, such as RPC-encoded Web services that are supported by JAX-RPC, which are not compatible with JAX-WS clients.

Part 3

Using the feature pack

Part 3 describes how to use the feature pack:

- Chapter 6, “Installation” on page 79 explains the installation and configuration of an application server cluster on distributed and z/OS versions of WebSphere Application Server.
- Chapter 7, “JAX-WS programming model” on page 175 explains the JAX-WS programming model with lots of programming snippets to demonstrate how to use it. In our view, JAX-WS is a great improvement in capability, usability and simplicity over JAX-RPC, and it is well worth spending the time studying this chapter to understand its benefits and how to exploit them.
- Chapter 8, “Policy sets” on page 247 describes the new concept of Policy sets which aim both to simplify Web service configuration, and also to separate, as far as it is possible, programming the functionality of a Web service from defining the security and reliability of a service — that is, its Quality of Service (QoS). There are examples in the chapter for you to study how to use predefined qualities of service, how to extend them, and how to program them, should the need arise.

- Chapter 10, “Reliable messaging” on page 355 explains reliable messaging with examples showing how to configure it.

  Reliable messaging is enabled with Policy sets, and is deceptively easy to implement, but as with any capability that aims to provide reliability and overcome network and system failures, there are deep architectural, design, management and administration considerations to think about before successfully deploying WS-Reliable Messaging into production. It is important to study this chapter carefully and understand how to make a success of using WS-Reliable Messaging, and what pitfalls to avoid.
- Secure conversation is the most difficult technology in the Feature pack for Web services to understand and get working and we have devoted the whole of Chapter 9, “Secure conversation” on page 309 to explain it and provide examples for you to work through.
- Finally, Chapter 11, “Interoperability” on page 403 brings together the constituent standards in the forthcoming Reliable and Secure Web services profile as a tutorial for building a fully interoperable solution connecting WebSphere Application Server (which you can deploy on either or both distributed and z/OS platforms) and Windows Communication Foundation.

6

Installation

In this chapter we show how to install and configure the Feature pack for Web services, and step through a detailed example, including verifying the installation and configuration by running the sample programs shipped with the feature pack.

- In 6.1, “Configurations used in this book” on page 80, we describe the installation configuration we use in this book.
- In 6.2, “Installation roadmap for Windows” on page 81, we give you an overview of the installation and configuration steps for Windows® and for the distributed platforms.
- In 6.3, “Windows installation and configuration example” on page 85, we take you through the detailed steps we followed to install and configure the application server, the tools we used to program the Feature pack for Web services, and verifying the installation using the sample programs shipped with the feature pack.
- In 6.4, “Installation roadmap for z/OS” on page 144, we provide an overview of the installation and configuration steps for z/OS.

6.1 Configurations used in this book

We use two configurations to run the examples. The configuration we used for developing and testing the code snippets and standalone examples ran in a VMWare image running Microsoft Windows Server® 2003. We also ran the installation, configuration, and interoperability examples on a z/OS system. For the interoperability example (Chapter 11, “Interoperability” on page 403) we added Microsoft Windows Communication Foundation (WCF), and Visual Studio® 2008 to the Windows Server 2003, to demonstrate configuration and interoperability with WCF[1].

Figure 6-1 shows the installation used to run the feature pack on a standalone Windows 2003 server.

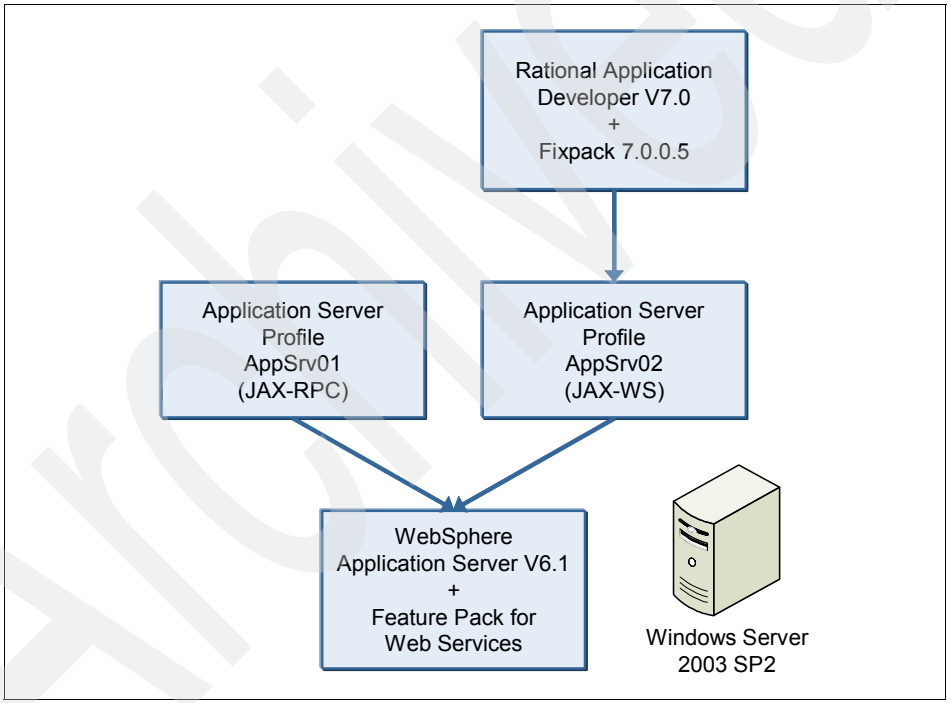


Figure 6-1 Installation and configuration on Windows 2003

[1] The Windows Communication Foundation sample does not require Visual Studio 2008 as long as you do not want to change the code. The WCF sample installation we provide automatically requests download of .Net 3.5. The Service Configuration Editor tool we use to configure .Net 3.5 Web services is available for free download as part of the Windows Vista® SDK.

The installation requires 6.85 GB of disk space, including a small amount of additional software. The running VMWare image size is 8.05 GB without Visual Studio 2008, and 15.1 GB with VS2008 and SQL Server® 2005 development edition[2]. We are using 1.2 GB of memory allocated to the VMWare image.

6.2 Installation roadmap for Windows

The installation and configuration of the Feature pack for Web services requires:

- Additional application server software:
  - Additional libraries and server functions
  - Extensions to the Administrator Console
- Fixes for the application server
- A new profile: the Feature pack for Web services profile
- New tooling to build applications:
  - Rational Application Developer 7.0.0.3
    or
  - Application Server Toolkit 1.1.0.3

The Feature pack for Web services installation wizard on Windows looks after getting an existing WebSphere Application Server to the right service level, without having to install additional Fix Packs using Update Installer.

Subsequent installation of fixes are managed by Update Installer on distributed platforms. We apply Fix Pack 13, to demonstrate how to apply maintenance to WebSphere Application Server v6.1 with Feature pack for Web services installed.

The four steps in the installation and configuration process for WebSphere Application Server are summarized in Figure 6-2, and described in the following sections:

- 6.3.1, “Install WebSphere Application Server”.
- 6.3.2, “Install Feature pack for Web services”.
- 6.3.3, “Create Feature pack for Web services server profile” on page 100.
- 6.3.5, “Update WebSphere Application Server v6.1 to 6.1.0.13” on page 107.

[2] We also had DB/2 9.1.4 in this image, although it is not used. Installing Visual Studio 2008 with SQL Server 2005 broke the DB/2 installation (it becomes unrunnable and difficult to uninstall cleanly). The problem probably lies in a conflict over the system JVM™. Try the fixes described in the note on page 435.

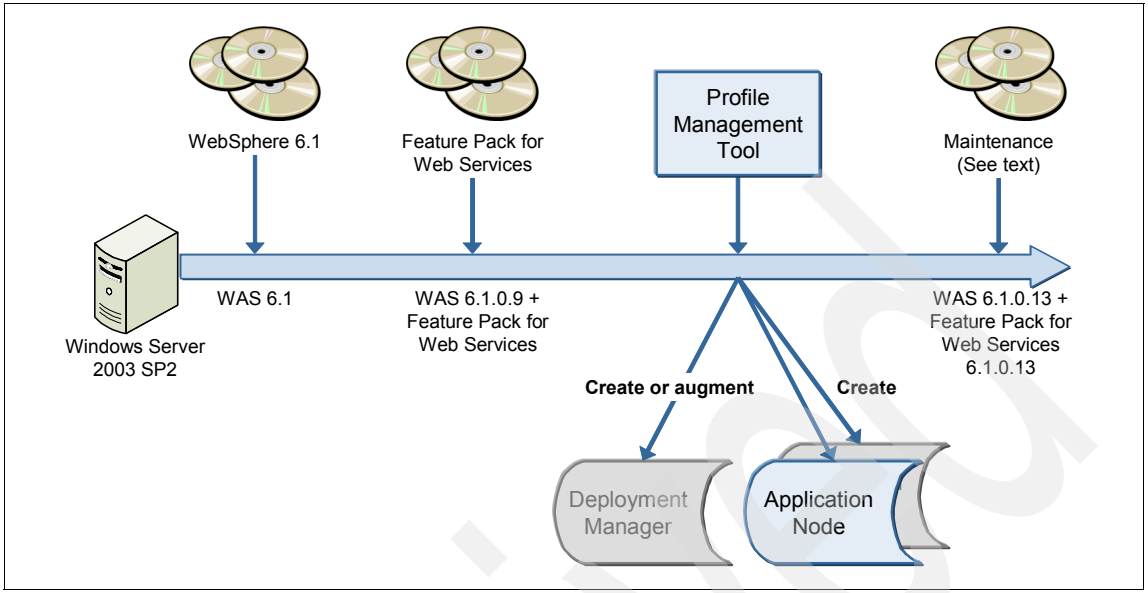


Figure 6-2 Feature pack for Web services installation and configuration steps

We also show an additional three steps to install the Rational Application Developer tool, to configure Rational Application Developer to use the installed WebSphere Application Server, and to verify the installation:

- "Install Rational Application Developer" on page 117.
- "Configure test server" on page 130.
- "Verify installation" on page 137.

6.2.1 Feature pack for Web services installation

The feature pack installs on top of an existing WebSphere Application Server v6.1 system, either Standalone or Network Deployment. The feature pack also installs into the base WebSphere Application Server, which underlies other WebSphere products such as WebSphere Process Server and WebSphere Enterprise Service Bus. This enables you to run J2EE applications as JAX-WS Web services on WebSphere Process Server and WebSphere Enterprise Service Bus. Installing the Feature pack for Web services does not (at the 6.1.0.0[3] level of WebSphere Process Server and WebSphere Enterprise Service Bus) enable mediation modules or business processes to run as JAX-WS Web services or to behave as JAX-WS clients.

The distributed installation package for the feature pack includes the Launchpad installation wizard, which automates the installation process. The feature pack is available from:

http://www-306.ibm.com/software/webservers/appserv/was/featurepacks/webservices/

6.2.2 Profiles

To run applications using the feature pack, you have to create a new profile for the application server. The profile is called the “Application Server with Feature pack for Web services”, frequently shortened to the “JAX-WS” profile. The Profile Management Tool (or the equivalent command scripts) is used to create new deployment manager and application server nodes and augment existing deployment manager profiles.

The Profile Management Tool (PMT) augments an existing deployment manager profile to include the Feature pack for Web services, but does not augment an existing application server profile to include the feature pack. To upgrade a server node to run JAX-WS you have to create a new profile, and then redeploy applications onto the new node.

In a clustered environment you can migrate a deployment manager to support the new feature pack (for example to have access to the new Web services capabilities in the Administrator Console), but to add the capability to run applications using the feature pack in the cluster, you have to create new nodes with the profile for the Feature pack for Web services, and withdraw the old nodes from the cluster. Figure 6-3 shows the process schematically. In the diagram we assume that the underlying WebSphere Application Server installation is already upgraded to support the feature pack.

- In the existing cell, the J2EE applications A and B (JAX-RPC if they are Web services) are deployed to the cell and replicated onto nodes 1 and 2.
- Step 1: The Deployment manager profile is augmented to support the Feature pack for Web services and new application nodes are created using the Feature pack for Web services profile. The new nodes 3 and 4 are federated into the cluster. The deployment manager automatically deploys the applications deployed to the cell level to the new nodes in the cluster.

[3] The 6.1.0.17 level of WebSphere Process Server and WebSphere Enterprise Service Bus require the 6.1.0.17 level of WebSphere Application Server, and require the 6.1.0.17 fix level of the Feature pack for Web services. We have found some incompatibilities remaining between the feature pack and WebSphere Process Server and WebSphere ESB at the 6.1.0.17 fix level when using the base WebSphere Application Server level of function. Rational Application Developer will not deploy an EJB project on to WebSphere Process Server when the project is configured with the feature pack facet.

- Step 2: The old nodes, 1 and 2, are withdrawn from the cluster
- Step 3: New J2EE applications are added to the cluster — in this case, a JAX-WS application.

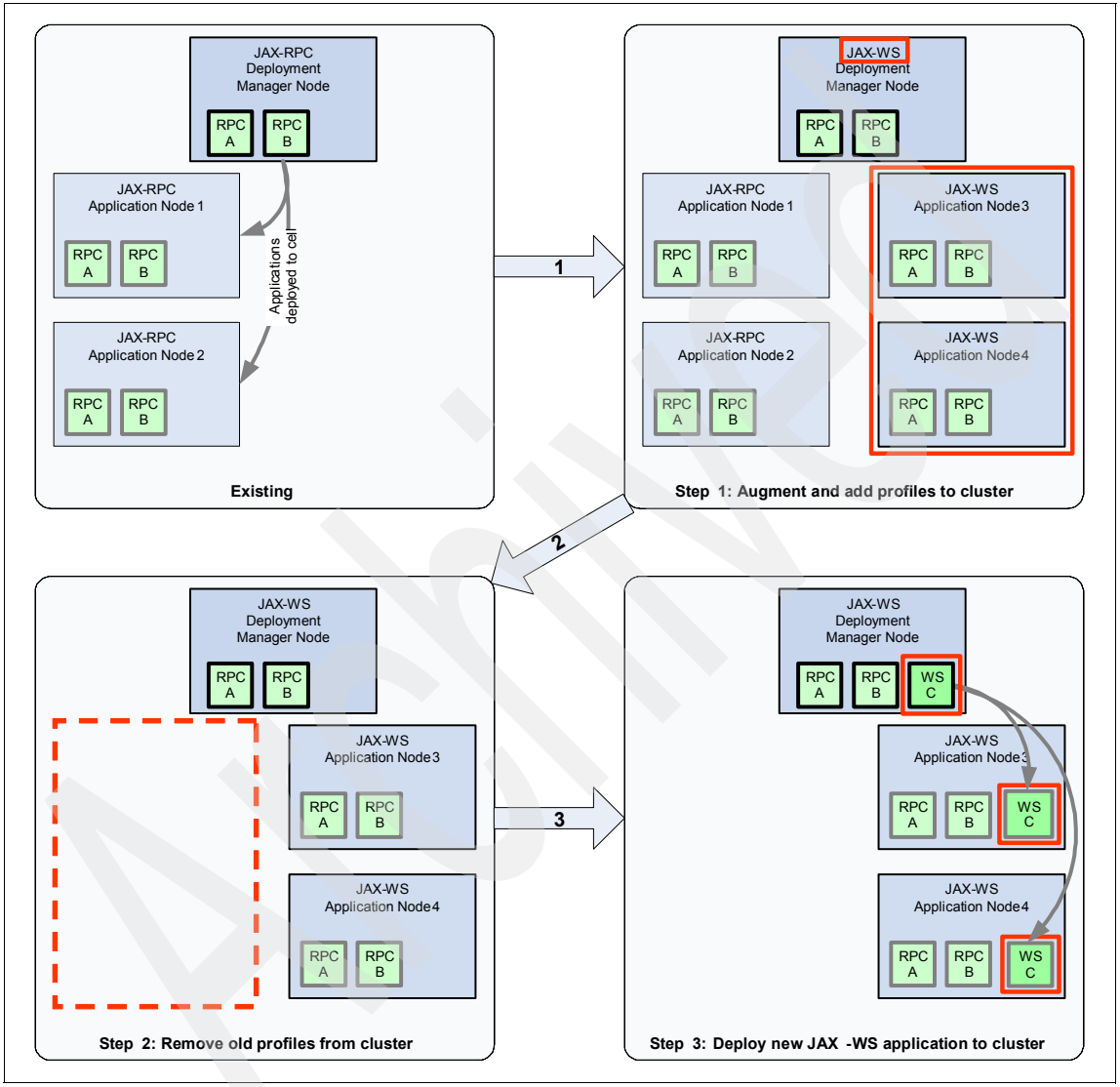


Figure 6-3 Upgrading an existing Network deployment cluster cell

6.2.3 Tools

You have a choice of two tools to work with the feature pack:

- Application Server Toolkit v6.1.1.3
- Rational Application Developer[4] V7.0.0.3

The feature pack tools functionality is the same in both products. The choice of toolkit depends on other factors beyond the scope of this book.

Fix Pack 3 for Application Server Toolkit (AST) 6.1.1 extends the toolkit to support the feature pack. You must have installed AST 6.1.1; then obtain Fix Pack 3 from:

http://www-1.ibm.com/support/docview.wss?uid=swg24016183

In this book we are using Rational Application Developer at v7.0.0.5 level. Instructions on how to install and configure Rational Application Developer are given in 6.3.6, “Install Rational Application Developer” on page 117.

A Technote has been issued about compilation errors wrongly reported by Application Server Toolkit and Rational Application Developer running on WebSphere Application Server v6.1.0.11:

http://www-1.ibm.com/support/docview.wss?uid=swg21268467

In 6.3.5, “Update WebSphere Application Server v6.1 to 6.1.0.13” on page 107 we show apply Fix Pack 6.10.13 to get around this problem.

6.3 Windows installation and configuration example

The installation and configuration instructions that follow install the WebSphere Application Server, Feature pack for Web services and Rational Application Developer configuration we used to build the examples in the book to run on Windows Server 2003. Because we are running with multiple WebSphere Application Server installations, one on Windows and one of z/OS, we chose to run with separately installed application servers, rather than running with the WebSphere Application Server installed with Rational Application Developer. There is no difference in the application servers, but there is a slight difference in the configuration of the server environment in Rational Application Developer.

[4] And all Rational Software Delivery Platform tools based on Rational Application Developer, such as Rational Software Architect.

Full installation instructions for other distributed platforms, and other configurations and for silent install are found in the following documents:

- Readme\Readme_base_en.txt file in the WebSphere Application Server installation folder
- WEBSV\docs\Getting_Started_en.html file in the Feature pack for Web services installation folder
- rad.QuickStartGuide\rad_qsg_en.pdf Quick Start guide in the Rational Application Developer installation folder

6.3.1 Install WebSphere Application Server

In this section we install the WebSphere Application Server, as follows:

1. Launch the installation of WebSphere Application Server.
   a. Unzip the installation package or insert the CD containing the WebSphere Application Server v6.1 installation package and double-click the launchpad.exe program (Figure 6-4).

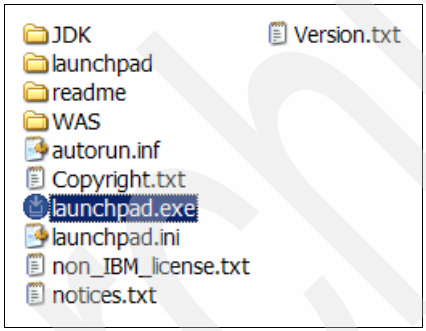


Figure 6-4 Start launchpad.exe

   b. Select the link to launch the installation wizard from the installation page (Figure 6-5).

Figure 6-5 Launching WebSphere Application Server installation

Note: If you get the error pop-up shown in Figure 6-5, then the probable cause is that you have navigated to the launchpad.exe program using “Network Places” and have not mapped the drive containing the installation folder to a drive letter.

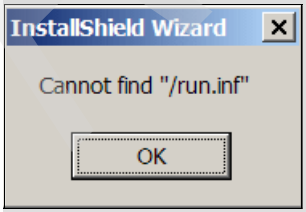


Figure 6-6 Incorrect drive mapping

2. Configure the installation options:

   a. Click **Next** on the first panel of the installation wizard (Figure 6-7) and accept the license terms and conditions → **Next**. The installer then checks installation prerequisites.

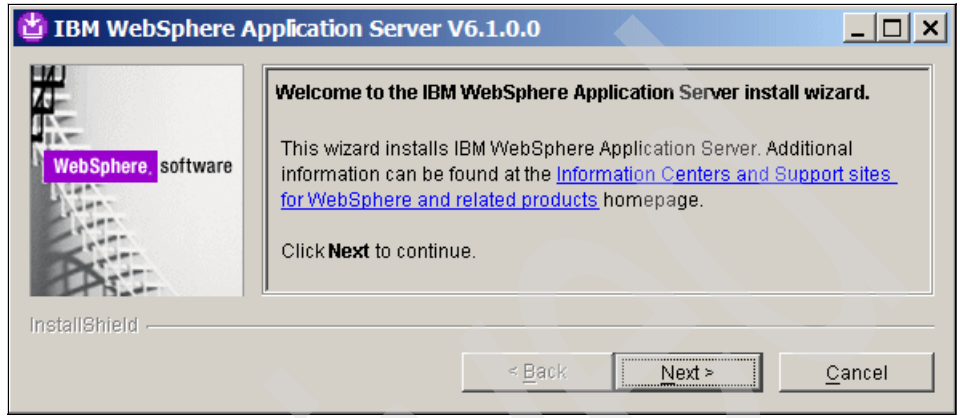


Figure 6-7 Installation wizard

   b. If you are running Windows Server 2003 SP2, then ignore the error shown in Figure 6-8 → **Next.**

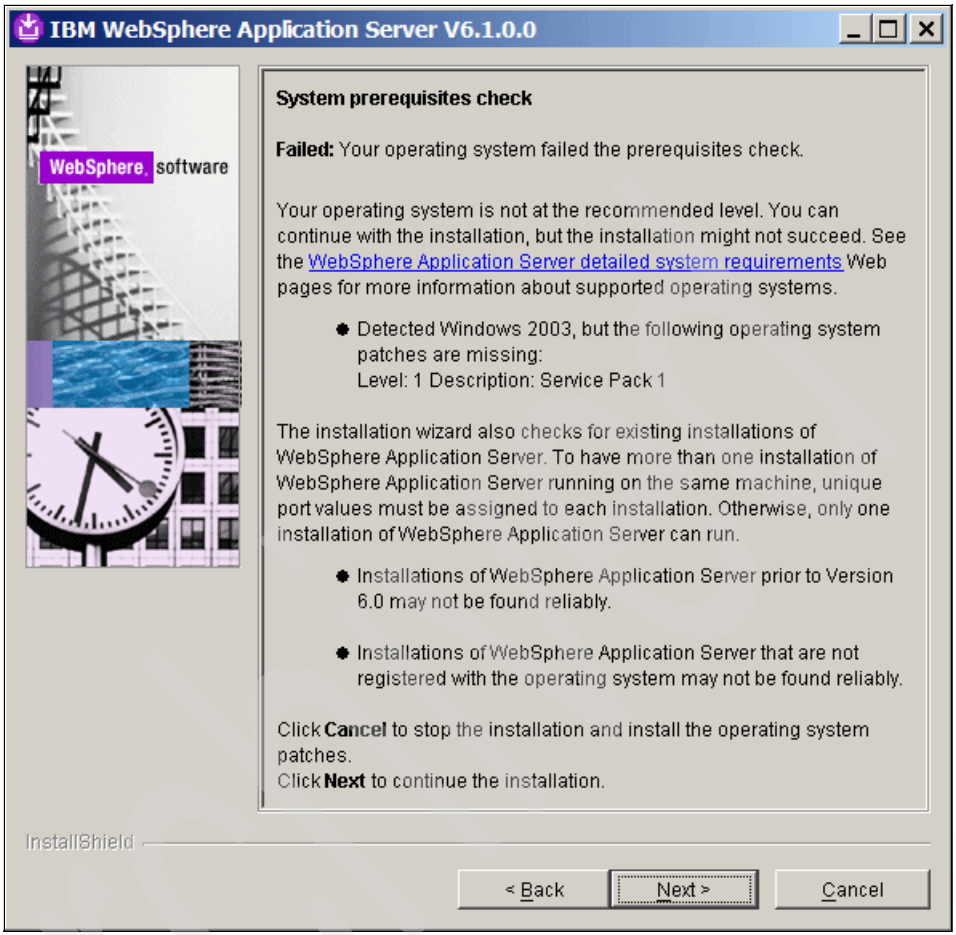


Figure 6-8 Windows Server 2003 SP2 prerequisites error

c. Choose whether to install the sample applications (we chose not) → **Next.**

d. In Figure 6-9, we chose to shorten the default program path by removing "Program Files". If you write command line scripts, it is easier to have program paths without embedded spaces. Long paths sometimes give rise to problems. Click **Next** to continue the installation.

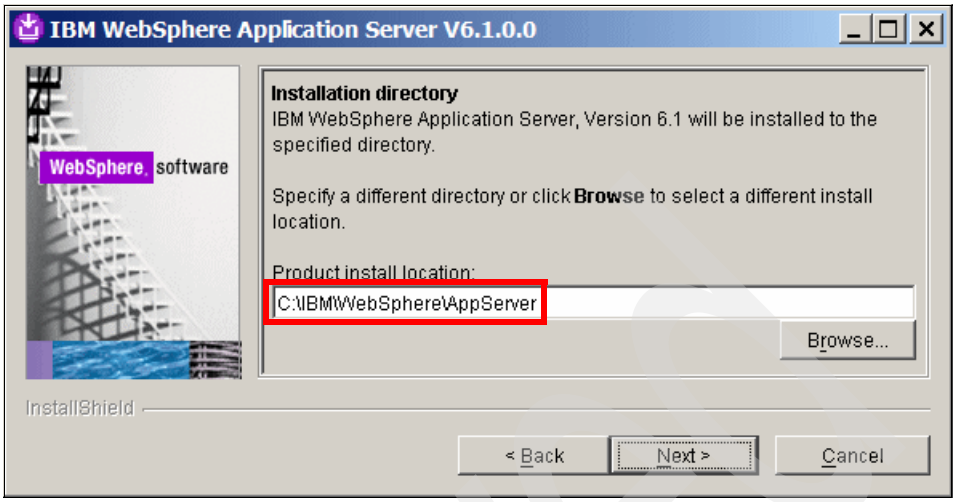


Figure 6-9 Shorten the program path

e. We have chosen to enable Administrative Security in Figure 6-10 → **Next.**

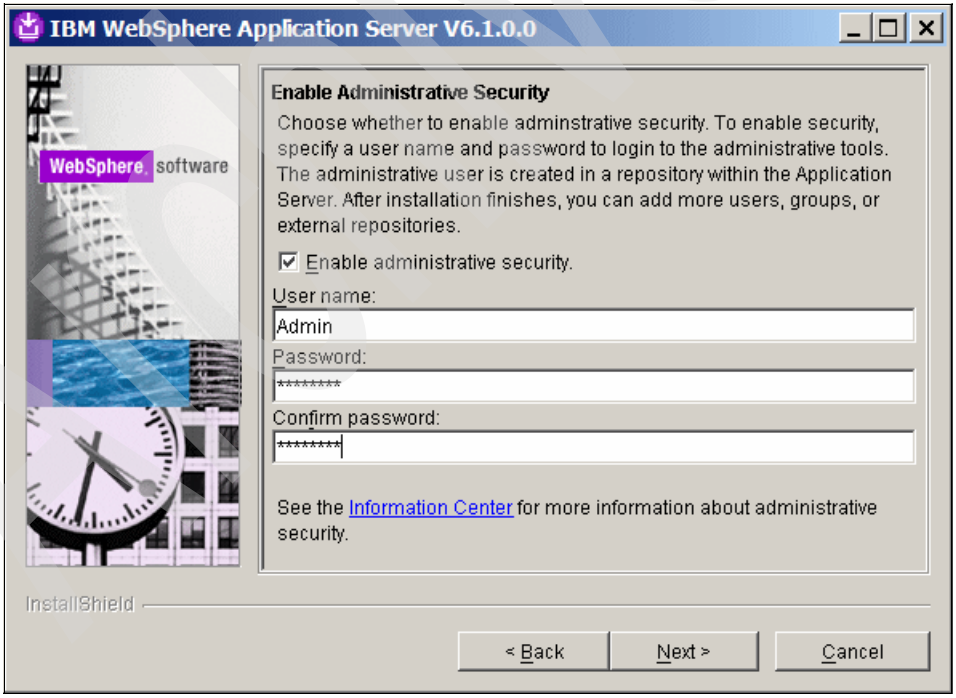


Figure 6-10 Enabling Administrative Security

3. The summary installation options are shown in Figure 6-11 → **Next**.

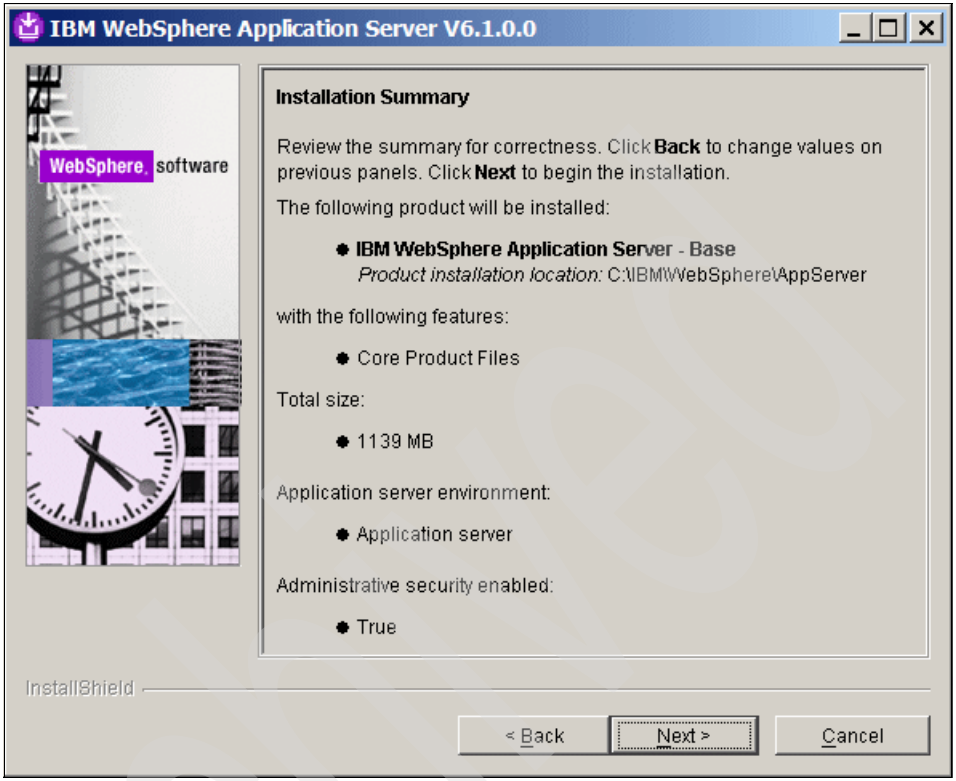


Figure 6-11 Summary installation options

The installation takes fifteen minutes or so. On completion you have the option to launch "First Steps" (Figure 6-12 on page 92) → **Finish**.

Figure 6-12 installation complete

4. If you choose to run the installation verification check at this point, remember to stop the server before installing the feature pack, which you can do by selecting the **Stop Server** option from the First Steps panel (Figure 6-13).

Note: By default, the server is installed as a Windows server with the *default* profile and *automatic start*. Because we want to run with the Feature pack for Web services profile, after installing the Feature pack for Web services, then open the Microsoft Management Profile and change the service startup option to `Manual`.

- Navigate to **My Computer** → **Manage** → **Services and Applications** → **Services** → **IBM WebSphere Application Server v6.1** - `servername` -
- Right-click → **Properties** → **Set Startup type to** `Manual` → **Apply** → **OK**

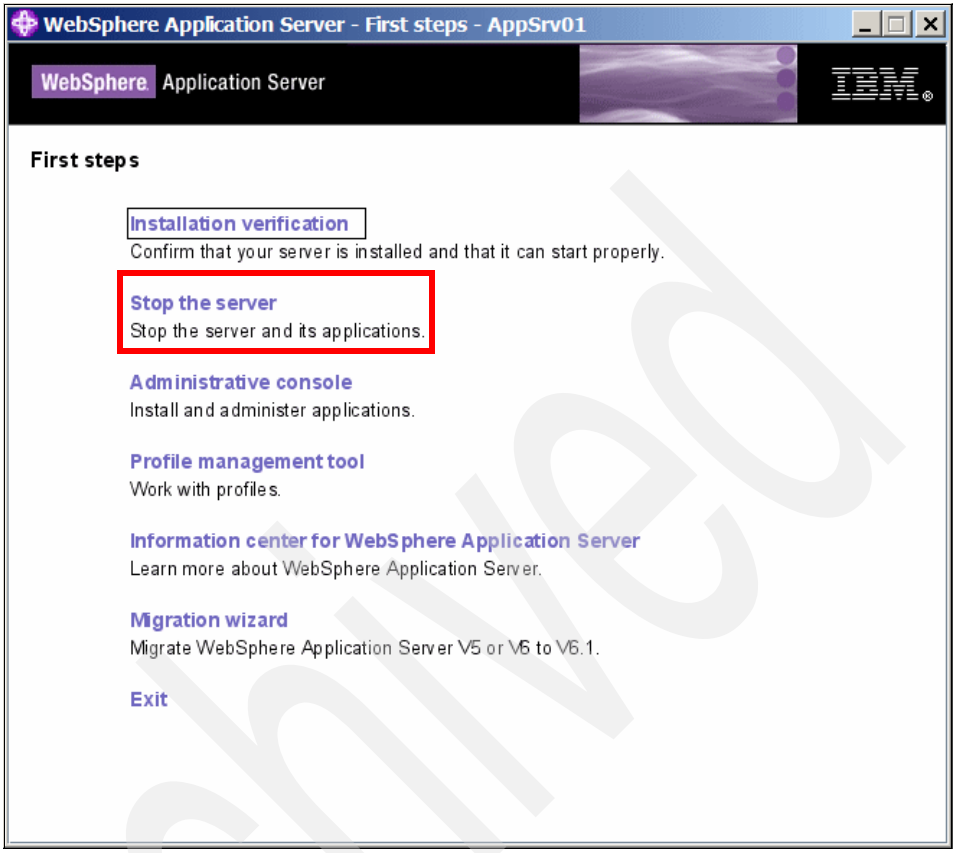


Figure 6-13 Stop the server after running installation verification

6.3.2 Install Feature pack for Web services

The Feature pack for Web services is packaged as a separate installable for each platform. You can find links to all the feature packs at:

http://www-1.ibm.com/support/docview.wss?rs=180&context=SSEQTP&dc=DB600&uid=swg21264563&loc=en_US&cs=UTF-8&lang=en

Follow these steps:

1. Unzip the installation package into a local directory (Figure 6-14) and run **6.1.0-WS-WAS-WSFEP-WinX32\WEBSV\install.exe.**

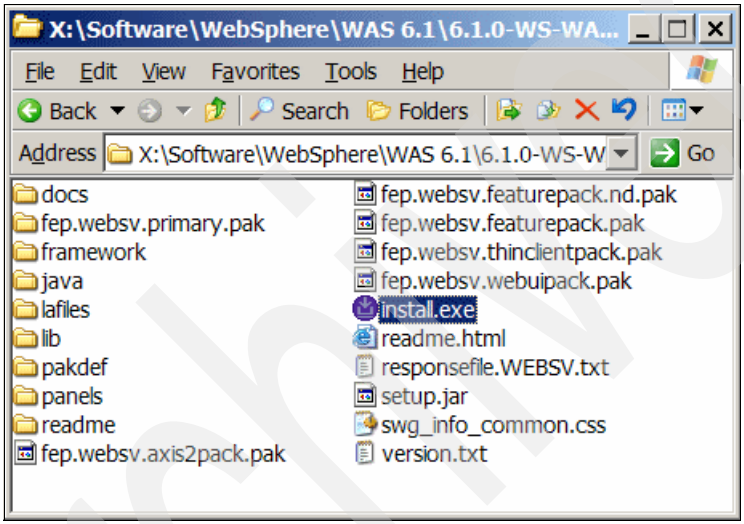


Figure 6-14 Running the feature pack installer

2. The installation wizard welcome panel warns you (Figure 6-15) that there are limitations deploying the feature pack to existing WebSphere Application Server Network Deployment environments — with the exception of the deployment manager server, you cannot augment existing WebSphere Application Server profiles, but you have to create new profiles for the feature pack (see “Profiles” on page 83).

   a. Click **Next** to continue the installation. Accept the license terms and conditions → **Next**.

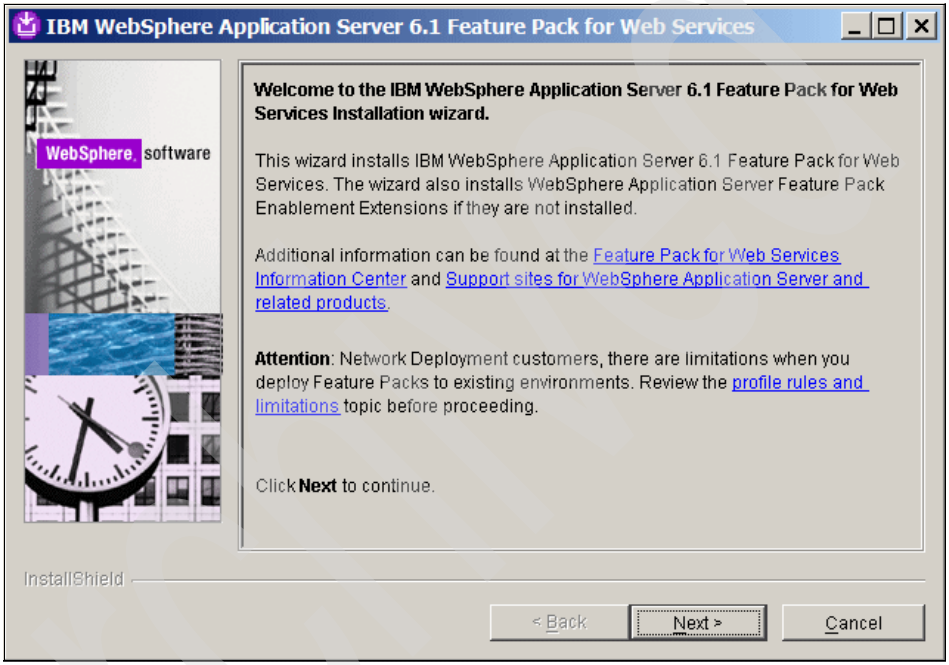


Figure 6-15 Feature pack for Web services installation welcome panel

b. Again, because we are installing onto Windows Server 2003 SP2, ignore the System prerequisite check warning (Figure 6-16) → **Next**.

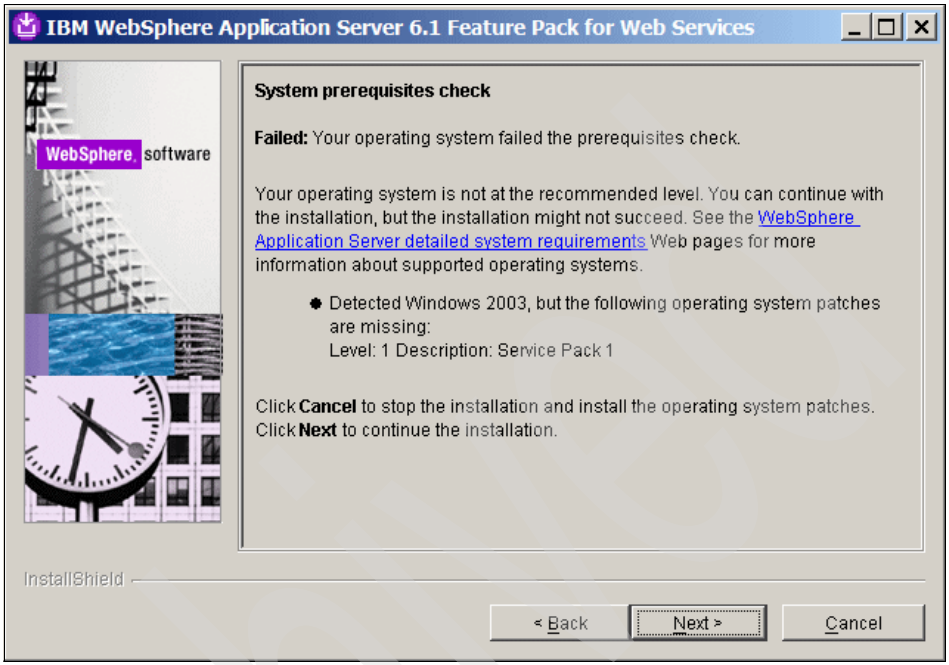


Figure 6-16 Prerequisite check warning

c. The next panel is the Version prerequisite check (Figure 6-17). Click **Next** to continue with the installation, which is automatically upgraded to 6.1.0.9.

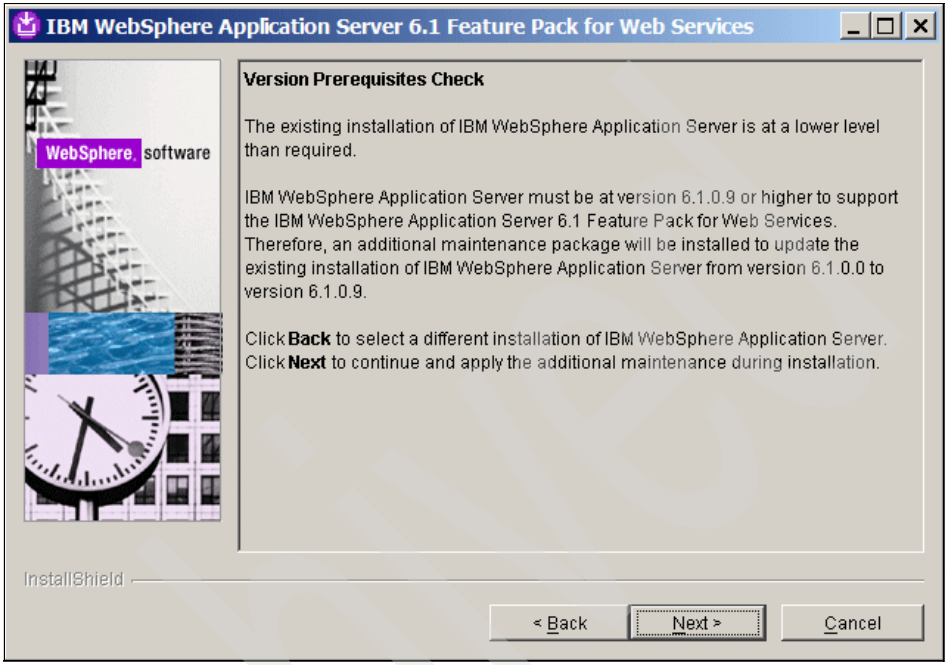


Figure 6-17 Version prerequisite check warning

3. Review the summary install panel (Figure 6-18) → **Next** to continue the Fix Pack installation.

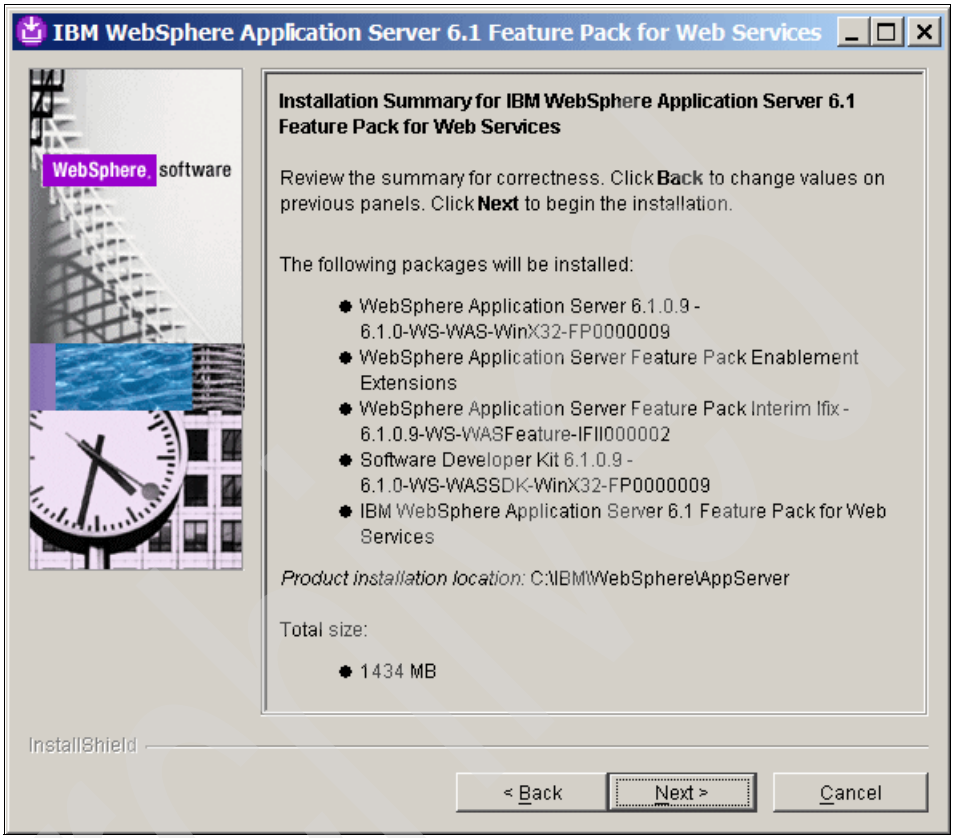


Figure 6-18 Summary of installation

On completion of the installation, the wizard has a check box to launch the profile management tool (Figure 6-19) → **Finish**.

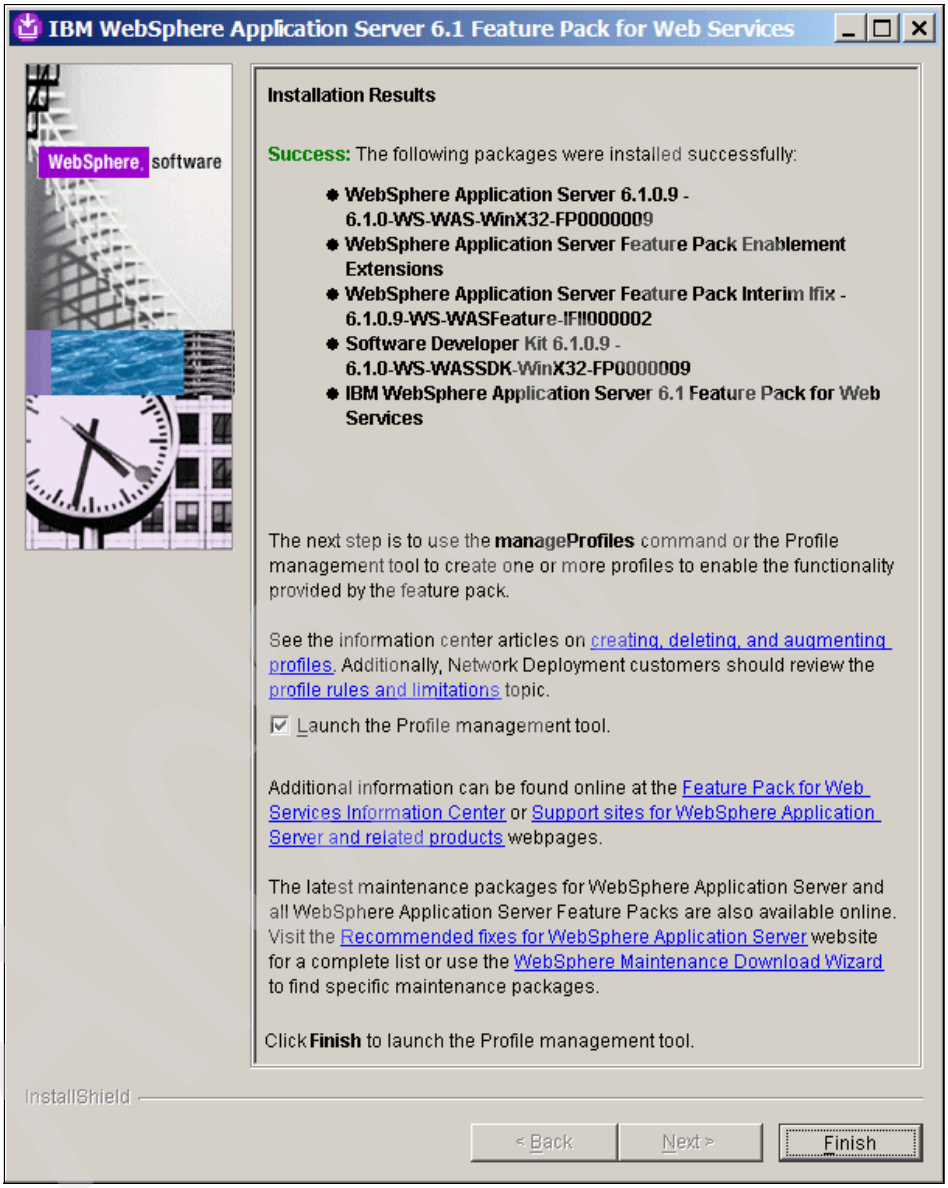


Figure 6-19 Launch the profile management wizard

6.3.3 Create Feature pack for Web services server profile

In this section, you learn how to create an application server profile that includes support for the Feature pack for Web services:

1. If you have not launched the Profile Management wizard from the previous step, click **Start** → **All Programs** → **IBM WebSphere** → **Application Server 6.1** → **Profile Management Tool** (see Figure 6-20).

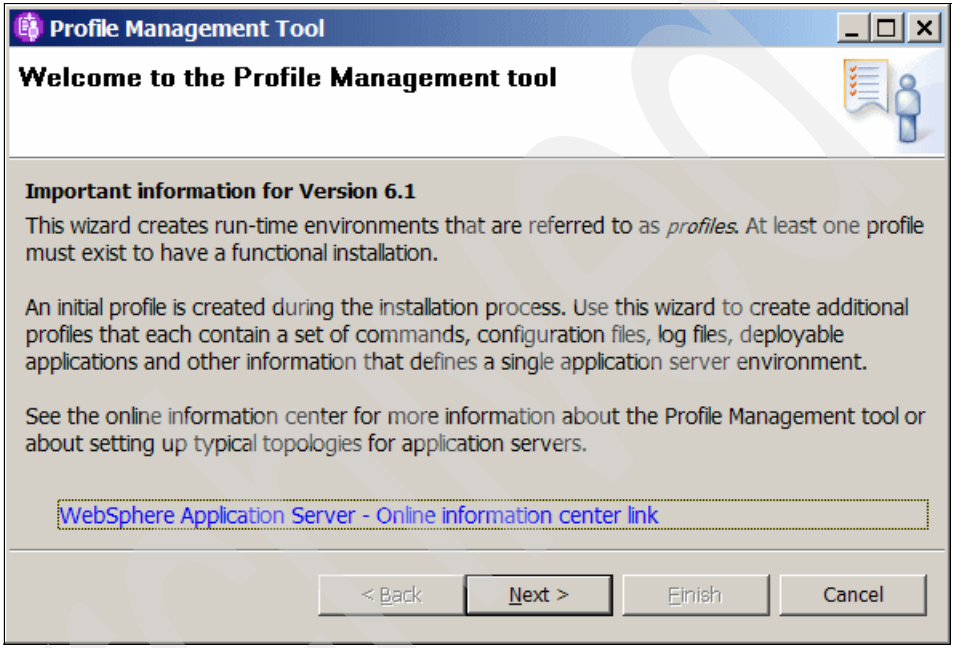


Figure 6-20 Launch the Profile Management Tool

2. On the Environment Selection panel, choose the **Feature pack for Web services** → **Next (**Figure 6-21)

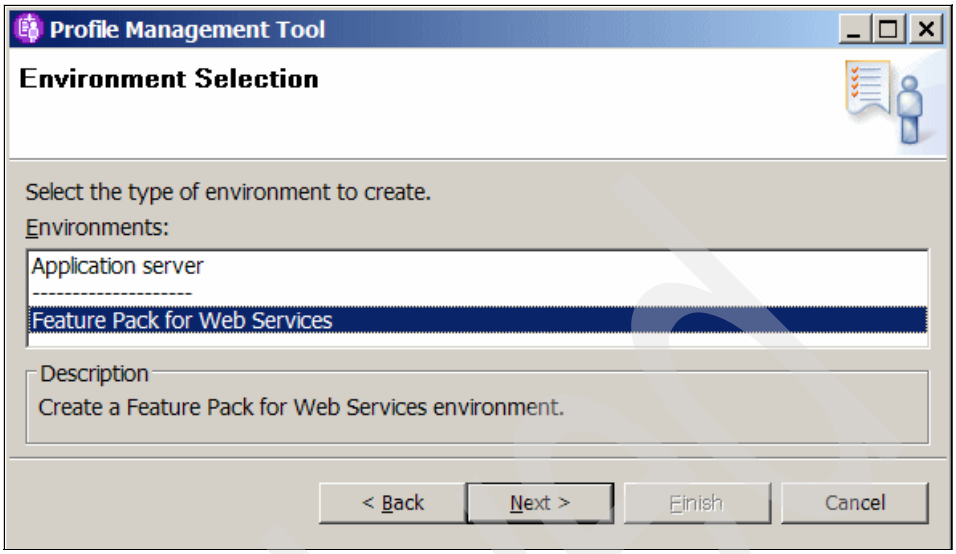


Figure 6-21 Environment Selection Panel

3. On the Profile Type Selection panel (Figure 6-22), select **Application Server with Feature pack for Web services** → **Next**.

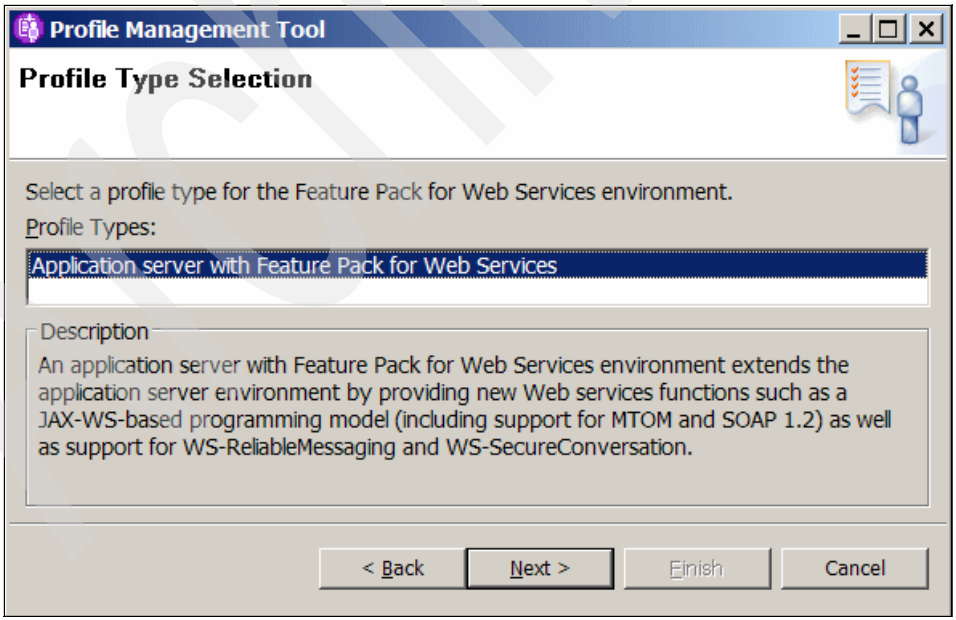


Figure 6-22 Select Application Server with Feature pack for Web services

4. On the Profile Creation Options panel (Figure 6-23 on page 102), select **Typical profile creation** option → **Next**

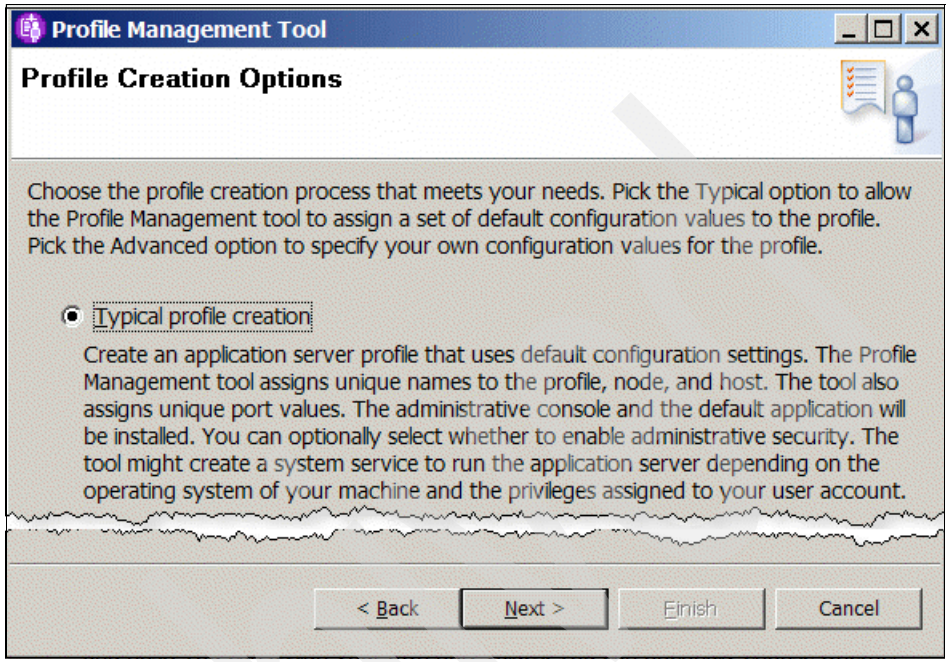


Figure 6-23 Create a typical profile

5. Review the Profile Creation Summary (Figure 6-24) → **Create**.

Figure 6-24 Profile creation summary

On the next panel, secure the administration. We have to do this to configure secure Web services. We chose to use the Windows Userid and password for our Windows account. You can enter any identifier string you choose for the Userid (Figure 6-25) → **Next**.

Figure 6-25 Make administration secure

6. On the Profile Creation Completion console (Figure 6-26), check **Launch the First steps console** → **Finish**.

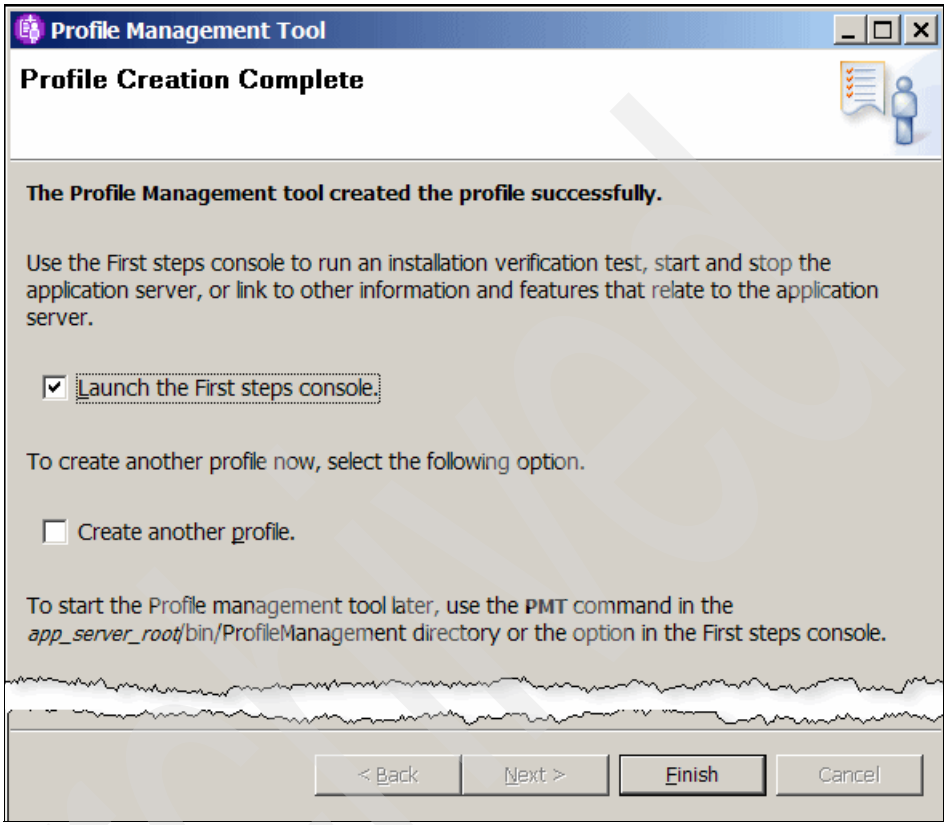


Figure 6-26 Complete profile creation

6.3.4 Verify installation

Verify the installation by launching Installation verification from the First steps console (Figure 6-27):

1. If you did not launch the First steps console from the Profile Management Tool, click **Start** → **All Programs** → **IBM WebSphere** → **Application Server 6.1** → **Profiles** → **AppSrv02**[5] → **First steps**.

[5] The profile name might be different if you are not following these instructions exactly.

2. When the verification has completed successfully, leave the server running, but change the settings of the Windows service to start manually:

   a. Navigate to **My Computer** → **Manage** → **Services and Applications** → **Services** → **IBM WebSphere Application Server v6.1** - IBM WebSphere Application Server V6.1 - itso-01Node02 and change the startup option to Manual.

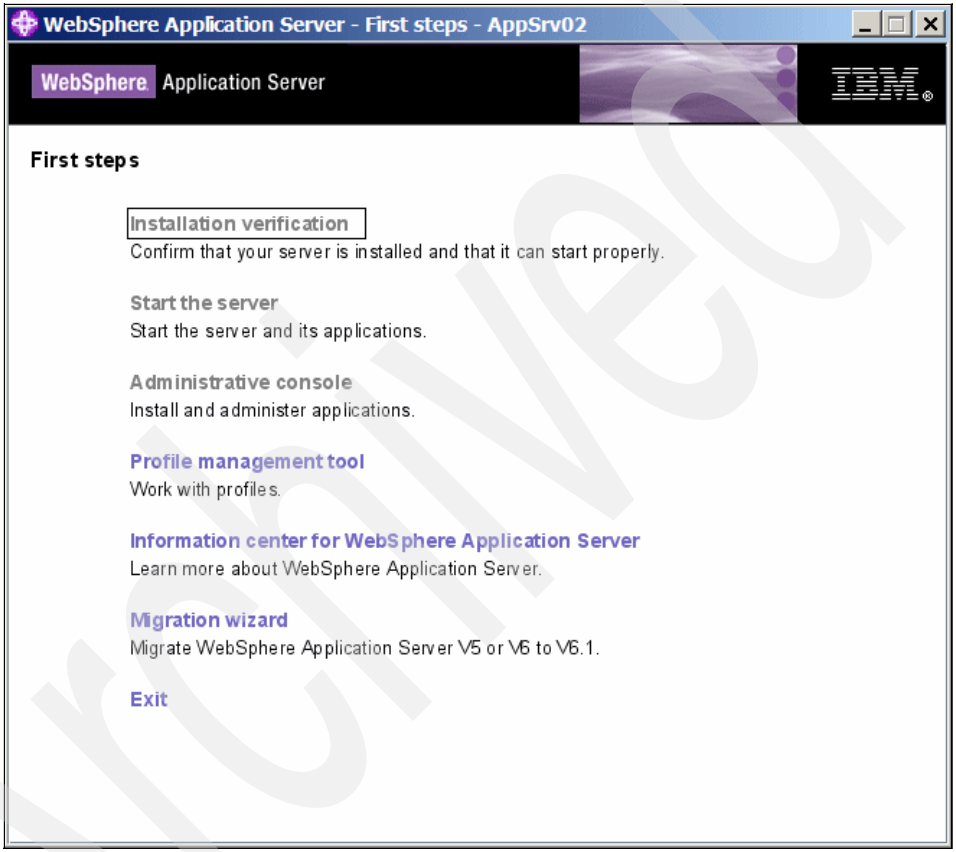


Figure 6-27 Installation verification

   b. You might also like to copy the AppSrv02 program group to the desktop, rename it more memorably, and configure it as a toolbar on the start menu bar (Figure 6-68).

Figure 6-28 Configure server profile program group as a toolbar

6.3.5 Update WebSphere Application Server v6.1 to 6.1.0.13

In this section we show you how to install updates to the Feature pack for Web services using Fix Pack 13 as an example. You have to download and install both Fix Pack 13 for WebSphere Application Server v6.1 and Fix Pack 13 for Feature pack for Web services. The Fix Packs are available from:

http://www-1.ibm.com/support/docview.wss?rs=180&uid=swg24017334

The release notes packaged with the Fix Pack caution you about necessary prerequisites (Figure 6-29). The automation of the IBM service download site simplifies the procedure somewhat by packaging the prerequisites for you. You have to perform two downloads, for both Fix Pack 13s. The prerequisite APAR, and the required version of Update Installer, are included in the packages. Duplicate downloads of Update Installer are eliminated by using Download Director, which eliminates duplicate downloads.

You do have to additionally download IFPK53210.pak if you have a standalone client installation to maintain.

```
You must download and install both WebSphere Application Server Fix
Pack 13 and Web Services Feature Pack Fix Pack 13 together, or your
Update Installer will not allow the installation to proceed. When
upgrading WebSphere Application Server and Web Services Feature Pack
for Application Server, as well as Application Client products from
version 6.1.0.11 to version 6.1.0.13 or higher level, a mandatory
Interim Fix must be installed prior to the installation of Fix Pack
6.1.0.13 or higher. The Update Installer will not allow you to
upgrade WebSphere Application Server and Web Services Feature Pack
to version 6.1.0.13 from version 6.1.0.11 without first installing
these Interim Fixes.
Download Interim Fix PK53084 which is required to be installed while
upgrading WebSphere Application Server and Web Services Feature Pack
for Application Server version 6.1.0.11 to version 6.1.0.13 or
higher level.
Download Interim Fix PK53210 which is required to be installed while
upgrading WebSphere Application Client and Web Services Feature Pack
for Application Client version 6.1.0.11 to version 6.1.0.13 or
higher level.
```

Figure 6-29 Prerequisite warnings

The full list of fixes required is shown in Table 6-1.

Table 6-1 Required fixes for 6.1.0.13

Fix	Description
6.1.0-WS-WAS-WinX32-FP0000013.pak	Fix Pack 13 for WebSphere Application Server v6.1
6.1.0-WS-WASWebSvc-WinX32-FP0000013.pak	Fix Pack 13 for Feature pack for Web services
6.1.0-WS-UPDI-WinIA32-FP0000013	Update Installer V6.1.0.13
6.1.0.9-WS-WASWebSvc-IFPK53084.pak	Prerequisite server fix
6.1.0.9-WS-AppClientWebSvc-IFPK53210.pak	Prerequisite client fix

Installing Update Installer

Before starting, copy all files listed in Table 6-1 into a drive letter mapped drive:

1. Unzip Update Installer into the root of your WebSphere Application Server installation directory, overwriting any previous version of Update Installer (Figure 6-30).

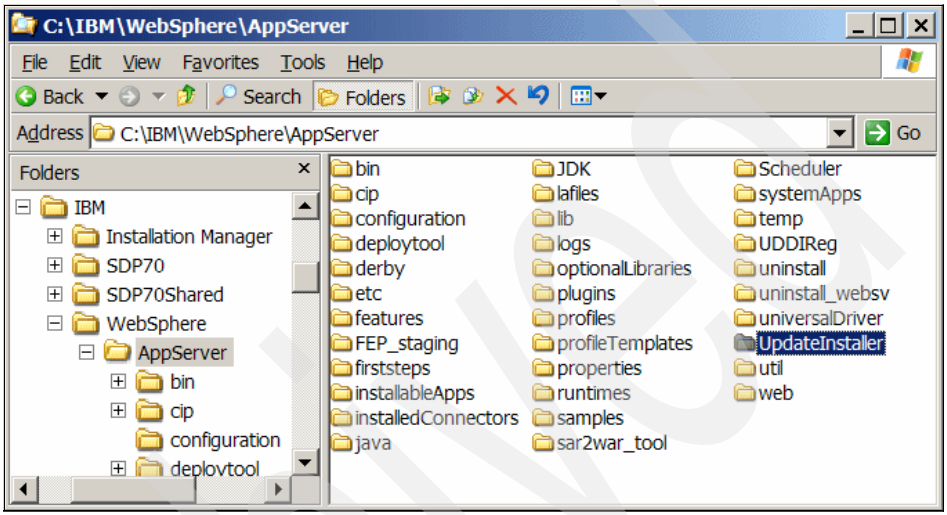


Figure 6-30 Unzipping Update Installer to the WebSphere Application Server directory

Note: Placing the unzipped Update Installer directory into the Appserver directory means it can find a JVM automatically. If you unzip to a temporary folder elsewhere, you have to set up a path to your JVM (Figure 6-31).

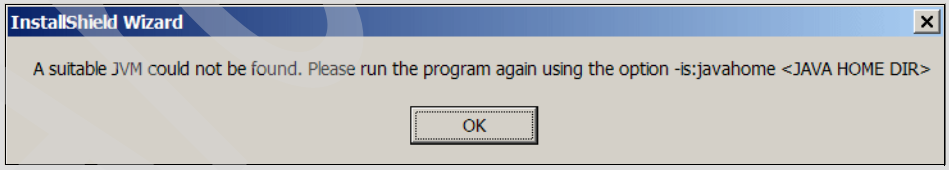


Figure 6-31 Installing Update Installer to a temporary directory

The installation directory of Update Installer must be empty, so we have to install Update Installer in a different location. We chose to install it one level up in the directory tree, at the same level as the WebSphere Application Server installation directory. This is unusual, and you might prefer to spend the time to find the path to your JVM, and unzip the Update Installer installation image to a temporary directory.

2. Click on install.exe to launch the installation of Update Installer from the unzipped directory (Figure 6-32).

Figure 6-32 Launch Update Installer installation

On the installation wizard welcome page, click **Next** → Accept the license terms and conditions → **Next** → Check the System Prerequisites; should be fine → **Next** → Select a different installation directory (for example, see Figure 6-33) → **Next.**

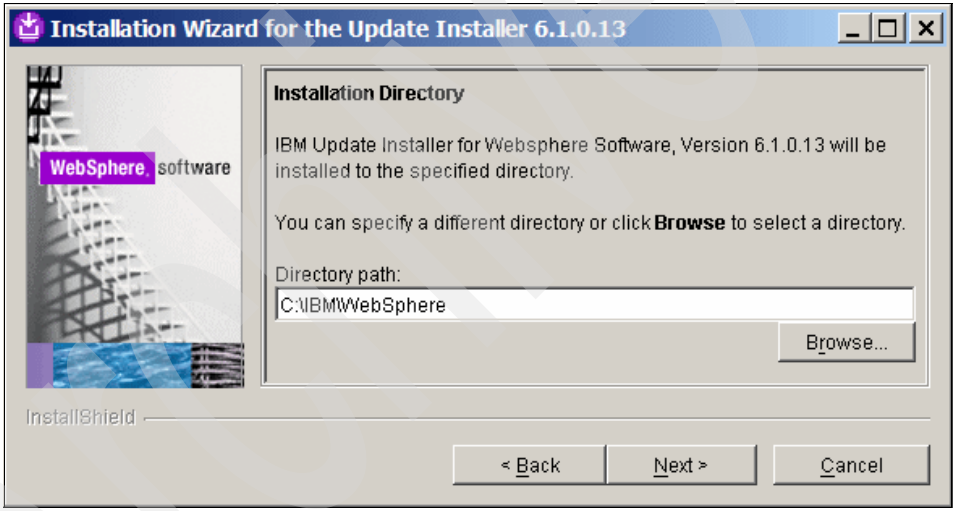


Figure 6-33 Choose a different installation directory to the unzipped Update Installer

Click **Next** on the Installation summary panel to start the installation (Figure 6-34) → Leave the Launch Update Installer check box selected → **Finish** to launch Update Installer.

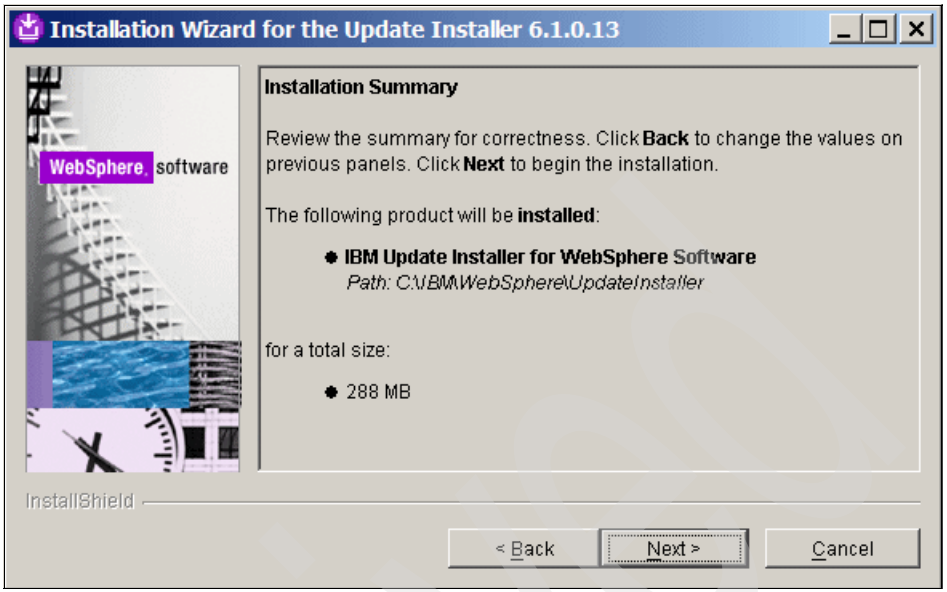


Figure 6-34 Installation summary

Install maintenance

To install maintenance on IBM WebSphere, use the IBM Update Installer tool. It checks prerequisites and manages the installation of fixes in the right order:

1. If you did not launch Update Installer automatically, launch it from the Start menu, **Start** → **All Programs** → **IBM WebSphere** → **Update Installer for WebSphere 6.1 Software** → **Update Installer** (Figure 6-35 on page 112). Check that WebSphere Application Server is stopped (look at the Task Manager, or issue the serverStatus.bat command from a command window → **Next.**

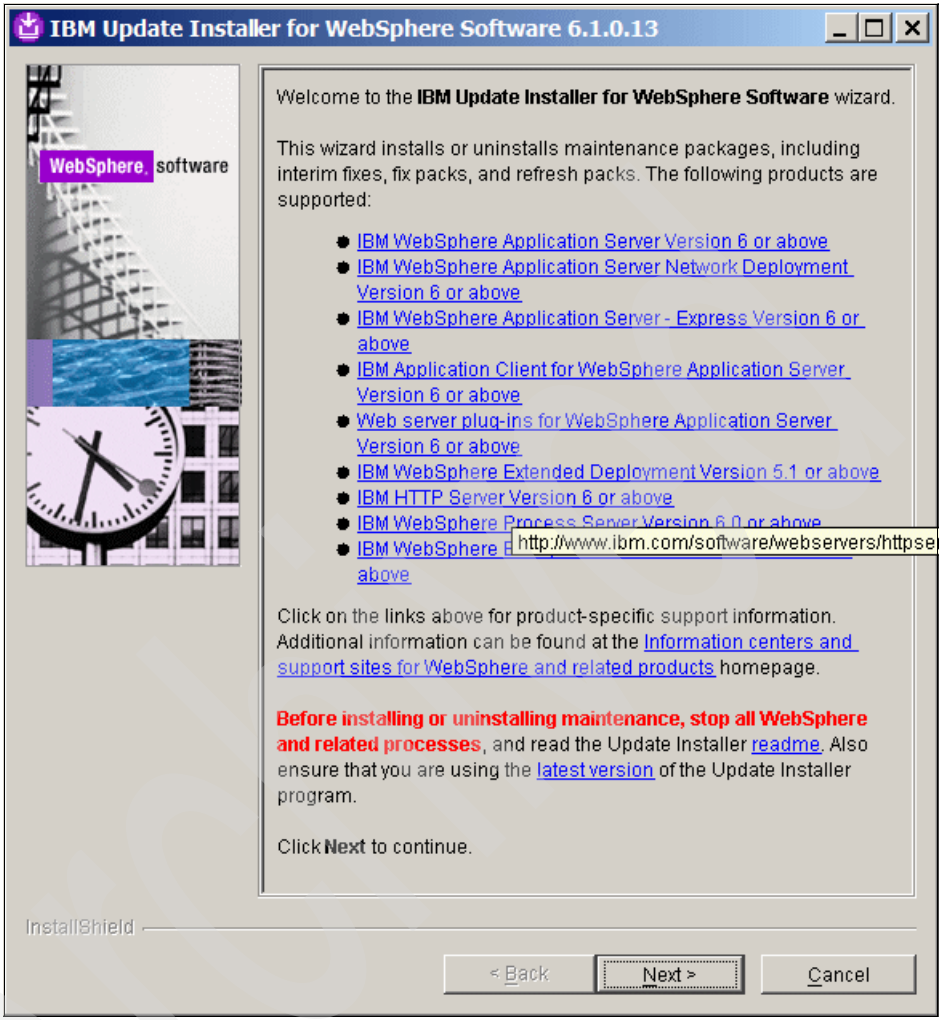


Figure 6-35 Launch Update Installer

2. Update Installer displays the application server path on the Product Selection panel (Figure 6-36), check that it is correct → **Next.**
3. On the Maintenance Package Directory Selection panel, select the folder to which you downloaded all the maintenance fixes (Figure 6-37).

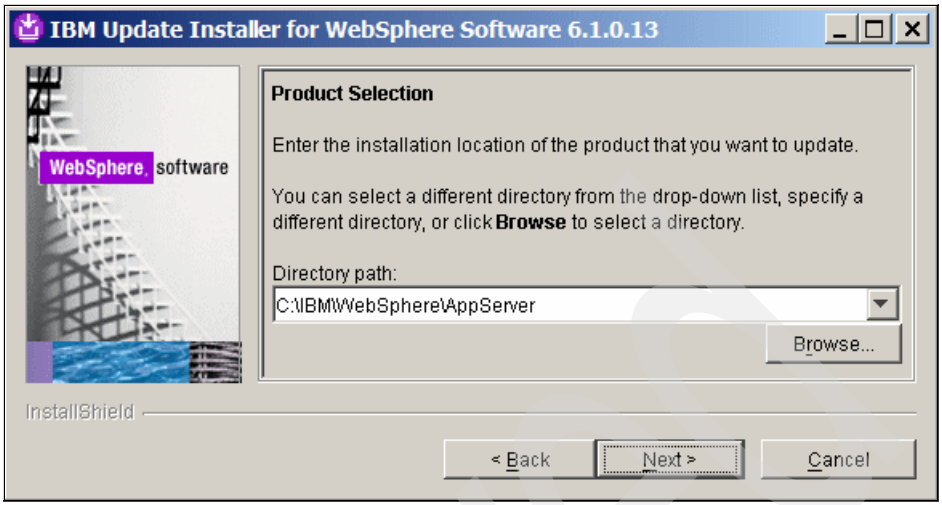


Figure 6-36 Product Selection panel

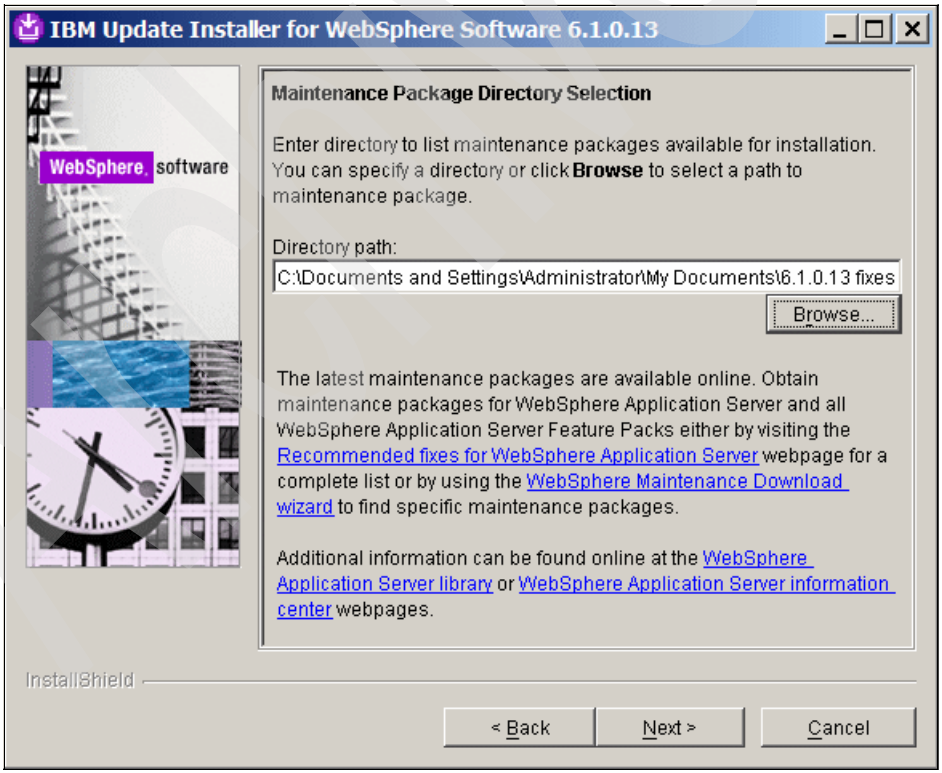


Figure 6-37 Point to the location of the fixes

4. The default (recommended) selection on the panel, Available Maintenance Packages to Install, is incorrect. Uncheck the recommended selection and check 6.1.0.9-WS-WASWebSvd-IPFK53084.pak (Figure 6-38) → **Next.**

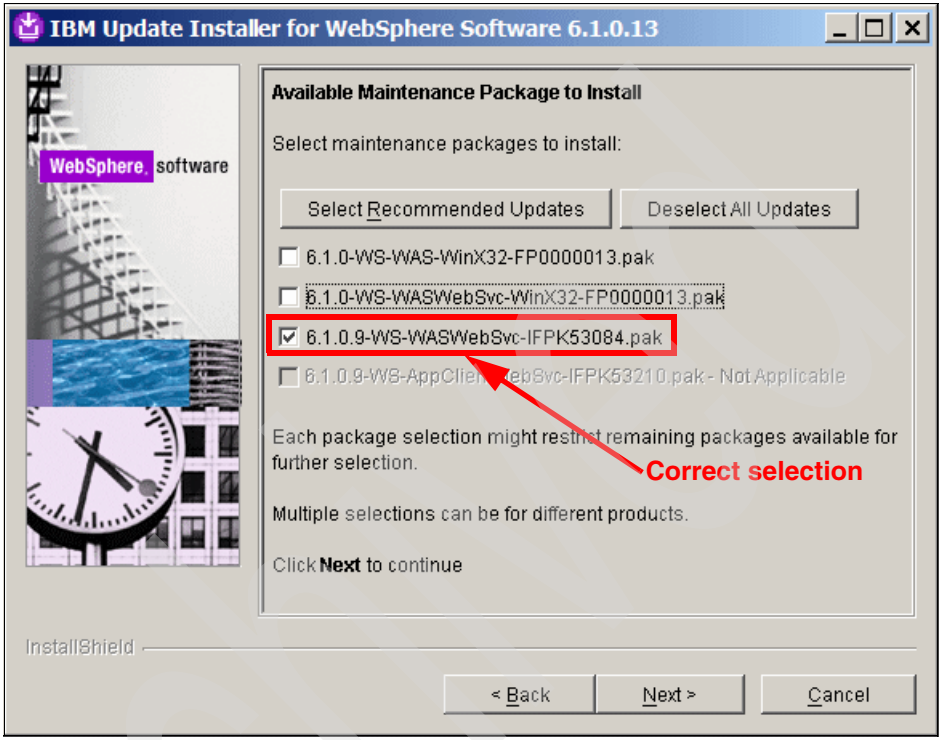


Figure 6-38 Selecting the maintenance to install

5. The next panel (not shown) summarizes the maintenance to be installed → **Next** → Click **Relaunch** on the installation complete panel (Figure 6-39).

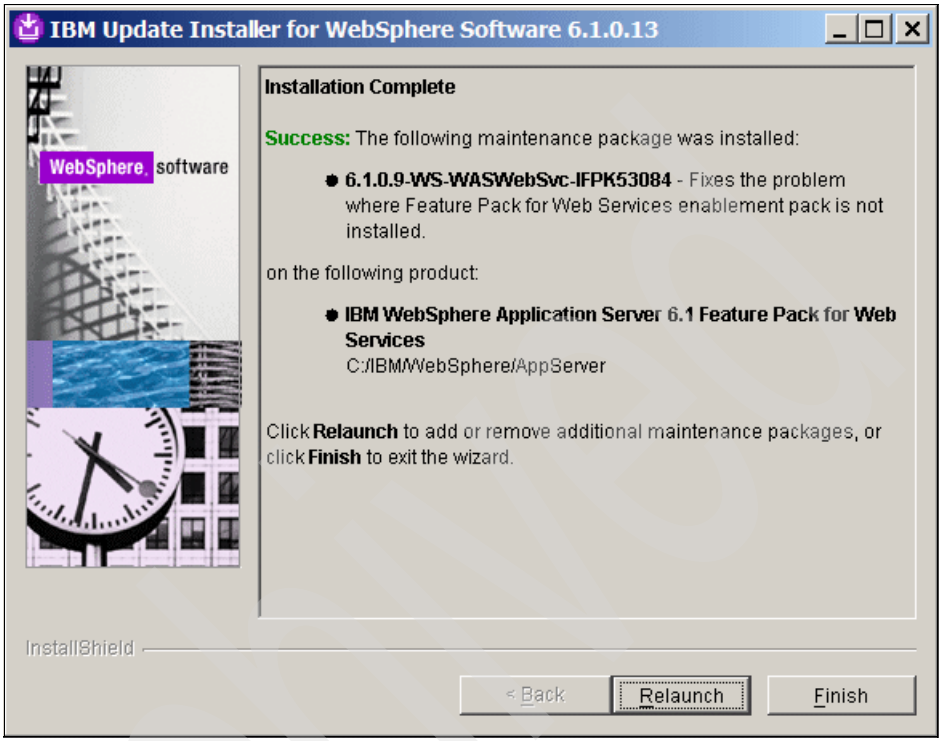


Figure 6-39 installation of the temporary fix completed successfully

6. The Available Maintenance Package to Install panel now shows the list of fixes to apply checked correctly (Figure 6-40) → **Next**.

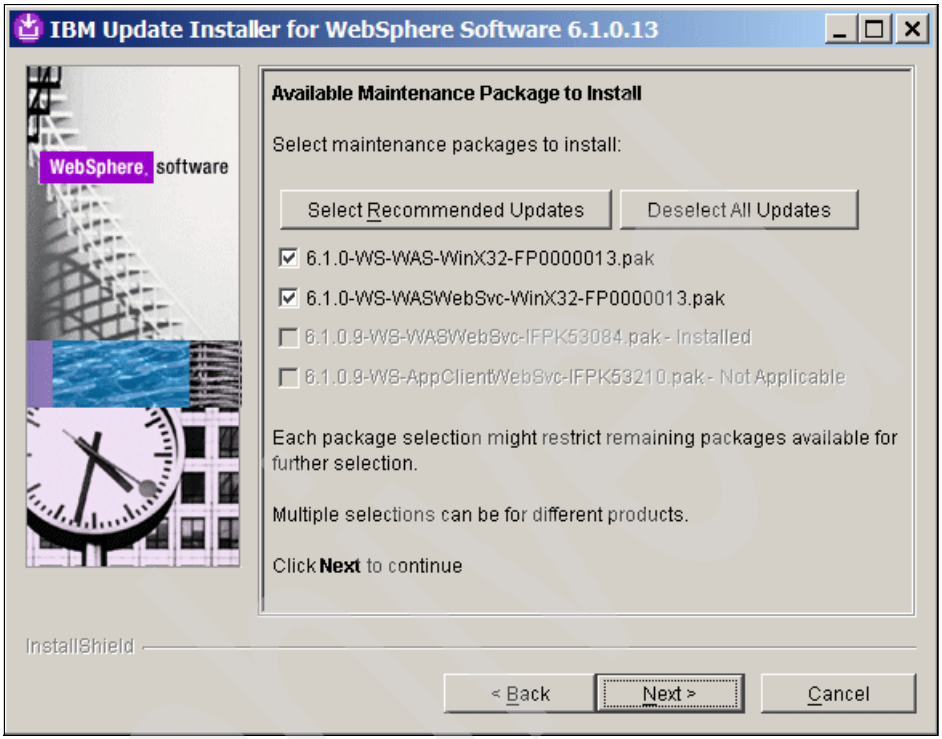


Figure 6-40 Correct selection of fixes to install

7. The Warning panel, (not shown) informs you that some fixes will be uninstalled → **Next.** The following panel (not shown) informs you that Feature pack for Web services was detected. → **Next** → There is an Installation Summary panel (not shown). → **Next** → Finally, click **Finish** on the Installation Complete panel (Figure 6-41).

This completes the installation of all the components required for Fix Pack 13 of the Feature pack for Web services.

You can delete the directory to which you unzipped the installation image of Update Installer (but keep the Update Installer installation directory, of course), and you can delete the directory to which you downloaded the Fix Packs.

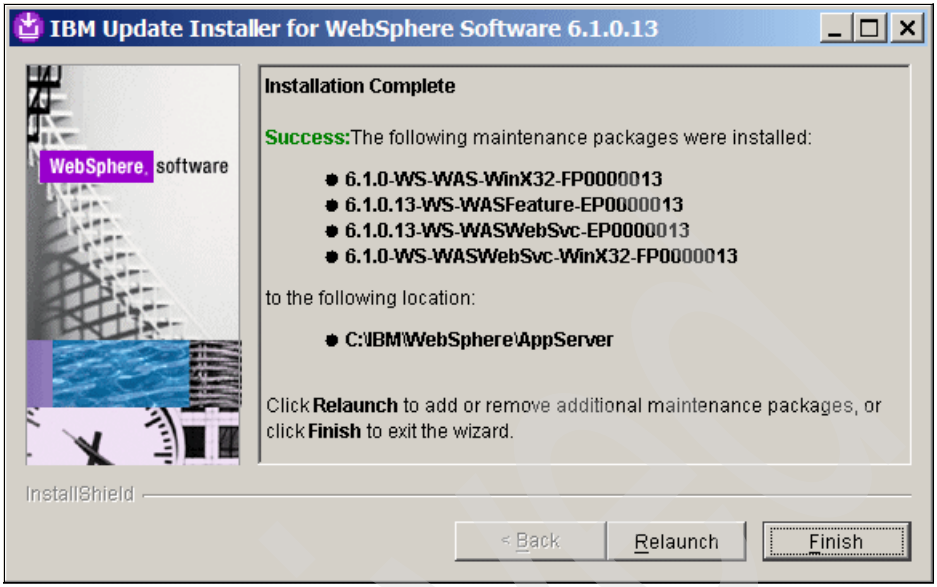


Figure 6-41 Installation of Fix Pack 13 for Feature pack for Web services complete

6.3.6 Install Rational Application Developer

At the time of writing this book, the current service level for Rational Application Developer v7.0 is 7.0.0.5.

You can choose to install the JVM v5 and Eclipse V3.2 separately before installing Rational Application Developer. Or you can choose to install the JVM and Eclipse bundled with Rational Application Developer v7. We shall do the latter.

You can also choose to install Rational Application Developer v7, and then apply the update (in our case 7.0.0.5) from the IBM service site on the Internet. Because 7.0.0.5 is so large (2.74 GB), or because like ours, your installation machine is not attached to the Internet, you might choose to download 7.0.0.5, and install it from your local disk[6].

[6] This procedure also saves you some installation time, because the Fix Pack is installed at the same time as Rational Application Developer, rather than in two separate installations.

Follow these steps for the installation:

1. Download the 7.0.0.5 Fix Pack from:
   http://www-1.ibm.com/support/docview.wss?uid=swg24017403&rs=0&cs=utf-8&context=SSRTLW&dc=D400&loc=en_US&lang=en&cc=US
2. Extract the installation package for Rational Application Developer v7 and the 7.0.0.5 to your local disk (this can take an hour or more).
3. Download the latest fixes for the IBM Installation Manager (currently 1.1.0.1). and extract to your local disk. This saves time in case the Fix Pack requires a later version than the installation image version. Find the latest installer fixes at:

   http://www-1.ibm.com/support/docview.wss?uid=swg24017183&rs=0&cs=utf-8&context=SSRTLW&dc=D400&loc=en_US&lang=en&cc=US
4. Open the directory you unzipped the Rational Application Developer installation images to and start **`disk1\launchpad.exe`** (Figure 6-42).

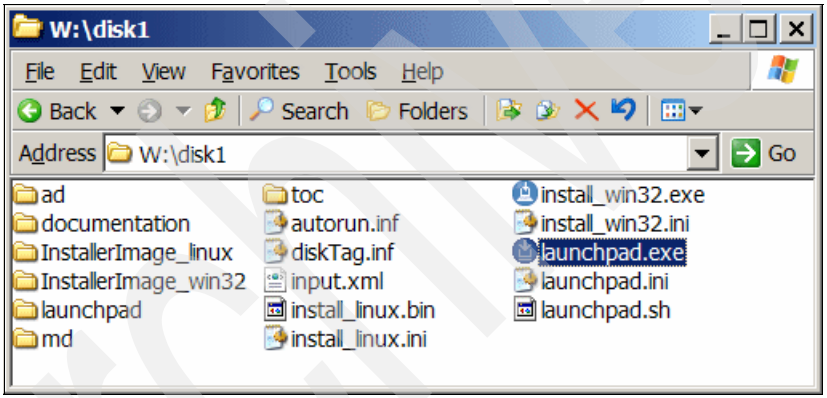


Figure 6-42 Installing the launchpad

Install the Installation Manager

Follow these steps for the installation:

1. Click **Install IBM Rational Application Developer V7.0** on the first panel of the Launchpad (Figure 6-43).

   The next series of panels configures and installs V1.0 of the Installation Manager. The installation process assumes that we are going to continue to install Rational Application Developer v7.0 immediately when the Installation Manager v1.0 installation completes. In our case, we want to interrupt the installation process, and configure the Installation Manager to pick up the v7.0.0.5 fixes to Rational Application Developer and the v1.1.0.1 fixes to the Installation Manager, before installing Rational Application Developer.

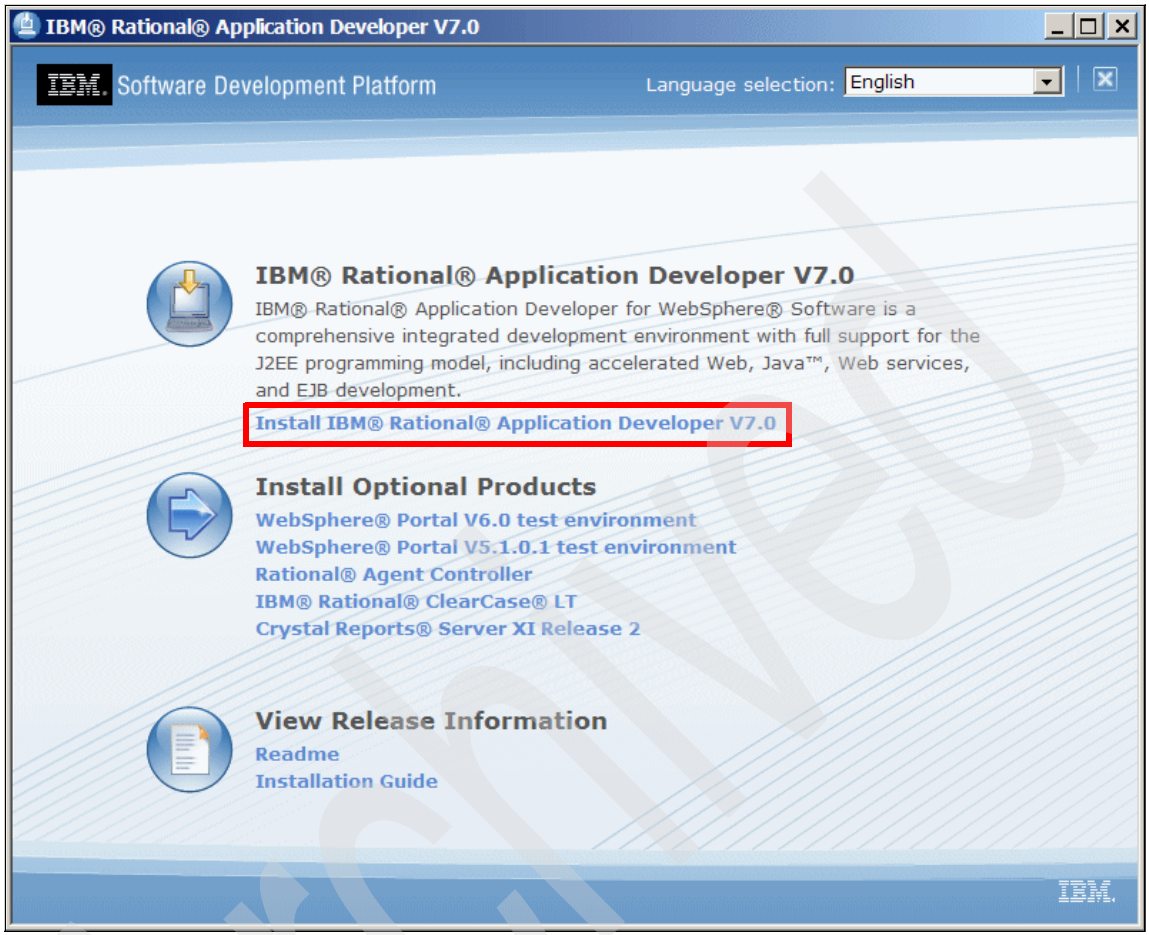


Figure 6-43 Launch Rational Application Developer installation

2. On the Welcome panel of the Installation Manager (Figure 6-44) → **Next** → Accept the license terms → **Next** → Change the destination folder path on the next panel to remove “Program files” (Figure 6-45) → **Install**.

Figure 6-44 Launchpad Welcome panel

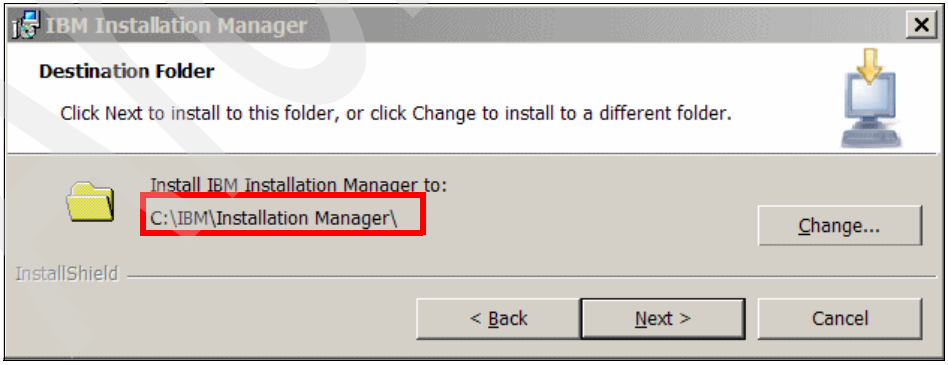


Figure 6-45 Change the destination folder

3. Click **Finish** on the completion panel to end the installation (Figure 6-46).

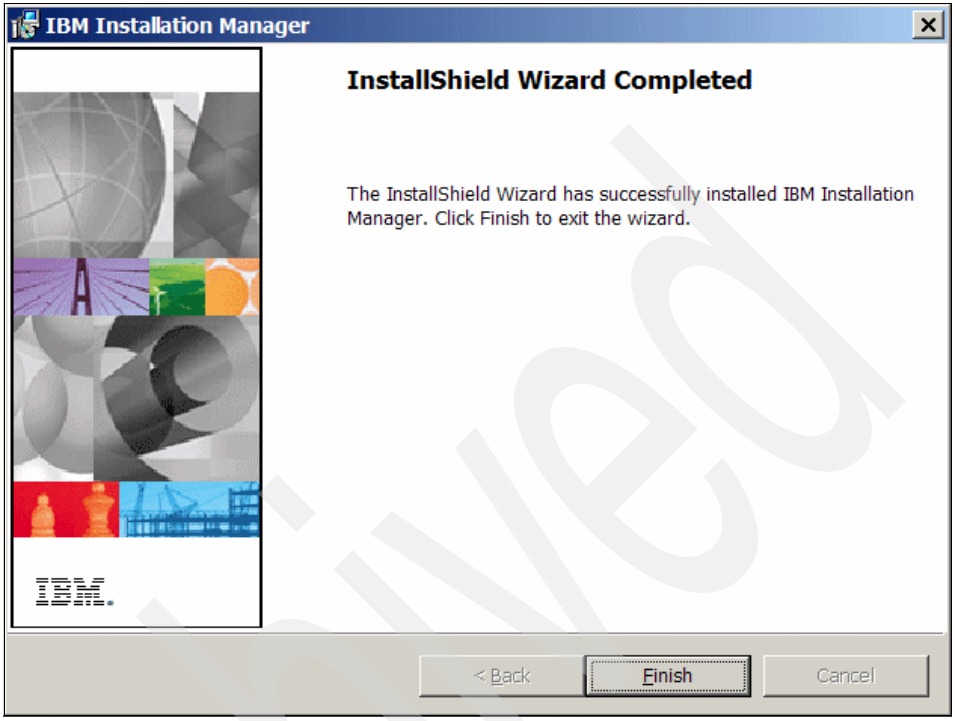


Figure 6-46 Complete the Installation Manager installation

4. The Installation Manager restarts the Rational Application Developer installation (Figure 6-43 on page 119). We want to open the Installation Manager at its start page so that we can configure its profile.

- Close the window, and launch the Installation Manager manually from the start menu: **Start** → **All Programs** → **IBM Installation Manager** → **IBM Installation Manager** (Figure 6-47).

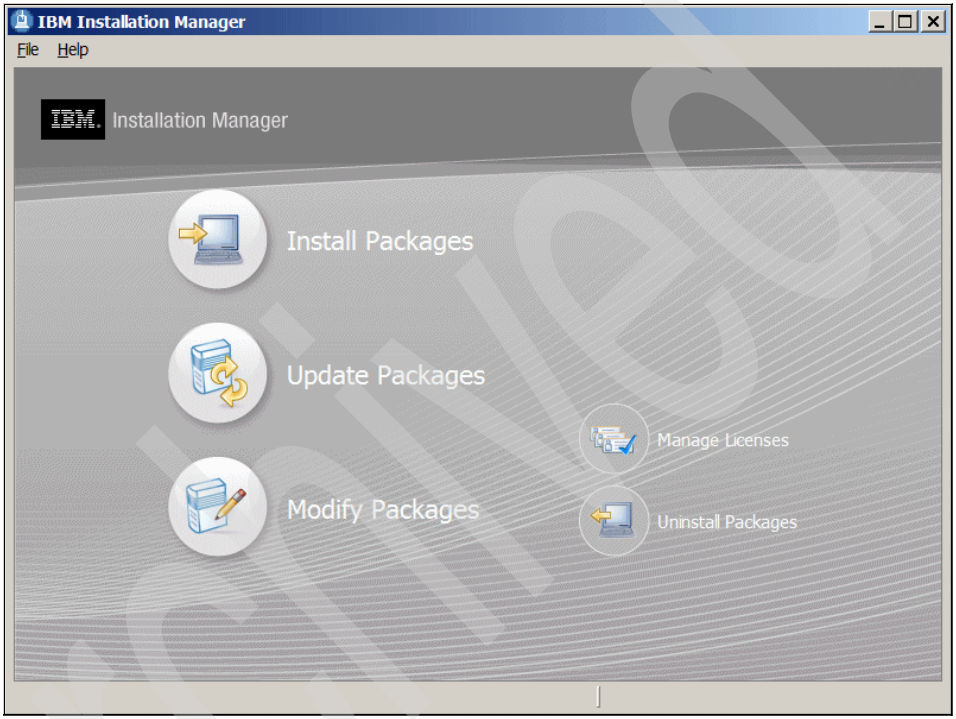


Figure 6-47 Installation Manager

5. We now add the three repositories to the Installation Manager that are used in the installation:

   a. Rational Application Developer v7.0
   b. Fix Pack 7.0.0.5
   c. Installation Manager 1.1.0.1

   Unpack all the three sets of zip files to a common directory, such as *RADv7*. We are going to add the repository information files for each installable image to the Installation Manager preferences. Figure 6-48 shows the location of the three files that define the installation contents. The next step is to add these files to the Installation Manager preferences.

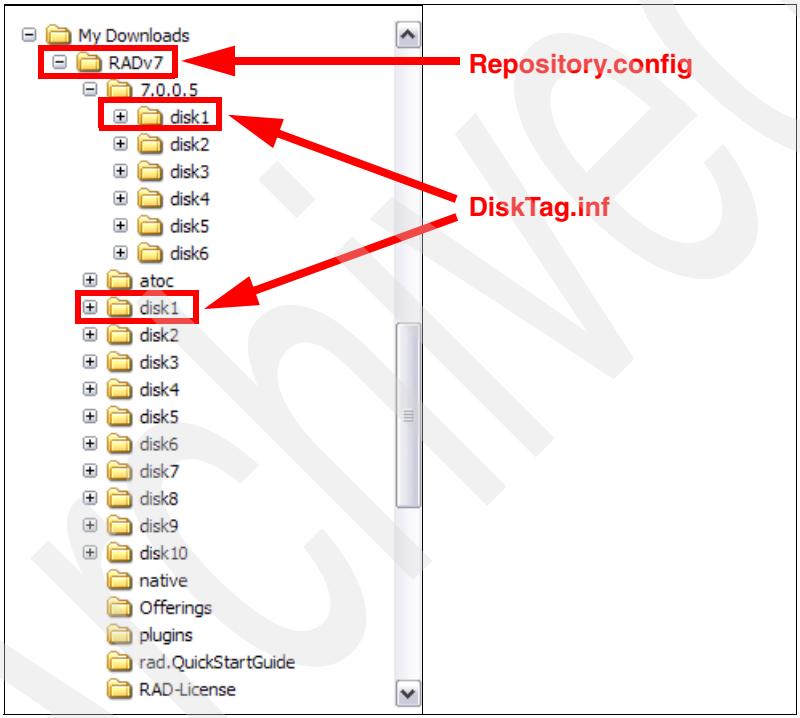


Figure 6-48 Repository information files

6. In the Installation Manager → **File** → **Preferences...** → **Add Repository** and add each of the three file locations (Figure 6-49).

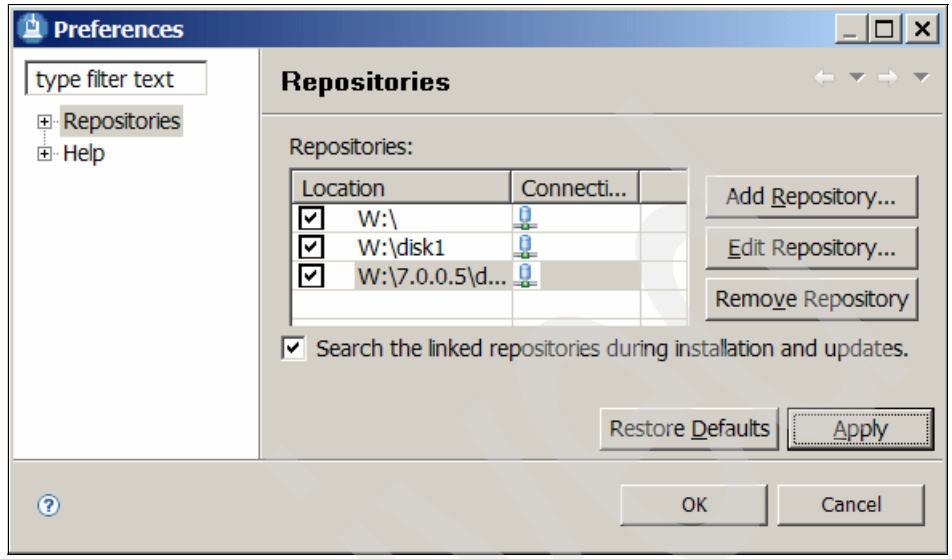


Figure 6-49 Added all three repositories

7. When you press **Apply** → **OK**, Installation Manager detects that there is a new version of Installation Manager available and displays as in Figure 6-50. Click **Yes** → **OK** to install the new version. The Installation Manager relaunches (Figure 6-47 on page 122).

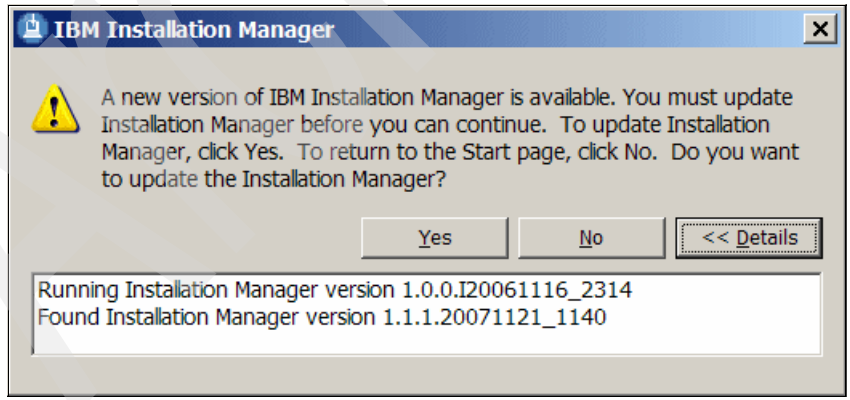


Figure 6-50 Detected a new version of Installation Manager

Install Rational Application Developer

1. Click the **Install Packages** button of the Installation Manager (Figure 6-47 on page 122). Figure 6-51 shows that now we are installing v7.0.0.5, and not v7.0 as was the case to begin with → **Next** → Accept the license terms and conditions → **Next**.

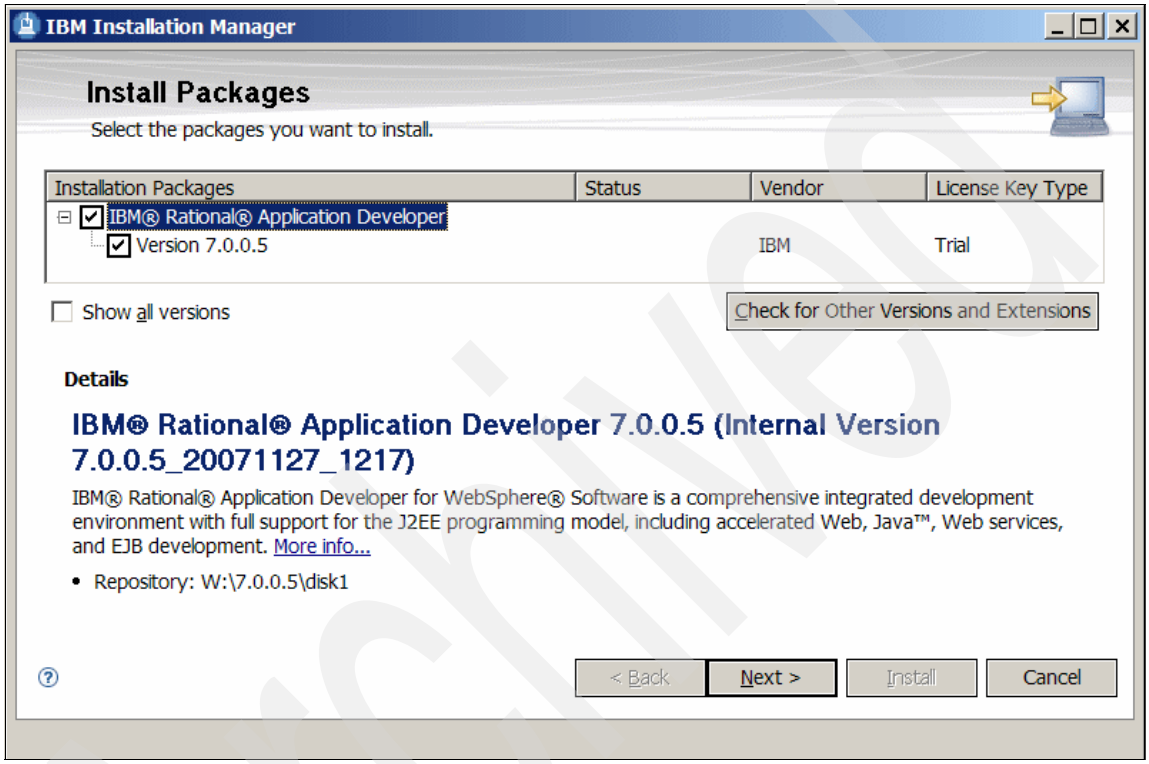


Figure 6-51 Installation of Rational Application Developer 7.0.0.5

2. You can choose, as shown in Figure 6-52 on page 126, to change the installation path, by removing the Program files node in the directory tree for both the installation of the shared resources → **Next** (screen capture omitted) and the package group → **Next**.

> **Note:** Some installations can share the same package group, and then they run in the same Eclipse shell. Rational Application Developer co-exists with Rational Software Modeler and Rational Data Architect for example, but currently cannot co-exist with WebSphere Integration Developer. However you can run tools in different package groups on the same workstation simultaneously, they run in different eclipse shells and obviously require a lot more memory.

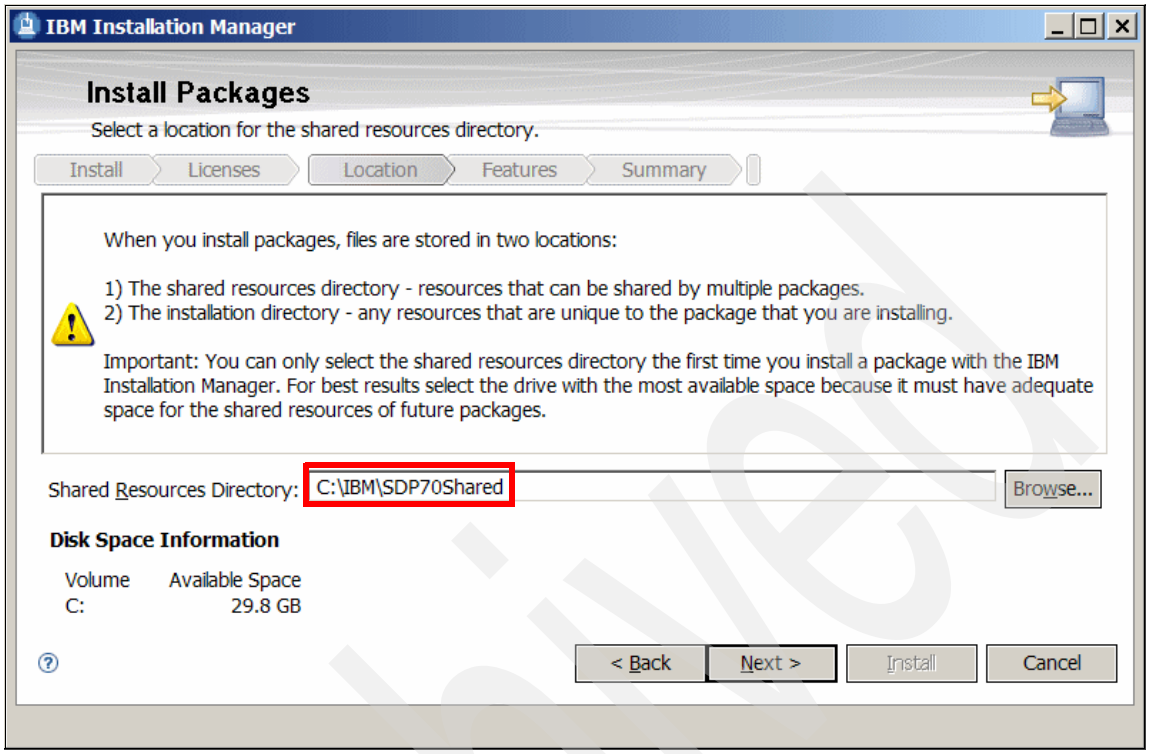


Figure 6-52 Configuring Shared resources directory

3. Because we chose not to install a JVM and Eclipse before installing Rational Application Developer, leave the options in Figure 6-53 empty → **Next**.
4. Leave the language choice as the default US English → **Next**.

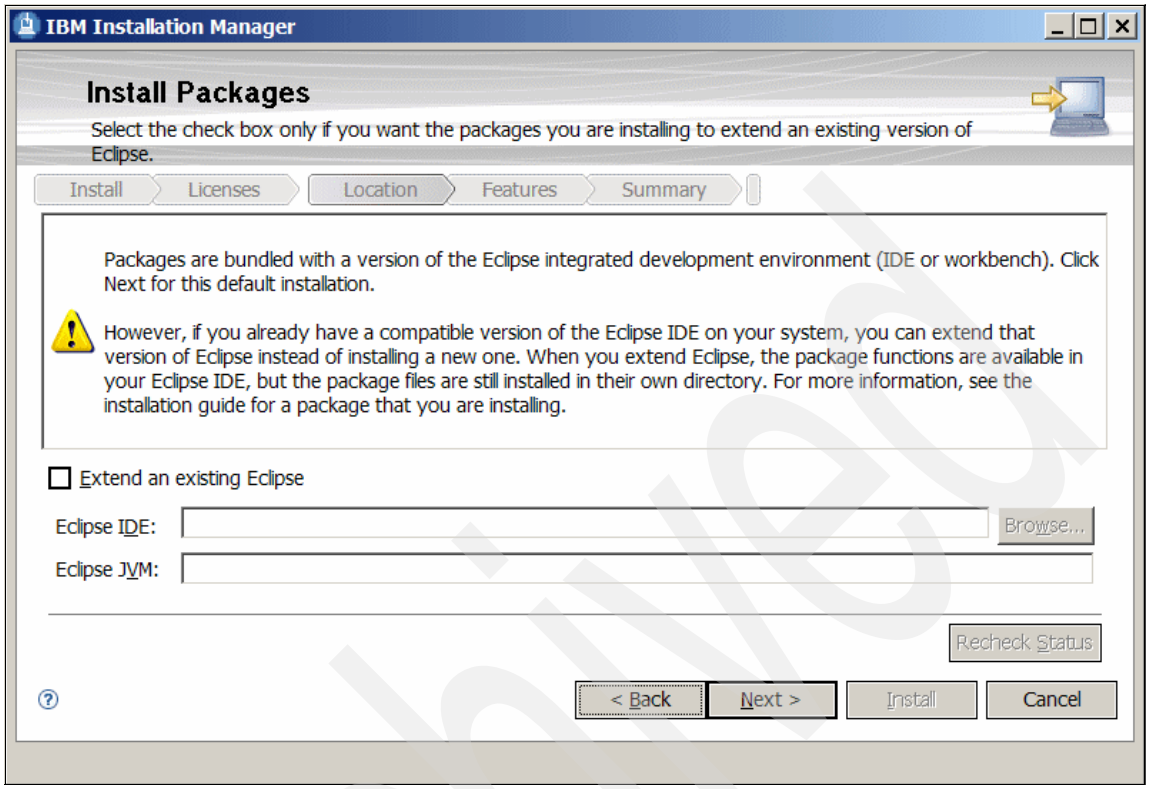


Figure 6-53 Extend existing Eclipse Environment

5. The next panel, Figure 6-54, has a large selection of choices to install. We take all the defaults, except that we *must* select the Feature pack for Web services. We have also deselected the choice of installing WebSphere Application Server v6.1 as the unit test environment because we have already installed it as a standalone runtime → **Install**.

> **Note:** If you choose to install WebSphere Application Server v6.1 as the unit test runtime, note that you also have to check the box to install the Feature pack for Web services runtime too.

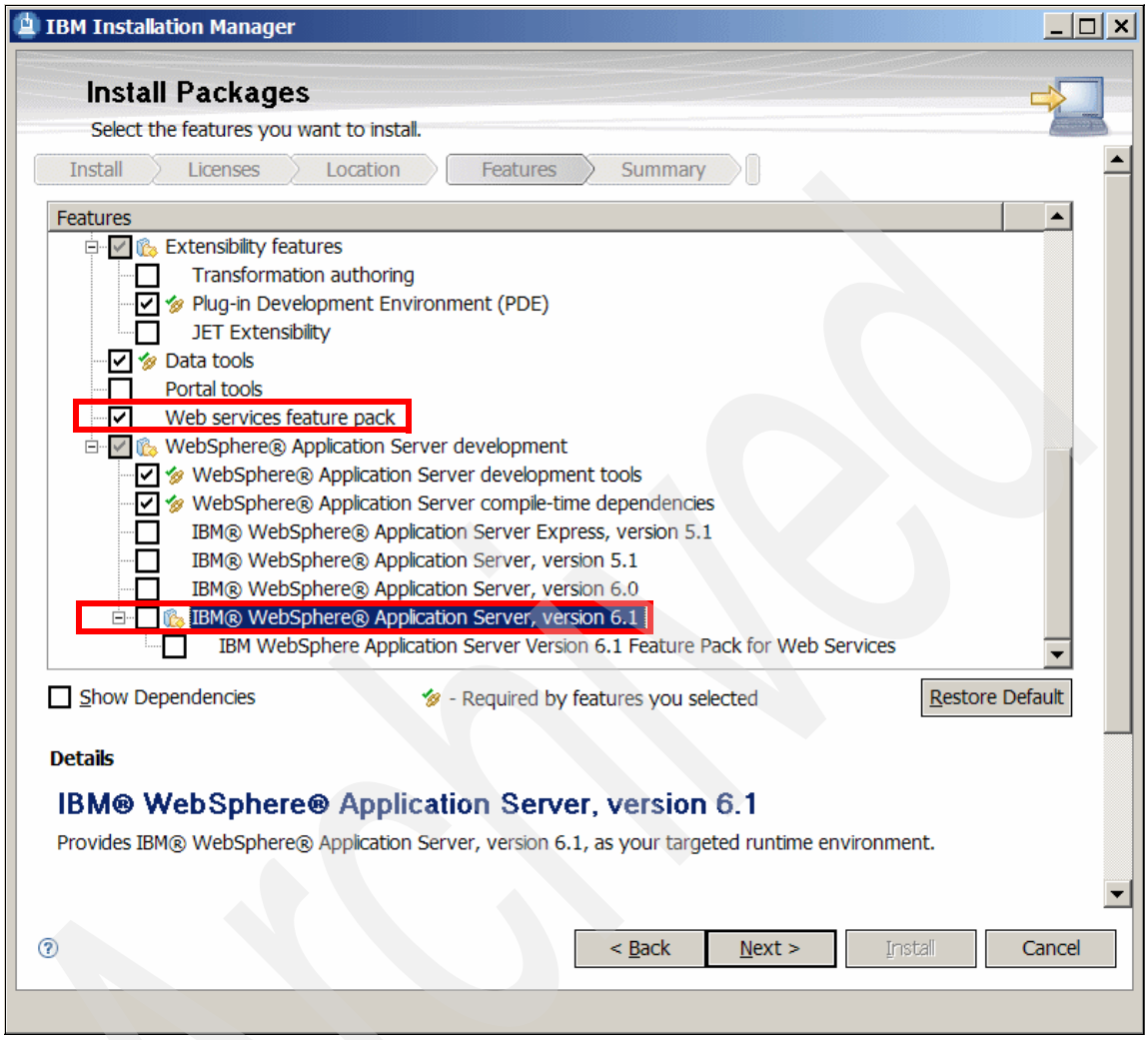


Figure 6-54 installation choices

Figure 6-55 shows the Progress panel. You can see the installation is sourced from drive W:, and there is no connection to the Internet during the whole installation.

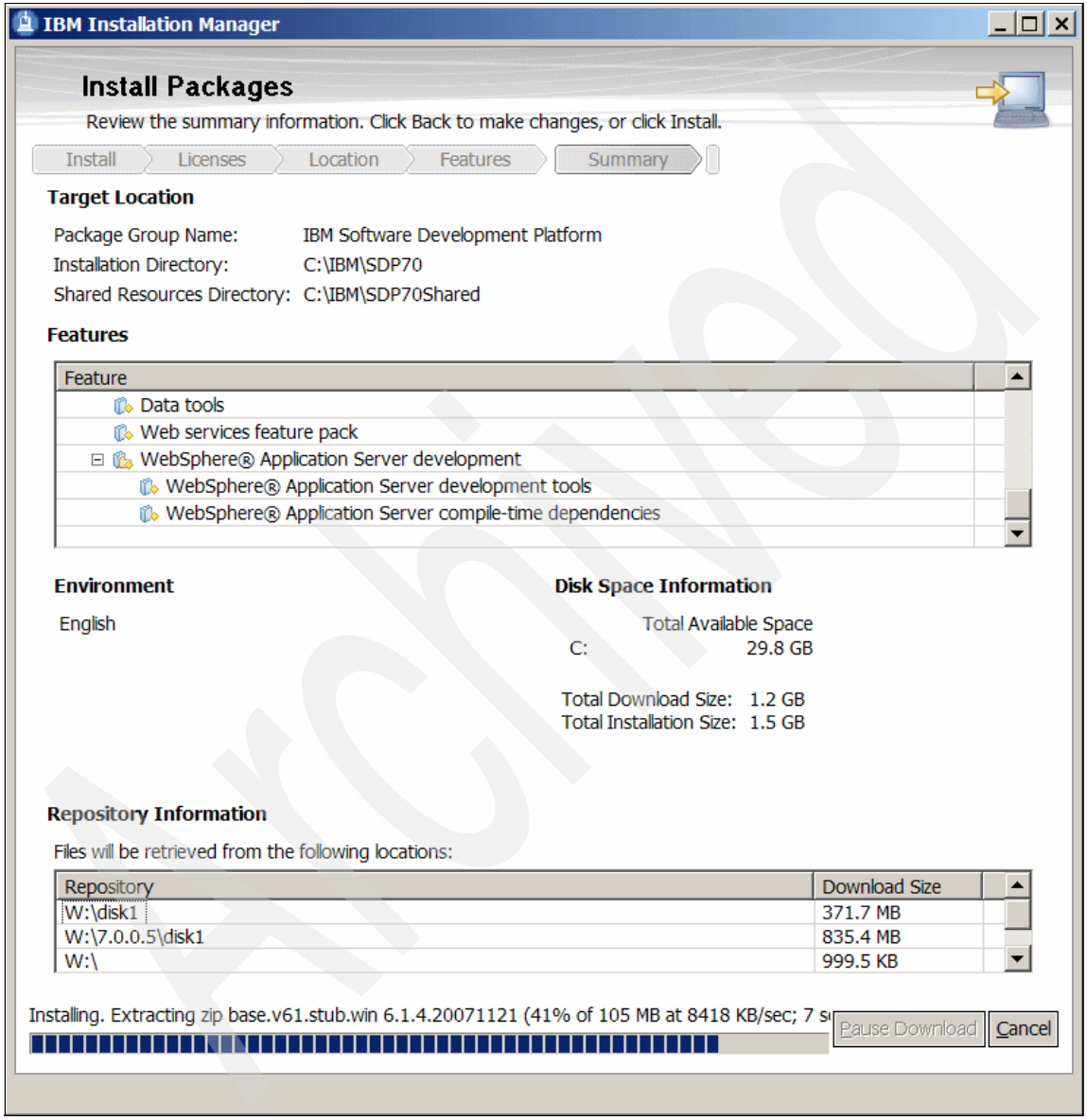


Figure 6-55 Progress panel

6.3.7 Configure test server

In this section we connect the server we configured with the Feature pack for Web services profile to the Rational Application Developer test environment.

1. If the server with the Feature pack for Web services profile is not already started, start it either from the Services list in the Microsoft Management Console, from the Start menu, from the **`StartServer.bat`** command in `C:\IBM\WebSphere\AppServer\bin`[7], or like us, from the toolbar on the start menu bar Figure 6-56.

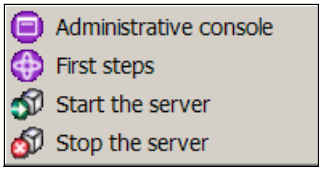


Figure 6-56 Short cuts to control a specific application server profile

2. Start Rational Application Developer in a new workspace — we have used a workspace folder called `C:\ITSO7618\Workspaces\Installation`
3. The workspace roles are already set up to match our needs (Figure 6-57). Click the **Go to Workbench** button, or close the welcome page.

Figure 6-57 Selected roles

[7] Run SetUpCmdLine.bat first to set up the appropriate command line variables.

4. You have to be in either the J2EE perspective or the Web perspective, and then select the **Servers** tab (Figure 6-58).

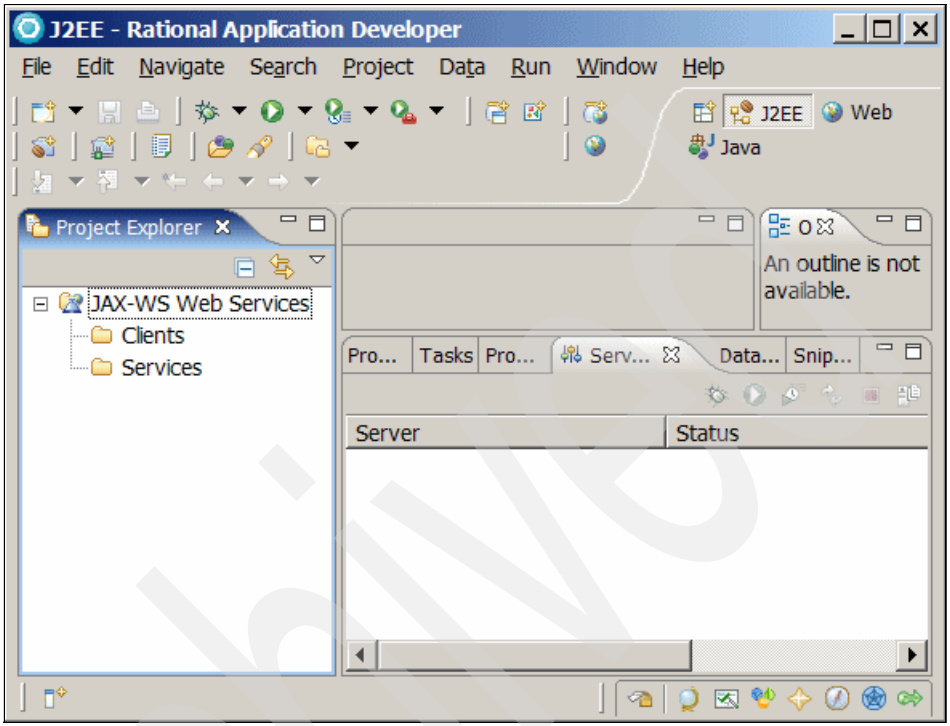


Figure 6-58 Servers tab in the J2EE perspective

5. Right-click in the Servers tab → **New** → **Server** and you are presented with the **Define a New Server** panel (Figure 6-59).

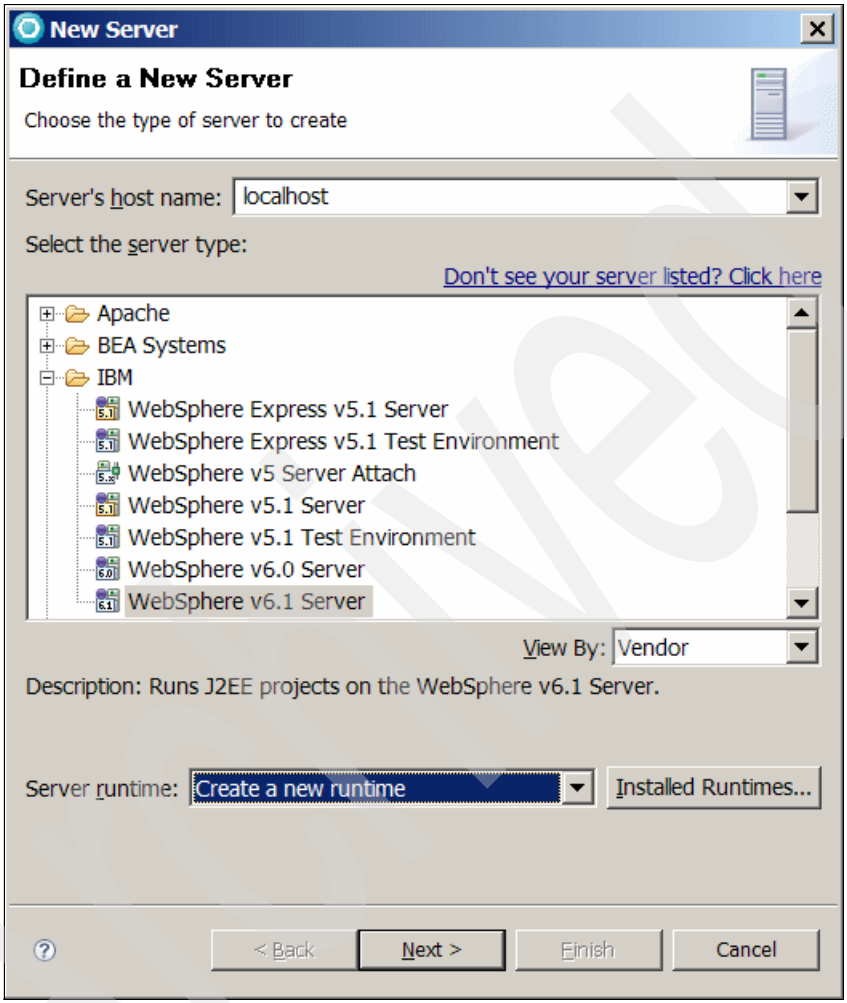


Figure 6-59 Define new server runtime

6. On this panel, you can:
   - Click **Next** and define the new server on the next panel.
   - Click **Installed Runtimes** → **Add...** and browse to the installed WebSphere Application Server v6.1 runtime in the directory tree.
   - Click **Installed Runtimes** → **Search** and Rational Application Developer finds the installed runtimes.
7. We chose the third option[8], and searched for installed runtimes. Figure 6-60 shows the Installed runtime panel with the new runtime added by Rational Application Developer → **OK**.

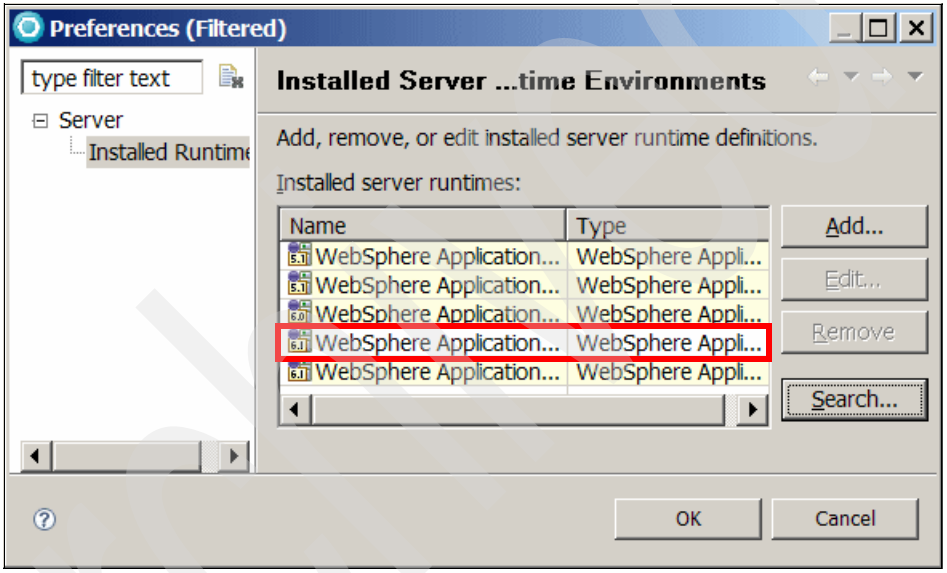


Figure 6-60 Located new installed runtime

[8] In our service level of Rational Application Developer the **Search...** option threw up two exception pop-up boxes. Ignore them by clicking the **Continue** or **Cancel** buttons. Probably because of this defect, the **Search...** option is slower than browsing for the runtimes manually.

Figure 6-61 shows the Define a New Server panel with the newly discovered runtime → **Next**.

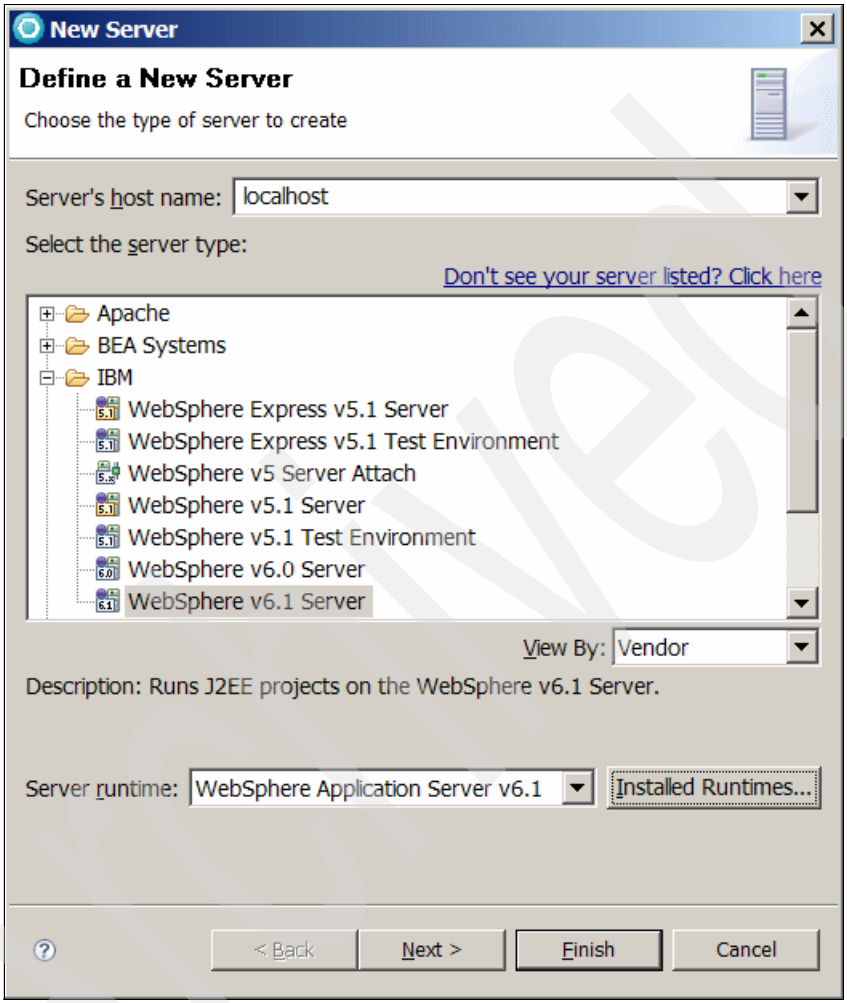


Figure 6-61 Define a new server

8. On the Server settings panel, scroll the drop-down box to select the profile configured with the Feature pack for Web services (AppSrv02 in our case), and provide the User Id and Password you supplied when you enabled administrative security when creating the profile (Figure 6-62) → **Finish**.

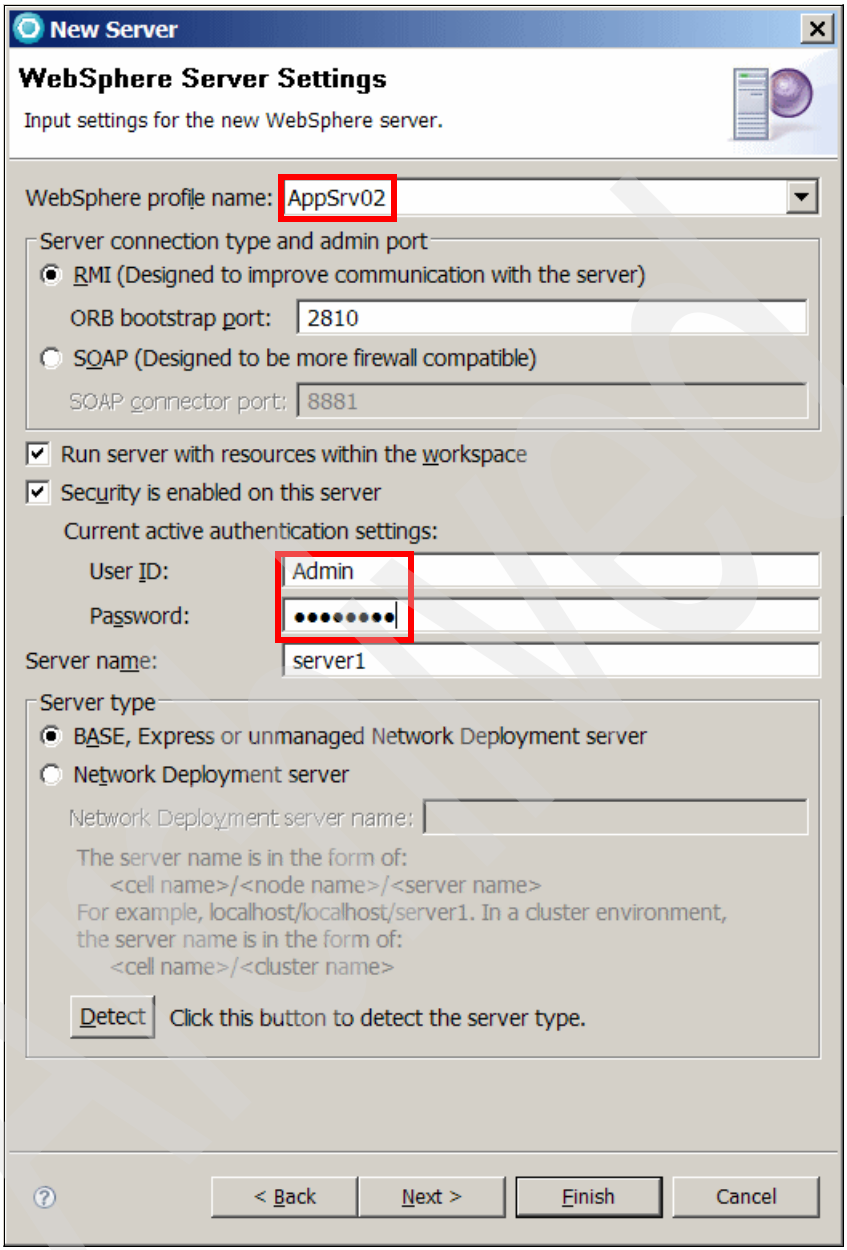


Figure 6-62 Provide profile name and security parameters

The server tab in Rational Application Developer (Figure 6-63) shows the running server.

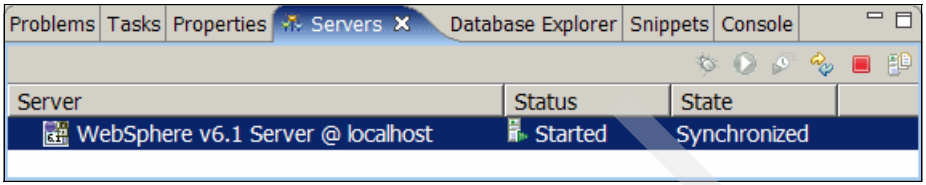


Figure 6-63 Running server

Note: We had already started the server. You can start and stop the server using the buttons in the server view. Because it can take a long time to start a server, you sometimes run into problems synchronizing Rational Application Developer with the server, unless you modify the timeouts in the workspace preferences (Figure 6-64). Alternatively, by starting and stopping the server using the WebSphere Application Server commands, you avoid this problem.

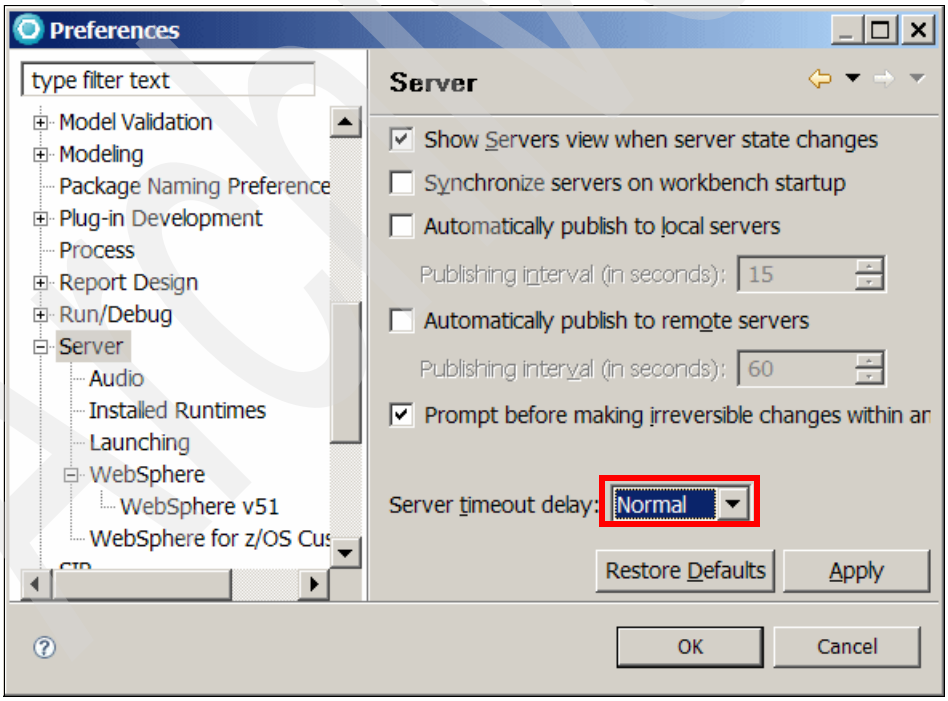


Figure 6-64 Adjusting the server timeout

6.3.8 Verify installation

To verify the complete installation, run the JAX-WS samples shipped with Rational Application Developer.

1. Either from the Welcome perspective (in the Help menu), or from the Welcome toolbar, select the **Samples** page (Figure 6-65).

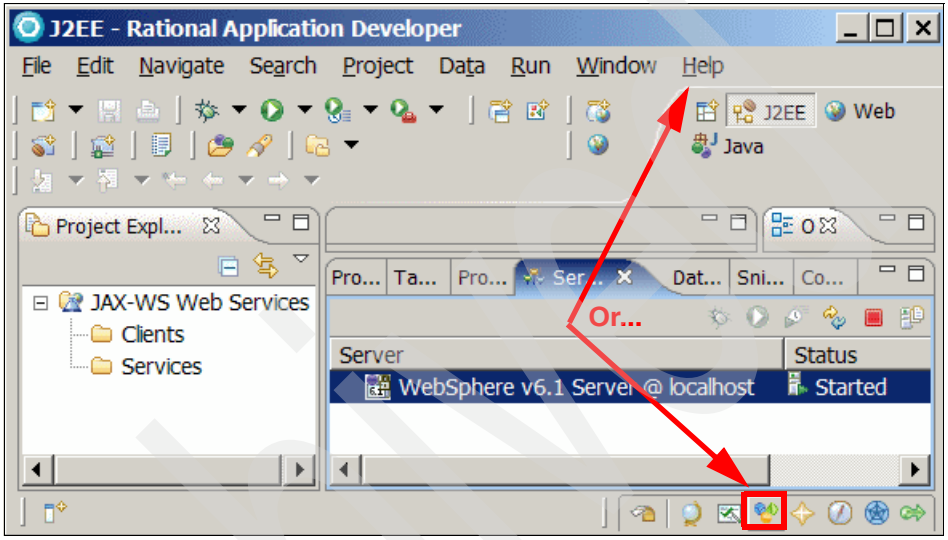


Figure 6-65 Samples button

2. Launch the Samples Gallery from the Samples welcome page (Figure 6-66).

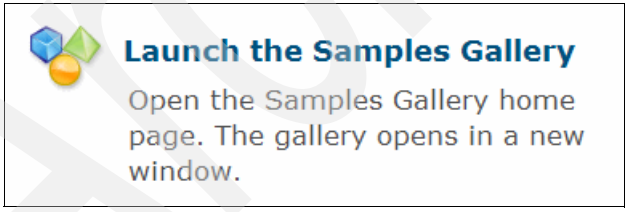


Figure 6-66 Launch the Samples Gallery

3. Expand the **Technology** → **Web services** samples in the Samples navigator and click **WebSphere JAX-WS address book Web service sample** (Figure 6-67).

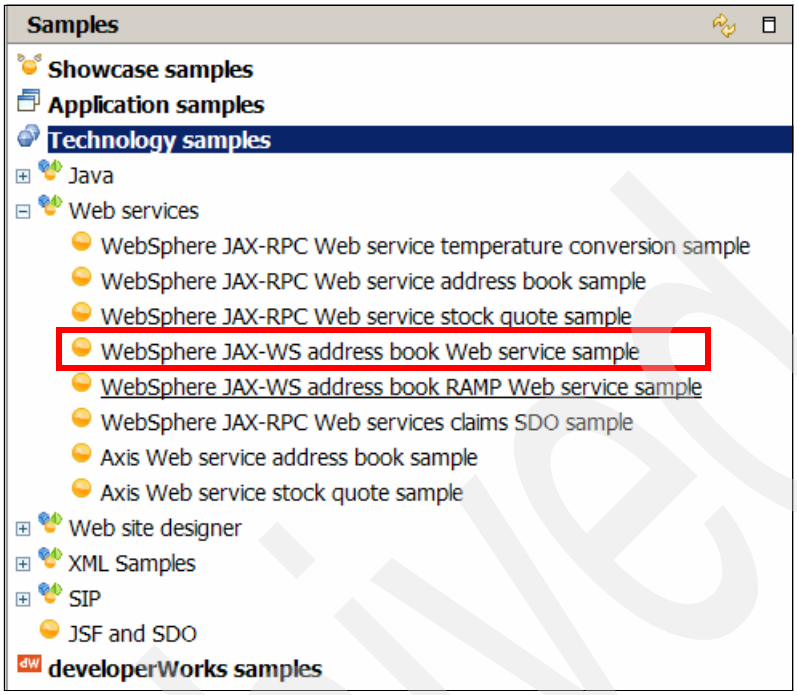


Figure 6-67 Select the WebSphere JAX-WS address book Web service sample

4. Click **Import the sample** (Figure 6-69 on page 139) → **Finish** (Figure 6-68).

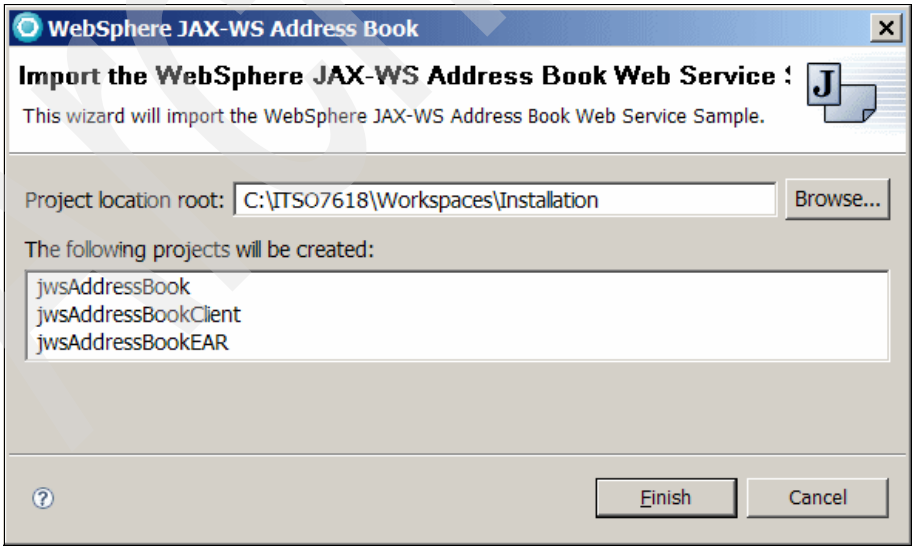


Figure 6-68 Importing the WebSphere JAX-WS Address book sample

Figure 6-69 Import the sample

5. In the project explorer, if there is an asterisk against JAX-WS Web services, then right-click the folder → **Refresh**.
6. Double-click the WSDL file under services to open it in an editor (Figure 6-70).

> **Note:** If the Services are starred, the WSDL will not open; do a **Refresh**.

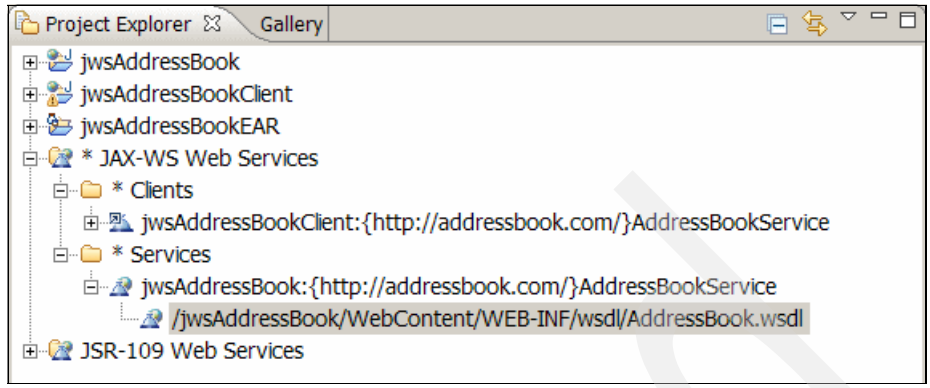


Figure 6-70 Open the WSDL file

Because the samples were built for the default server, the SOAP address does not necessarily match the SOAP address of the server that we have configured. We have to modify the SOAP address from 9080 to 9081 in our example (Figure 6-24)[9].

7. In the properties panel of the AddressBookService port, change the SOAP address to 9081 (Figure 6-71) → **Save.** The project rebuilds automatically.

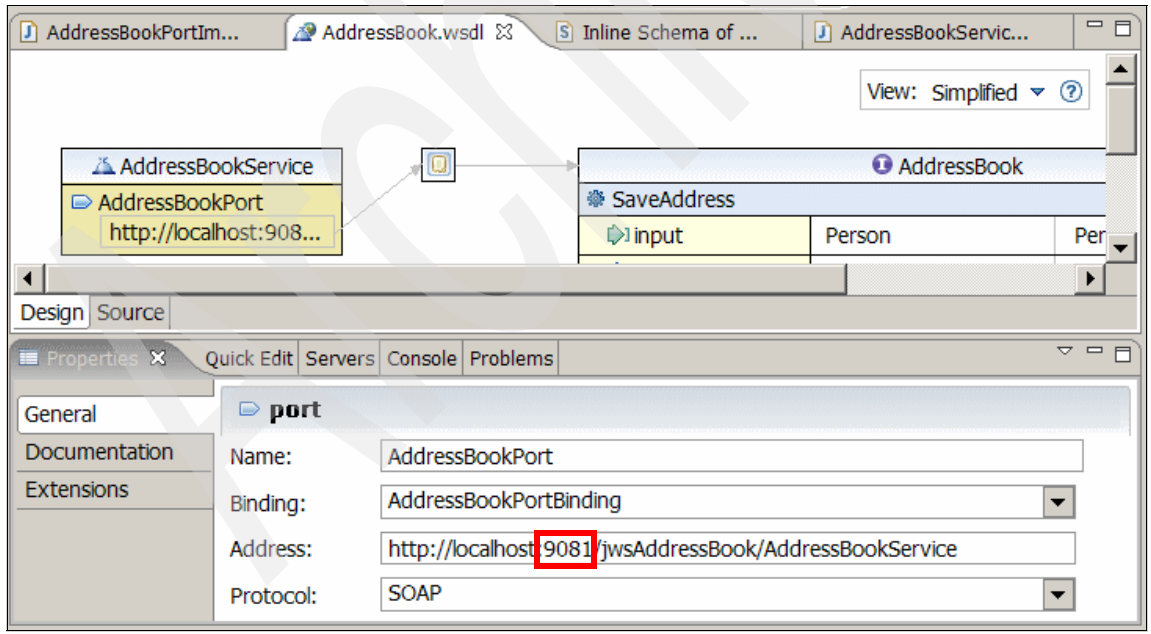


Figure 6-71 Change the SOAP address to 9081

[9] The SOAP transport port is also available in the Administrator Console.

8. Open up the project explorer and right-click TestClient.jsp → **Run As** → **Run on Server** (Figure 6-72).

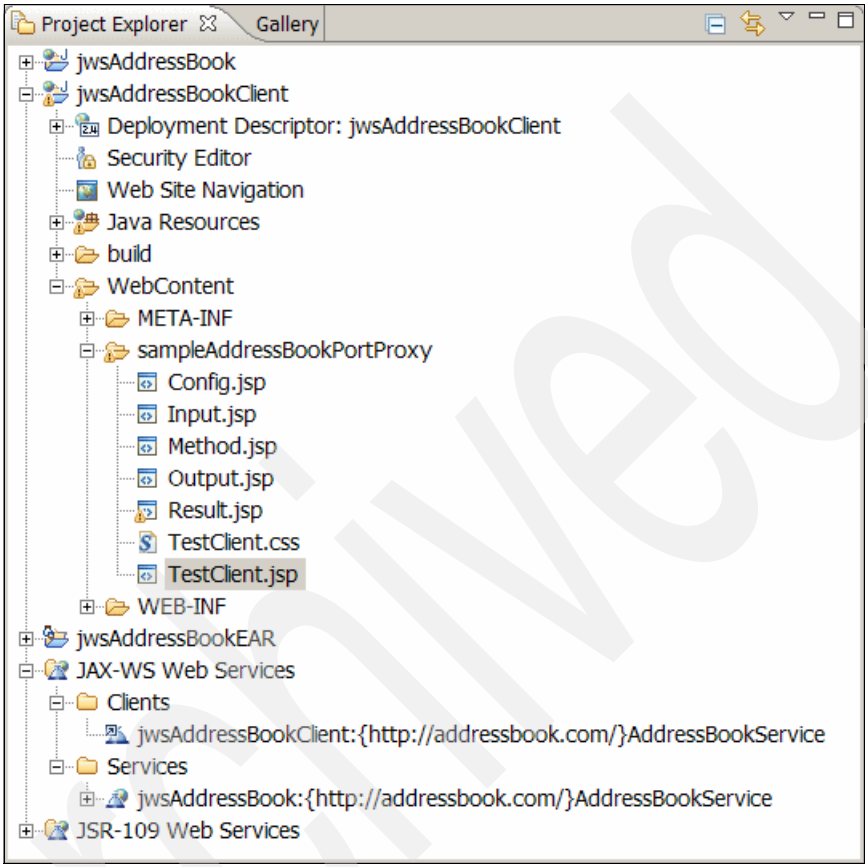


Figure 6-72 Run the TestClient.jsp on the server

This step triggers the deployment of the Web service, which is defined in the same project and is a known dependency. You are asked to define a new server for the deployment.

9. Select the server that you added as a test environment (Figure 6-73). Setting it as the default eliminates this step in future deployments.

10. Either select **Finish** to complete the deployment, or **Next** through the remaining steps of the deployment without making any changes.

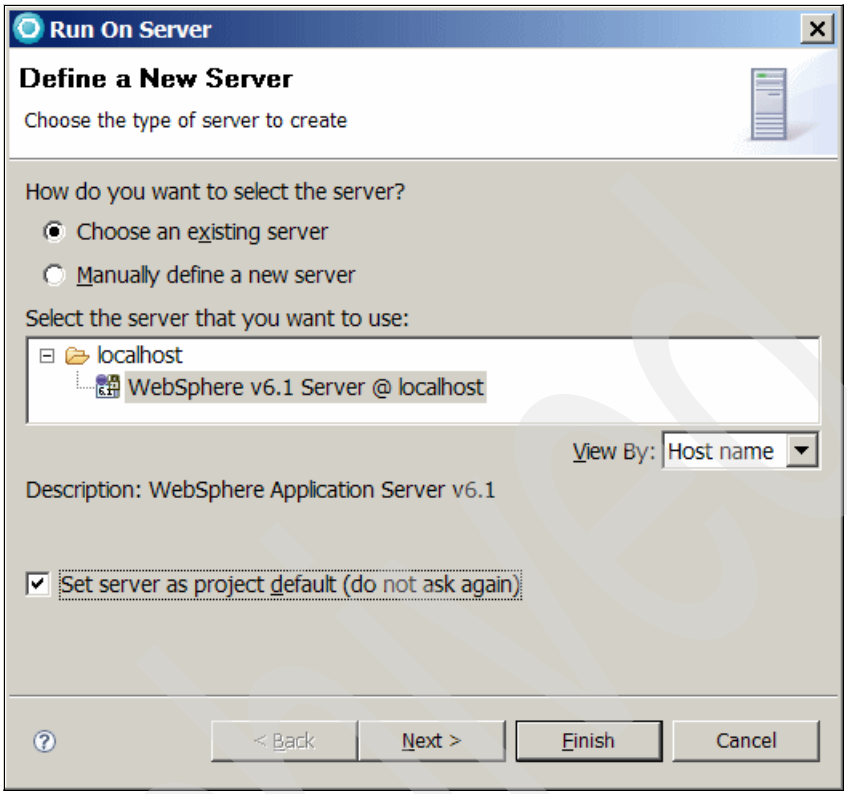


Figure 6-73 Define the server for the JAX-WS test

Test the service

The test client launches as a new view, which you use to test the service:

1. On the TestClient.jsp panel, select the saveAddress method, and enter at least a name and a city (Figure 6-74) → **Invoke**.

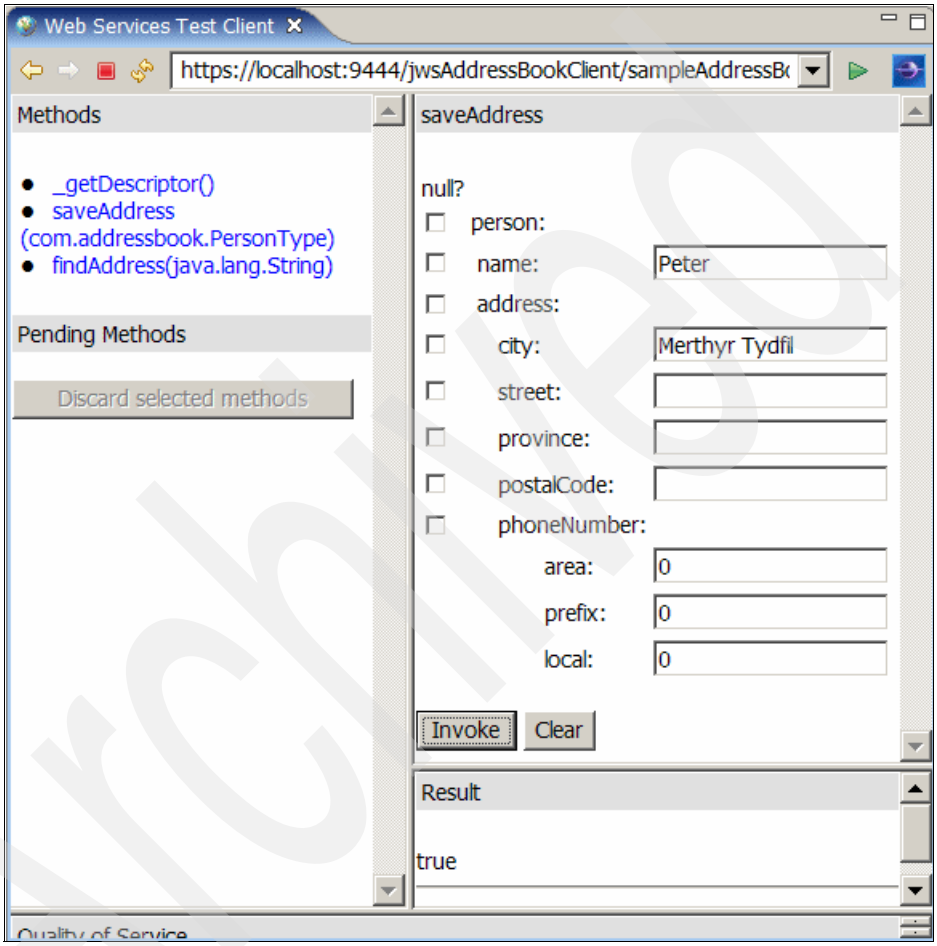


Figure 6-74 Invoke saveAddress method

2. Then select the findAddress method and type the same name again (Figure 6-75).

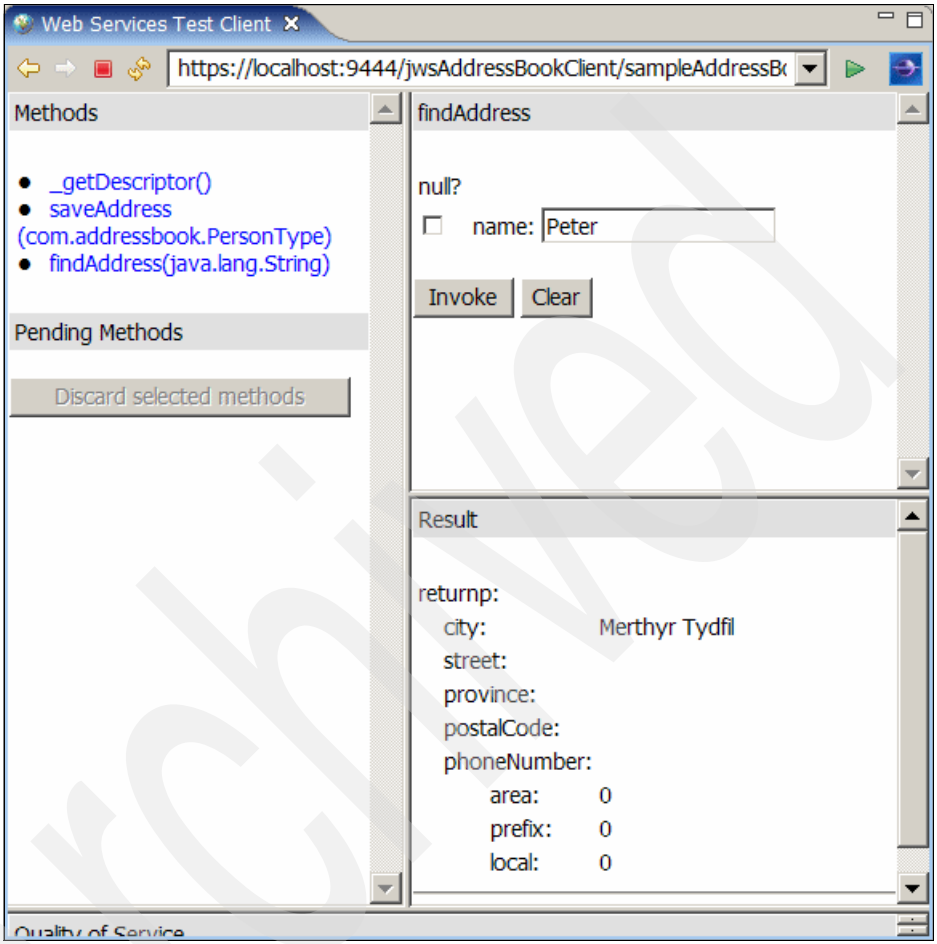


Figure 6-75 Invoke findAddress method

3. To verify the behavior of the Feature pack for Web services with the RAMP profile, repeat this test by importing the WebSphere JAX-WS Address book RAMP Web service sample.

6.4 Installation roadmap for z/OS

The installation and configuration roadmap for the Feature pack for Web services on z/OS follows essentially the same path as described in 6.2, “Installation roadmap for Windows” on page 81.

The major difference is in how the feature pack is delivered and installed, and the details concerning configuring the tools environment to run the Profile Management Tool to augment or create deployment manager nodes and to create application server nodes that support the Feature pack for Web services. But essentially, once the feature pack code is installed, configuration of application servers to use the feature pack follows the same model as on the distributed platforms.

The installation of the feature pack is kept separate from the installation of the application server itself. The feature pack is installed into a different place (a different mount point) in the hierarchical file system (HFS) on z/OS from the one that is used to install WebSphere Application Server for z/OS to make it easier to manage the server and feature packs separately. The new mount point is used for all feature packs for WebSphere Application Server for z/OS and is called the Optional Materials mount point. The initial installation steps are illustrated in Figure 6-76.

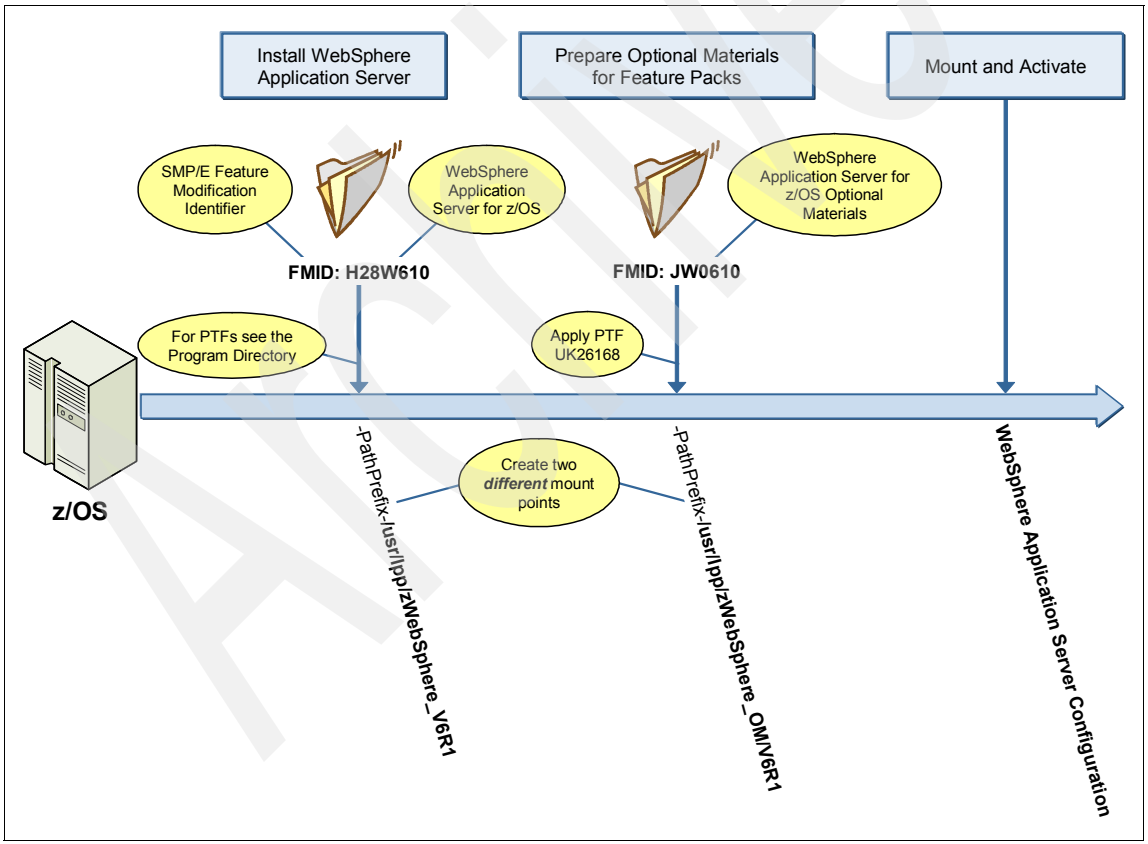


Figure 6-76 Part 1: Install WebSphere Application Server and prepare optional materials

The next steps are to install the feature pack and necessary maintenance, and then to configure the application server to use the feature pack — see Figure 6-77.

The application server is not configured to use the feature pack using ISPF tools running on z/OS. Instead the same Profile Management Tool that is used for distributed platforms is also used to manage the configuration of application servers on z/OS to use the feature pack.

In using the PMT rather than ISPF to configure application server nodes, the Feature pack for Web services is little different from base WebSphere Application Server for z/OS. The only difference is in one of the configuration panels, and one of the generated jobs (BBO*HFSA) takes longer than before.

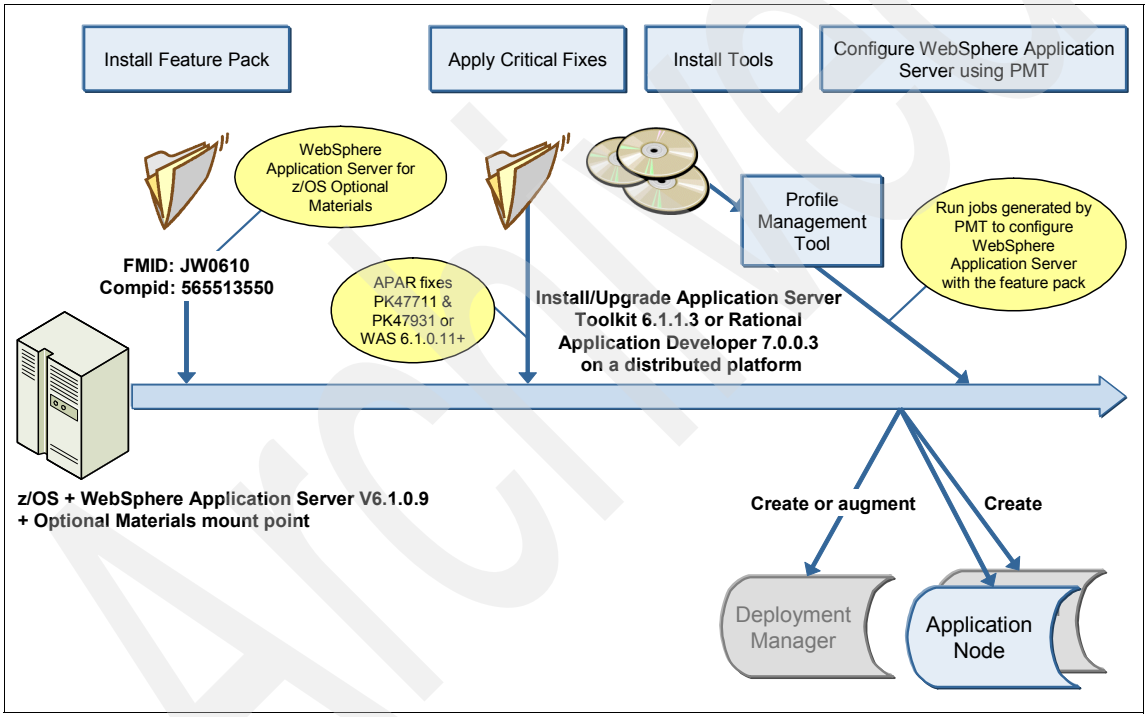


Figure 6-77 Part 2: Install Feature pack, fixes and configure

The Profile Management Tool is launched part of the Application Server Toolkit or Rational Application Developer environment (Figure 6-78). There is some additional configuration of the z/OS server profile in the tool to customize the jobs that the tool creates to execute the configuration on z/OS, which is described in 6.4.2, “Profiles” on page 148.

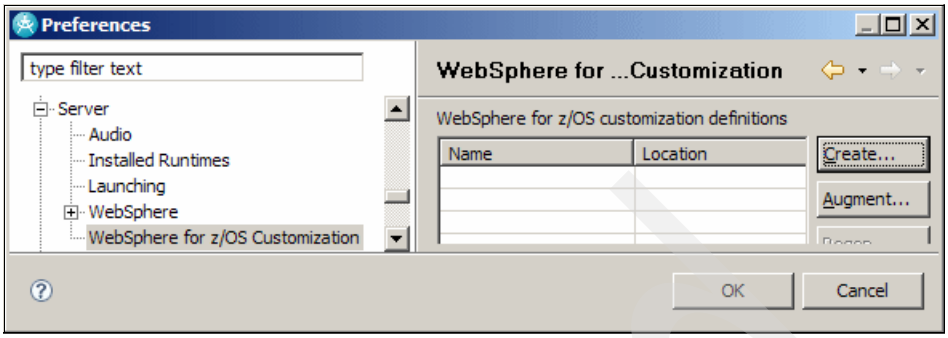


Figure 6-78 z/OS customization (PMT tool embedded in Rational Application Developer)

Subsequent installation of fixes uses SMP/E on z/OS in the expected manner, and the Profile Management Tool[10] continues to be used to configure application server nodes.

6.4.1 Installation

For installation and configuration of WebSphere Application Server for z/OS v6.1, consult its program directory, to be found at:

`ftp://ftp.software.ibm.com/software/webserver/appserv/library/v61/BOS5G111.PDF`

You will also find Chapter 1, *"WebSphere Application Server Network Deployment configuration"*, of the Redpaper, *"WebSphere Process Server for z/OS: Configuring a Network Deployment Environment"*, useful. It was published in February 2008 and can be found at:

`http://www.redbooks.ibm.com/redpapers/pdfs/redp4388.pdf`

You can obtain the Feature pack for Web services from:

`http://www-1.ibm.com/support/docview.wss?rs=404&uid=swg27009131`

This page directs you to the appropriate download depending on the fix level of WebSphere Application Server for z/OS, and also any necessary prerequisites — such as to create the Optional Materials mount point.

A very useful source of extra information about installing and configuring the feature pack is to be found in the white paper by Don Bagwell from the Washington System Center called, simply, *"Feature pack for Web services"*. It is available at:

`http://www-03.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/WP101084`

[10] Abbreviated to zPMT on z/OS

6.4.2 Profiles

The capabilities and restrictions for WebSphere Application Server profiles on z/OS are identical to distributed environments; see 6.2.2, “Profiles” on page 83.

Unlike the distributed environment, there is no separately packaged Profile Management Tool in the z/OS environment. Instead you use the server customization wizards in Rational Application Developer or Application Server Toolkit. We could have used the server customization wizards instead of launching the PMT tool to configure our Windows Server 2003 WebSphere Application Server installation too.

if you follow the instructions described in 6.3.6, “Install Rational Application Developer” on page 117, then you can customize WebSphere Application Server for z/OS by opening the WebSphere for z/OS customization wizard and clicking **Window** → **Preferences** → **Server** → **WebSphere for z/OS customization** ((Figure 6-78 on page 147)**.** We show you an example configuration of WebSphere Application Server for z/OS to use the Feature pack for Web services in 6.5, “z/OS configuration example” on page 149.

Alternatively, you can use the Application Server Toolkit 6.1.1, which is included with your WebSphere Application Server for z/OS license. You have to upgrade the shipped Application Server Toolkit to 6.1.1.3 using the Rational Product Updater. If you are unsure how to do this, details are given in the Washington Systems Center Feature pack for Web services white paper referred to in 6.4.1, “Installation” on page 147.

6.4.3 Tools

There are no differences in the tools used to create and deploy JAX-WS applications on z/OS. Refer to 6.2.3, “Tools” on page 85.

6.5 z/OS configuration example

We start the configuration using the WebSphere Application Server for z/OS customization wizard shown in Figure 6-78 on page 147.

1. Click **Create...**

   The wizard responds with some important information for V6.1 (Figure 6-79).

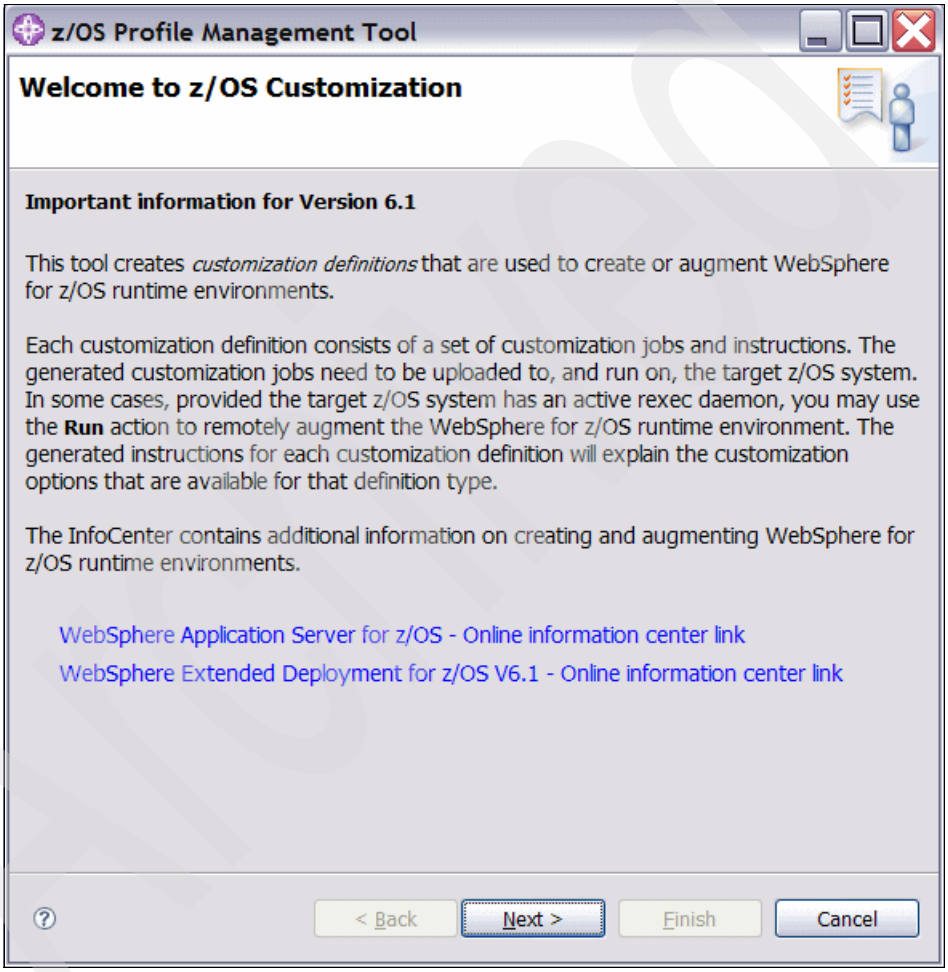


Figure 6-79 z/OS Feature pack for Web services customization - Step 1

2. Read this information, and click **Next**.

The following panel is presented (Figure 6-80).

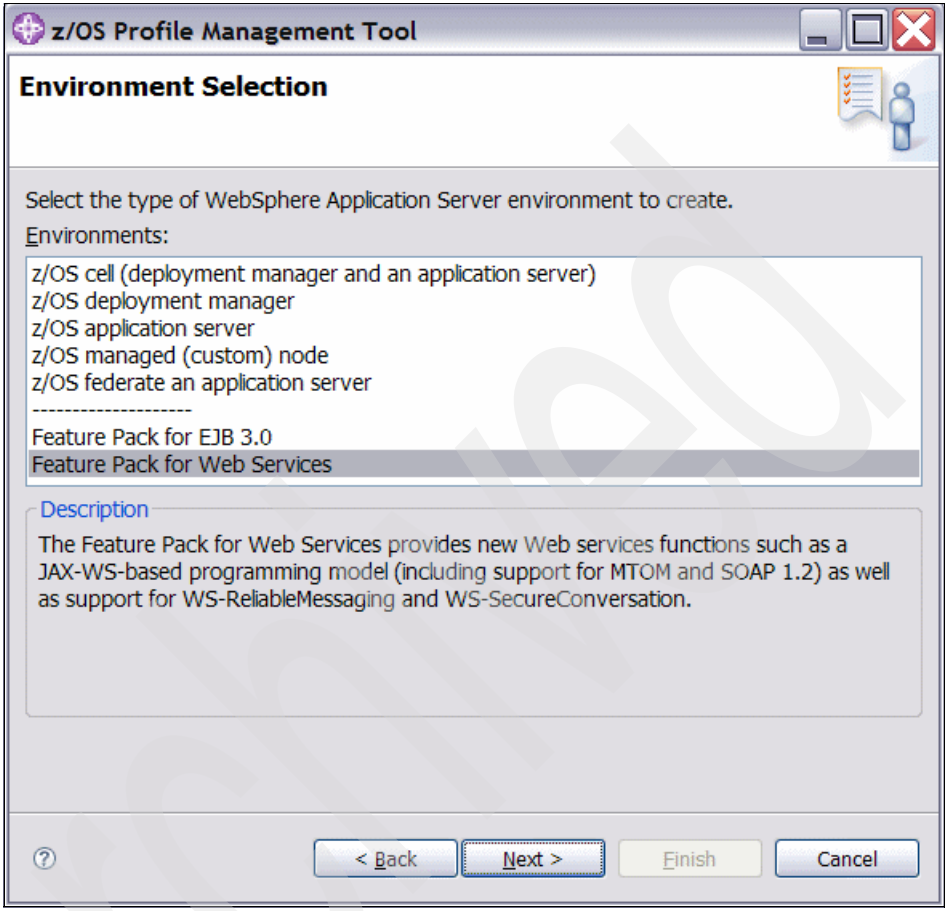


Figure 6-80 Environment selection - 1

3. Select **Feature pack for Web services** → **Next**.

The following panel is presented (Figure 6-81).

Figure 6-81 Environment selection - 2

4. Select **z/OS application server with Feature pack for Web services** → **Next**.

The following panel is presented (Figure 6-82).

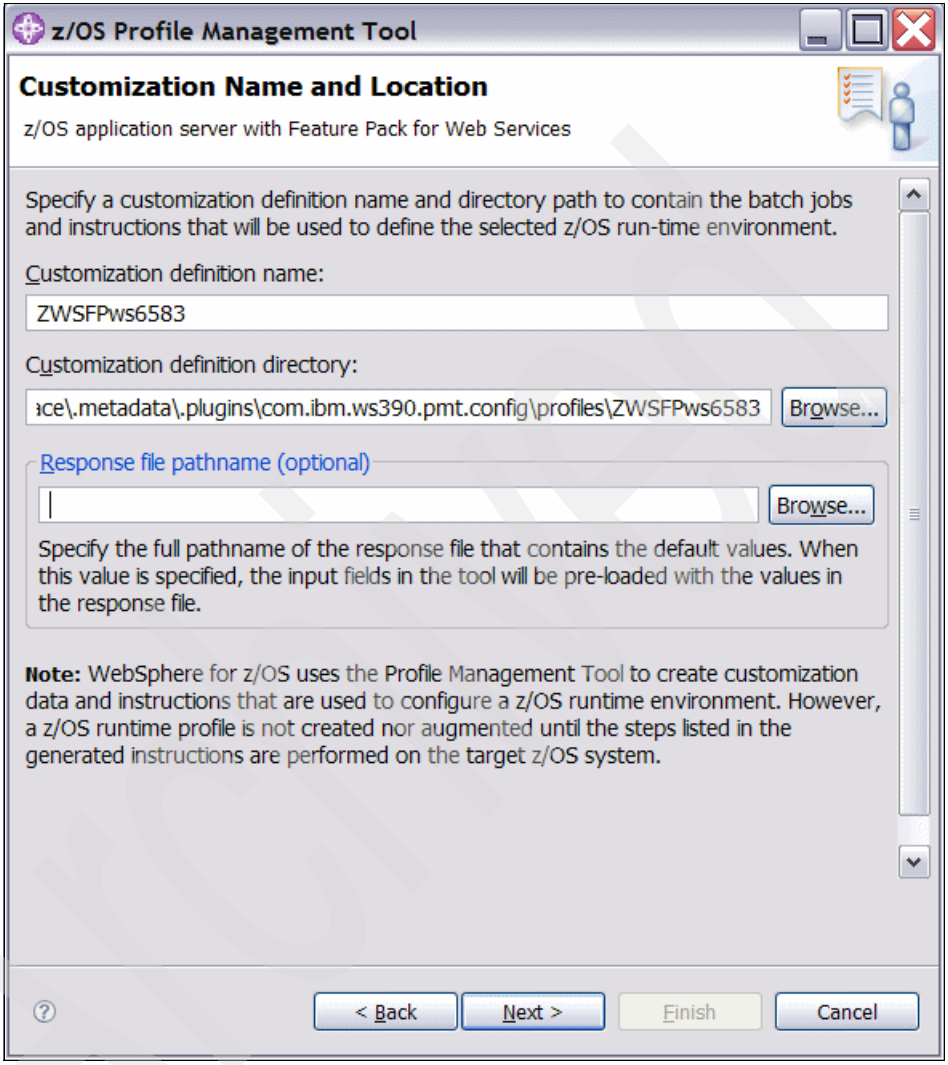


Figure 6-82 Customization Name and Location

5. Specify a location for the generated batch file, and if you have prepared one before, the response file to provide default values for the configuration → **Next**.

The following panel is presented (Figure 6-83).

z/OS Profile Management Tool

Default Values

z/OS application server with Feature Pack for Web Services

Specify options for generating default values for this customization definition.

☑ Set each default GID and UID value to indicate OS security is to assign an unused value

Name and userid defaults

☐ Set default names and userids based on cell, cluster and system identifiers.

Two-character cell identifier: AZ

Two-character cluster identifier: 00

Single-character system identifier: A

Port defaults

☑ Select default port values from the following port range.

Lowest default port number 9140

Highest default port number 9159

The port range must specify at least 20 ports. (The daemon port defaults are not assigned within this range.)

Note: If you specified a response file for setting default values, any default selected on this panel will override the corresponding response file values.

< Back | Next > | Finish | Cancel

Figure 6-83 Default values

6. Set any default values you want applied by the wizard → **Next**.

The following panel is presented (Figure 6-84).

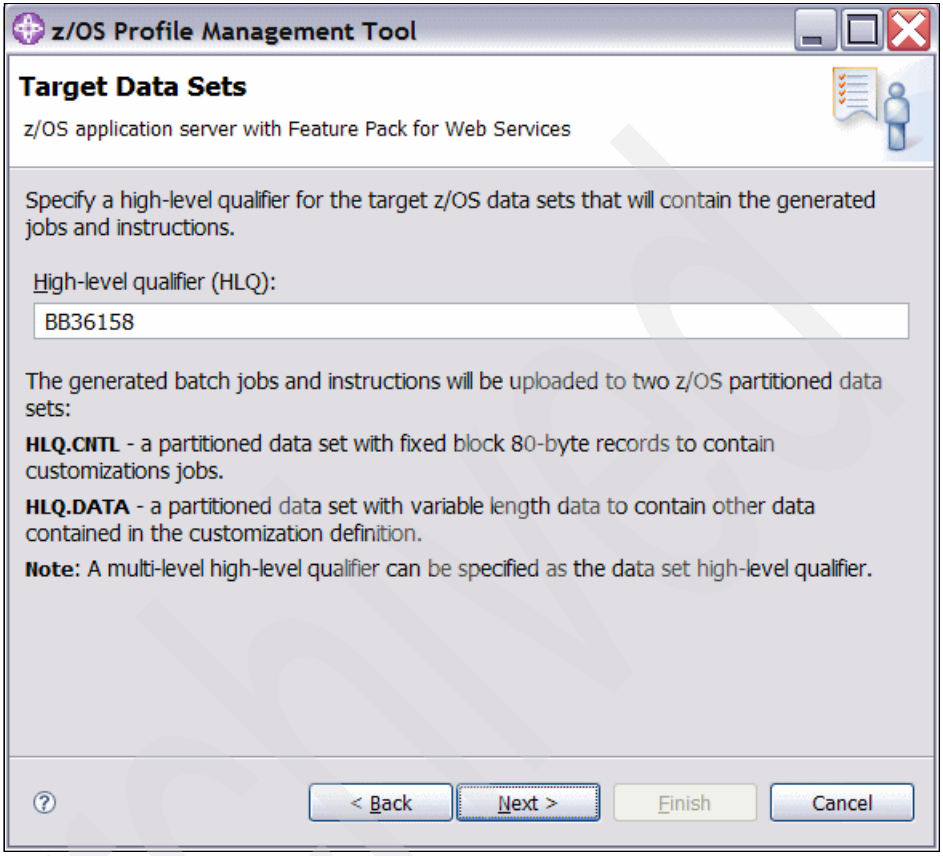


Figure 6-84 Target Data Sets

7. Provide a High Level Qualifier name for the target z/OS datasets that will contain the generated jobs and instructions → **Next**.

The following panel is presented (Figure 6-85).

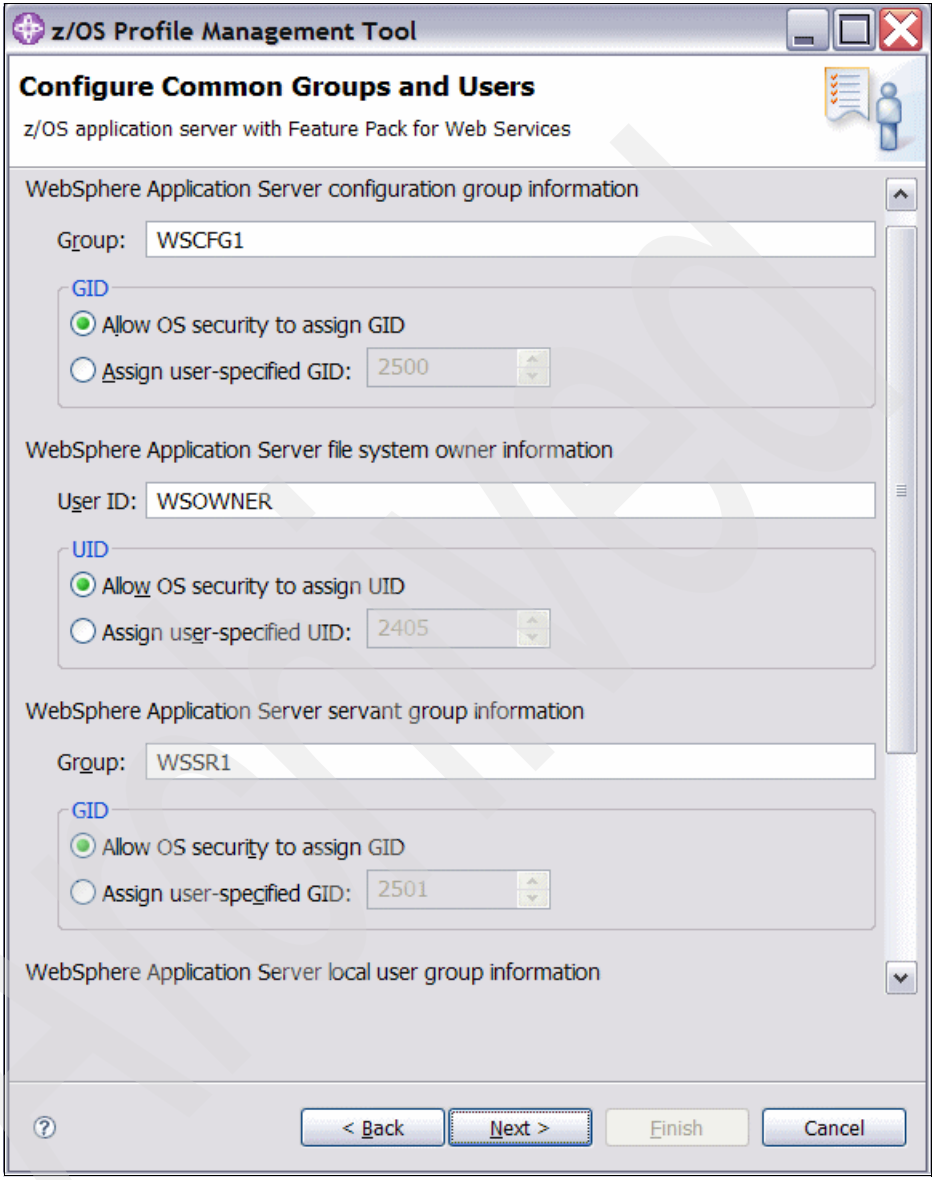


Figure 6-85 Configure Common Groups and Users

8. Provide the common group and user information → **Next**.

The following panel is presented (Figure 6-86).

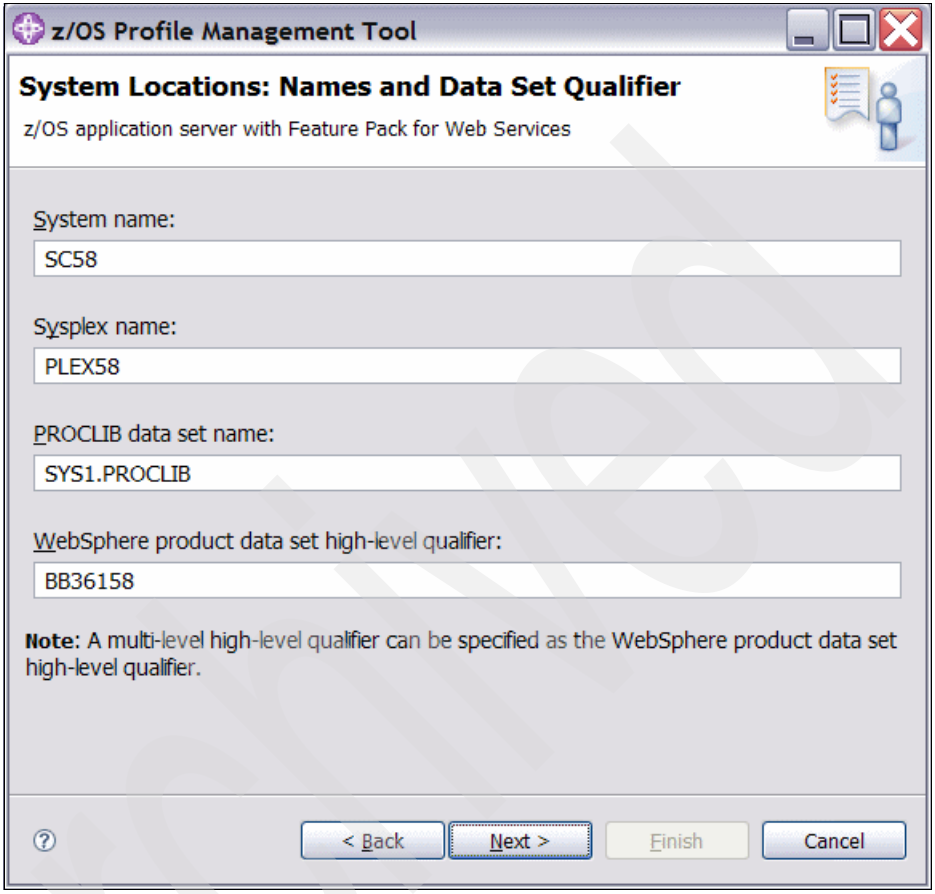


Figure 6-86 Names and Data Set Qualifier

9. Provide the names for the data set qualifier → **Next**.

The following panel is presented (Figure 6-87).

z/OS Profile Management Tool

System Locations: Product Data Sets

z/OS application server with Feature Pack for Web Services

SBBOLPA data set name or catalog alias:

BB36158.SBBOLPA

SBBOEXEC data set name:

BB36158.SBBOEXEC

SBBOMSG data set name:

BB36158.SBBOMSG

SBBOLOAD data set name or catalog alias:

BB36158.SBBOLOAD

SBBGLOAD data set name or catalog alias:

BB36158.SBBGLOAD

SBBOLD2 data set name or catalog alias:

BB36158.SBBOLD2

☑ Run WebSphere Application Server from STEPLIB

Note: Refer to the information center for information on how catalog aliases can be used to isolate service environments.

Online information center link

< Back | Next > | Finish | Cancel

Figure 6-87 Product Data Set Locations

10. Provide the locations of the product data sets from the installation → **Next**.

The following panel is presented (Figure 6-88).

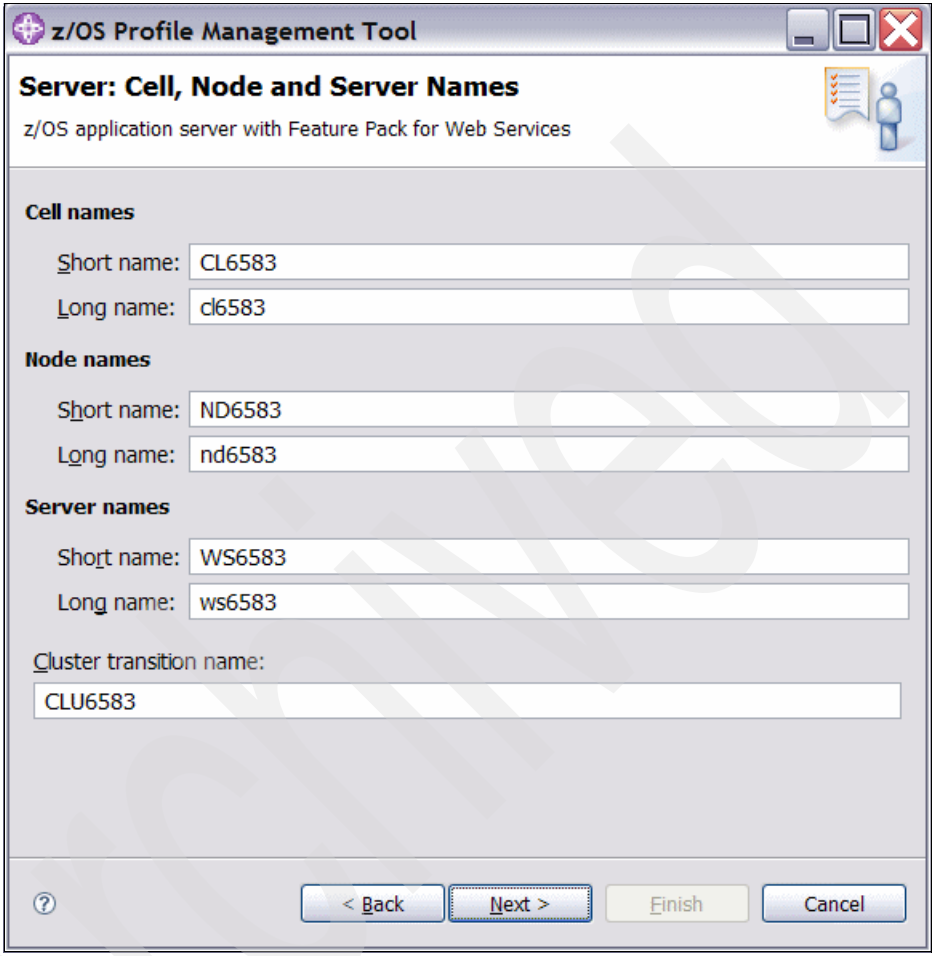


Figure 6-88 Cell, Node and Server names

11. Provide the cell, node, and server names → **Next.**

The following panel is presented (Figure 6-89).

z/OS Profile Management Tool

System Environment: Configuration File System

z/OS application server with Feature Pack for Web Services

Configuration file system

Mount point:

/WebSphereBS3/V6R1

Name:

OMVS.WAS61.BS3.CONFIG.HFS

Volume, or '*' for SMS:

TARHFB

Primary allocation in cylinders: 420

Secondary allocation in cylinders: 100

File system type:

Hierarchical File System (HFS)

zSeries File System (ZFS)

Product file system

Product file system directory (or pathname of intermediate symbolic link):

/usr/lpp/zWebSphereBS3/V6R1

< Back | Next > | Finish | Cancel

Figure 6-89 Define the Configuration File System

12. Identify the configuration file system being used → **Next**.

The following panel is presented (Figure 6-90).

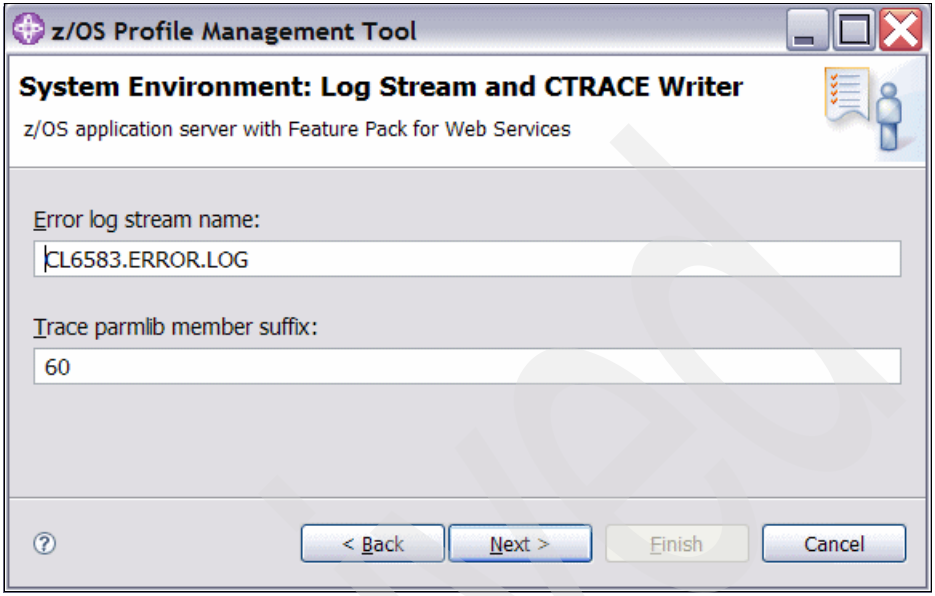


Figure 6-90 Log stream and CTRACE Writer

13. Define the name and suffix for the Log Stream and CTRACE writer → **Next**.

The following panel is presented (Figure 6-91).

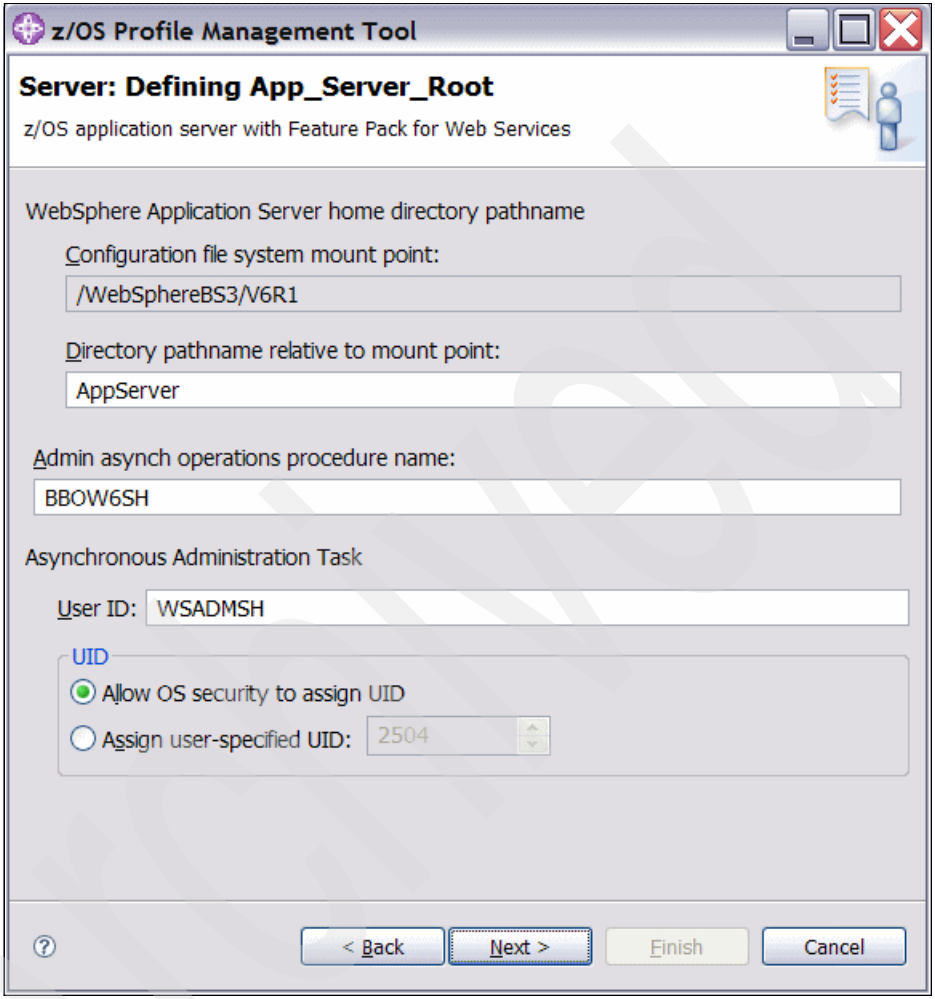


Figure 6-91 \Define App_Server_Root

14. Identify the WebSphere Application Server for z/OS location and Administrator userid → **Next**.

The following panel is presented (Figure 6-92).

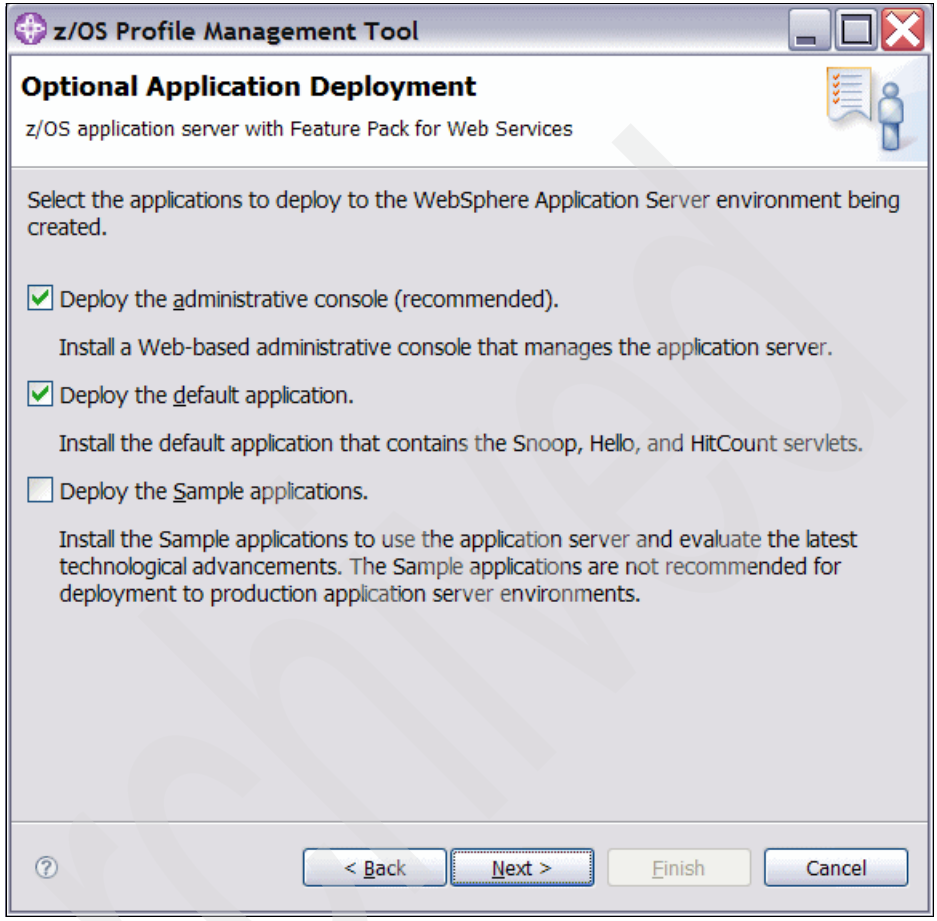


Figure 6-92 Optional Application Deployment

15. Select the optionally deployed components → **Next**.

The following panel is presented (Figure 6-93).

z/OS Profile Management Tool

Server: Process Definitions

z/OS application server with Feature Pack for Web Services

Controller process

Job name: WS6583

Procedure name: WS6583C

User ID: ASCR1

UID

Allow OS security to assign UID

Assign user-specified UID: 2431

Controller adjunct process

Job name: WS6583A

Procedure name: WS6583A

User ID: ASCRA1

UID

Allow OS security to assign UID

Assign user-specified UID: 2433

Servant process

Job name: WS6583S

Procedure name: WS6583S

< Back | Next > | Finish | Cancel

Figure 6-93 Provide Process Definitions

16. Provide the process definitions for the configuration → **Next**.

The following panel is presented (Figure 6-94).

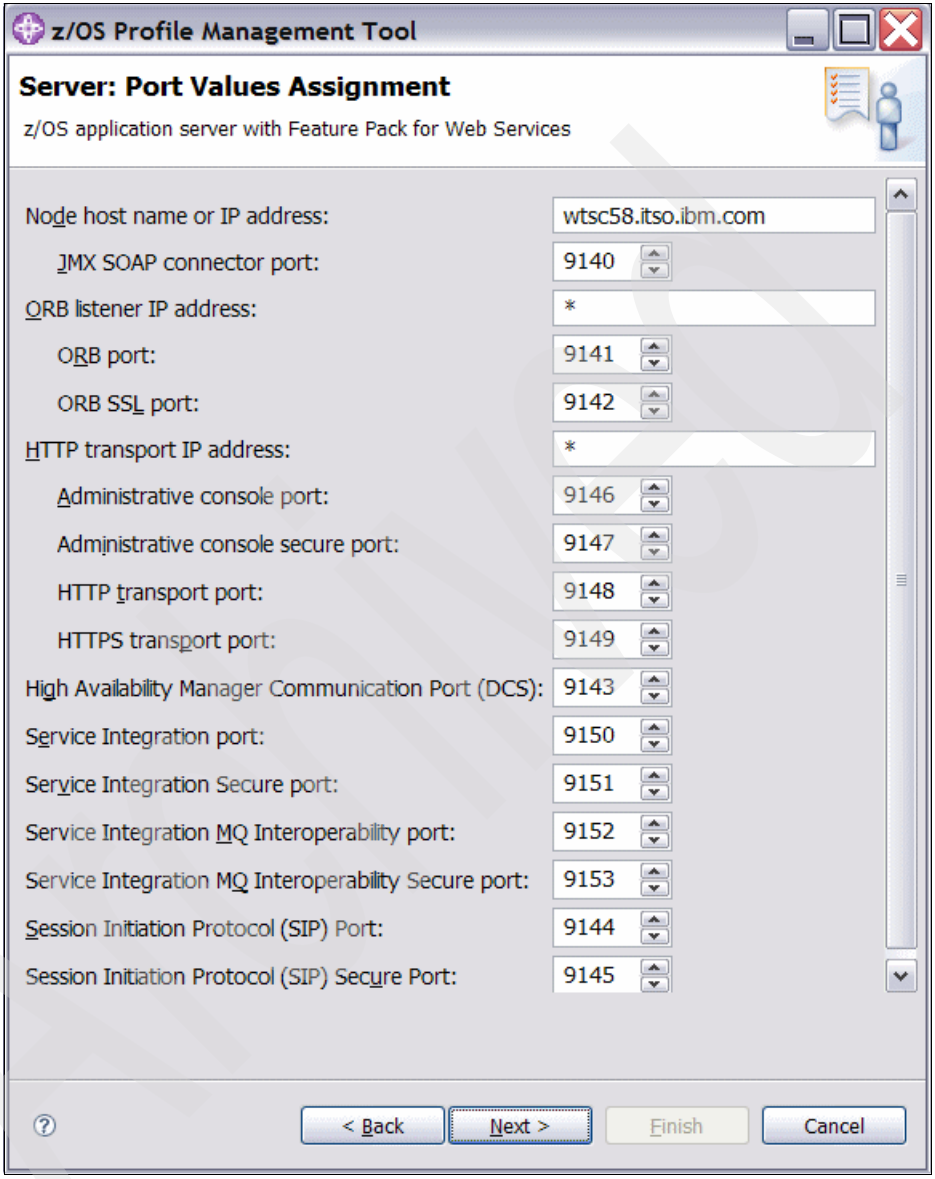


Figure 6-94 Confirm port value assignments

17. Confirm the port values assignments → **Next**.

The following panel is presented (Figure 6-95).

z/OS Profile Management Tool

Server: Location Service Daemon Definitions

z/OS application server with Feature Pack for Web Services

Specify the location service daemon settings. The location service daemon is the initial point of client contact in WebSphere Application Server for z/OS. The server contains the CORBA-based location service agent, which places sessions in a cell. All RMI/IIOP IORs (for example, for enterprise beans) establish connections to the location service daemon first, then forward them to the target application server.

Daemon home directory: /WebSphereBS3/V6R1/Daemon

Daemon job name: WS6583D

Procedure name: WS6583D

IP name: wtsc58.itso.ibm.com

Listen IP: *

Port: 9154

SSL port: 9155

User ID: WSDMNCR1

UID

Allow OS security to assign UID

Assign user-specified UID: 2411

Register daemon with WLM DNS

< Back | Next > | Finish | Cancel

Figure 6-95 Specify the location of the service Daemon

18. Specify the location of the service daemon → **Next**.

The following panel is presented (Figure 6-96).

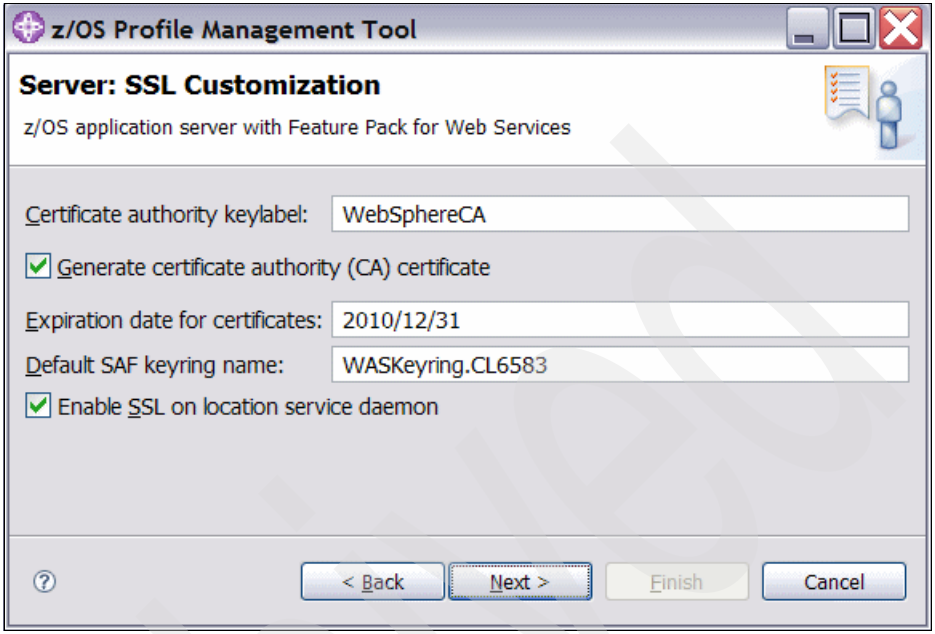


Figure 6-96 SSL Customization

19. Provide the SSL parameters → **Next**.

The following panel is presented (Figure 6-97).

Figure 6-97 Security selection

20. Select the security configuration → **Next**.

The following panel is presented (Figure 6-98).

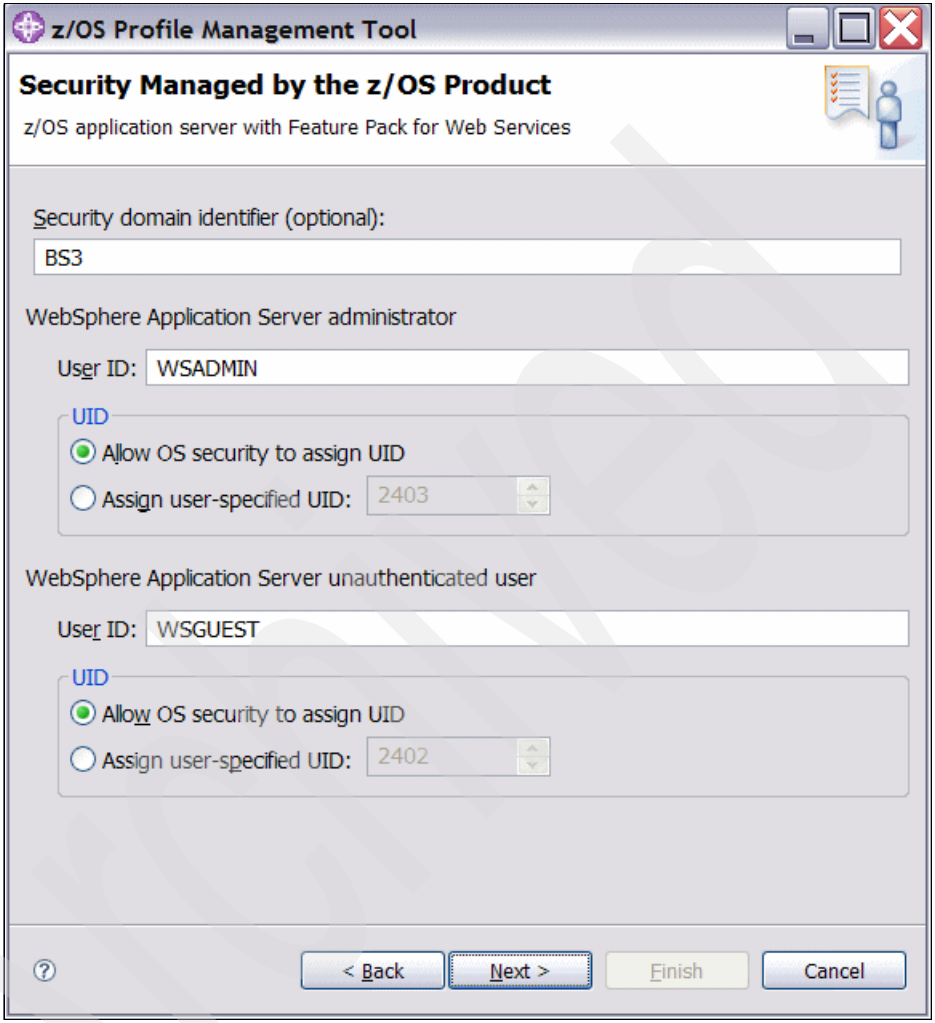


Figure 6-98 Security parameters

21. Specify the z/OS security parameters → **Next**.

The following panel is presented (Figure 6-99).

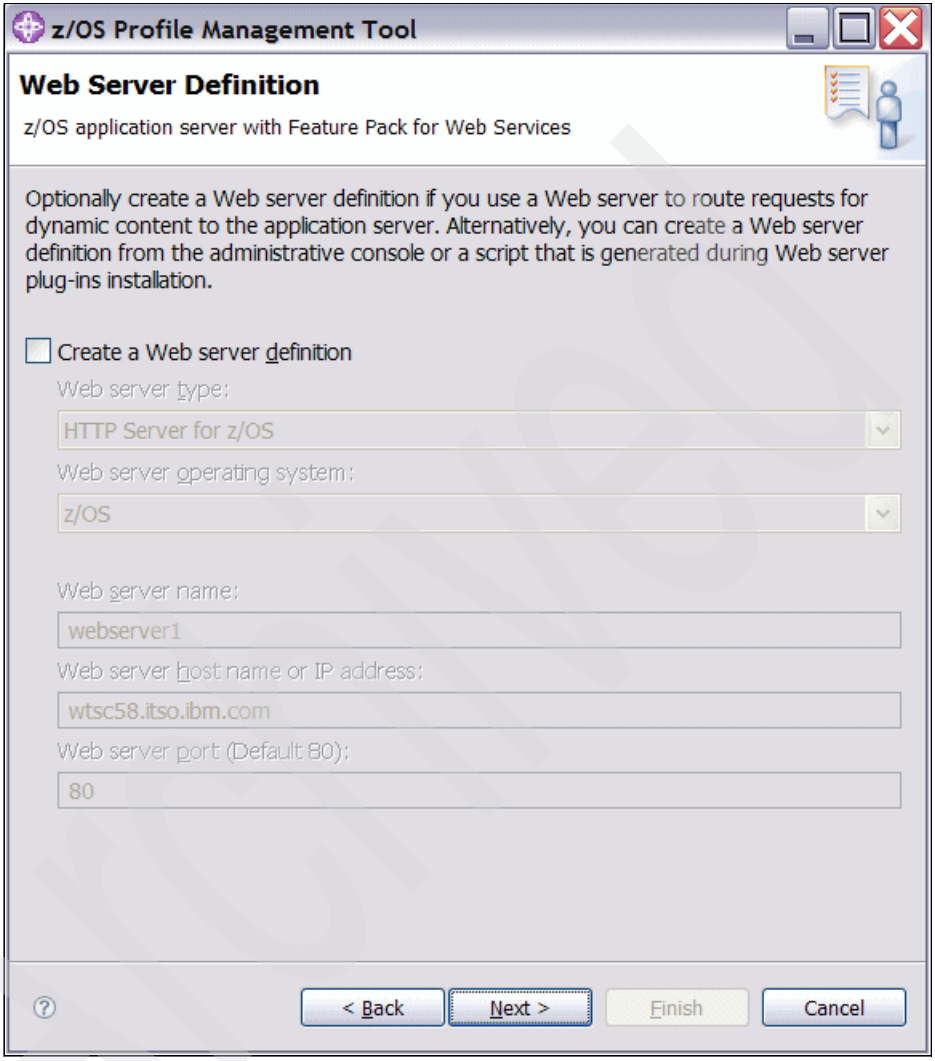


Figure 6-99 Omit Web server definition

22. We do not bother with a Web server → **Next**.

The following panel is presented (Figure 6-100).

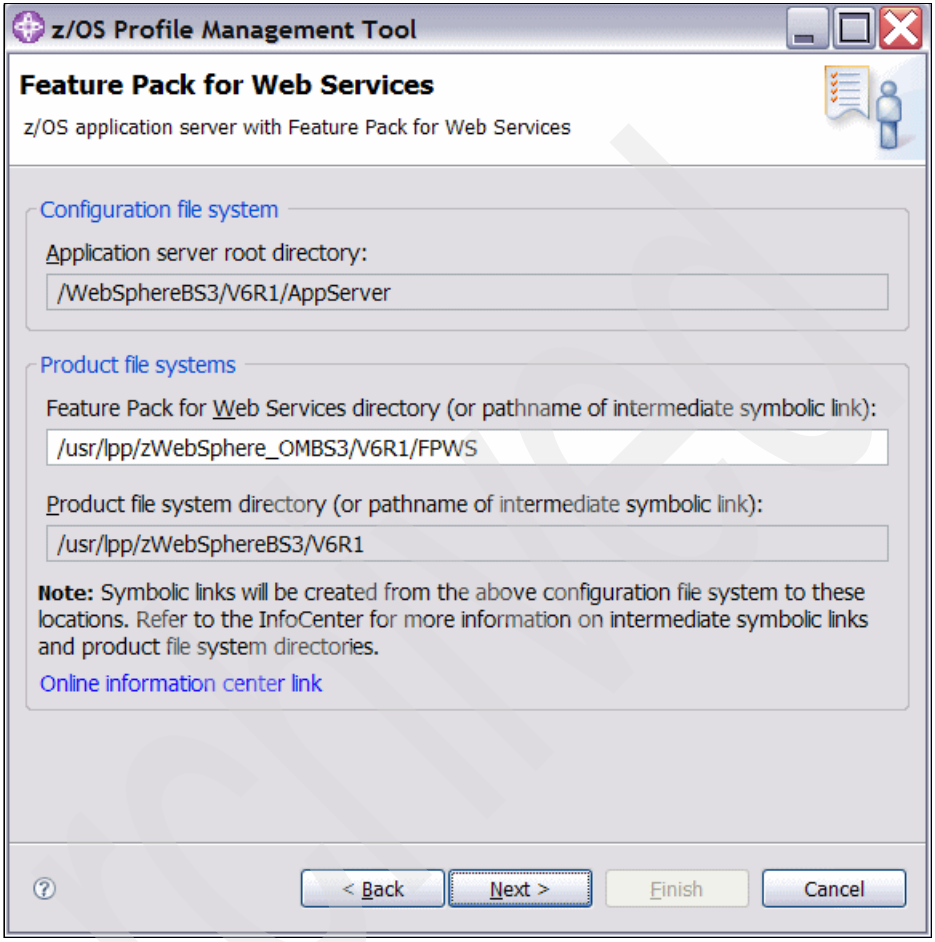


Figure 6-100 Identify the feature pack directory

23. Specify the link to the feature pack installation directory → **Next**.

The following panel is presented (Figure 6-101).

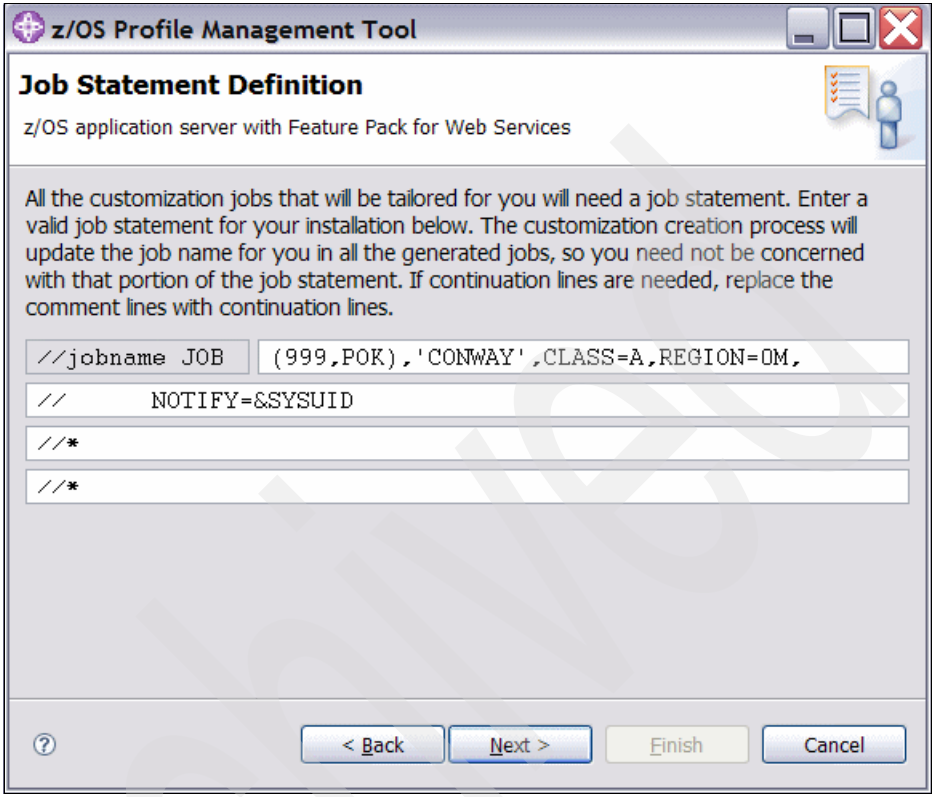


Figure 6-101 Provide a Job statement

24. Provide a job statement → **Next**.

The following panel is presented (Figure 6-102).

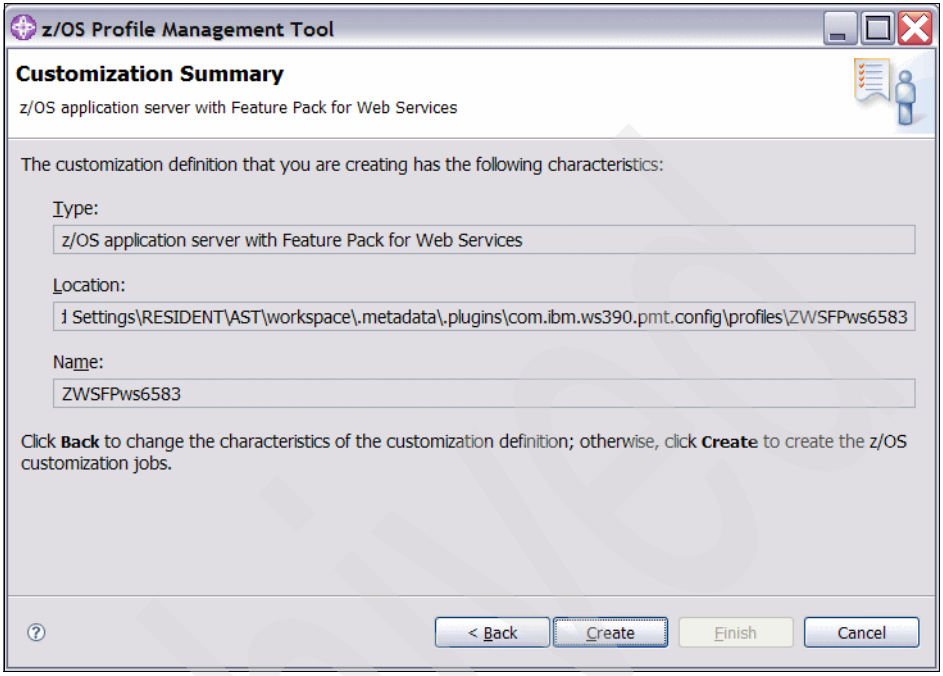


Figure 6-102 Summary panel

25. Review the customization summary → **Next** (Figure 6-103).

There is a little delay while the customization is generated.

The following panel is presented (Figure 6-103).

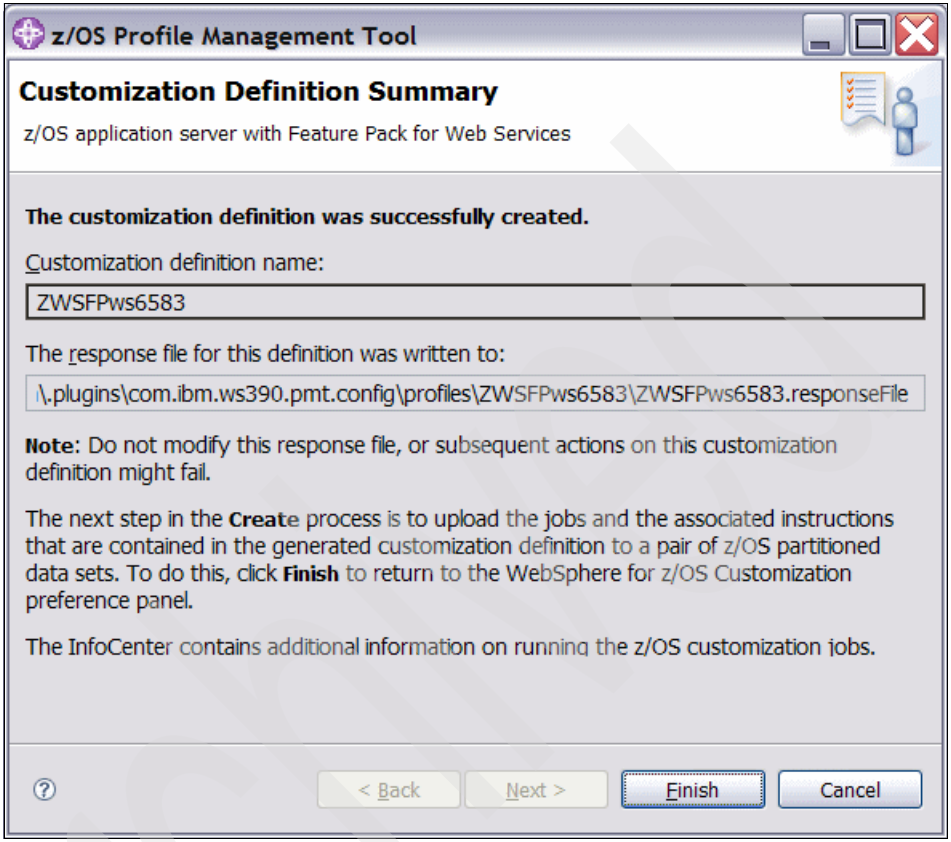


Figure 6-103 Customization creation complete

26. Click **Finish** on this panel to submit the job and create the profile.

6.6 Summary

In this chapter we reviewed the installation and configuration of the Feature pack for Web services, and went through a detailed example of installing and configuring the feature pack in WebSphere Application Server v6.1 and Rational Application Developer v7.0, and then configuring and verifying the installation using the sample programs provided with the feature pack.

7

JAX-WS programming model

JAX-WS 2.0 is the latest incarnation of a mapping between WSDL and Java. The primary reason for introducing JAX-WS in the feature pack is because it provides an asynchronous client model, which is necessary for some reliable messaging scenarios, but that is only a small part of JAX-WS.

This chapter goes into some of the details of building JAX-WS Web services. We first walk you through the feature pack tools to build a couple of Web services. Then we use those services to explain the functionality that JAX-WS provides.

However, this book is not a handbook. It is not a reference manual. It does not show you every feature of JAX-WS and every option in the tools. You can go to the Web services feature pack InfoCenter for those details[1]. In this chapter of the book, we attempt to give you a basic understanding of what you can do with the new JAX-WS Web services and get you started with some simple examples.

Note: In this and subsequent chapters, the square box icon is used to indicate steps that you can follow to reproduce the examples in the text.

[1] For the InfoCenter, see: http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/index.jsp?topic=/com.ibm.websphere.wsfep.multiplatform.doc/info/ae/htmlnav_fepnew.html

7.1 Generate a Web service and client

Before we describe the new features of the Feature pack for Web services, we need an example. We start with a WSDL that contains a simple echo Web service. It has a request/response operation called echoOperation that also raises a fault.

The WSDL for this chapter is in the additional materials for this book (see Appendix B, “Additional material” on page 561 for instructions on how to obtain the additional materials .zip file). When you unzip the zip file, there is a directory tree, the root of which is a directory called `ITSO7618`. Within the directory tree, there is a project interchange file named `Chapter8Staging.PI.zip`.

- Import this file into your Rational Application Developer workspace as a project interchange file. Select and import the project named `staging`. The steps in this chapter assume that you are in the J2EE perspective. See Figure 7-1.

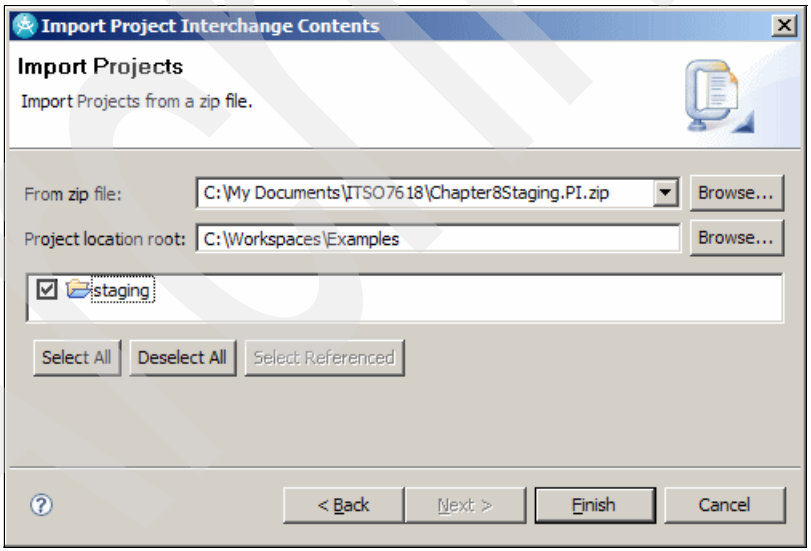


Figure 7-1 Import staging from the additional materials

There is another project interchange file called `Chapter8Solutions.PI.zip`. This file contains the services and clients that we create and walk through in this chapter.

7.1.1 Generate the echo service

This Web service example is simple. It is merely a dynamic Web project.

- Select the Echo.wsdl file. It is in the staging project: **staging** → **Echo.wsdl.**
- Right-click Echo.wsdl → **Web services** → **Generate Java bean skeleton**. See Figure 7-2.

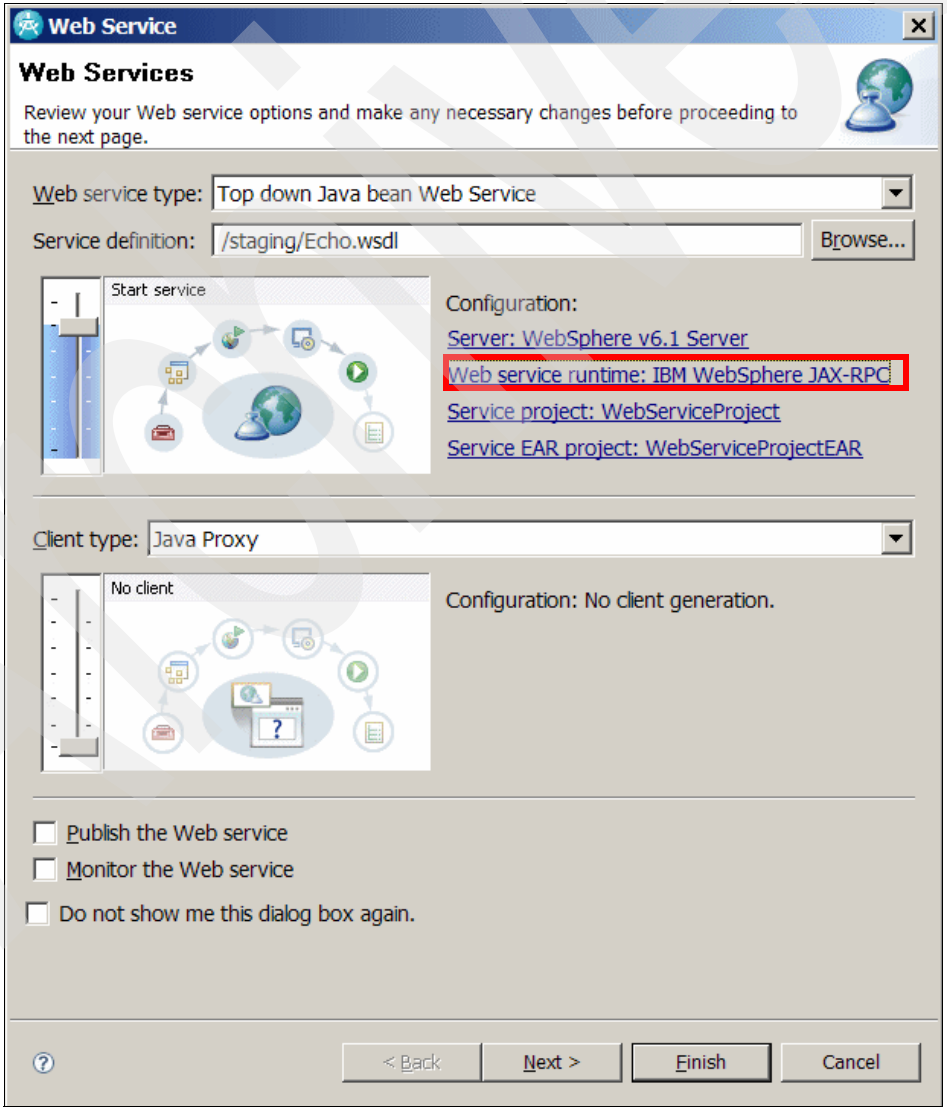


Figure 7-2 Create Java bean skeleton

Notice that the configuration section's second entry tells us that the Web service runtime is JAX-RPC. The feature pack still supports the JAX-RPC runtime. Both the JAX-RPC and the new JAX-WS engines exist within the WebSphere Application Server. In Rational Application Developer, JAX-RPC is the default, as you can see here. But we want to program for the JAX-WS runtime.

- Click **Web service runtime: IBM WebSphere JAX-RPC** → Choose **IBM WebSphere JAX-WS** from the options → **OK.**

  The wizard defaults to create your project in the same directory as the WSDL resides in, but that is not what we want. You must have already created the project to use as the service project.

- Click **Service project: staging** → **EchoService** to select the Dynamic Web project EchoService as the service project. The wizard will fill in the other fields appropriately (Figure 7-3) → **OK**.
  - Service Project type: Dynamic Web Project
  - Service EAR project: EchoServiceEAR

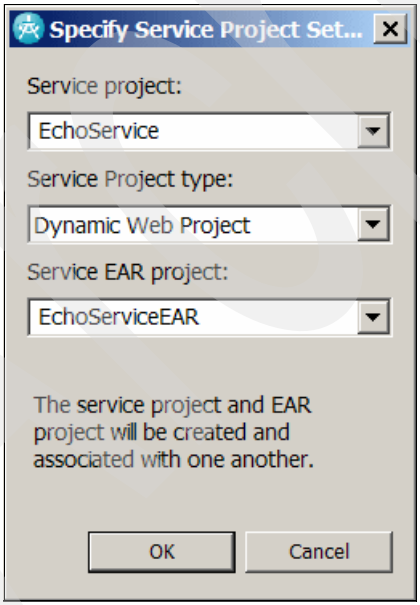


Figure 7-3 Create Dynamic service project

You can see that this wizard allows us to build a client as well as the server skeleton. But we are starting with a WSDL from the staging area, which is not the real WSDL of the service (the address is not the real address) — the real WSDL will not exist until you build the service — you will defer building the client until later.

- **Next** → to the Web service options panel (Figure 7-4).

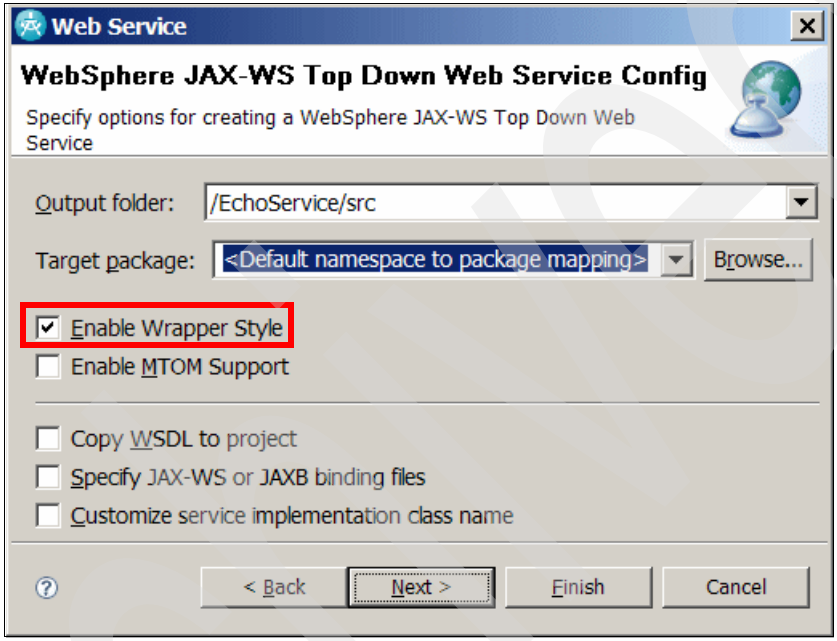


Figure 7-4 Java bean skeleton options panel

- Ensure that **Enable Wrapper Style** is checked.

This option is on by default. What it means is that, assuming you have a document/literal wrapped WSDL[2], the wrapper element (that is, the element whose name is the same as the operation) is peeled apart and each of its content elements becomes a parameter to the generated Java method signature. For example, suppose that your wrapper element is:

```
<xsd:element name="operation">
   <xsd:sequence>
      <xsd:element name="field1" type="xsd:string"/>
      <xsd:element name="field2" type="xsd:int"/>
   <xsd:sequence>
</xsd:element>
```

The generated Java method's signature is:

```
public void operation(String field1, int field2);
```

[2] For information about the document/literal wrapped pattern, see: http://www.ibm.com/developerworks/webservices/library/ws-whichwsdl/

If you uncheck the **Enable Wrapper Style** box, then the generated Java method's signature becomes:

```
public void operation(Operation parameters);
```

- ☐ Leave **Enable MTOM Support** unchecked.

  JAX-WS allows a Web service receiver to enable or disable MTOM support. See "MTOM" on page 213 for details about MTOM.

- ☐ Leave **Copy WSDL to project** unchecked.

  A Web service project need not contain a WSDL file. When you query a Web service for its WSDL (for example, when you type `<serviceURL>?WSDL` into a browser), a SOAP engine constructs its WSDL from the WSDL file contained within it[3]. However, if the Web service contains no WSDL file, the SOAP engine has enough information from the deployed EAR file, in the Java source, to generate the entire WSDL file from scratch. We will generate the WSDL from the service.

- ☐ Leave JAX-WS **or JAXB binding files** unchecked.

  Both JAX-WS and JAXB support additional bindings held in external files that describe additional options for the Java generation tools, such as:

  - Override the default *namespace to package* mapping.
  - Override the default generated file names.
  - Enable the wrapper style.

  There are other customizations available too. When you check this box, the wizard asks you for files containing these bindings.

- ☐ Leave **Customize service implementation class name** unchecked.

  The default name of the skeleton implementation class is based on the port name in the wsdl: `service element <port>Impl.java`. Check this box to override the default name generation.

- ☐ Click **Finish** after accepting the defaults described on this panel. If the test server is not running, the wizard starts it before continuing.

> **Note:** Because the server takes some time to start, it is a good practice to start the server before running the Web services wizard.

[3] It might modify the SOAP address, because a given Web service can be deployed to many places and the address within the contained WSDL might not be accurate for a given deployment.

- Click **Next** to see the decisions you could make to publish the Web service.

When the wizard completes, Rational Application Developer presents the Java bean skeleton. Give it a simple implementation such as the highlighted line in Figure 7-5. (There is a pasteable example of the service in Example 7-7 on page 200 that we use later).

```
package com.ibm.was.wssample.sei.echo;

@javax.jws.WebService(
    endpointInterface =
        "com.ibm.was.wssample.sei.echo.EchoServicePortType",
    targetNamespace = "http://com/ibm/was/wssample/sei/echo/",
    serviceName = "EchoService",
    portName = "EchoServicePort")
public class EchoServicePortImpl {
    public String echoOperation(String echoInput)
            throws Fault Exception {
        return "JAX-WS echo >> " + echoInput;
    }
}
```

Figure 7-5 The EchoService implementation

On saving your work, Rational Application Developer redeploys the application and you have a fully functional Echo Web service.

You can also use the command line tool called wsimport to do everything that this wizard does. See the InfoCenter for details at:

http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/index.jsp?topic=/com.ibm.websphere.wsfep.multiplatform.doc/info/ae/ae/rwbs_wsimport.html

7.1.2 Generate the echo client

This book's echo Web service client is simple; it is a thin client.

- Right-click in the Project Explorer view on **staging** → **Echo.wsdl**, → **Web services** → **Generate Client**".

You started with the staging Echo.wsdl, but now that EchoService is deployed, there is a real WSDL available:

- Change the Service Definition field to refer to this real WSDL(Figure 7-6):

http://localhost:9080/EchoService/EchoService?WSDL

Your machine name and port number might be different depending on how you installed WebSphere Application Server.

- Select IBM WebSphere JAX-WS for the service runtime.

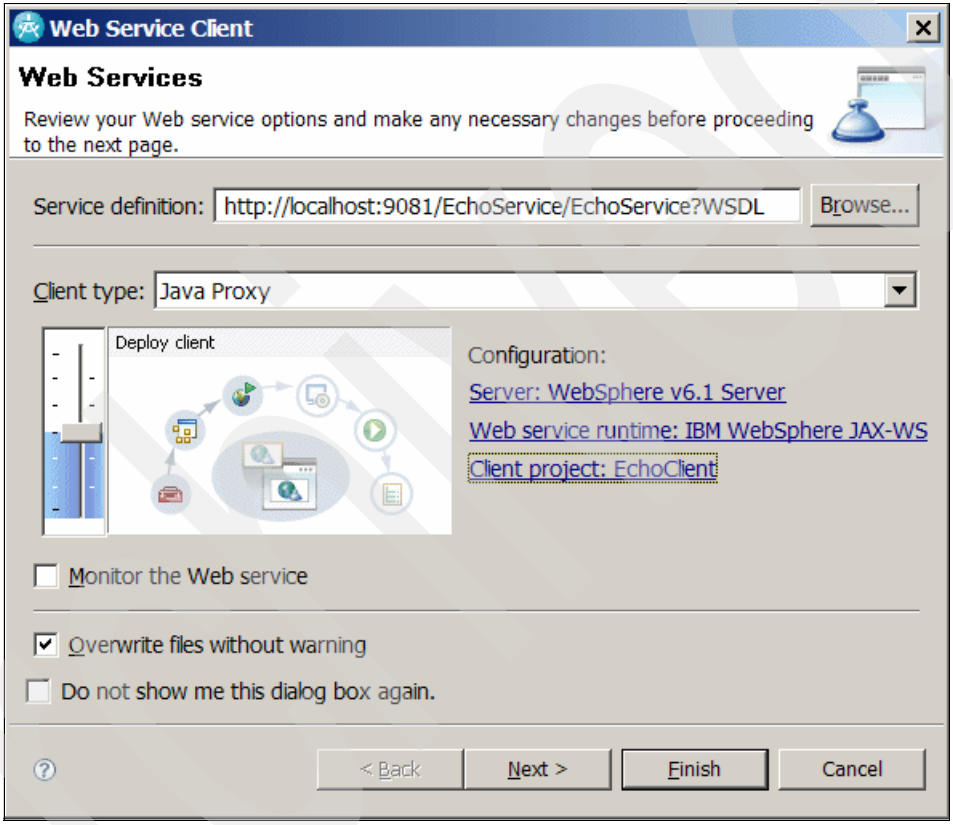


Figure 7-6 Create Client

- Define the client project with the name `EchoClient` and the type `Java Utility Project` (Figure 7-7) → **OK**.

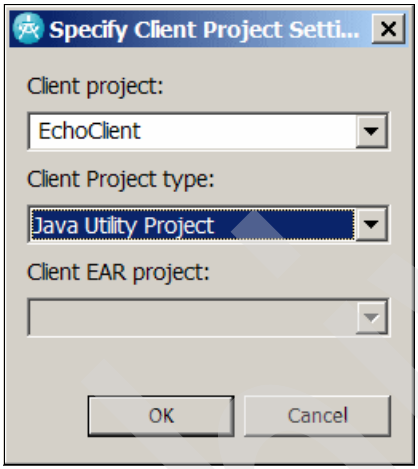


Figure 7-7 Specify client

- Click **Next**. In the options panel (Figure 7-8), check **Enable asynchronous invocation for generated client**. We do this so that we can discuss asynchronous programming later → **Finish**.

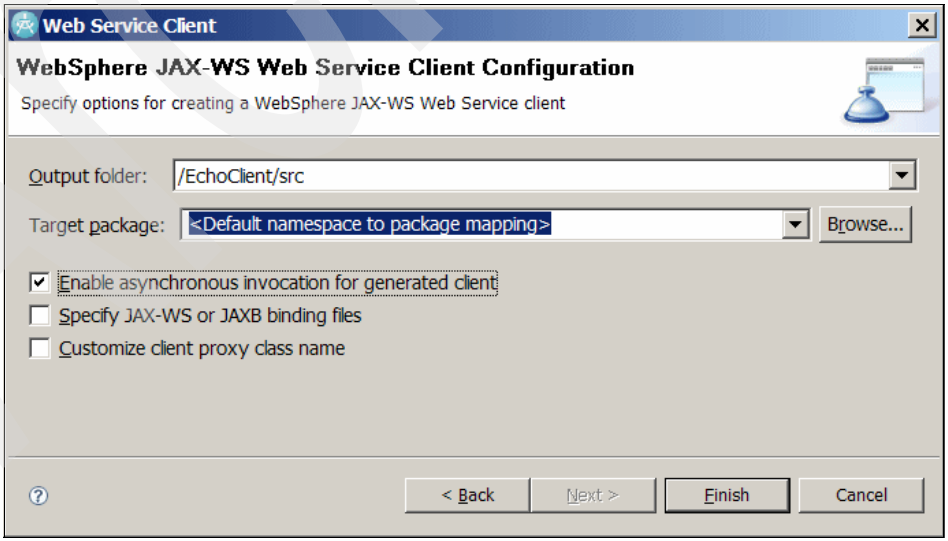


Figure 7-8 Enable Asynchronous client

We now have the client-side bindings for the Echo Web service.

- There is one more step that you must do in order to actually run this client. This is a thin (or unmanaged) client. It is not part of any container and has no association with a server. We have to include the Web services thin client JAR file in the build class path. This JAR file contains the code for the client-side SOAP engine.

- Right-click on the **EchoClient project** → **Properties**.

- Select **Java Build Path** on the left of the panel → Select the tab labeled **Libraries** on the right hand side → **Add External JARs...** → Browse to and **Open**:

  `%WAS_HOME%\runtimes\com.ibm.jaxws.thinclient_6.1.0.jar`

  Be sure about the name; that same directory also contains the thin client JAR file for JAX-RPC Web services

  Figure 7-9 shows the resulting panel with the thin client on the build path.

  Note: The wizard also has a variable which points to this thin client JAR file, but if your WAS runtime is not the test server runtime, this variable points to a non-existent location.

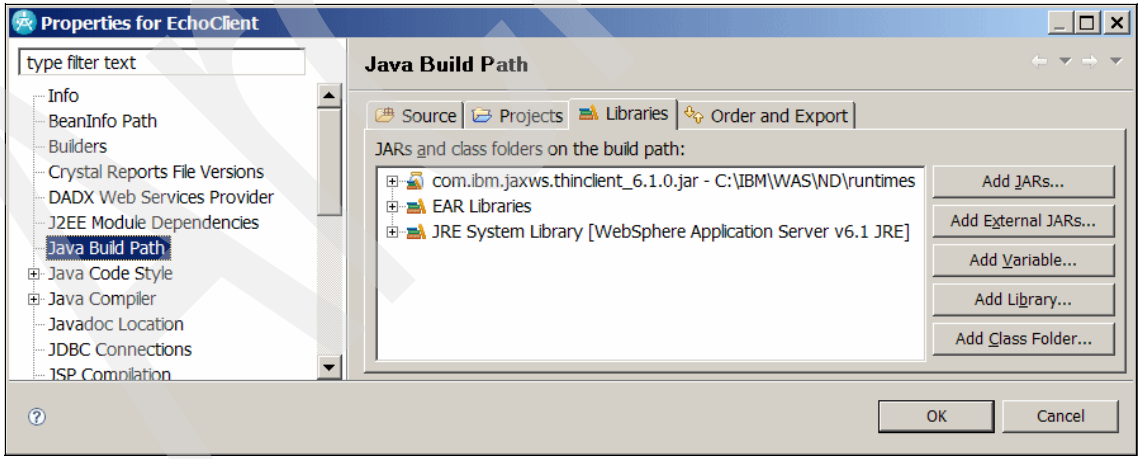


Figure 7-9 Java Build Path — Libraries tab

Make sure that this thin client JAR is first in the CLASSPATH, otherwise you get an exception with a stack trace that contains com.sun... classes. We do not use the com.sun classes.

- Select the “Order and Export” tab → Select the thin client JAR → Click the **Up** button until the thin client JAR is at the top as in Figure 7-10[4] → **OK**.

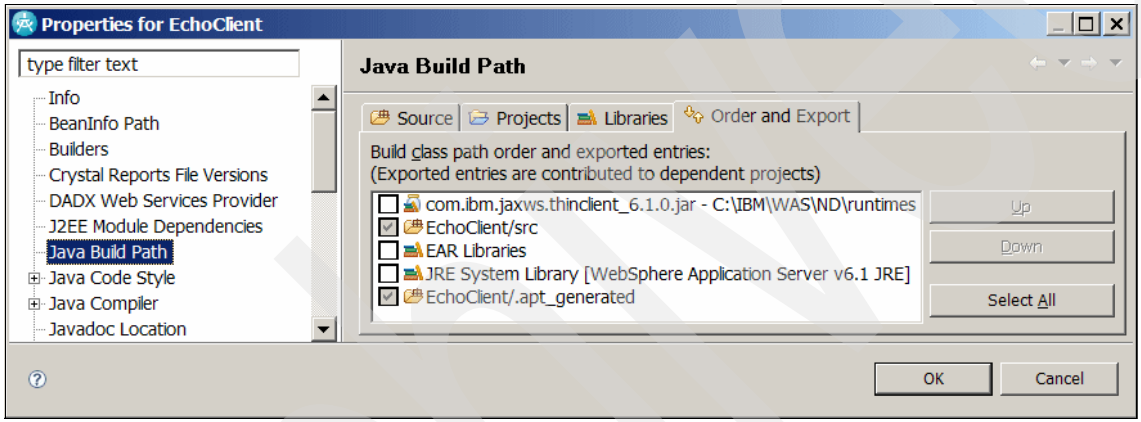


Figure 7-10 Java Build Path order- Order and Export tab

We have now finished generating the Java client bindings for the echo Web service. We start using the client in 7.4, “The synchronous client API” on page 193.

> **Note:** Some documentation tells us to create a dynamic Web project first, then generate the Web service client into that project. That process very nicely adds the thin client JAR variable to the build path, however, that variable will be wrong when your runtime is not the test server runtime optionally installed with Rational Application Developer. So we chose here to walk you through manually adding the thin client JAR.

7.1.3 Command-line tools

Rational Application Developer provides wizards to generate client and server code. We can perform the same operations with command-line tools.

[4] We have removed the WebSphere Application Server library from this list. This is a thin client, so it cannot really rely on the WAS library.

wsimport

This is the command line WSDL-to-Java tool. For details, see the InfoCenter at:

http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/index.jsp?topic=/com.ibm.websphere.wsfep.multiplatform.doc/info/ae/ae/rwbs_wsimport.html.

xjc

This command-line tool generates Java beans from XML schema. Normally we do not have to call this tool. It is called implicitly by wsimport and it generates the WSDL-related artifacts. For details, see the InfoCenter at:

http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/index.jsp?topic=/com.ibm.websphere.wsfep.multiplatform.doc/info/ae/ae/twbs_jaxbschema2java.html.

wsgen

Although we have not shown you how to use the wizard to generate a JAX-WS Web service from existing Java code, the wizard does exist. Because the wizard exists, it is only reasonable to assume that the command-line tool also exists. wsgen is that command-line tool. For details, see the InfoCenter at:

http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/index.jsp?topic=/com.ibm.websphere.wsfep.multiplatform.doc/info/ae/ae/rwbs_wsgen.html.

7.2 The generated artifacts

Now that we have generated for Web service and its client bindings, you can take a look ahead at what was generated (see 7.2.2, “Client generated files”):

- Figure 7-11 on page 188 shows you the contents of the EchoClient project.
- Figure 7-12 on page 189 shows you the contents of the EchoService project.

7.2.1 Common generated files

The highlighted class files are the files that are common to both the client and server. As you can see, most of the files are common[5]. Table 7-1 gives a short explanation of each of these common files.

Table 7-1 Common generated files

File type	Echo-specific file	Notes
Request message type	EchoOperation.java	This is mapped from the request message wrapper element. Normally not used for document/literal wrapped Web services.
Response message type	EchoOperation Response.java	This is mapped from the response message wrapper element. Normally not used for document/literal wrapped Web services except for asynchronous responses.
Interface	EchoServicePortType .java	This is mapped from the WSDL portType. Each WSDL operation becomes a Java method. The client interface can also contain asynchronous methods for each operation.
Fault types	Fault.java	There can be zero or more fault types.
Fault exceptions	FaultException.java	There can be zero or more exceptions.
Data types	Only the fault type	There can be zero or more data types.
Object factory	ObjectFactory.java	This is a factory class through which we create instances of the data types, faults, and message types.
Package info	package-info.java	This file contains the namespaces from which the Java packages are mapped.

If you are familiar with JAX-RPC, you might realize that there are far fewer classes generated for JAX-WS than JAX-RPC. JAX-RPC implementations generate helper classes and files. The JAX-WS engine does not require helper classes generated by tooling. It uses the annotations in the Java files to determine dynamically how to invoke Web services. Because annotations were not available in Java 1.4, JAX-RPC required helper classes.

[5] For a description of the artifacts generated by *wsimport, s*ee: http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/index.jsp?topic=/com.ibm.websphere.wsfep.multiplatform.doc/info/ae/ae/rwbs_wsimport.html

7.2.2 Client generated files

Figure 7-11shows the Java files generated for the client by Rational Application Developer and the Application Server Toolkit. Because we created EchoClient as a Java Utility Project (Figure 7-11), it is simply a POJO (Plain Old Java Object), so there are no deployment descriptors.

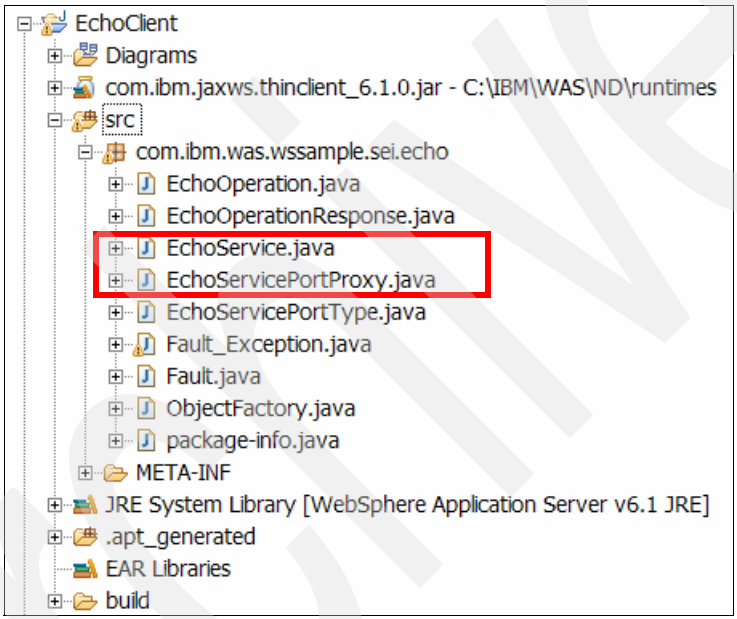


Figure 7-11 Client specific generate files

The highlighted files, EchoService.java and EchoServicePortProxy.java, are generated by the toolkit and are specific to the client. See “The synchronous client API” on page 193 for details of these two classes. Briefly:

- EchoService.java is generated from the WSDL service. It is a factory class that returns an instance that implements the service’s interface. In JAX-RPC, this implementation class is called a stub. In JAX-WS, no stub class exists; the “stub” is a class dynamically generated from WSDL[6].
- EchoServicePortProxy.java is an IBM-proprietary proxy class. JAX-WS does not define this class. It is a convenience class which implements the Web service’s interface and hides programming details such as the service factory and BindingProvider calls. Do not program to this class if you want to write portable code.

[6] JAX-WS clients require access to the service WSDL at runtime to instantiate the service interface. This is a significant change to JAX-RPC clients, which do not require access to WSDL at runtime. See 7.9, “Change the address of the Web service in the client” on page 225.

The unhighlighted Java files in Figure 7-11 on page 188 and Figure 7-12 here are common to both the client and server.

7.2.3 Server generated files

Only one generated file is specific to the server: EchoServicePortImpl.java. When this file is generated, it is a skeleton for the implementation. This is where we code the service implementation.

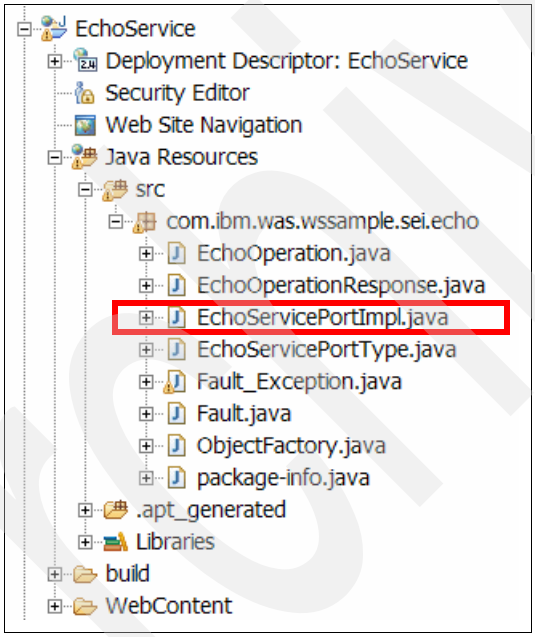


Figure 7-12 Server specific generated file

7.2.4 Hidden round-tripping

Round-tripping is the process of mapping from one representation to another and back again. Typically, we want to end up where we started. For instance, take a number from the Arabic numeral system, such as 9. Map this number to the Roman numeral system. We end up with IX. Now map that back to the Arabic numeral system and we end up with, no surprise, 9.

But why is this no surprise? It is no surprise because the mapping from Arabic to Roman numerals, and back again, is what is mathematically known as one-to-one and onto. In short, for each Arabic numeral, there is one, and only one, Roman numeral which maps to it. The same is true for the mapping of Arabic to Roman numerals.

Or is it? What about the Arabic numeral 0? There is no mapping of Arabic 0 to a Roman numeral. So while it seems that there are no surprises, there often are. That is especially true in the world of Web services. JAX-WS defines two mappings: WSDL-to-Java describes the Java artifacts generated from WSDL; Java-to-WSDL describes the WSDL generated from Java code. When we apply WSDL-to-Java mappings to get Java classes, and then Java-to-WSDL mappings to those Java classes to get another WSDL, we have done WSDL round-tripping. Often the resulting WSDL looks just like the WSDL that we started with. However, JAX-WS does not guarantee this, and if you do not keep this in mind, you might, for example, be surprised by an unexpected element name.

The issue

Why should we care about round-tripping? Because it occurs frequently. It might not be obvious, but when we create the Echo service and client in "Generate the echo service" on page 177 and "Generate the echo client" on page 181, we are performing round-tripping, as shown in Figure 7-13. We generated the Echo Web service from the WSDL in the staging project — this is the first step in WSDL round-tripping. The EchoService project contains no WSDL, so when we ask the EchoService for the WSDL (by referring to the WSDL's URL of `http://localhost:9080/EchoService/EchoService?WSDL`, the SOAP engine dynamically generates that service's WSDL from the service's Java code. At that point we have just completed the second half of WSDL round-tripping.

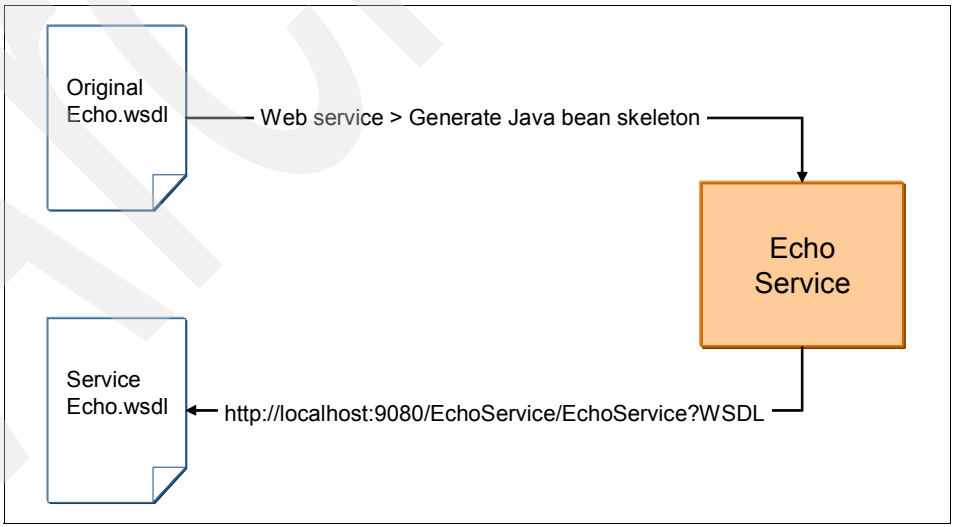


Figure 7-13 Round-tripping in the Echo scenario

As in mapping numerals from one representation to another, round-tripping often appears to work and we get lulled into forgetting that it sometimes does not. We have built Echo.wsdl to show one case where it does not work: name collisions.

In Echo.wsdl, the name “Fault” is overused, which would result in a name collision in the generated Java classes if JAX-WS did not have rules to manage name collisions.

Figure 7-14 contains the relevant portion of the original WSDLFault used in the WSDL to name the operation, message, schema element and the complexType.

```
<wsdl:message name="Fault">
  <wsdl:part name="Fault" element="tns:Fault"/>
</wsdl:message>
```

Figure 7-14 Fault information in original WSDL

For a fault, the JAX-WS WSDL-to-Java generator creates two Java classes: an exception class, and a Java bean containing the fault data. The exception class name is derived from the message name. The Java bean name is derived from the complexType name. Because they are both the same name, the naming collision has to be resolved. JAX-WS dictates that this is resolved by suffixing the exception with the string “_Exception”. That is where the name “Fault_Exception” in Figure 7-12 on page 189 comes from.

Now if we look at the WSDL generated from the Echo service[7] — see Figure 7-15 — we can see that it derived the message name from the name of the Java exception. The message name in the generated WSDL is different from the message name in the original WSDL.

```
Address http://localhost:9081/EchoService/EchoService/echoservice.wsdl   Go   Links »
- <fault name="Fault_Exception">
    <soap:fault name="Fault_Exception" use="literal" />
  </fault>
```

Figure 7-15 Fault information in generated WSDL

The final twist in this round-tripping issue is the Java code generated on the client. If we compare the names of the Java classes generated in both the client and the server in Figure 7-11 on page 188 and Figure 7-12 on page 189, respectively, you will notice that the server side has “Fault_Exception.java” and the client side has “FaultException.java”. Because Java naming conventions frown on underscores in class names, when a schema name containing an underscore is mapped to a Java name, JAX-WS dictates that the underscore is removed and we end up with the name on the client: “FaultException”.

[7] We use Internet Explorer®, rather than the Web services Explorer, to display the WSDL, because as of WebSphere Application Server v6.1 FP 13, the Web services Explorer does not support the feature pack and does not display WSDL generated by the Feature pack for Web services correctly.

Note that this is Java round-tripping (Java → WSDL → Java) interleaved with WSDL round-tripping: WSDL → Java → WSDL → Java. JAX-WS does not promise that either WSDL round-tripping or Java round-tripping preserves the original artifacts.

Does the issue matter?

Does this round-tripping issue matter? From a practical point of view, no. We might end up with Java code that looks different, as we did here, but generally we do not share Java, we share WSDL. We have been discussing how a WSDL maps to Java, and vice versa, but WSDL also maps to SOAP. It is the SOAP messages that are important for communications. The portions of the WSDL that describe the contents of the SOAP messages are generally only in the XML schema. As you can see, the XML schema has not changed in WSDL round-tripping. The XML schema will properly round-trip, so the part of the WSDL that describes the SOAP messages does not change, and the communications are not affected by WSDL round-tripping issues.

When might this issue matter?

- It might matter if your build, deployment, or maintenance environment relies on common Java files.
- If your source code control process requires common WSDL for services and clients, rather than relying on dynamic creation of WSDL, you might be concerned.
- Comments in the original WSDL are not preserved. If you want to preserve comments, you have a problem.
- MTOM expected types are not preserved from WSDL to WSDL and your generated client does not have as much information as the generated service. See “MTOM” on page 213 for details.

Avoiding the issue

This issue arises because the SOAP engine dynamically generates the round-tripped WSDL. We can avoid this generation step and use the original WSDL by telling the Java bean skeleton generation wizard to generate a Java bean that contains the original WSDL. We do this by checking the “Copy WSDL to project” box on the panel shown in Figure 7-4 on page 179.

Copying the WSDL is not necessary in most cases, but the extra file in the project takes up very little space, and we suspect that it might become a best practice to copy the WSDL into the service project simply to avoid having to understand the details of when round-tripping might become an issue.

7.3 The service API

Programming a JAX-WS service is very similar to programming a JAX-RPC service. In the top-down scenario we presented here, the tools generate an interface and a skeleton implementation. We do not have to know anything about XML schema or WSDL to write the implementation. We are presented with the skeleton and all we have to do is implement its operations.

The bottom-up scenario is also similar to the JAX-RPC model, however, we now have the opportunity to decorate the legacy Java code with Java 5 annotations to give the Java-to-WSDL generation process hints about how to generate the WSDL. We cover annotations in "Annotations" on page 211.

7.4 The synchronous client API

If you are familiar with the JAX-RPC client programming model, there is really only one surprise in the JAX-WS synchronous client programming model. The static client has to read the WSDL for the service at runtime, whereas for JAX-RPC static clients, the development tools generated proxy code from the WSDL at compile time, and the WSDL was not referred to by the runtime. We return to this important difference in 7.9, "Change the address of the Web service in the client" on page 225. Otherwise, there is not much difference between programming a JAX-WS client and programming a JAX-RPC client.

The WSDL-to-Java tools generate an interface; this interface potentially contains references to generated types and exceptions. There are slight differences between the JAX-RPC data mapping and the new JAXB 2.0 data mapping[8], but nothing exceptional.

In order to make a call to a Web service, we follow either of two approaches:

- Use the pure JAX-WS generated APIs. These are the classes that the JAX-WS specification defines. Use this interface to code Java that ports to other J2EE application servers, and is close as possible to the Web service coding model on other platforms, such as WCF.
- Use the IBM-specific generated proxy class. This proxy class is a convenience class that simplifies the programming model. Frequently portability of the Java code implementing a Web service is not a requirement, and using the IBM-specific generated proxy class is likely to be the most popular choice.

[8] See the article titled "Web services hints and tips: JAX-RPC versus JAX-WS, Part 2 — for a look at the differences in data mapping, at: http://www.ibm.com/developerworks/webservices/library/ws-tip-jaxwsrpc2.html

7.4.1 The pure JAX-WS API

The pure JAX-WS classes are those that JAX-WS defines should be generated. To call the Web service via the pure JAX-WS APIs, you do three things:

- Get an instance of the service factory.
- Get an instance of an object that implements the interface, from the factory.
- Call the Web service via the stub.

These steps are illustrated for the echo client in Example 7-1.

Example 7-1 Call the echo service via a JAX-WS stub

```
// Get an instance of the service factory
EchoService serviceFactory = new EchoService();

// Get an instance of an EchoServicePortType object from the factory
EchoServicePortType stub = serviceFactory.getEchoServicePort();

// Call the Web service via the stub
System.out.println("stub.echoOperation(\"Hello\") returns \""
      + stub.echoOperation("Hello") + "\"");
```

7.4.2 The IBM-specific proxy

The IBM-specific proxy is a convenience class that relieves us from some of the detailed work that is necessary when coding to the pure JAX-WS APIs. In the case of simply calling a Web service, all it saves you is one line of code to create a client stub using the Web service factory. But there is more detail that the proxy class hides from you. Examine the source code generated in the EchoClient project if you are curious about these details. When you use a proxy to call a Web service, you make two Java calls:

- Get an instance of the proxy class.
- Call the Web service via the proxy.

These steps are illustrated for the echo client in Example 7-2.

Example 7-2 Call the echo service via an IBM proxy

```
// Get an instance of the proxy class
EchoServicePortProxy proxy = new EchoServicePortProxy();

// Call the Web service via the proxy
System.out.println("proxy.echoOperation(\"Hello\") returns \""
      + proxy.echoOperation("Hello") + "\"");
```

7.4.3 Test the EchoService using the synchronous IBM proxy

- Add a new class to the com.ibm.was.wssample.sei.echo package in the EchoClient project using the **New** → **Class** wizard calling the new class EchoClientMainSync and check the **public static void main(String[] args)** check box (Figure 7-16) → **Finish**.

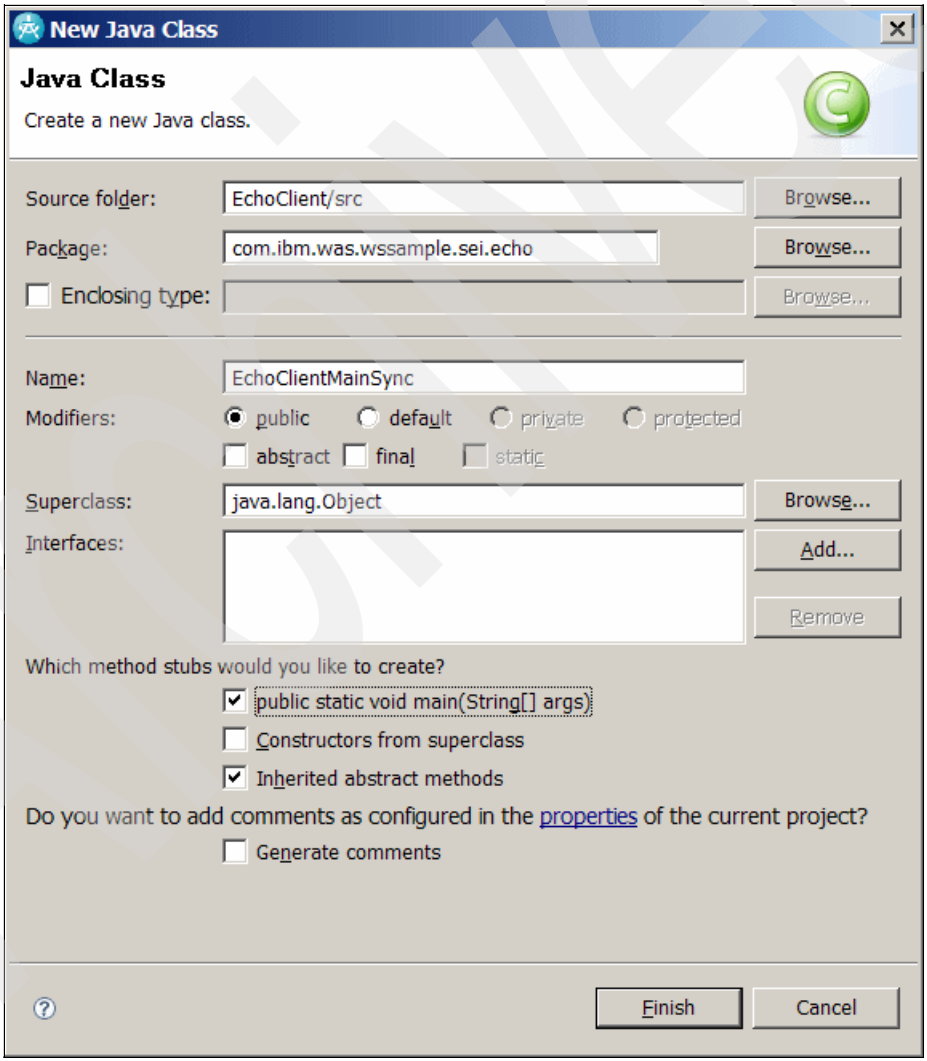


Figure 7-16 Create EchoClientMainSync

- In the EchoClientMainSync.java generated stub, copy the code from Example 7-2 on page 194, and add `throws Exception` to the class definition (Example 7-3).

Example 7-3 EchoClientMainSync

```
package com.ibm.was.wssample.sei.echo;
public class EchoClientMainSync {
   public static void main(String[] args) throws Exception {
//     Get an instance of the proxy class
      EchoServicePortProxy proxy = new EchoServicePortProxy();
//     Call the Web service via the proxy
      System.out.println("proxy.echoOperation(\"Hello\") returns \""
            + proxy.echoOperation("Hello") + "\"");
   }
}
```

- Click on **EchoClientMainSync.java** in the Project Explorer → **Run as** → **Java Application.** The results are displayed in the Console tab:

```
proxy.echoOperation("Hello") returns "JAX WS echo >> Hello"
```

7.5 The asynchronous client API

JAX-RPC does not have an asynchronous model. It is new in JAX-WS. There are two asynchronous APIs: callback and polling. When we enable asynchronous invocation for the generated client in the generation wizard, the Web service wizard adds two additional methods to the Web service interface as shown for the echo client in Example 7-4 (annotations have been removed for brevity).

Example 7-4 Asynchronous EchoServicePortType methods

```
public Response<EchoOperationResponse> echoOperationAsync(
   String echoInput);

public Future<?> echoOperationAsync(
   String echoInput,
   AsyncHandler<EchoOperationResponse> asyncHandler);
```

The echoOperationAsync method has two signatures. The echoOperationAsync version, which returns a Response, is used for polling; and the version that returns a Future is used for callback. Each version of the method uses the EchoOperationResponse class as an parameter of a generic type.

The EchoOperationResponse class is the wrapper element for the response. The asynchronous response might well contain more than one field, and the wrapper element, class works, bundles up all the fields nicely for this purpose.

7.5.1 Asynchronous polling API

After an asynchronous client invokes a service sending a request, it must have a way to get a response. One mechanism is to periodically poll for the response. As we saw in the first operation in Example 7-4 on page 196, the echoOperationAsync method returns a Response object containing the EchoOperationResponse data. The Response class inherits two methods that we are interested in, `isDone` and `get`. The client periodically calls `isDone`. Once it returns true, then the client retrieves the response data by calling get.

- Create a new Java class called EchoClientMainAsyncPolling similar to EchoClientMainSync in 7.4.3, “Test the EchoService using the synchronous IBM proxy” on page 195. Example 7-5 shows the polling code embedded into the new class.

Example 7-5 Polling client

```
package com.ibm.was.wssample.sei.echo;
import javax.xml.ws.Response;
public class EchoClientMainAsyncPolling {
   private static final int SLEEPER = 2; // Polling interval
   private static final int TIMEOUT = 240; // Error timeout in seconds
   public static void main(String[] args) throws Exception {
      EchoServicePortProxy proxy = new EchoServicePortProxy();
      Response<EchoOperationResponse> resp = proxy.echoOperationAsync("Hello");
      // Poll for the response.
      Thread.sleep(1000);
      int waiting = TIMEOUT;
      while (!resp.isDone()) {
         // Check for timeout
         if (waiting <= 0) {
            System.out.println("echoOperation async timed out.");
            System.exit(0);
         }
         System.out.println("echoOperation async still not complete.");
         Thread.sleep(1000 * SLEEPER);
         waiting -= SLEEPER;
      }
      // Get the response and print it.
      EchoOperationResponse eor = resp.get();
      System.out.println("echoOperation async invocation complete.");
      if (eor != null) {
         System.out.println("echoOperation async response is: "
               + eor.getEchoResponse());
      }
   }
}
```

The output is written to the console:

```
echoOperation async still not complete.
echoOperation async invocation complete.
echoOperation async response is: JAX WS echo >> Hello.
```

One detail that we have neglected to discuss to this point is how to handle the WSDL fault that we have defined on Echo.wsdl. In the polling case, the get method throws `java.util.concurrent.ExecutionException`. This exception is defined by the Futures API to contain the exception which the service throws.

- ☐ Add the code fragment shown in Figure 7-6 to retrieve the response handling the service exception.

Example 7-6 Handling an asynchronous fault

```
import java.util.concurrent.ExecutionException;
   // Change the input to trigger the fault
   Response<EchoOperationResponse> resp =
      proxy.echoOperationAsync("Fault");
   // ...
   // Get the response and print it.
   try {
      EchoOperationResponse eor = resp.get();
      System.out.println("echoOperation async invocation complete.");
      if (eor != null) {
         System.out.println("echoOperation async response is: "
               + eor.getEchoResponse());
      }
   } catch (ExecutionException ee) {
      FaultException fe = (FaultException) ee.getCause();
      System.out.println("echoOperation failed:  reason = " +
            fe.getFaultInfo().getReason());
   }
```

- ☐ Modify the echoService Web service from the simple EchoService in Figure 7-5 on page 181 to the code shown in Example 7-7 to throw a FaultException when the input is the string “Fault”. The console displays the following line when the fault is thrown.

```
echoOperation failed:  reason = You told me to throw a fault
```

Example 7-7 EchoServicePortImpl.java

```
package com.ibm.was.wssample.sei.echo;
@javax.jws.WebService (endpointInterface =
   "com.ibm.was.wssample.sei.echo.EchoServicePortType",
   targetNamespace = "http://com/ibm/was/wssample/sei/echo/",
   serviceName = "EchoService", portName = "EchoServicePort")
public class EchoServicePortImpl {
   public String echoOperation(String echoInput)
      throws Fault_Exception {
      if ("Fault".equals(echoInput)) {
         Fault faultInfo = new Fault();
         faultInfo.setReason("You told me to throw a fault");
         throw new Fault_Exception("echo fault", faultInfo);
      }
      return "JAX WS echo >> " + echoInput;
   }
}
```

7.5.2 Asynchronous callback API

An alternative means of handling asynchronous responses is by using callback. When the client makes the request, it gives the receiver a class to call back once the receiver has a response.

- Create another new Java application in the EchoClient project called EchoClientMainAsyncCallback. The code is shown Example 7-8. In the interests of keeping the example as simple as possible, and because this book is not a handbook of the Java Futures design, rather than synchronize on the response object for the completion of the callback routine to continue with the main thread, we simply “sleep” for five seconds.

Example 7-8 EchoClientMainAsyncCallback

```
package com.ibm.was.wssample.sei.echo;
import java.util.concurrent.Future;
public class EchoClientMainAsyncCallback {
   public static void main(String[] args) throws Exception {
      EchoServicePortProxy proxy = new EchoServicePortProxy();
      // Set up the callback handler.
      EchoClientCallbackHandler callbackHandler = new EchoClientCallbackHandler();
      // Make the Web service call.
      Future<?> response = proxy.echoOperationAsync("Hello", callbackHandler);
      System.out.println("Wait 5 seconds.");
      // Give the callback handler a chance to be called.
      Thread.sleep(5000);
      System.out.println("echoOperation async is now ending.");
   }
}
```

- ☐ Create the call back class EchoClientCallbackHandler as shown in Example 7-9

  The code highlighted in blue is almost identical to code in the polling example. There is the additional Interrupted Exception to handle, and obviously the additional class and method signature definitions. The output in the console is:

```
Wait 5 seconds.
echoOperation async invocation complete.
echoOperation async response is: JAX WS echo >> Hello
echoOperation async is now ending.
```

Example 7-9 EchoClientCallbackHandler

```
package com.ibm.was.wssample.sei.echo;
import java.util.concurrent.ExecutionException;
import javax.xml.ws.AsyncHandler;
import javax.xml.ws.Response;
public class EchoClientCallbackHandler implements
      AsyncHandler<EchoOperationResponse> {
   public void handleResponse(Response<EchoOperationResponse> resp) {
      try {
         EchoOperationResponse eor = resp.get();
         System.out.println("echoOperation async invocation complete.");
         System.out.println("echoOperation async response is: "
               + eor.getEchoResponse());
      } catch (ExecutionException ee) {
         FaultException fe = (FaultException) ee.getCause();
         System.out.println("echoOperation failed:  reason = "
               + fe.getFaultInfo().getReason());
      } catch (InterruptedException ie) {
         System.out.println("EchoOperation received Interrupted Exception");
      }
   }
}
```

7.5.3 Asynchrony on the wire

The examples so far show you how to write clients that call a Web service asynchronously. But 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, as written, they are not. The SOAP/HTTP protocol is no different in the asynchronous case from the synchronous one. The client Web services engine opens a connection, sends the request, and receives the response back along the same connection. The client-side SOAP engine implements the asynchrony via a response thread.

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; and 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 does matter 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. IBM has provided a feature that goes beyond the JAX-WS specification.

First let us see how to set up wire level asynchrony using the Feature pack for Web services, then discuss how this is implemented, the firewall considerations, and what the wider architectural choices are when considering whether to use client async and wire async.

Enabling wire asynchrony

- Copy EchoClientMainAsyncCallback.java into a new Java application called EchoClientMainAsyncCallbackWireAsync.java, and add the code shown in **blue** in Example 7-10 to set the Wire Async property (compare with Example 7-8 on page 201). The results of running EchoClientMainAsyncCallbackWireAsync are shown in the following console output:

```
[WAShttpAsyncResponseListener] listening on port 1085
Wait 5 seconds.
echoOperation async invocation complete.
echoOperation async response is: JAX WS echo >> Hello
echoOperation async is now ending.
```

Example 7-10 EchoClientMainAsyncCallbackWireAsync

```
package com.ibm.was.wssample.sei.echo;
import java.util.concurrent.Future;
import javax.xml.ws.BindingProvider;
public class EchoClientMainAsyncCallbackWireAsync {
   public static void main(String[] args) throws Exception {
      EchoServicePortProxy proxy = new EchoServicePortProxy();
      // Set wire async
      BindingProvider bp = (BindingProvider) proxy._getDescriptor().getProxy();
      bp.getRequestContext().put("com.ibm.websphere.webservices.use.async.mep",
            Boolean.TRUE);
      // Set up the callback handler.
      EchoClientCallbackHandler callbackHandler = new EchoClientCallbackHandler();
      // Make the Web service call.
      Future<?> response = proxy.echoOperationAsync("Hello", callbackHandler);
      // Sleep for a while here and let the callback thread do its thing.
      System.out.println("Wait 5 seconds.");
      // Give the callback handler a chance to be called.
      Thread.sleep(5000);
      System.out.println("echoOperation async is now ending.");
   }
}
```

Feature pack for Web services implementation of wire async

To accomplish this wire-level asynchrony, the Web services engine relies on WS-Addressing headers. Example 7-11 shows you an example request message, and Example 7-12 shows you an example response message.

Example 7-11 echoOperation asynchronous request message

```
<soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:wsa="http://www.w3.org/2005/08/addressing">
  <soapenv:Header>
    <wsa:To>http://localhost:9081/EchoService/EchoService</wsa:To>
    <wsa:ReplyTo>
      <wsa:Address>
        http://192.168.86.131:1120/axis2/services/EchoService.EchoServicePort
      </wsa:Address>
    </wsa:ReplyTo>
    <wsa:MessageID>urn:uuid:791F4B43453573FB7D1204876983384</wsa:MessageID>
    <wsa:Action>echoOperation</wsa:Action>
  </soapenv:Header>
  <soapenv:Body>
    <ns2:echoOperation xmlns:ns2="http://com/ibm/was/wssample/sei/echo/">
      <echoInput>Hello</echoInput>
    </ns2:echoOperation>
  </soapenv:Body>
</soapenv:Envelope>
```

Example 7-11 contains a typical set of WS-Addressing headers for an asynchronous request message:

To: This is the Web service's endpoint address.

ReplyTo: The receiver of the request where to send the response; typically it is a unique port open specifically for Web services responses.

MessageID: The response message will contain this message ID so that the receiver of the response can correlate the response message to the request message.

Action: The name of the operation being invoked; in the case of document/literal wrapped WSDL, this header is superfluous because the body already has all the information the receiving engine needs to dispatch the message.

The asynchronous response message in Example 7-12 has many of the same headers as the request message. The unique one for a response is the *RelatesTo* header. The contents of this header are the same as the contents of the

MessageID header of the request. This provides the correlation from the request to the response.

Example 7-12 echoOperation asynchronous response message

```
<soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:wsa="http://www.w3.org/2005/08/addressing">
  <soapenv:Header>
    <wsa:To>
      http://192.168.86.131:1120/axis2/services/EchoService.EchoServicePort
    </wsa:To>
    <wsa:ReplyTo>
      <wsa:Address>http://www.w3.org/2005/08/addressing/none</wsa:Address>
    </wsa:ReplyTo>
    <wsa:MessageID>urn:uuid:70A594E9BAACEFF6EC1204876987200</wsa:MessageID>
    <wsa:Action>
      http://com/ibm/was/wssample/sei/echo/EchoServicePortType/echoOperationResponse
    </wsa:Action>
    <wsa:RelatesTo>urn:uuid:791F4B43453573FB7D1204876983384</wsa:RelatesTo>
  </soapenv:Header>
  <soapenv:Body>
    <ns2:echoOperationResponse xmlns:ns2="http://com/ibm/was/wssample/sei/echo/">
      <echoResponse>JAX-WS echo &gt;&gt; Hello</echoResponse>
    </ns2:echoOperationResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

Note: With wire-level asynchrony, debugging becomes a bit more of a chore. The TCP/IP monitor loses much of its usefulness. It only captures request messages, not response messages. By turning on trace for the feature pack's Web service engine, you can view all the messages:

```
com.ibm.ws.websvcs.trace.MessageTrace=all
```

The SOAP messages now appear in the trace.log file. They are not formatted well, so it takes a bit more effort to read them, but at least they are available.

Asynchrony and firewalls

If you plan to use wire-level asynchrony, you must investigate whether there are firewalls between the client and the service, and how they are configured. It is very likely, if there are firewalls, that they will not permit the Reply message to pass through.

When wire-level asynchrony is enabled, WS-Addressing headers are used. You can see in Example 7-11 on page 204 that the request message contains the header wsa:ReplyTo. In this example, the address in this header contains port 1120. This is a random port. Random ports and firewalls do not mix. If there is a firewall between the sender and receiver of asynchronous request/response messages, we must explicitly set the response port to a number that the firewall allows. We do this with a JVM property:

```
java -Dcom.ibm.websphere.webservices.http.listenerPort=xxxx ...
```

Where “xxxx” is a port that is open through the firewall.[9]

Architectural considerations about using asynchrony

The advantage of using wire level asynchrony is that it decouples the transport and the application layer more. There are other alternatives. In Chapter 10, “Reliable messaging” on page 355, we shall see that WS-ReliableMessaging decouples the transport layer and application layer to an even greater extent than wire level asynchrony, because it not only decouples the transport session used for the reply from the session used for the request, it even decouples the application from relying on a continuous session to complete either the request or the reply. But the decoupling achieved by asynchrony comes at a cost: The decoupled application can be harder to deploy, and it is likely to be less interoperable because it makes greater use of WS-* specifications, and greater demands on WS-* implementations.

So the choice whether to use wire level asynchrony is not open and shut — it is not simply a “better” implementation for the developer to choose. The capability is there as one of many different options to consider when deciding how to implement a distributed application. It improves the throughput of a system, but has to be weighed against other ways of improving performance.

In an attempt to illustrate some of alternative options to implement a SOAP application, the table shown in Figure 7-17 takes the request/reply pattern and looks at how different Web services programming models compare[10].

[9] For more details, see the InfoCenter at:
http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/index.jsp?topic=/com.ibm.websphere.wsfep.multiplatform.doc/info/ae/ae/twbs_httpasyncresplist.html

[10]

HTTP sync	One HTTP session used in both directions
HTTP 2*Sync	Two HTTP sessions used, one for each direction (aka HTTP wire async)
Request/Reply	SOAP two-way request/reply
2*Request	SOAP one-way in each direction to construct a request/reply pair

				Client			Quality of Service						
Application	Transport	SOAP	Transport Use	Client	Flexibility	Simplicity	Interoperability	Firewall	Reliability	Availability	Throughput	Response	Score
Request/Reply	HTTP	SOAP	Sync	Sync	1	3	3	3	1	1	1	3	16
				Async	2	2	3	3	1	1	1	3	16
			2*Sync	Sync	1	3	2	2	1	2	2	2	15
				Async	2	2	2	2	1	2	2	2	15
		SOAP-RM	Sync	Sync	1	2	2	3	2	2	1	3	16
				Async	2	1	2	3	2	2	1	3	16
			2*Sync	Sync	1	2	1	2	2	3	2	2	15
				Async	2	1	1	2	2	3	2	2	15
	JMS	SOAP	One-way	Sync	1	2	1	1	3	3	3	1	15
				Async	2	2	1	1	3	3	3	1	16
2*Request	HTTP	SOAP	2*Sync	Async	3	1	2	2	1	2	2	2	15
		SOAP-RM			3	1	1	2	2	3	2	2	16
	JMS	SOAP	One-way		3	1	1	1	3	3	3	1	16

Figure 7-17 Alternative ways to implement request/reply using Web services

The table is indicative of the number of choices and the weightings are only suggestive, and you might well disagree with the choice of factors and the weightings used. The table does show a large number of alternative implementations of the request/reply pattern using Web services.

It also suggests that the advantages and disadvantages of each choice balance out, making it difficult to generalize about there being one programming model that is clearly more likely to be preferable in all cases. You have to study particular cases in context to choose the factors that are important, weight them appropriately, and decide which solution is best.

When the Web Services Interoperability organization RSP profile is finalized, and if it is widely adopted, then as a result of giving SOAP-RM an additional interoperability point in the score sheet, SOAP-RM choices do score higher than any of the other possibilities.

7.6 The dynamic programming model

The dynamic programming model makes it possible for clients to call services without knowing the name, location or signature of the service until runtime. In the JAX-WS client programming model, the static client is known as the Dispatch client, and the dynamic client as the Dynamic Proxy client. Dynamic services are services that are created programmatically at runtime, rather than being fixed at deployment. The dynamic service API is known as the Provider API.

First we list the common APIs used to program to the dynamic model, and then look at the specific APIs, javax.xml.ws.Dispatch and javax.xml.ws.Provider to program the client and server respectively.

JAX-WS defines dynamic APIs that are used both in client and the server. They are very similar. They manipulate the XML message using one of three mechanisms:

javax.xml.transform.Source API to work directly with XML producing/consuming APIs. The programmer can work at either the payload or the message level.

JAXB Java objects generated from XML schema gives a programmer a Java view of the message payload.

javax.xml.soap.SOAPMessage The SAAJ 1.3 API to manipulate SOAP messages.

There is a fourth API, but it only is applicable to HTTP not SOAP bindings:

javax.activation.DataSource API to work with MIME-typed messages. It is only usable with HTTP bindings, but the Feature pack for Web services does not support HTTP bindings.

7.6.1 Dynamic client API: The Dispatch class

To build a dynamic client, you call the javax.xml.ws.Dispatch object[11]. In brief, you start with a Service object, which describes the WSDL:

```
Service svc = Service.create(serviceQName);
```

[11] For more details, see the InfoCenter at:
http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/index.jsp?topic=/com.ibm.websphere.wsfep.multiplatform.doc/info/ae/ae/twbs_jaxwsdynclient.html
Or see the article titled Creating a Dispatch client at:
http://www.ibm.com/developerworks/websphere/library/techarticles/0707_thaker/0707_thaker.html

```
svc.addPort(portQName, SOAPBinding.SOAP11HTTP_BINDING, address);
```

Then you instantiate one of three Dispatch objects. To build an XML-based message:

```
Dispatch<Source> ds = svc.createDispatch(serviceQName, Source.class,
   Mode.PAYLOAD);
```

The first two parameters are self-evident. The last is either `Mode.PAYLOAD` or `Mode.MESSAGE`. Chose the appropriate mode to work with the message body (payload) or the complete message.

To build a JAXB-based message:

```
Dispatch<Object> dj = svc.createDispatch(serviceQName, jaxbContext,
   Mode.PAYLOAD);
```

To build a Dispatch object that uses the SAAJ APIs:

```
Dispatch<SOAPMessage> dsm = svc.createDispatch(serviceQName,
   SOAPMessage.class, Mode.MESSAGE);
```

With the Dispatch object, you build the message with the appropriate objects, then invoke the Web service:

```
ds.invoke(source);
dj.invoke(jaxBObject);
dsm.invoke(soapMessage);
```

The Dispatch object also has methods for asynchronous invocations and one-way invocations.

7.6.2 Dynamic service API: The Provider class

JAX-RPC does not have a dynamic server-side programming model. JAX-WS introduces the javax.xml.ws.Provider interface[12] to build dynamic services. On the server-side, you must implement the Provider interface.

The client-side Dispatch features have corresponding server-side Provider features:

- To describe the service, instead of instantiating a Service object, you insert an annotation into your implementation.
- Instead of instantiating a Dispatch object of a given type (Source, Object, or SOAPMessage), you implement the Provider interface with the given type.

[12] For more details, see the InfoCenter at: http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/index.jsp?topic=/com.ibm.websphere.wsfep.multiplatform.doc/info/ae/ae/twbs_devjaxwsendpt.html

Example 7-13 shows you a skeleton for a javax.xml.transform.Source Provider. The skeletons for the JAXB and SOAPMessage Providers would be similar. The annotation would be identical, only the invoke method's signature would change.

Example 7-13 Provider<Source> skeleton

```
package com.ibm.clients;
import javax.xml.transform.Source;
import javax.xml.ws.Provider;
import javax.xml.ws.WebServiceProvider;

@WebServiceProvider(wsdlLocation = "Echo.wsdl",
      targetNamespace = "http://com/ibm/was/wssample/sei/echo/",
      serviceName = "EchoService", portName = "EchoServicePort")

public class MyProvider implements Provider<Source> {
   public Source invoke(Source arg0) {
      return null;
   }
}
```

7.7 Annotations

We have given examples of coding to the generated Java classes, but we have not yet shown you the classes themselves. In this section we describe the annotations associated with an interface by looking at the interface for the echo service: EchoServicePortType. You can see its source in Figure 7-18.

```
package com.ibm.was.wssample.sei.echo;

import javax.jws.WebMethod;

@WebService(
    name = "EchoServicePortType",
    targetNamespace = "http://com/ibm/was/wssample/sei/echo/")
public interface EchoServicePortType {

    @WebMethod(action = "echoOperation")
    @WebResult(name = "echoResponse", targetNamespace = "")
    @RequestWrapper(
        localName = "echoOperation",
        targetNamespace = "http://com/ibm/was/wssample/sei/echo/",
        className = "com.ibm.was.wssample.sei.echo.EchoOperation")
    @ResponseWrapper(
        localName = "echoOperationResponse",
        targetNamespace = "http://com/ibm/was/wssample/sei/echo/",
        className = "com.ibm.was.wssample.sei.echo.EchoOperationResponse")
    public String echoOperation(
        @WebParam(name = "echoInput", targetNamespace = "")
        String echoInput)
        throws Fault_Exception;
}
```

Figure 7-18 EchoServicePortType interface

Java annotations all begin with an ampersand (@). Most of the interface is annotated. The interface itself is annotated with WebService. This tells us that this interface describes a Web service with the given portType name and the given targetNamespace.

Each Java method signature is heavily annotated:

- @WebMethod

  This annotation sets the SOAPAction field, which maps to the operation description in the WSDL binding.

- @WebResult

  When the WSDL is document/literal wrapped, as Echo.wsdl is, this annotation describes the response wrapper element contents. Because echoResponse is an unqualified element, the targetNamespace is empty.

- @RequestWrapper

  Because this is a document/literal wrapped WSDL, it has a wrapper element which this annotation describes. It not only describes the WSDL view of this wrapper element — with the localName and targetNamespace variables — it also describes the Java class name to which the wrapper maps.

- @ResponseWrapper

  This annotation plays the same role to the response wrapper element as the RequestWrapper annotation plays to the request wrapper element.

Within the parameter list, each parameter has an associated annotation. Each WebParam annotation serves the same purpose for the inputs as WebResult served for the outputs — it describes each wrapper element's contents.

This is just a brief introduction of the annotations available from JAX-WS and JAXB. For details of all the annotations available, see the InfoCenter at:

http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/index.jsp?topic=/com.ibm.websphere.wsfep.multiplatform.doc/info/ae/ae/rwbs_jaxwsannotations.html

Using the annotations

The process you have followed in this chapter has been top-down generation of Web services. We started with a WSDL file and generated the Web service. Generally speaking, top-down is the best practice. The interface should precede the code, and not change should the code change.

But sometimes the nature of a project precludes top-down generation. Many projects are "meet in the middle": We study an existing implementation and decide how the implementation should be encapsulated into an interface. Often a façade pattern is the best way forward. Having studied the implementation, we handcraft a WSDL interface, generate the Web services skeleton top-down, and then hand-code a mapping from the façade to the existing implementation.

Another way to create a façade is to use the façade wizard in Rational Application Developer to generate a session EJB. You could then do bottom-up generation to create the Web service. In other words, start with existing Java code and generate the Web service and the corresponding WSDL from it. The WSDL that is generated is determined by the annotations you create. The Java-to-WSDL wizards will generate your WSDL according to direction of the annotations. If you do not create annotations, then the java-to-WSDL wizard uses default mappings.

7.8 MTOM

MTOM is the new Message Transmission Optimization Mechanism. It supplements, and in time will likely succeed, the SOAP with Attachments (Sw/A) specification. The feature pack supports both models of passing binary data in conjunction with SOAP messages.

As we did with Echo.wsdl, we give you an MTOM.wsdl from which you build a Web service and its client. From these, we describe the features of the feature pack's implementation of MTOM. If you do not wish to create your own client and server, we have provided solutions in the file, Chapter8Solutions.PI.zip.

7.8.1 Generate the MTOM service

You have already generated the echo service, so we assume that you now understand how to use the Web service creation wizards. In this section we give you just the highlights to generate the MTOM service.

- Import the Chapter8Staging.PI.zip file if you have not already done so (see instruction on page 176).
- Find the staging project in the Project Explorer and select **MTOM.wsdl** → **Web service** → **Generate Java bean skeleton**.
- Name the service MTOMService and match the other details shown in Figure 7-19 on page 214 → **Next**.

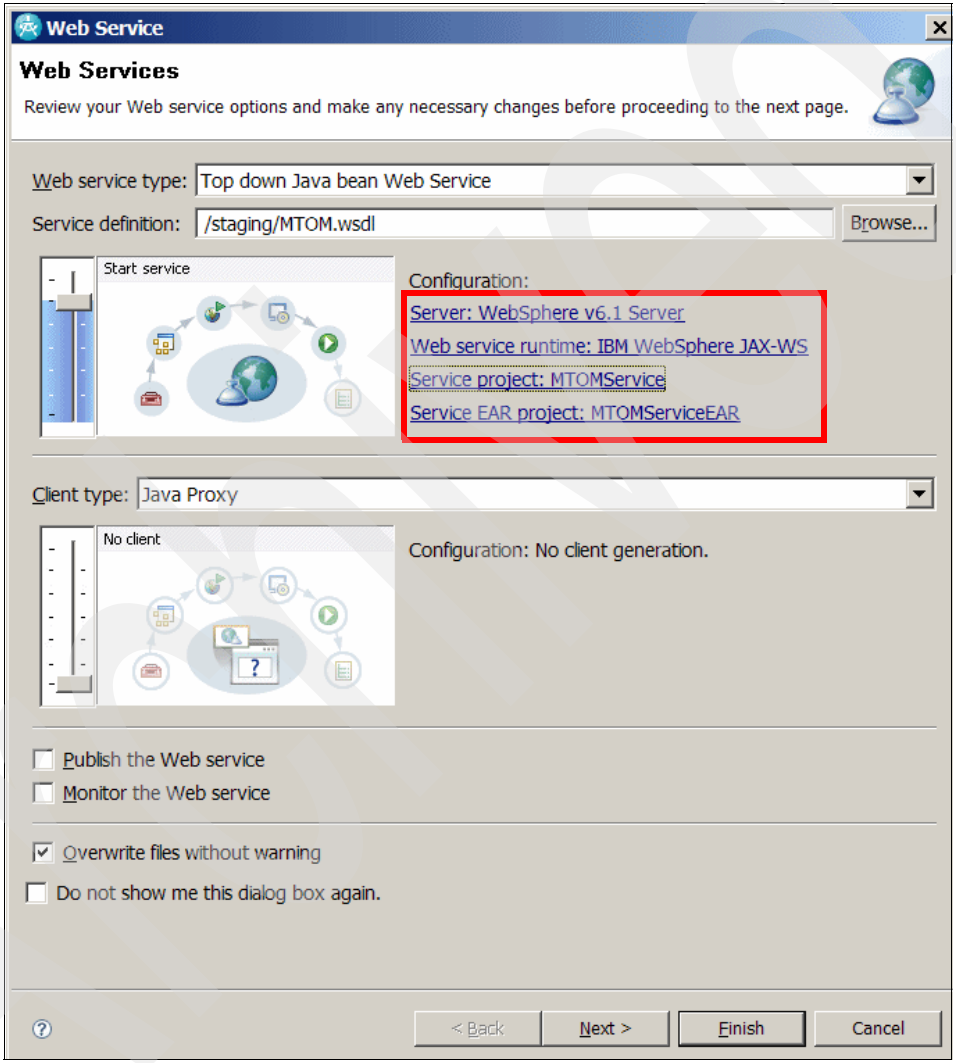


Figure 7-19 Create MTOMService

☐ Check the two additional boxes shown in Figure 7-20 → **Finish**.

We copy the WSDL to avoid the hidden round-tripping issue described in “Does the issue matter?” on page 192.

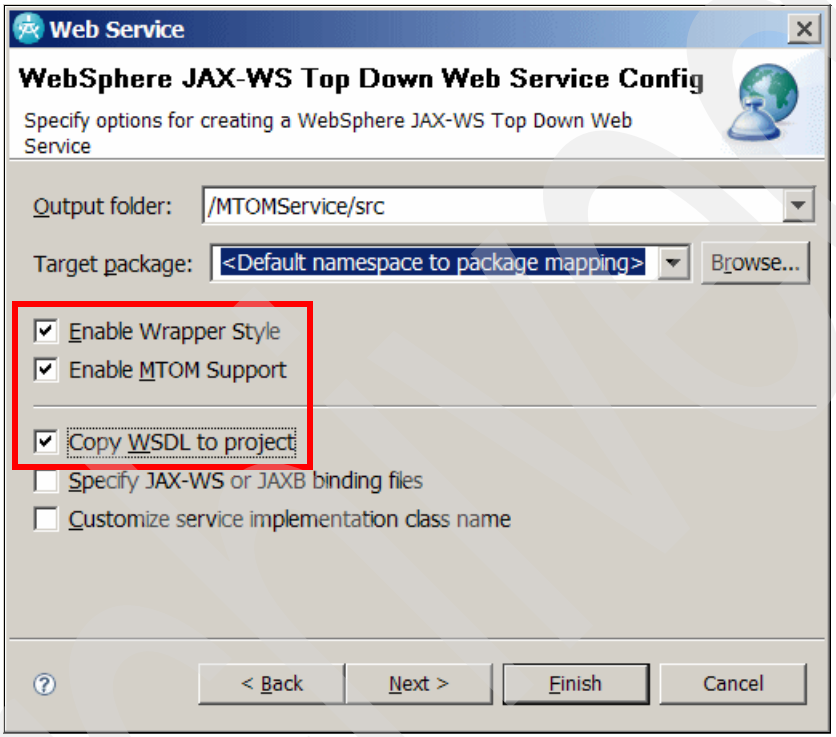


Figure 7-20 Java bean skeleton options panel for MTOM service

A warning window pops up (see Figure 7-21). Ignore this warning; MTOM is a new specification and not included in the Web Services Interoperability organization Basic Profile 1.1. The WS-I's Basic Profile 1.2 incorporates MTOM but is not yet finalized.

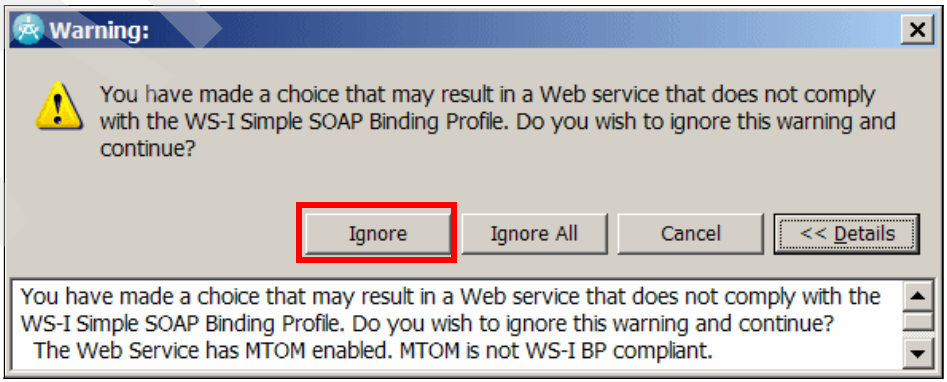


Figure 7-21 WS-I non-compliance warning

When the wizard completes, Rational Application Developer opens the Java editor on AttachmentPortImpl.java, which is the Java bean skeleton for the MTOM service. You can build whatever implementation you like.

- Example 7-14 shows the very simple implementation we use. Change the default null return value for sendXML and sendBinary to return their input parameter.

Example 7-14 AttachmentPortImpl implementation

```
package com.ibm.was.wssample.sei.mtom;
import javax.xml.transform.Source;
@javax.jws.WebService (
      endpointInterface="com.ibm.was.wssample.sei.mtom.Attachment",
      targetNamespace="http://com/ibm/was/wssample/sei/mtom/",
      serviceName="AttachmentService", portName="AttachmentPort",
      wsdlLocation="WEB-INF/wsdl/MTOM.wsdl")
@javax.xml.ws.BindingType (
      value=javax.xml.ws.soap.SOAPBinding.SOAP11HTTP_MTOM_BINDING)
public class AttachmentPortImpl{
    public Source sendXML(Source xml) {
        return xml;
    }
    public byte[] sendBinary(byte[] binary) {
        return binary;
    }
}
```

7.8.2 Generate the MTOM client

- Generate your MTOM client in the same manner as you generated your echo client into a POJO project (section 7.1.2, “Generate the echo client” on page 181). Choose the name “MTOMClient” (Figure 7-22) → **Finish**.

> **Note:** The URL for the MTOM WSDL is
>
> http://localhost:9080/MTOMService/AttachmentService?WSDL

We do not bother to check the box labeled **Enable asynchronous invocation for generated client** on the next panel. MTOM support does not add any new aspects to the asynchronous APIs, so there is nothing new to discuss with respect to asynchronous invocations.

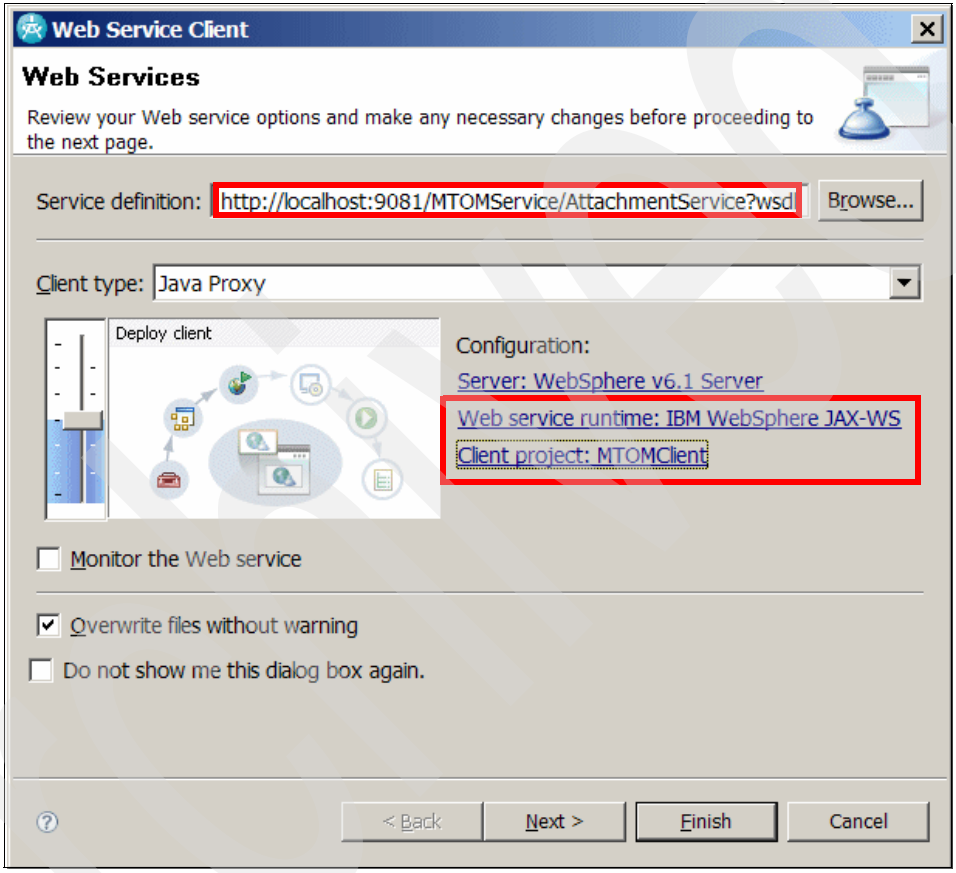


Figure 7-22 Generate MTOMClient into a POJO project

7.8.3 Examine the WSDL-to-Java mapping

You can examine the files in the project that you generated, or you could examine the projects in the file Chapter8Solutions.PI.zip, which we have provided. The examples presented in this section are excerpts from those files.

MTOM.wsdl contains two operations: sendXML and sendBinary. Figure 7-23 contains the XML schema snippet for the wrapper element types for these operations, and Figure 7-24 contains the Java mapped Java method signatures for these operations.

Note: The generated MTOM Java interface methods in Figure 7-24 are annotated in the generated code. We have removed the annotations for clarity.

```
<xsd:complexType name="sendXML">
  <xsd:sequence>
    <xsd:element
           xmlns:ns1="http://www.w3.org/2005/05/xmlmime"
           ns1:expectedContentTypes="application/xml"
           name="xml"
           type="xsd:base64Binary"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="sendBinary">
  <xsd:sequence>
    <xsd:element name="binary" type="xsd:base64Binary"/>
  </xsd:sequence>
</xsd:complexType>
```

Figure 7-23 MTOM schema

What Figure 7-23 shows you are two variations on how to define binary data. The simplest is sendBinary: Typing the elements of the sequence to base64Binary identifies the data as binary. The more elaborate definition, sendXML, in addition to identifying the elements of the sequence as base64Binary, uses the attribute expectedContentTypes to qualify the element definition and tell the SOAP engine the MIME type for this binary data that it should expect; in this example, the SOAP engine should expect “application/xml”.

```
package com.ibm.was.wssample.sei.mtom;

import javax.jws.WebMethod;

public interface Attachment {
    public Source sendXML(Source xml);

    public byte[] sendBinary(byte[] binary);
}
```

Figure 7-24 MTOM Java interface signatures

Figure 7-24 shows you how the element types in the WSDL are mapped to Java types. The simple form of binary data maps to Java simply as “byte[]”. The sendXML type is mapped to Source by virtue of being defined with an expectedContentType attribute of application/xml. What is Source, and how was that mapping chosen?

When you indicate an expected content type, the generation tools map that to a Java type defined in the JAXB specification. In this case, “application/xml” maps to “javax.xml.transform.Source”. The complete set of MIME expected type mappings to Java is shown in Table 7-2 (this table is extracted from the JAXB specification).

Table 7-2 JAXB MIME to Java mappings (from specification)

MIME Type	Java type
image/gif	java.awt.Image
image/jpeg	java.awt.Image
text/plain	java.lang.String
text/xml	javax.xml.transform.Source
application/xml	javax.xml.transform.Source
/	javax.activation.DataHandler

Note: All MIME type information in MTOM is in the type space, within the XML schema. This is unlike Sw/A, which places MIME information in the SOAP binding. You can see by examining the type that it is binary data and its MIME type.

7.8.4 Implementing an MTOM client

- Create the MTOM client as a new Java application called MTOMClientMain as in Figure 7-25 and add the code shown in Example 7-15.

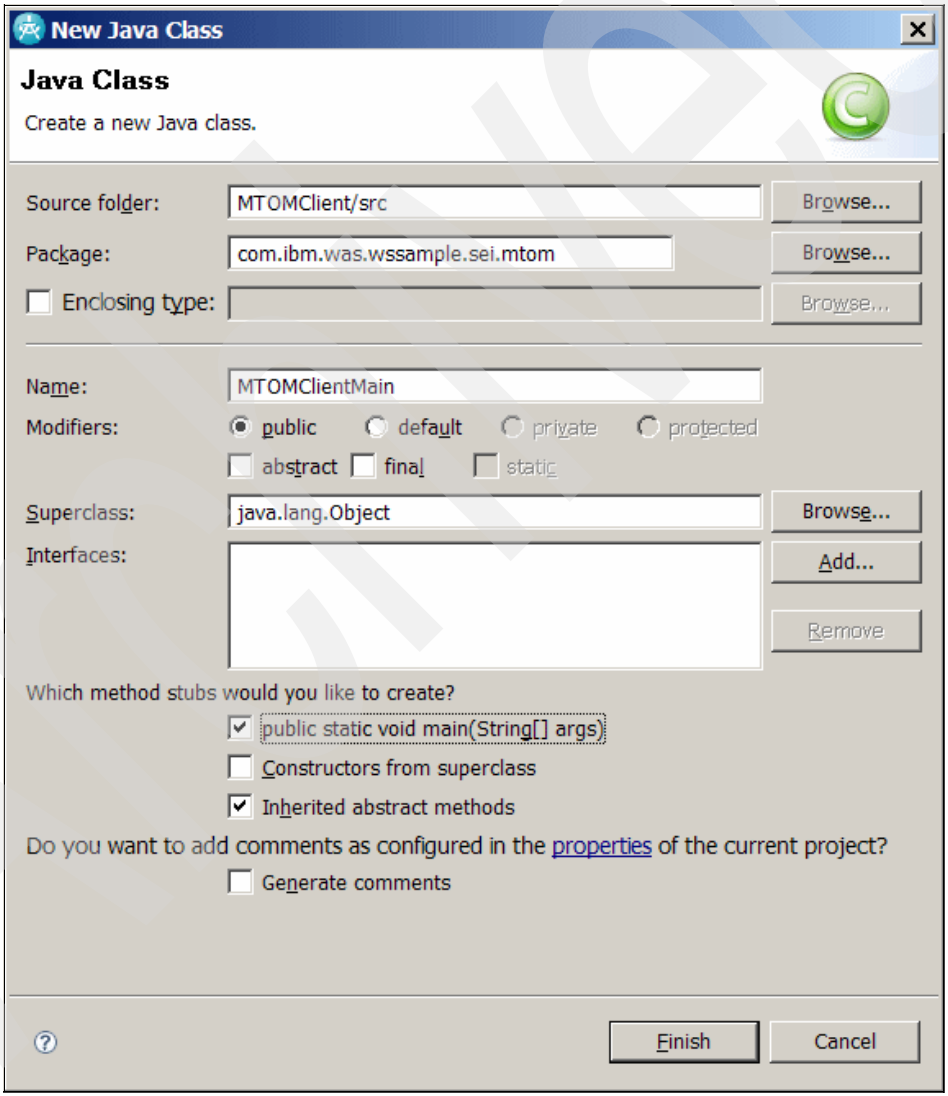


Figure 7-25 MTOM Client

Example 7-15 MTOMClientMain

```
package com.ibm.was.wssample.sei.mtom;
import java.io.InputStreamReader;
import java.io.StringReader;
import javax.xml.transform.Source;
import javax.xml.transform.stream.StreamSource;
public class MTOMClientMain {
   public static void main(String[] args) throws Exception{
      // Get an instance of the proxy class
      AttachmentPortProxy proxy = new AttachmentPortProxy();
      // Make the Web service call to sendBinary.
      System.out.println("sendBinary(\"Hello\") returned \""
            + new String(proxy.sendBinary("Hello".getBytes())) + "\"");
      // Make the Web service call to sendXML.
      Source ss = new StreamSource(new StringReader("<Hello/>"));
      ss = proxy.sendXML(ss);
      InputStreamReader reader = (InputStreamReader) ((StreamSource) ss)
            .getReader();
      char[] chars = new char[100];
      int length = reader.read(chars);
      System.out.println("sendXML(\"<Hello/>\") returned \""
            + new String(chars, 0, length) + "\"");
   }
}
```

We have kept the attachments *really* simple to focus on the code template. Consider the coding and explanation supplied in Figure 7-26.

When this code executes, the output is:

```
sendBinary("Hello") returned "Hello"
sendXML("<Hello/>") returned "<?xml version="1.0" encoding="UTF-8"?><Hello/>"
```

Tip: *"Exception in thread "main" java.lang.IllegalStateException: writeNamespace() can only be called following writeStartElement() or writeEmptyElement()"?* ***Turn the page...***

Note: You probably will not see this error the first time you use the Feature pack for Web services with a thin client, but perhaps now, when you are not so much focused on the configuration steps, you might get this sort of trace running a thin client. The trace is not very helpful in suggesting the nature of its cause. It is due to not configuring the thin client correctly — see 7.1.2, "Generate the echo client" on page 181.

```
Exception in thread "main" java.lang.IllegalStateException: writeNamespace() can only be called following
writeStartElement() or writeEmptyElement().
at com.ibm.xml.xlxp.api.stax.msg.StAXMessageProvider.throwIllegalStateException(StAXMessageProvider.java:45)
at com.ibm.xml.xlxp.api.stax.XMLStreamWriterBase.writeNamespace(XMLStreamWriterBase.java:514)
at com.ibm.xml.xlxp.api.stax.XMLOutputFactoryImpl$XMLStreamWriterProxy.writeNamespace(XMLOutputFactoryImpl.java:148)
at com.sun.xml.ws.encoding.soap.SOAPEncoder.startEnvelope(SOAPEncoder.java:467)
at com.sun.xml.ws.encoding.soap.client.SOAPXMLEncoder.toSOAPMessage(SOAPXMLEncoder.java:177)
at com.sun.xml.ws.protocol.soap.client.SOAPMessageDispatcher.doSend(SOAPMessageDispatcher.java:241)
at com.sun.xml.ws.protocol.soap.client.SOAPMessageDispatcher.send(SOAPMessageDispatcher.java:139)
at com.sun.xml.ws.encoding.soap.internal.DelegateBase.send(DelegateBase.java:86)
at com.sun.xml.ws.client.EndpointIFInvocationHandler.implementSEIMethod(EndpointIFInvocationHandler.java:174)
at com.sun.xml.ws.client.EndpointIFInvocationHandler.invoke(EndpointIFInvocationHandler.java:108)
at $Proxy14.sendBinary(Unknown Source)
at com.ibm.was.wssample.sei.mtom.AttachmentPortProxy.sendBinary(AttachmentPortProxy.java:85)
at com.ibm.was.wssample.sei.mtom.MTOMClientMain.main(MTOMClientMain.java:15)
```

Figure 7-26 Running the thin client with the server library

Everything worked, so we are done, right? Not exactly. A feature of MTOM is that it is an optional optimization mechanism. If a sender does not support MIME attachments, it need not send MIME attachments. The XML schema says that the data is "xsd:base64Binary", so that is all that the sender really must send. By default, Web services clients are not MTOM-enabled, so by default they send base-64 binary data inline within the SOAP message as shown in Example 7-16. This message is the request message of the sendBinary call.

Example 7-16 SOAP request message for sendBinary - no MTOM

```
POST /MTOMService/AttachmentService HTTP/1.1
Host: localhost
Accept: application/soap+xml,multipart/related,text/*
User-Agent: IBM WebServices/1.0
Cache-Control: no-cache
Pragma: no-cache
SOAPAction: ""
Connection: Keep-Alive
Content-Type: text/xml; charset=UTF-8
Content-Length: 270
Date: Sun, 02 Mar 2008 10:41:50 GMT

<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Body>
    <ns2:sendBinary xmlns:ns2="http://com/ibm/was/wssample/sei/mtom/">
      <binary>SGVsbG8=</binary>
    </ns2:sendBinary>
  </soapenv:Body>
</soapenv:Envelope>
```

To send binary data as a MIME attachment via the MTOM protocol, enable MTOM in the implementation as shown in Example 7-17.

Example 7-17 Code snippet to enable MTOM in the client

```
import javax.xml.ws.soap.SOAPBinding;
// ...
AttachmentPortProxy proxy = new AttachmentPortProxy();
// Enable MTOM
BindingProvider bp = (BindingProvider)
      (proxy._getDescriptor().getProxy());
((SOAPBinding) bp.getBinding()).setMTOMEnabled(true);
```

Once MTOM is enabled, the MTOM client sends SOAP messages with MIME attachments. The MTOM-enabled version of the sendBinary request message is shown in Example 7-18.

Example 7-18 SOAP request message for sendBinary - MTOM enabled

```
POST /MTOMService/AttachmentService HTTP/1.1
Host: localhost
Accept: application/soap+xml,multipart/related,text/*
User-Agent: IBM WebServices/1.0
Cache-Control: no-cache
Pragma: no-cache
SOAPAction: ""
Connection: Keep-Alive
Content-Type:
    multipart/related;
    boundary=MIMEBoundaryurn_uuid_89923BED9D6FF1F1B41204457564556;
    type="application/xop+xml";
    start="<0.urn:uuid:89923BED9D6FF1F1B41204457564557@apache.org>";
    start-info="text/xml"
Content-Length: 896
Date: Sun, 02 Mar 2008 11:32:44 GMT

--MIMEBoundaryurn_uuid_89923BED9D6FF1F1B41204457564556
content-type: application/xop+xml; charset=UTF-8; type="text/xml"
content-transfer-encoding: binary
content-id: <0.urn:uuid:89923BED9D6FF1F1B41204457564557@apache.org>

<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Body>
    <ns2:sendBinary xmlns:ns2="http://com/ibm/was/wssample/sei/mtom/">
      <binary>
        <xop:Include
            href="cid:urn:uuid:89923BED9D6FF1F1B41204457565308@apache.org"
            xmlns:xop="http://www.w3.org/2004/08/xop/include"></xop:Include>
      </binary>
    </ns2:sendBinary>
  </soapenv:Body>
</soapenv:Envelope>
--MIMEBoundaryurn_uuid_89923BED9D6FF1F1B41204457564556
content-type: application/octet-stream
content-transfer-encoding: binary
content-id: <urn:uuid:89923BED9D6FF1F1B41204457565308@apache.org>

Hello
--MIMEBoundaryurn_uuid_89923BED9D6FF1F1B41204457564556--
```

You can see that the content type is now multipart/related rather than simply text/xml. The first part is the SOAP message. The second part is the attachment — the string “Hello”. The SOAP message refers to the attachment in the xop:Include statement.

7.9 Change the address of the Web service in the client

WSDL was intentionally designed to be fully declarative. As well as describing what a Web service looks like, it describes where it is — in other words, it contains the address of the Web service. This address is a static part of the WSDL. However, in practice, this address is frequently not static. For example: Services move; messages to services flow through intermediaries. Client programs must be able to adapt to this dynamic nature of addresses.

Dynamic clients (which are similar to JAX-RPC DII clients) do not require a WSDL file at compile time or runtime — see 7.6.1, “Dynamic client API: The Dispatch class”. Using a dynamic client, we do not have to know where a service is, what it is called, or what its signature is until runtime. All the information required is supplied programmatically. But there is the rub. Dynamic clients are seriously more complicated than static clients and the complexity is only really justified when the identity or signature of the service is unknown at compile time. When it is just the address of the service that changes, we only want the address of the service the client is calling to be dynamic, while still benefiting from the simplicity of a static interface in all other respects.

Change the address of the service

To solve this problem, JAX-WS provides a method call similar to one in JAX-RPC, to change the address of the service to which the request is sent. We show you how to do this in 7.9.1, “Change the address of the service” on page 229.

This method works very well if, for example, the service has not changed its location, and we just want to send the SOAP message to an intermediary first. The intermediary might be the TCP/IP monitor in a development environment, or perhaps a gateway in a production environment.

The important word that keeps being repeated is *change*. We are changing the service address. This is unlike the similar call in a JAX-RPC client where we would be *setting* the service address. What are we changing the address from? We are changing it from the service address statically encoded in the WSDL. If you recall in 7.4, “The synchronous client API” on page 193, we pointed out a significant difference between JAX-RPC and JAX-WS static clients.

JAX-WS static clients require a copy of the service WSDL available at runtime but JAX-RPC clients do not. So in using the technique we describe in 7.9.1, “Change the address of the service”, although we have succeeded in being able to direct the message flow to a different address, we have not removed the requirement to know the static address of the WSDL to obtain the service description. We have simply changed the SOAP address in the WSDL, which is obtained from a fixed location.

Change the address of the WSDL

JAX-WS provides another method that changes the address where we obtain the WSDL from. Rather than instantiate the service proxy using the WSDL supplied at compile time by the developer, there is an alternative Service constructor that instantiates the service proxy using a WSDL location and a service name provided by the runtime client program. We show you how to do this in 7.9.2, “Change the address of the WSDL” on page 229.

Now both the location of the WSDL and the location of the service are dynamic. We would seem to be home dry on the requirement that, “Services move, messages to services flow through intermediaries. Client programs must be able to adapt to this dynamic nature of addresses” — without getting involved in the complexities of dynamic clients. But we have one more requirement to think about. Not only do services move, but clients move too. This is called the portable client problem.

Portable clients

The portable client issue has three aspects to it. Being able to route service requests through an intermediary, and using WSDL provided at runtime to instantiate the service proxy, are two aspects that we have already discussed. The final issue is more of an architectural question than of implementation: The client needs to get WSDL from somewhere to be able to run, but where should it get the WSDL from? For a client to be portable, we have to consider where to locate the WSDL. How do we make WSDL portable?

A reasonable initial reaction to this question is to think that JAX-WS has made clients less portable than JAX-RPC, by coupling the deployment of the client to the location of the WSDL; whereas JAX-RPC clients do not have to know the location of the WSDL used to define the service. In fact, the reverse is true, and JAX-WS clients are more portable. By providing the JAX-WS client with the location of the WSDL to use, rather than the location of the service to call, JAX-WS uncouples its clients from more of the details of the service description and integrates better with modern Web service deployment patterns. We describe some patterns in 7.10, “Deployment patterns” on page 230.

Problems with portable clients and Rational Application Developer

JAX-WS more portable than JAX-RPC, you ask? That is not the initial reaction of developers. Developers have built JAX-WS clients using Rational Application Developer by consuming WSDL from the service. But this requires the WSDL to remain at the initial service location even when the service itself moves. Using the procedure described in 7.9.1, “Change the address of the service” on page 229. to dynamically change where the client sends its service request does not work. See “Anti-pattern 1: Default service constructor with service WSDL” on page 238, for more explanation.

Developers have also built JAX-WS clients using Rational Application Developer from the WSDL file copied or generated into the service project. Again, this does not deploy without more work. See “Anti-pattern 2: Default service constructor with local WSDL” on page 239.

Oops! Early users of the Rational Application Developer JAX-WS support have discovered two problems that have naturally confused early developers:

- In Example 7-19, the @WebServiceClient annotation contains the address of the WSDL. If there is no WSDL at that address, the client shows a compile time error. The code still works fine. The problem here is that the error has masked a good deployment pattern (perhaps the best), which is to override the default WSDL location by using the `Service(WSDL, QName)` constructor described in “Change the address of the WSDL” on page 229. Consequently users have been casting around for other, but less desirable solutions.
- Another good deployment pattern (see “Pattern 1: Deployed WSDL” on page 232) is to import a copy of the service WSDL into the client project. The WSDL is then deployed with the project, and the address of the service is mutable using the procedure described in 7.9.1, “Change the address of the service” on page 229. Unfortunately the Rational Application Developer wizard to generate a JAX-WS client, unlike the wizard to develop a service, did not offer an option copy WSDL into the client project. The result has been users have created clients that work fine in development, but fail as soon as they are deployed onto another system.

Both these problems will be fixed in the near future (Figure 7-27). They are an indication that lots of thought needs to going into planning client as well as service deployment. Although without these problems, users would have experienced fewer problems, the Rational Application Developer fixes are not in themselves *the* solution to the portable client requirement. The fixes make it easier to implement the patterns described in 7.10, “Deployment patterns” on page 230.

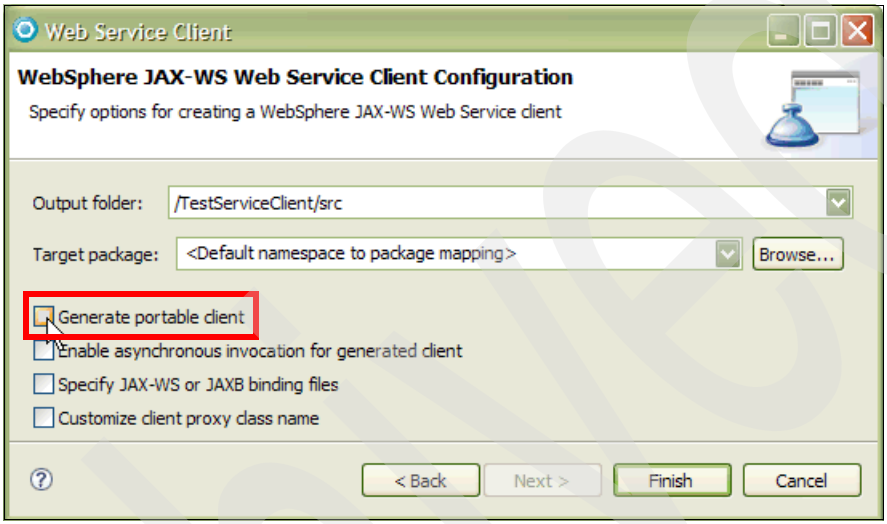


Figure 7-27 New option planned for JAX-WS client generation wizard

Example 7-19 EchoService generated code

```
@WebServiceClient(
   name = "EchoService",
   targetNamespace = "http://com/ibm/was/wssample/sei/echo/",
   wsdlLocation = "http://localhost:9081/EchoService/EchoService/echoservice.wsdl")
public class EchoService extends Service {
   private final static URL ECHOSERVICE_WSDL_LOCATION;
   static {
      URL url = null;
      try {
         url = new URL(
               "http://localhost:9081/EchoService/EchoService/echoservice.wsdl");
      } catch (MalformedURLException e) {
         e.printStackTrace();
      }
      ECHOSERVICE_WSDL_LOCATION = url;
   }
```

7.9.1 Change the address of the service

To change the address of the message flow, you set a property. Example 7-20 shows you how to do it for the echo client using the JAX-WS stub class. Example 7-21 shows you how to do it using the proxy.

Example 7-20 Change the address via the JAX-WS stub

```
EchoServicePortType stub = serviceFactory.getEchoServicePort();
BindingProvider bp = (BindingProvider) stub;
bp.getRequestContext().put(
      BindingProvider.ENDPOINT_ADDRESS_PROPERTY,
      endpointURL);
```

Example 7-21 Change the address via the proxy

```
proxy._getDescriptor().setEndpoint(endpointURL);
```

Setting the address via the proxy eliminates a lot of knowledge requirements from the programmer. The programmer does not have to know that the stub is really a BindingProvider, nor that it contains a request context, nor does the programmer have to know the key for the endpoint address property.

Note that by setting the address property without setting the address for service factory (as we describe in 7.9.2), the service factory will continue to query the original WSDL's address. If the WSDL no longer exists at that address, then the client will fail. It can be considered a best practice to set both addresses. If you do this, you do not have to consider the two aspects as separate aspects. You would merely see setting the address as two steps. That might be a simpler way of looking at the issue of changing the address.

7.9.2 Change the address of the WSDL

To change the address of the WSDL, you have to change it when you instantiate the service factory class. Example 7-22 shows how to do it for the echo client.

Example 7-22 Change the address via the service factory

```
EchoService serviceFactory = new EchoService(
   new URL(endpointURL + "?WSDL"),
   new QName("http://com/ibm/was/wssample/sei/echo", "EchoService"));
```

When you are using the proxy class, you give the proxy the necessary information instead of the service factory as shown in Example 7-23.

Example 7-23 Change the address via the proxy

```
EchoServicePortProxy proxy = new EchoServicePortProxy(
   new URL(endpointURL + "?WSDL"),
   new QName("http://com/ibm/was/wssample/sei/echo", "EchoService"));
```

Note that the WSDL URL does not have to refer to a remote Web service location. The URL could be of the form “file:/...”.

When you change the address of a WSDL, you should be fully aware of what you are doing. You are telling the SOAP engine to get a new WSDL to describe the service. That new WSDL’s address is used for subsequent Web service invocations (unless the code detailed in Example 7-19 on page 228 is implemented).

7.10 Deployment patterns

A major criticism of Web services from an Enterprise Architecture point of view is that the standard static Web services client is tightly coupled to its service. This makes development and simple deployments easy, but subsequent maintenance and administration difficult. Various techniques and products have been developed to reduce the coupling and simplify maintenance and management of services. See Chapter 3 in the Redbooks publication, *“Patterns: Implementing an SOA using an Enterprise Service Bus*,” SG24-6346 for a good discussion of client and server coupling, and how an SOA architecture resolves some of the weaknesses in Web services when used in an Enterprise architecture. The book can be found at:

http://www.redbooks.ibm.com/abstracts/sg246346.html

One source of the coupling is the encapsulation of the service contract into the standard static JAX-RPC Web services client. JAX-WS, in contrast, by decoupling the client from its service contract, enables a more flexible deployment model without the added programming complexity of a fully dynamic and interpretative client. Separation of the service contract from the client opens the door to managing the way clients attach to a service by modifying service descriptions, rather than by modifying the clients themselves. The performance of some client and server maintenance tasks is simplified into a standard administrative procedure rather than requiring a full development cycle.

Figure 7-28 shows four components that are commonly found in a services architecture, and which we use to describe some good and bad approaches to deploying JAX-WS clients and services.

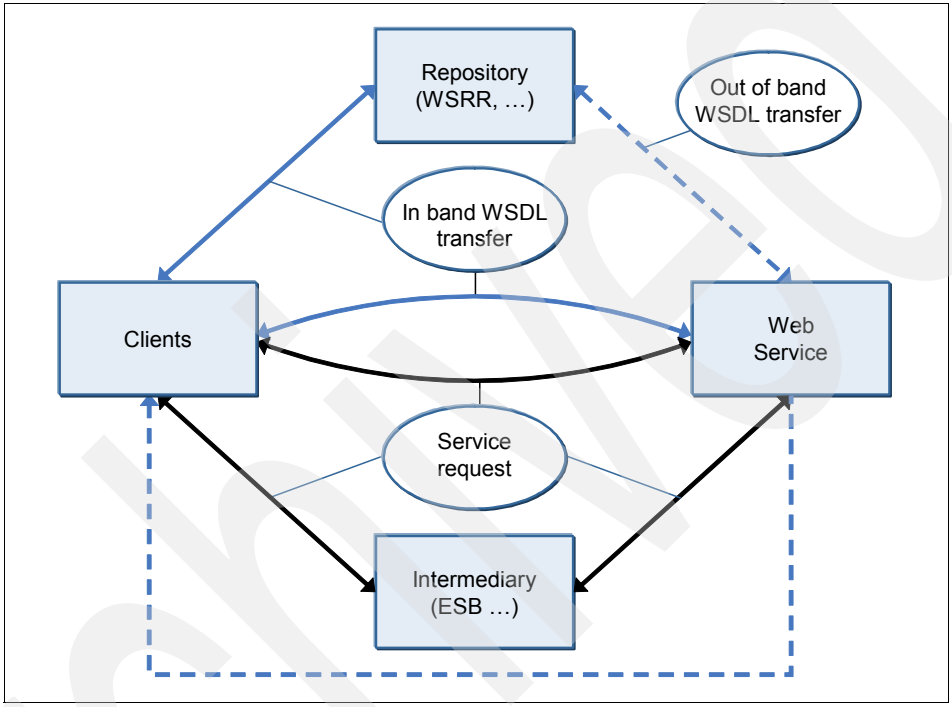


Figure 7-28 Components used in deployment patterns

The components have the following basic characteristics:

Clients JAX-WS static clients, can contain WSDL files

Web Service JAX-WS service, returns WSDL to *service QName?WSDL* query

Repository Returns WSDL to an HTTP *get* request

Intermediary Maps address of service,
(minimally) *host1:port1* → *host2:port2*

The three patterns we describe are:

1. Pattern 1: Deployed WSDL

   WSDL is deployed with the client.

2. Pattern 2: Service WSDL

   WSDL is dynamically requested from the service.

3. Pattern 3: Registry WSDL

   WSDL is deployed to a registry.

7.10.1 Pattern 1: Deployed WSDL

This pattern[13] includes the simplest and most widely used approach to client deployment. The WSDL is co-located with the client (Figure 7-29). It also performs the best, in that the WSDL is obtained locally, rather than across a network. It is just as flexible as the other approaches, but requires some administration at the client, and is more error prone if poor administration at the client results in the WSDL that the client is using becoming out of step with the service.

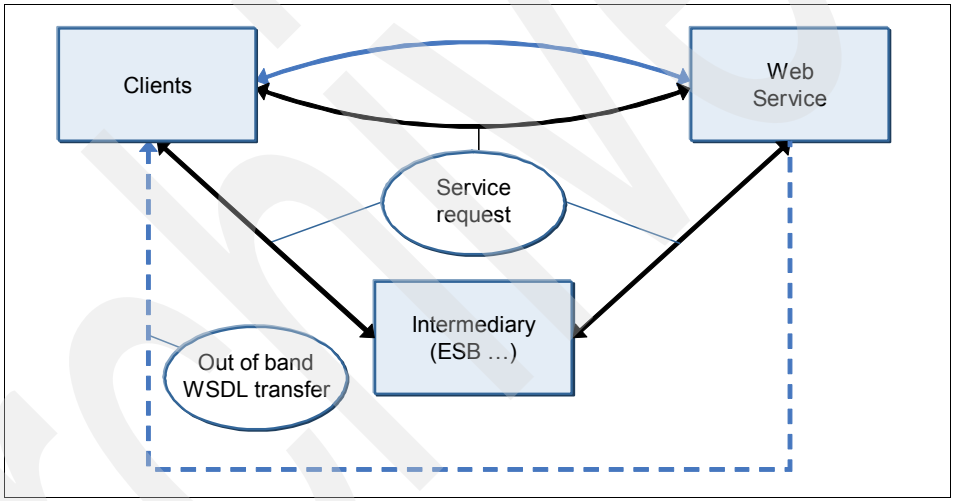


Figure 7-29 Components used in Pattern 1: Deployed WSDL

Build time

Create a project for the client; either generate a copy of the service WSDL and place it into the client project and then generate the client from the WSDL in the project, or, when the Rational Application Developer fix for copying WSDL into the client project is available, generate the client with WSDL copied into the client project.

Use the default Service constructor to create a service proxy.

Runtime

There are three runtime variations, allowing us to change the location of the service. Variation 3 also allows us to change the QName of the service. Note that Pattern 1: Runtime variation 2 is only really appropriate for debugging.

[13] This is the pattern that is automated by the “portable client” option to be introduced to the Rational Application Developer JAX-WS client generation wizard

Pattern 1: Runtime variation 1

Provide the address of the service at runtime and use the code described in 7.9.1, “Change the address of the service” route the service request to the server running the service.

Pattern 1: Runtime variation 2

Provide the address of the intermediary at runtime and use the code described in 7.9.1, “Change the address of the service” route the service request to the intermediary.

Pattern 1: Runtime variation 3

Deploy a new copy of the WSDL to the clients (“out-of-band”) overwriting the original copy.

Using this approach, you can change both the QName of the service and the address of the service without redeploying the client, using the simple default Service Constructor, and not loading the service address from anywhere else.

> **Tip:** We would recommend variation 3 as exhibiting the simplest separation of configuration from code. Variations 1 and 2 introduce an additional dependency between the configuration and the client code that has to be recognized in any subsequent relocation of the service.

Note that in all these three variations, the SOAP address contained in the WSDL might be the address of an intermediary, and not the service itself. Therefore Pattern 1: Runtime variation 2 is only needed for debugging.

7.10.2 Pattern 2: Service WSDL

The client, in the Service WSDL approach, gets its WSDL directly from the service (Figure 7-30). This pattern is slightly more complicated to program than the other two, to the extent of providing two additional parameters on the Service constructor at the client. The payback is that it is a robust approach, in the sense there is no danger that the WSDL used by the client gets out of step with the WSDL at the server. However, it has two main downsides:

- It still requires you to deploy the address of the service (and the QName if you so wish) to the client, so in this respect, it is no better from an administration perspective than all the variations on Pattern 1: Deployed WSDL.

- Unlike Pattern 1: Runtime variation 3 on Pattern 1: Deployed WSDL, it does not allow you to define an arbitrary intermediary by setting the SOAP address in the WSDL — the SOAP address in the WSDL is going to be the address of the service, unless the intermediary is a Web services router, such as the Web services Gateway or WebSphere Enterprise Service Bus, which responds correctly to a *?wsdl* query.

Of course, one can use the approach described in "Change the address of the service" on page 229 to route through an intermediary as a temporary expedient (for debugging say). But adding an additionally deployed SOAP address as part of the deployment architecture reintroduces another dependency on administration at the client.

Pattern 2: Service WSDL is almost an anti-pattern when compared to Pattern 1: Deployed WSDL, because it couples the service address to the service location. Its main redeeming feature is that if the client is poorly administered, and the deployed service address is wrong, the client fails with a simple 404 error — whereas failures in Pattern 1: Deployed WSDL can be more obscure.

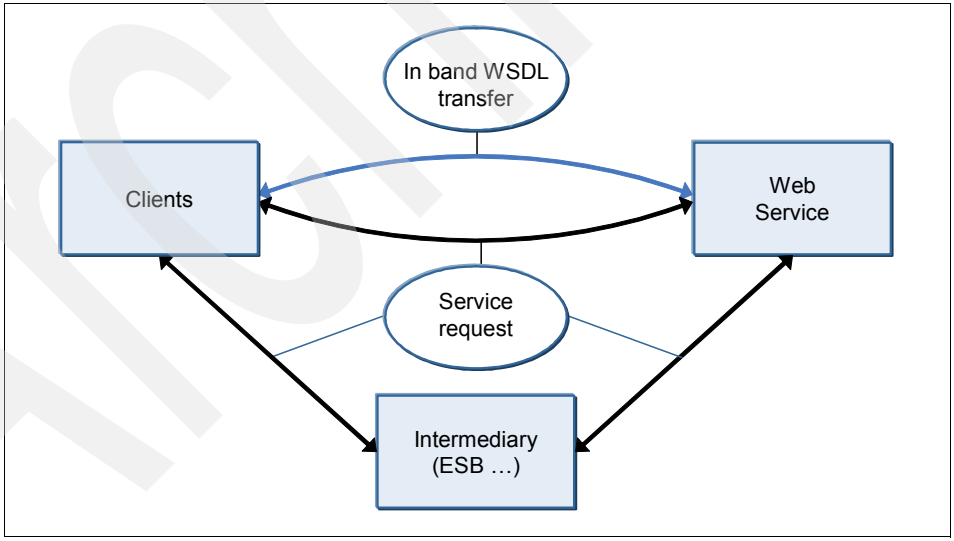


Figure 7-30 Components used in Pattern 2: Service WSDL

Build time

Create a client project, effectively using WSDL for the service from anywhere you like to generate the client, because the client does not have to refer to this WSDL at runtime. We might as well use the address of the service in our development environment.

Create the client Service proxy using the technique described in 7.9.2, “Change the address of the WSDL” on page 229, loading the URL endpoint from a parameter. Optionally load the QName from a parameter too.

> **Note:** Until Rational Application Developer 7.0.0.6 is fixed, if we move our development project to a workspace or machine that is set up differently, we might get a compile error in the generated client Service factory code. We can safely ignore the error.

Runtime

There are three variations to consider. Pattern 2: Runtime variation 3 is only really appropriate for debugging.

Pattern 2: Runtime variation 1

Provide the address of the service at runtime.

Pattern 2: Runtime variation 2

Provide the address of Web services router, such as WebSphere Enterprise Service Bus at runtime.

Pattern 2: Runtime variation 3

Combine Pattern 2: Runtime variation 1 or Pattern 2: Runtime variation 2, with using the code described in 7.9.1, “Change the address of the service” route the service request to the intermediary for debugging.

> **Tip:** We prefer Pattern 1 to Pattern 2: But if you can ensure the longevity of the location of the service or service intermediary from which the service is obtained, and if you want to retain the capability of modifying other parts of the WSDL that do not impact the client, then there are some merits to Pattern 2.

7.10.3 Pattern 3: Registry WSDL

The client gets its WSDL from a registry rather than from a service. The registry must be able to respond to a URL get request, with the WSDL for the service, for the pattern not to require additional client programming. Effectively this is a RESTful pattern.

Our final pattern, “Pattern 3: Registry WSDL”, is a step change on “Pattern 2: Service WSDL” from the perspective of simplifying client development and administration and managing the routing of service requests from clients. All the clients are the same, all use the default constructor, and there is no client administration (Figure 7-31).

If the clients have to behave differently, for example, to version a service and have some clients use the old service and some the new, or to insert a different intermediary for some clients, then that requires significantly more complex client programming to use the more sophisticated features of a repository.

The more complex variation is beyond the scope of this book, as it requires knowledge of programming to specific Web service registries.

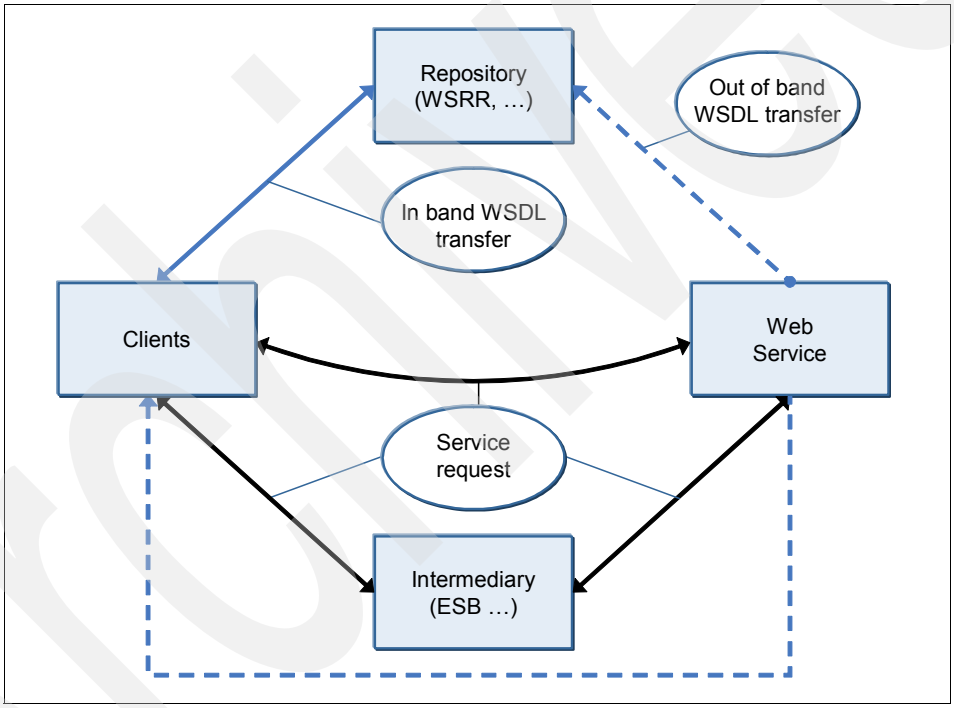


Figure 7-31 Components used in Pattern 3: Registry WSDL

Build time

There are two Build time approaches:, a simple RESTful implementation, and a more complex services design.

Pattern 3: Build time variation 1

Generate a JAX-WS client project using WSDL from repository using a RESTful query. For example, we used the following URL to return a WSDL from WebSphere Services Registry and Repository to the JAX-WS client:

`http://flail.hursley.ibm.com:9080/WSRR/6.1/Content/d66aaed6-49c5-457a.be2d.0993ac092dbb`

Create the service proxy instance using the default Service constructor.

Pattern 3: Build time variation 2

Build a client to the registry, and build a client to the service which instantiates its service proxy using the WSDL retrieved by the registry client.

Runtime

There are only two variations, and variation 2 is only really appropriate for debugging.

Pattern 3: Runtime variation 1

There is no change or configuration to the client. Any runtime variations, such as routing the service request through an intermediary, or changing the address of the service, are made by changing the contents of the WSDL at the registry.

Pattern 3: Runtime variation 2

Use the code described in 7.9.1, “Change the address of the service” route the service request to the intermediary for debugging.

Caveat: WebSphere Services Registry and Repository was not really designed be used as in Pattern 3: Build time variation 1. For this pattern variation to work, it is essential that the URL to retrieve the WSDL for the service does not change.

The way WSRR is designed to work, is to assign new ids (hence new URLs) to new versions of the service WSDL. This requires new clients, or deployment of new URLs to clients, to pick up the new version of the service, which defeats the purpose of using the registry.

WSRR is intended to work as in Pattern 3: Build time variation 2, but this requires considerably more client programming and knowledge of WSRR.

The work around is for the WSRR administrator to publish changes to WSDL by updating the documents that are made visible to clients, rather than reversioning them.

Tip: Currently we prefer Pattern 1 to Pattern 3, because of the additional complexity of coding to Pattern 3, and managing updating WSDL at the registry. The complexity issues should get resolved in time, making Pattern 3 a better choice than Pattern 1 in the future from a maintainability perspective.

7.10.4 Anti-patterns

There are at least two definite anti-patterns, that you should not normally use. Both involve using the default Service constructor, and result in clients that are tied to particular client or service addresses.

And as we have said, arguably the whole of “Pattern 2: Service WSDL” on page 233 might be regarded as an anti-pattern, because it does not allow for simple administrative routing of service requests by configuring resources at the server. But, it is more robust than the wholly client variations provided by “Pattern 1: Deployed WSDL” on page 232.

Anti-pattern 1: Default service constructor with service WSDL

In this anti-pattern, the client ends up tied to the service address used by the developer to attach to the service. In fact, this anti-pattern is, in a sense, the natural Web service pattern, and it might be thought a little perverse to call it an anti-pattern. In the standard Web services scenario, and part of the benefit of Web service architecture, is that client development is wholly separated from the development of, and knowledge about the service. The assumption is the service does not change and will not move. If these assumptions hold, then this is a satisfactory pattern.

It is more often the case within enterprises, that client and service deployment are not conducted in isolation, that services are being deployed at the same time as clients, and the deployment of clients and services changes, if only from the development environment through test to production.

Tip: For Internet scale applications, where zero client administration is a must and WSDL is expected to change, we would advise Pattern 3: Build time variation 1 rather than this anti-pattern.

Pattern 3: Build time variation 1is easy to implement without a registry product. Build a Web application directory that returns WSDLs for all your services using the <url>?wsdl form of http request.

Build time

The developer generates a JAX-WS client project using the URL of the service being used to test the client.

The default constructor is used to instantiate the service proxy.

Runtime

The deployed client uses the same service URL used to test the client.

Anti-pattern 2: Default service constructor with local WSDL

In this anti-pattern, the client picks up the WSDL of the service from either a colleague, or from another project (such as the service project) in the workspace being used to develop the client. No attention is paid to the location of the WSDL, it is used simply to generate the JAX-WS client.

Something can be said in defense of a developer taking this approach, but nothing can be said in defense of the approach itself! One might hope that the tool the developer is using would defend the developer against “shooting themselves in the foot.” And we have all “shot” ourselves, encouraged by this being a reasonable development approach for JAX-RPC clients.

Build time

The developer locates the WSDL for the service somewhere on the file system or within the workspace used to develop the client. The JAX-WS client project is generated directly from the WSDL.

Runtime

As long as the WSDL is still visible to the client at runtime, the client operates correctly. As soon as the client is moved onto another system, or a new workspace is created, or some other aspect of the environment upsets the path to the WSDL, the client breaks.

7.10.5 Deployment summary

We have labored heavily over describing the deployment of JAX-WS clients and their accompanying infrastructure. This is partly due to our own experience, and the experience of early customers of JAX-WS, who have not unlearned the practices that worked for JAX-RPC, and have fallen into one or other of our anti-patterns — ourselves, and perhaps the implementers of Rational Application Developer, included!

When one understands the role of the WSDL file in JAX-WS, the benefits of the JAX-WS client model become apparent. Closer integration of the JAX-WS client model registries will make deployment of manageable clients even easier.

7.11 The handler model

The purpose of the handler framework is to allow interception of a message at various points in its transmission. With outgoing messages, handlers are invoked before a message is sent out to the wire. With incoming messages, handlers are invoked before the receiving application receives the message. There are very

detailed rules about the order of handler invocation, particularly during fault handling. We do not cover those rules here. Read the JAX-WS 2.0 specification if you need those details.

JAX-WS provides two levels of handlers: logical and protocol. Logical handlers deal with the payload level of the message. Protocol handlers deal with protocol information, such as SOAP headers. Because manipulating SOAP headers is a common use of handlers, logical handlers often are not as useful as protocol handlers, but you can build handler logic that is common across protocols — such as logging — with logical handlers.

7.11.1 Logical handler

Example 7-24 shows a logical handler skeleton. The most important 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 a response message contains a fault. There might be very important code in those methods, but the handleMessage method is the main part of the handler.

Example 7-24 LogicalHandler skeleton

```
package com.ibm.handlers;
import javax.xml.ws.handler.LogicalHandler;
import javax.xml.ws.handler.LogicalMessageContext;
import javax.xml.ws.handler.MessageContext;
public class MyLogicalHandler implements LogicalHandler<LogicalMessageContext> {
   public void close(MessageContext arg0) { }
   public boolean handleFault(LogicalMessageContext arg0) { return false; }
   public boolean handleMessage(LogicalMessageContext arg0) { return false; }
}
```

The handleMessage method (and handleFault) return a boolean. Returning true tells the engine that processing should move to the next handler in the chain. Returning false tells the engine that processing of the handler chain should end.

The parameter to handleMessage is a LogicalMessageContext. It is an extension of java.util.Map, and it contains <key, value> pairs of context properties. There are properties for such things as WSDL element names and attachment information (if any). One important property is the direction of the message — inbound or outbound:

```
Boolean outbd = arg0.get(MessageContext.MESSAGE_OUTBOUND_PROPERTY);
```

Another important property is the message itself:

```
LogicalMessage lmsg = arg0.getMessage();
```

Get the payload either as XML data using the javax.xml.transform.Source:

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

Or, get it as JAXB objects:

```
Object jaxBPayload = lmsg.getPayload(jaxbContext);
```

In either case, what we get is 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).

7.11.2 Protocol handler

The only protocol handler that the feature pack supports is the SOAP handler. Example 7-25 shows an example of a SOAP handler skeleton.

Example 7-25 'SOAP Handler skeleton

```
package com.ibm.handlers;
import java.util.Set;
import javax.jws.HandlerChain;
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;
@HandlerChain(file = "MyHandlers.xml")
public class MySOAPHandler implements SOAPHandler <SOAPMessageContext>{
   public Set<QName> getHeaders() { return null; }
   public void close(MessageContext arg0) { }
   public boolean handleFault(SOAPMessageContext arg0) { return false; }
   public boolean handleMessage(SOAPMessageContext arg0) { return false; }
}
```

It has the familiar methods: close, handleFault, handleMessage. The parameter to handleFault and handleMessage, however, is now SOAPMessageContext instead of LogicalMessageContext. There is a new method to implement, getHeaders.

The getHeaders method returns the set of the header names that the handler understands. The engine calls this method to determine SOAP mustUnderstand processing. 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, and the getMessage method returns a SOAPMessage, which is an SAAJ class. From the SOAPMessage class we can traverse the entire SOAP message. The primary reason for writing a SOAP handler is to manipulate SOAP headers. But the SAAJ APIs are not pretty. SOAPMessageContext provides a convenience method that allows us to get SOAP headers as JAXB objects rather than SAAJ objects:

```
Object[] headers = arg0.getHeaders(qName, jaxbContext, allRoles);
```

7.11.3 Handler deployment

To deploy a handler, it must be listed in a handler chain. To deploy the handler chain, it must be listed, as an annotation, in the Java implementation:

```
@HandlerChain(file = "MyHandlers.xml")
```

MyHandlers.xml must reside in the same directory as the Java implementation. Example 7-26 shows an example of this handler chain file.

Example 7-26 MyHandlers.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<jws:handler-chains xmlns:jws="http://java.sun.com/xml/ns/javaee">
  <jws:handler-chain name="MyHandlerChain">
    <jws:protocol-bindings>##SOAP11_HTTP</jws:protocol-bindings>
    <jws:handler>
      <jws:handler-class>
        mysample.MyLogicalHandler
      </jws:handler-class>
    </jws:handler>
    <jws:handler>
      <jws:handler-class>
        mysample.MySOAPHandler
      </jws:handler-class>
    </jws:handler>
  </jws:handler-chain>
</jws:handler-chains>
```

There is much more detail about handlers than we have covered here. If you have to know more, we suggest reading in the InfoCenter[14] and the JAX-WS specification itself.

[14] See the following URL:
http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/index.jsp?topic=/com.ibm.websphere.wsfep.multiplatform.doc/info/ae/ae/twbs_jaxwshandler.html

7.12 SOAP 1.2

The Web service examples in this chapter have been SOAP 1.1 examples. They could just as easily have been SOAP 1.2 examples by changing the soap prefix to point to a different namespace. A SOAP 1.1 WSDL's definitions element has the attribute:

```
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
```

A SOAP 1.2 WSDL's definitions element has the attribute:

```
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap12/"
```

7.12.1 Issues

Although SOAP 1.2 is becoming popular, it is not yet fully integrated with the tools nor with the industry as a whole. Microsoft have made it the default binding in Windows Communication Foundation for the WSHttpBinding class that supports distributed transactions and secure, reliable sessions[15].

WS-I does not address SOAP 1.2

Be aware that SOAP 1.2 is not yet discussed in any WS-I profile. The profiles simply have not yet caught up to SOAP 1.2. This means that, when you run a SOAP 1.2 WSDL through the tools, you will be warned that the WSDL is not WS-I compliant. If you want to use SOAP 1.2 Web services, you must accept that this is the situation at this point in time and simply tolerate this particular non-compliance.

The Web services explorer

At the time that this book was written, the Web services explorer did not support SOAP 1.2 WSDL. It is a known limitation.

The WSDL editor

The WSDL editor nominally supports SOAP 1.2. You can use the properties view to change the soap prefix's namespace. But when you do, the visual presentation of the service's port's address disappears. The GUI does not recognize that namespace to know that it contains an address field that is displayable.

[15] The BasicHttpBinding class provides classes compatible with Web Services Interoperability organization BP 1.1, and hence uses SOAP 1.1 in the Windows Communication Foundation class library.

7.13 Deployment

This section exists for those people familiar with the JAX-RPC/JSR-109 deployment model. When they look at a JAX-WS deployment, it will seem that something is missing. Something *is* missing, but that is a good thing. In this section, we explain why.

Figure 7-32 shows the deployment process for a JAX-RPC Web service. As the EAR file deploys, the JAX-RPC engine examines the Web service deployment descriptors to gather detailed information about the Web service.

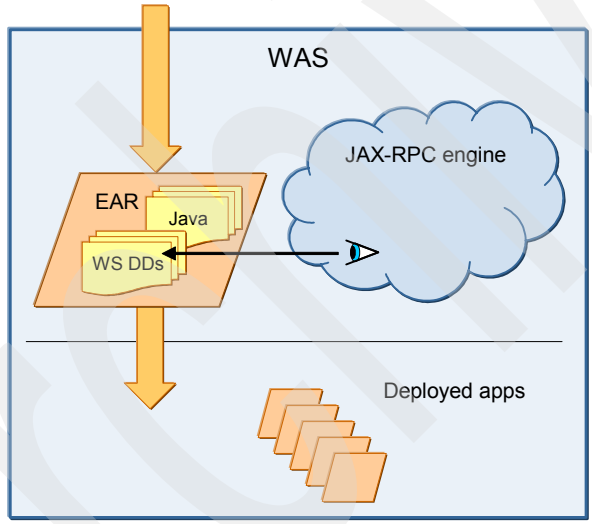


Figure 7-32 JAX-RPC deployment process

The deployment model for JAX-WS is different than it is in JAX-RPC, as shown in Figure 7-33. There are no Web service deployment descriptors! Instead of examining deployment descriptors, the JAX-WS engine introspects on the Java implementation itself, examining the annotations for Web service information.

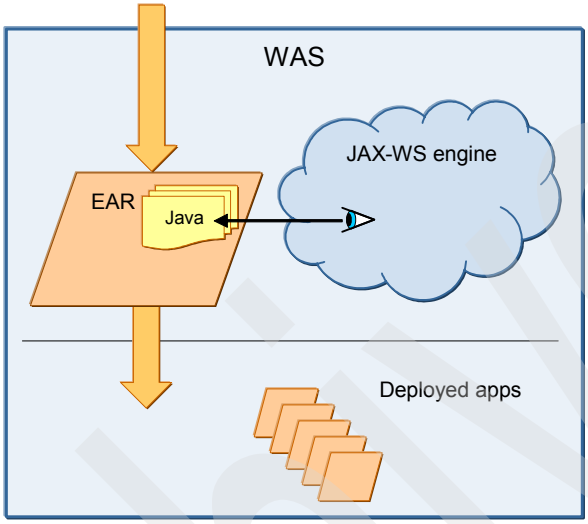


Figure 7-33 JAX-WS deployment process

In a JAX-RPC environment, if there were deployment issues, they would be resolved by examining the deployment descriptors. In a JAX-WS environment, we must examine Java annotations.

Another thing that could be missing is the WSDL file. Annotations contain all the Web service-relative information, and WSDL is constructed from the Java code dynamically if required (for example, by asking for it using the “<service URL>?WSDL” string). If you do not check the “Copy WSDL to project” box in the creation wizard (see Figure 7-4 on page 179), the project will not contain a WSDL file.

Because a JAX-WS Java code generator generates fewer files, the deployed footprint is smaller and it is simpler to deal with. If you are used to the JAX-RPC files, this might take a little getting used to.

7.14 Summary

In this chapter we have walked you through a number of detailed aspects of the feature pack’s JAX-WS implementation. But this chapter is not a JAX-WS handbook. It does not cover all of the details. Where appropriate, we have provided links to more details for those who want to get even deeper into the implementation of JAX-WS Web services. The best place to start is often the Info Center for the feature pack.

8

Policy sets

In this chapter, we explore the Policy set feature introduced in Feature pack for Web services. Policy set usage greatly reduces the amount of configuration that must be done for Web Services by providing reusable configurations.

First, we describe the motivation to use Policy set and the Policy set definitions. Then we guide you through some samples to administer the Policy set in WebSphere Application Server. This includes applying Policy set to your Web services using a default Policy set and a custom Policy set. We also show you how to configure a custom binding for your Web service. Finally, we explore the Policy set tools support in Rational Application Developer.

8.1 Motivation

The motivation for using Policy sets is threefold:

1. 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 that is most simply tackled in a separate step from writing interfaces and implementing the clients and services.

2. Multi-vendor environment

   The configuration task is made all the more daunting in a multi-vendor environment, with the necessity to match client and server configurations. Using 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 multi-vendor environment. The second step is to standardize the way the configuration is expressed (such as WSDL, for example), so that configuration files are exchangeable.

3. Development and management of qualities of service

   As described in our business scenario in 3.4, “Outsourced development” on page 46, separating Policy sets from the services is preferable from 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 services themselves.

8.1.1 Managing Web services configuration complexity

Working with Web Services has certain difficulties; one particular concern 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 have been few defaults, and configuration data had to 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 contains multiple services. Each service exposes multiple endpoints. And finally, each of those endpoints has one or more operations associated with it. This can lead to a large number of configuration points for a single application, as seen in Figure 8-1:

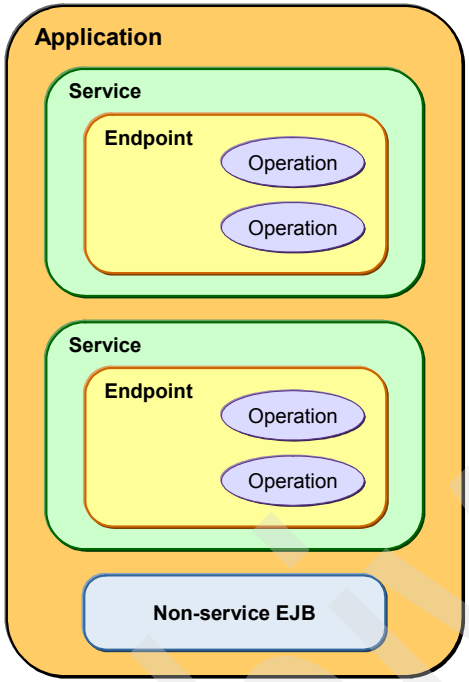


Figure 8-1 Configuration points for a Web service application

In previous releases of WebSphere Application Server, each service had its own quality of service separately defined. In many cases, quality of service configurations were not reusable and you had to repeat the configuration for each service, which is error-prone and difficult to manage.

As more and more qualities of service get added to the Web service with additional Web services specifications, such as WS-Addressing, WS-ReliableMessaging, and WS-Security, there is a requirement to be able to manage more information about the Web service effectively.

In addition, there are relationships between the various qualities of service. For example, to prevent sequence attack against WS-ReliableMessaging, you have to use WS-SecureConversation to establish a secure context between the two parties. Managing the individual configuration of each set of qualities of service is a daunting task.

Managing a single set of configurations for these qualities of service is a much simpler model, especially if you can relate these configuration groupings to a well-defined name, and reuse it in different services and across multiple application servers. Managing the combination of WS-ReliableMessaging and WS-SecureConversation to secure WS-ReliableMessaging headers is a good example of how combining related policies in a single Policy set helps to make management of configurations easier. See 9.4, “Secure conversation example” on page 336 for an example.

8.2 Policy set overview

The Feature pack for Web services introduces a new method of configuring and applying quality of service to deployed JAX-WS service providers and service clients using Policy set*s*. Policy sets are aimed at reducing the complexity of configuring Web services in WebSphere Application Server. By providing reusable configurations, administrators are able to 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.

8.2.1 Qualities of service

With the widespread adoption of Web services as the key technology for implementing a service-oriented architecture, quality of service has become a major priority for service providers and consumers. Qualities of service cover a whole range of requirements that match the requirements of service consumers with those of the service provider's. This includes reliability, security, accessibility, availability, interoperability, performance, transaction, management, and so on.

8.2.2 Policy set definitions

Figure 8-2 illustrates the elements of Policy sets, which we describe next.

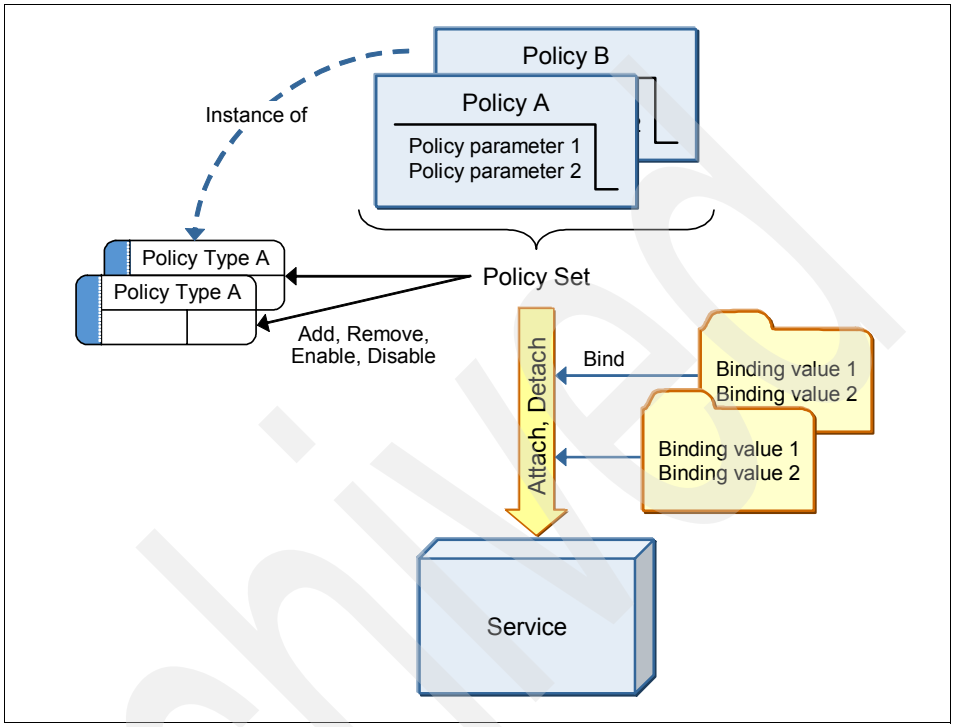


Figure 8-2 Policy sets

Policy type

A policy type is a single type of quality of service defined by a set of assertions. Policy types can also be defined based on specific Web Services standards. Examples of policy types include:

- 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 from the Organization for the Advancement of Structured Information Standards (OASIS) standards.

Policy

A policy is a named, configured instance of policy type. A policy does not include environment or platform specific information such as key for signing, key store information, persistent store information and so on. These types of information are defined in the binding.

Binding

A binding is the specific configuration of a quality of service. It contains the environment and platform specific information, such as keys for signing, key store information, or persistent store information, which indicates they are not normally shared. The Feature pack for Web services includes a set of preconfigured Policy sets and bindings. If no binding for a Policy set is specified, a default binding is used for that Policy set (except for the thin client which has no access to the default bindings)[1].

Policy set

A Policy set is a collection of policy types that are configured and associated with a given Web service provider or requestor. A Policy set is either pre-defined 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 Asynchronous Messaging Profile (RAMP) 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.

The Feature pack for Web services ships with various Policy sets already defined, that can be used out of the box. Examples of Policy sets include:

- WSAddressing default
- Username RAMP default
- Username SecureConversation
- WSHTTPS default
- WSReliableMessaging persistent

Policy sets in the Feature pack for Web services are not based on the WS-Policy specification, but are instead specific to WebSphere Application Server.

[1] Note that if you are using WS-Security, you will want asymmetry in the keys used by the client and server and so you cannot use the defaults for the WS-Security policy on both the client and server; one must be customized.

8.2.3 Using Policy sets

After Policy sets are created, they are associated with bindings that tailor specific details about the policy to the application (Figure 8-3). It is the combination of the Policy set and the binding that is applied to Web services or their clients.

Policy set configurations are stored at a cell wide level, but binding data is application specific.

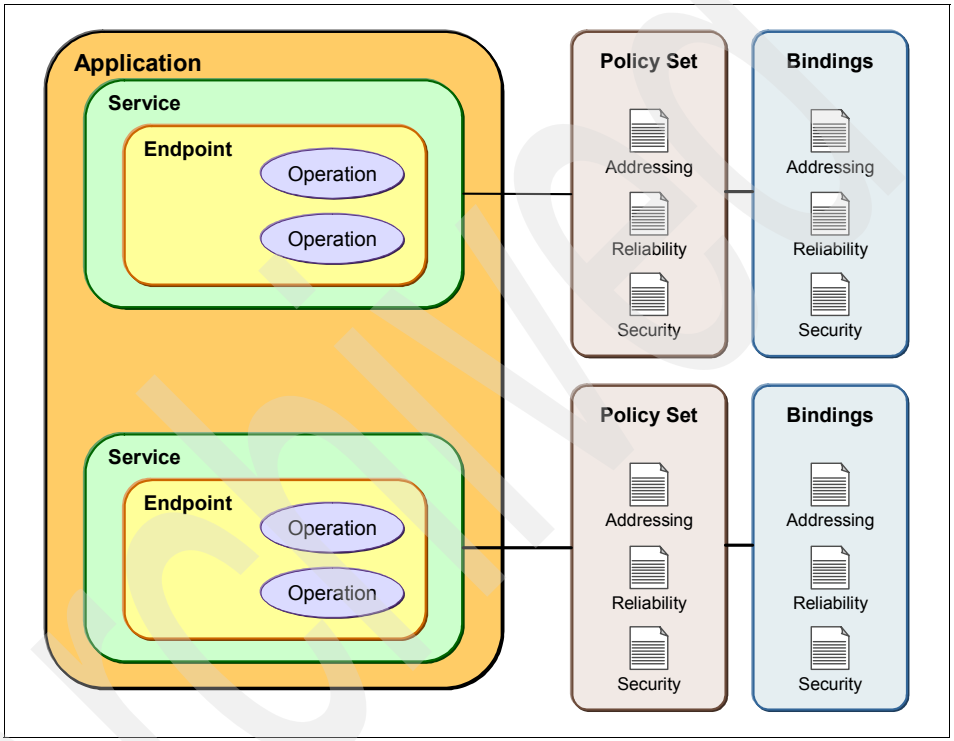


Figure 8-3 Attach Policy sets and bindings to Web services

8.3 Policy set administration

This section explains how to manage and administer Policy sets in a WebSphere Application Server V6.1. We use examples to show you how to perform the following tasks,

- "View Policy sets" on page 254
- "Attach a Policy set to a Web service" on page 257
- "Use a customized Policy set" on page 266
- "Configure the custom binding" on page 272

First we briefly discuss the life cycle of a Policy set.

8.3.1 Policy set life cycle

We strongly recommend that you create new Policy sets only by copying existing ones, rather than starting from scratch. You then modify the copies to configure them to your quality of service requirements.

You attach a Policy set to an application, service, endpoint, or operation. You can do this either at deployment or after an application has been deployed. When a Policy set is attached, the application must be restarted in order 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; and an endpoint-level attachment is inherited by all child operations.

Existing User Policy sets are modifiable, but be cautious about unintended side-effects, which is why we recommend taking a copy of a Policy set and working with the copy. With an unattached Policy set, such as a new copy, there is little to be concerned with. However, modifying an attached Policy set alters the configuration for a deployed application, although the changes will not be made until the application is restarted.

If the changes had never been intended to affect a particular application, and the application is not restarted for days or weeks, perhaps on a different shift, then the possibility for confusion is strongly present. To help alert you to the danger, a warning does let the administrator know that specific endpoints will be affected. After changes have been made, the associated application has to be restarted to pick up the changes.

User Policy sets are deletable. An attached Policy set first must be removed from any associations with applications. In order to accomplish this, the applications must be stopped, and then have the Policy set removed from their configuration. Only when the Policy set is not in use, is it deletable.

8.3.2 View Policy sets

To view Policy sets from the administrative console, do the following tasks:

- Start the WebSphere Application Server that uses the Feature pack for Web services enabled profile.

- ☐ Start the WebSphere Administrative Console by selecting **Start** → **All Programs** → **IBM WebSphere** → **Application Server v6.1** → **profiles** → **Web services feature pack enabled** profile → **Administrative Console**. Alternatively, you can open a browser and go to http://localhost:port/ibm/console. Replace the port with the console port number that your WebSphere Application Server is running on — for example, 9062.

> **Note:** If you are using the WebSphere Application Server shipped with Rational Application Developer, you can right-click the server in the Servers view and select **Run Administrative Console**.

- ☐ Click **Services** → **Policy sets**.

  There are two types of Policy sets listed: Application Policy sets and System Policy sets.

  Application Policy sets are used by application resources. System Policy sets are used by system resources the Security Token Service. We describe the System Policy sets in detail in Chapter 9, “Secure conversation” on page 309.

  Default Policy set bindings are also provided and listed here to minimize the configuration required by Policy sets.

- ☐ Click **Application Policy sets**, as seen in Figure 8-4.

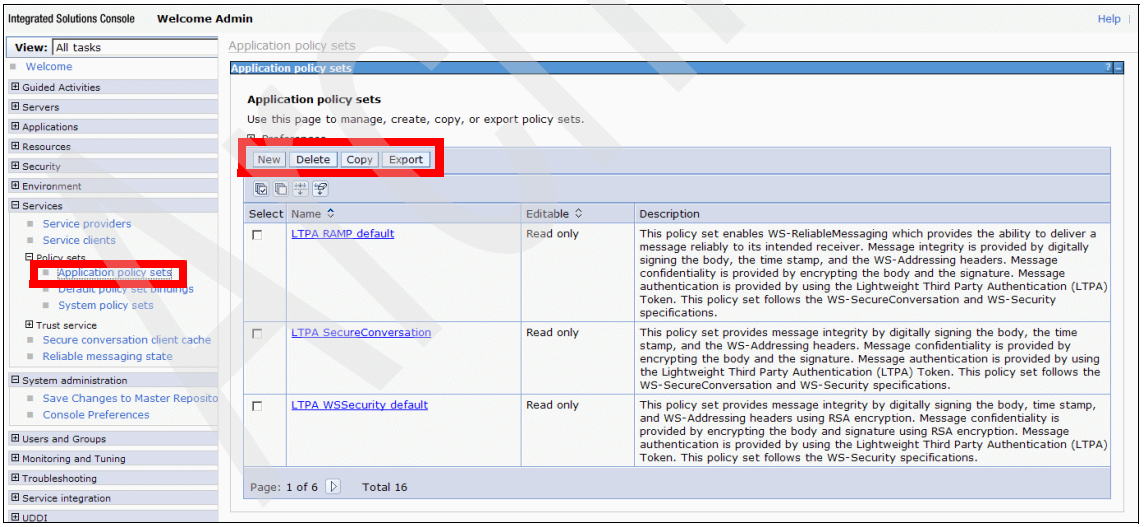


Figure 8-4 Application Policy sets

The available Policy sets are listed in this panel, including a description of the qualities of service provided by the Policy set. Using this panel, we create, copy, delete, and export Policy sets — obviously we are only allowed to modify user Policy sets. If the Policy set is a user Policy set, the fields on the page are editable. If the Policy set is provided as a default, then the Policy set cannot be changed.

Review the list of the Policy sets

Default Policy sets are provided with the Feature pack for Web services, this allows users to configure most qualities of service out of the box quickly. For example, the Username RAMP default Policy set enables WS-ReliableMessaging, WS-Addressing, and WS-Security. Look at the Username RAMP default Policy set,

- Click **Username RAMP default** on page 2 of the panel displayed in Figure 8-4 on page 255.The result is shown in Figure 8-5.

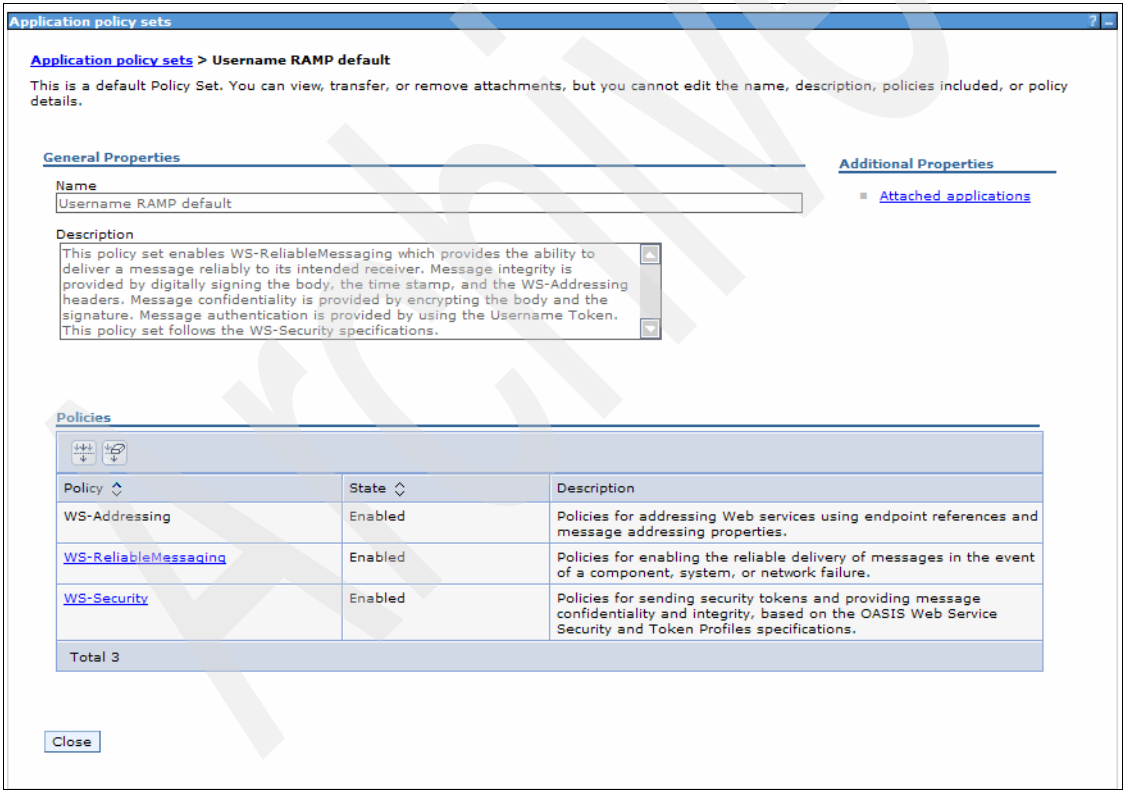


Figure 8-5 Details of Username RAMP default Policy set

The Username RAMP Policy set consists of the WS-Addressing, WS-ReliableMessaging and WS-Security policy types. The WS-Addressing policy type is not customizable, hence it is not selectable.

8.3.3 Attach a Policy set to a Web service

In this section, we walk you through an example applying the WS-Addressing Policy set. We also monitor the SOAP messages to confirm the WS-Addressing Policy set is successfully applied.

Prepare for the sample

WebSphere Application Server ships a set of JAX-WS samples that demonstrate simple message exchange patterns using both synchronous and asynchronous programming models. In this example, we apply the WS-Addressing Policy set to the Service Endpoint Interface (SEI) sample application shipped with WebSphere Application Server. To install the sample, complete these steps:

- ☐ Start WebSphere Application Server, if it is not already running.
- ☐ Navigate to `%WS_HOME%\samples\lib\WebServicesSamples` directory → Double-click **installapps.cmd** to install the applications.

> **Note:** It is very likely that you are not running the default WebSphere Application Server configuration. You have to supply command line parameters to installapps.cmd, or edit the script. Find your server parameters on the runtime page of the server page in the Administrator Console.
>
> The parameters to installapps.cmd are:
> ```
> installapps [profile] [cell] [node] [server] [-user user -password password]
> ```
>
> Example arguments we used are:
> ```
> installapps ws01 itso-01Node02Cell Node02 server1 -user Admin -password itso4you
> ```

Run the sample without applying the WS-Addressing policy

To run the sample, do the following steps:

- Open a Web browser and open the page with the URL (Figure 8-6):

  `http://<hostname>:port/wssamplesei/demo`

  For example, if your server is running at port 9081, then the URL is:

  `http://localhost:9081/wssamplesei/demo`

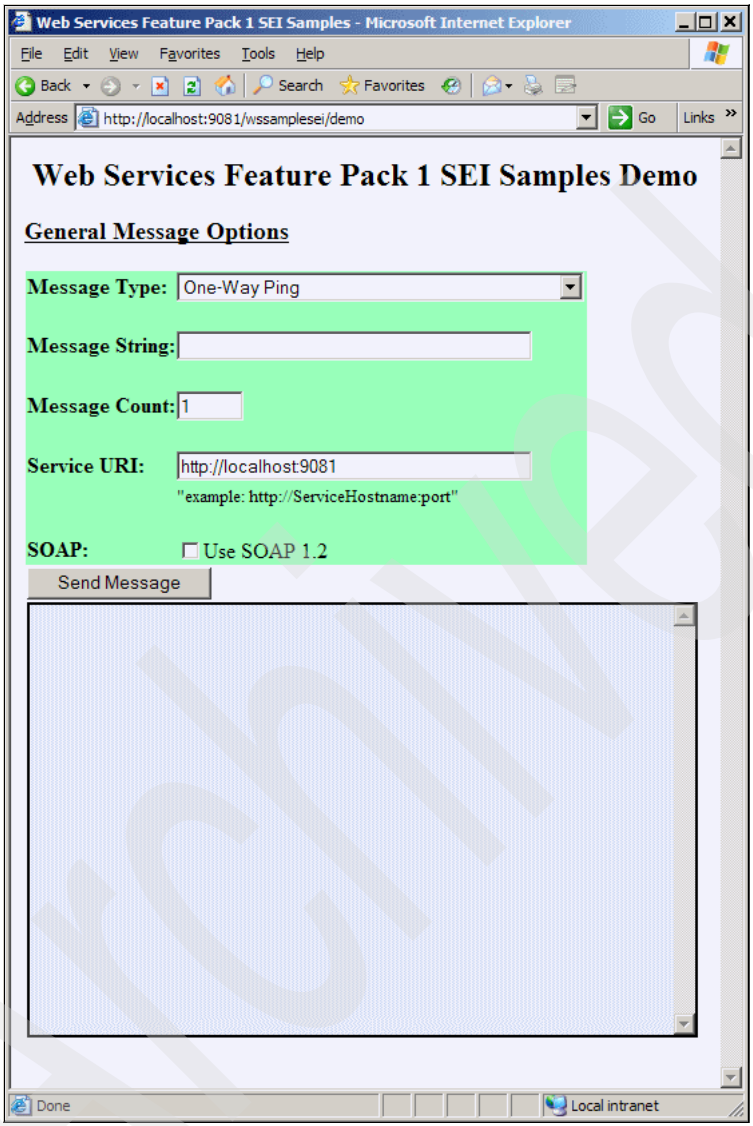


Figure 8-6 SEI Sample application

- *Message Type* lists the available message exchange patterns.
- *Message String* specifies the text to be sent to the service.
- *Message Count* specifies how many times the message exchange pattern is executed.
- *Service URL* is the service endpoint for the Web services.
- *SOAP* checkbox determines whether the SOAP 1.2 or SOAP 1.1 is used for the message transmission.

- Select **One-Way Ping** as the Message Type → type `Hello` as the Message String → **Send Message**.

  You should see the output, similar to Figure 8-7.

```
Connecting to... http://localhost:9081

Message Request:



Message Response:
Message delivered successfully. Please check server logs to
confirm message delivery.
```

Figure 8-7 One-Way Ping result

The server log is in:

```
%WebSphere Install Directory%\profiles\%Feature Pack enabled
profile%\logs\server1\SystemOut.log
```

If you are using the Rational Application Developer, then the server log is displayed in the console. Simply run the client from a browser, as before. Example 8-1 shows the message delivery result from the console.

Example 8-1 Message delivery result in the server log

```
[14/04/08 17:12:45:078 BST] 00000021 SystemOut     O >> SERVLET: Request count = 1
[14/04/08 17:12:45:078 BST] 00000021 SystemOut     O >> SERVLET: Request index: 1
[14/04/08 17:12:45:093 BST] 00000021 SystemOut     O >> CLIENT: SEI Ping to http://localhost:9081/WSSampleSei/PingService
[14/04/08 17:12:45:093 BST] 00000021 WSChannelFram A   CHFW0019I: The Transport Channel Service has started chain
HttpOutboundChain:localhost:9081.
[14/04/08 17:12:45:109 BST] 00000023 SystemOut     O >> SERVICE: SEI Ping JAX-WS Service: Request received.
[14/04/08 17:12:45:109 BST] 00000023 SystemOut     O >> SERVICE: SEI Ping Input String ''
[14/04/08 17:12:45:109 BST] 00000021 SystemOut     O >> CLIENT: SEI Ping SUCCESS.
```

Attach WS-Addressing Policy set

To apply the WS-Addressing default Policy set to the Web services and Web services client, we do the following steps:

- In the Administrator Console expand **Services** → **Services providers**, (Figure 8-8).

  A list of Web services are displayed. You attach the WS-Addressing Policy set to PingService.

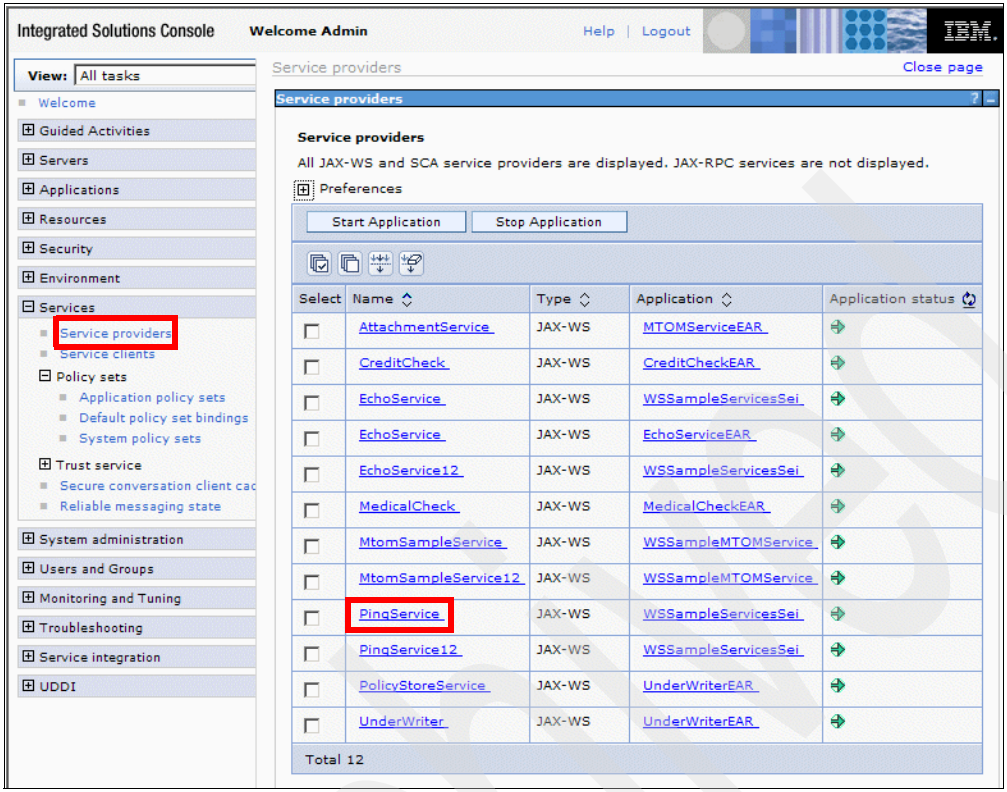


Figure 8-8 Service providers → PingService

- Click **PingService** to configure it with a policy (Figure 8-8).

  A Web service has three levels of generality: *service*, *endpoint*, and *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.

- Check **PingService** (Figure 8-9).

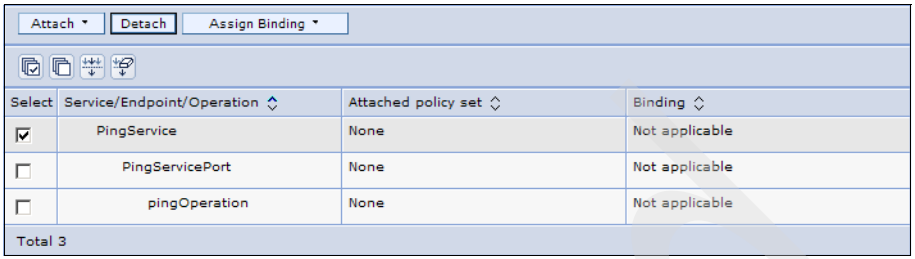


Figure 8-9 Attach Policy set to the PingService

- Click **Attach** to display a list of available Policy sets to attach → click **WS-Addressing default** Policy set (Figure 8-10).

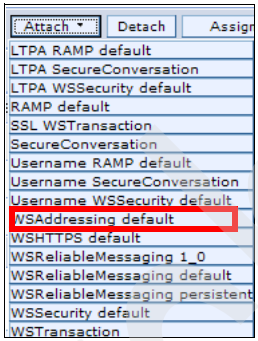


Figure 8-10 Attach WSAddressing default Policy set

- The WS-Addressing default Policy set is applied to the PingService (Figure 8-11) → **Save**.

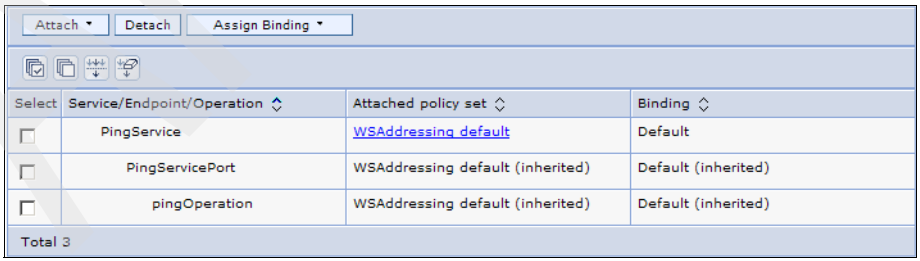


Figure 8-11 WSAddressing default policy set is attached

- In the left pane, select **Service clients** under Services. A list of service clients is displayed (Figure 8-12).

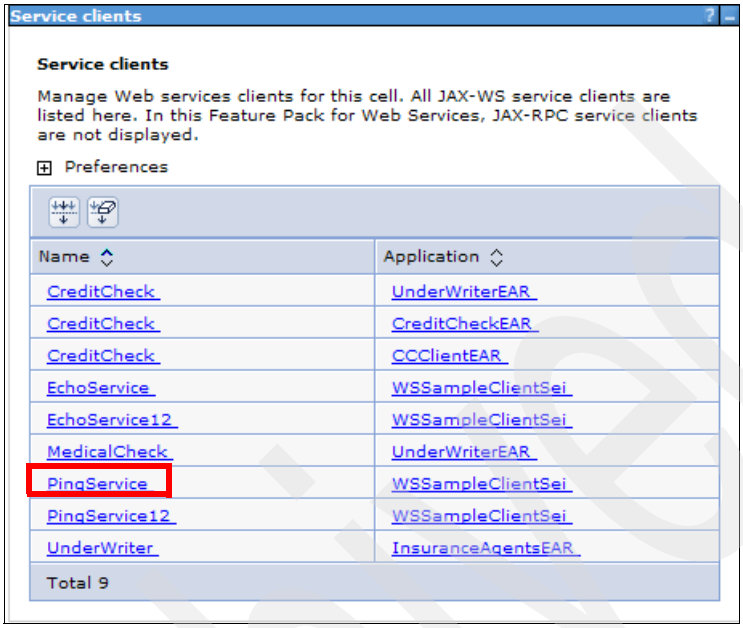


Figure 8-12 Click PingService

- Click **PingService** → Check **PingService** → **Attach** to display a list of available Policy sets to attach. → Click **WSAddressing default** Policy set from the list (Figure 8-13) → **Save** to save the changes directly to the master configuration

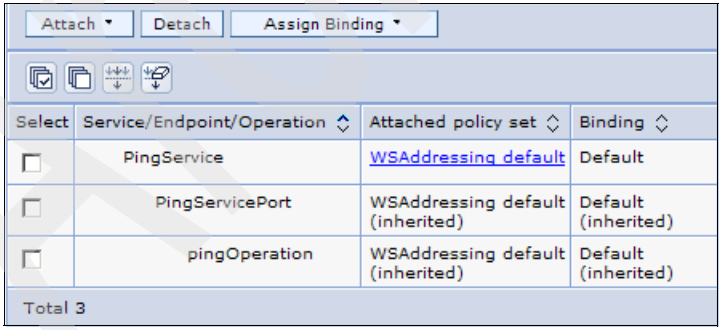


Figure 8-13 Attached WS-Addressing default to client

We must restart the service and the client to pick up the configuration changes.

- In the left pane, select **Applications** → **Enterprise Applications** → Check **WSSampleClientSei** and **WSSampleServerSei** → **Stop** → **Start** (Figure 8-14).

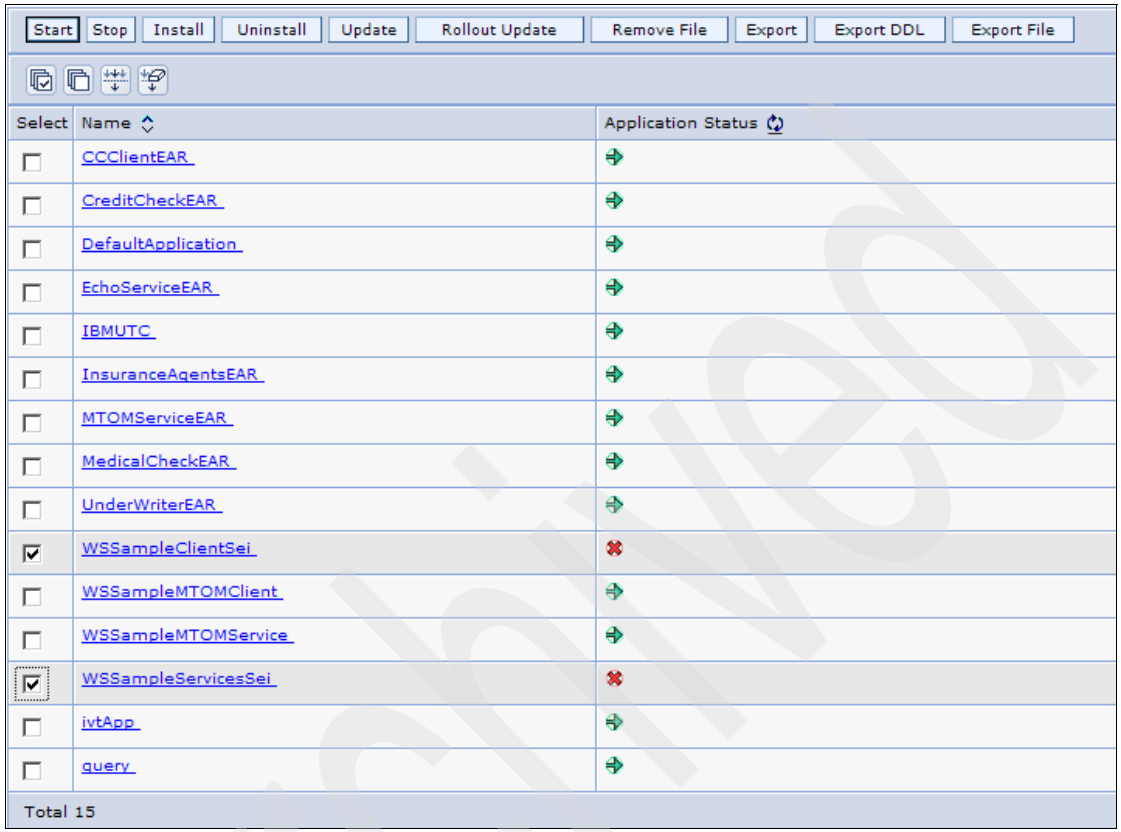


Figure 8-14 Stop → Start WSSampleClientSei and WSSampleServerSei

The WS-Addressing default Policy set has been applied to the WSSample Web service and Web service client. To make sure that the policy has been taken into effect, monitor the SOAP traffic to see if WS-Addressing information is added to the SOAP message.

Monitor the SOAP traffic

Rational Application Developer provides a TCP/IP monitor that we 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, and 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 being used.

To monitor the SOAP traffic, do the following tasks:

- ☐ Start Rational Application Developer if it is not already running.
- ☐ Select **Window** → **Preferences** → **Run/Debug** → **TCP/IP Monitor** → Click **Add**... The **New Monitor** dialog box opens. Alternatively, in the Servers tab, right-click the server → **Monitoring** → **Properties** → **Add...** The **Monitoring Ports** dialog box opens. The dialogs are different, but you can achieve the same result with either. We shall continue with the New Monitor dialog.
- ☐ Figure 8-15 shows how you should fill in the fields. In the Local monitoring port field, specify a unique port number on your local machine that's not used by any process (for example, **9089**). In the Host name field, type **localhost**. For the port, specify the port number of your WebSphere Application Server V6.1 for Web services provider → **OK** → **Start** → **OK**

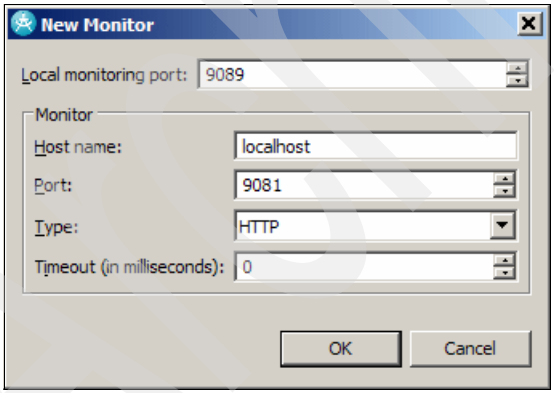


Figure 8-15 Create TCP/IP Monitor

To display the TCP/IP monitor window, and check the new monitor is running, do the following tasks:

- ☐ **Window** → **Show view** → **Other** and scroll down to the TCP/IP monitor → **OK** (Figure 8-16).

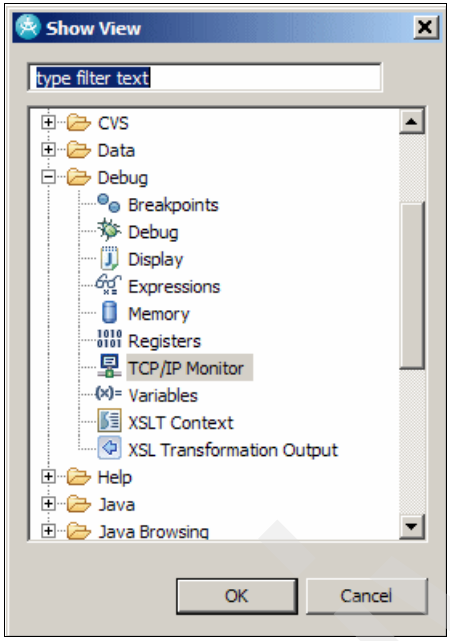


Figure 8-16 Display TCP/IP monitor view

- Click the little down arrowhead on the right side of the view → **Properties** (Figure 8-17).

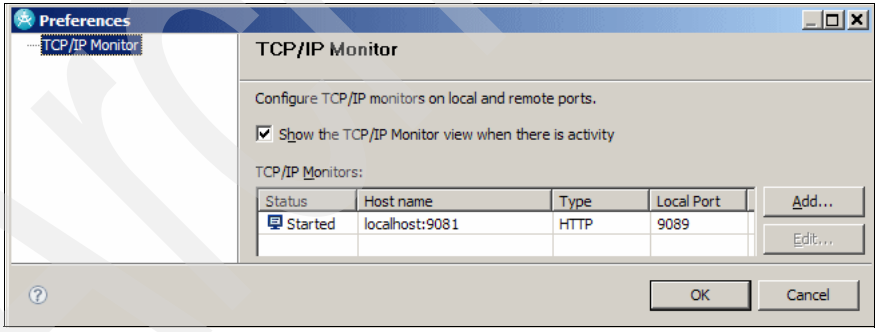


Figure 8-17 verifying TCP/IP monitor

- Open a Web browser and open the same page as explained in “Run the sample without applying the WS-Addressing policy” on page 257.

  For example, if your server is running at port 9081, then the URL should be:

  `http://localhost:9081/wssamplesei/demo`

- This time, specify **http://localhost:9089**, for the **Service URI**, which points to port that the TCP/IP monitor is listening to → **Send Message**. The result is the same as in Figure 8-7 on page 259.
- In the TCP/IP monitor view, select the request → select the XML format to view its contents (Figure 8-18).

Figure 8-18 Use 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.

8.3.4 Use a customized Policy set

The Feature pack for Web services ships several default Policy sets that are configured for use and are also 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 SEI application. Here is a summary of the requirements:

- Enables WS-Addressing which provides a transport-neutral way to uniformly address Web services and messages.
- Encrypts the message body of the SOAP request using RSA encryption. The SOAP response is not required to be encrypted.
- Message integrity is not required.

None of the default Policy sets match these requirements exactly. We note that the WS-Security default Policy set most closely meets our needs, so we use the WS-Security default Policy set as a template to create our own Policy set.

Keystores for the sample

Two key stores that contain the following keys are provided with the book (Table 8-1).

Table 8-1 Key store properties

Property	Service	Client
Key store path	C:\ITSO7618\PolicySet\receiver.jks	C:\ITSO7618\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 8-19).

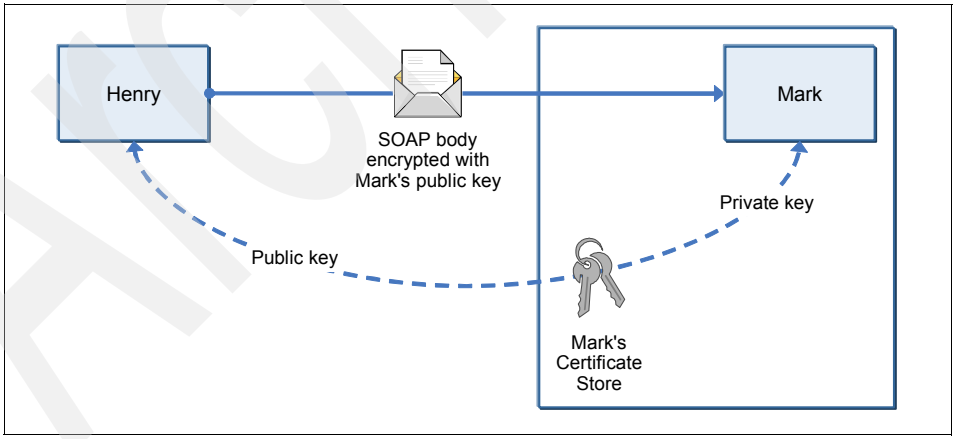


Figure 8-19 Customized Policy set example scenario

Create the custom Policy set

First you have to make a copy of the WSSecurity default policy. To do this, complete the following steps:

- ☐ Start the WebSphere Administrative Console and enter your Application Server administrative **user ID** and, if required, your password, and click **Log in**.
- ☐ In the left pane, select **Services** → **Policy sets** → **Application Policy sets**.
- ☐ In the right pane, check **WSSecurity default** → **Copy** at the top of the page.
- ☐ In the Name field, enter **ITSO WSSecurity** → **OK**.
- ☐ Click the newly created **ITSO WSSecurity** Policy set.

This scenario encrypts only the body, so we have to update the WS-Security policy.

- ☐ Under the **Policies** section → **WS-Security** → **Main policy**. The Main policy dialog displays, as shown in Figure 8-20. (You might not have saved the changes yet, and would see a message box. That is not a problem.)

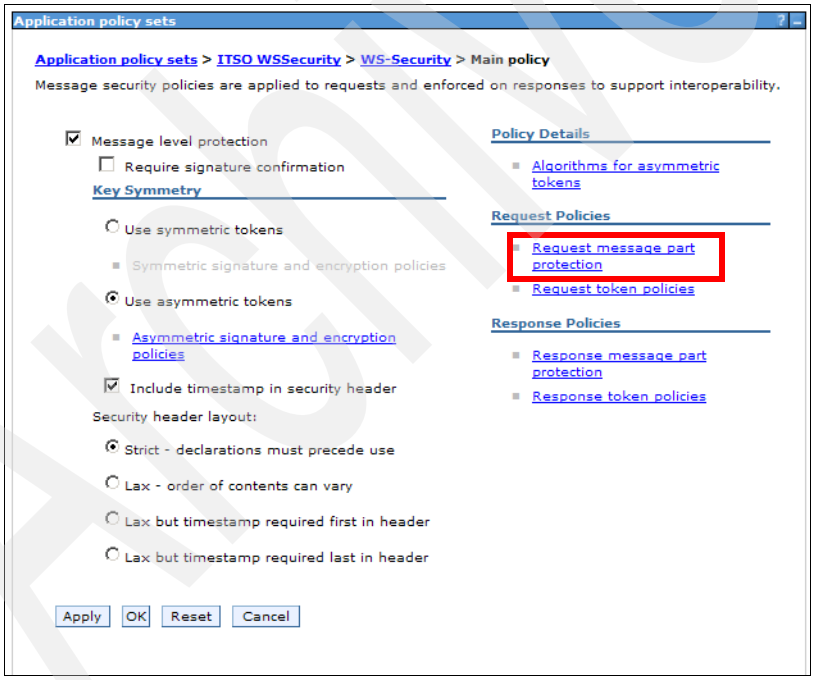


Figure 8-20 Main policy page

- Select **Request message part protection** (Figure 8-21).

Figure 8-21 Request message part page

For this scenario, we only apply message encryption. The digital signature is not required.

- Under the Integrity protection section → **app_signparts** → **Delete** → **Save**.
- Under Encrypted parts → **app_encparts** → **Edit** (Figure 8-22).

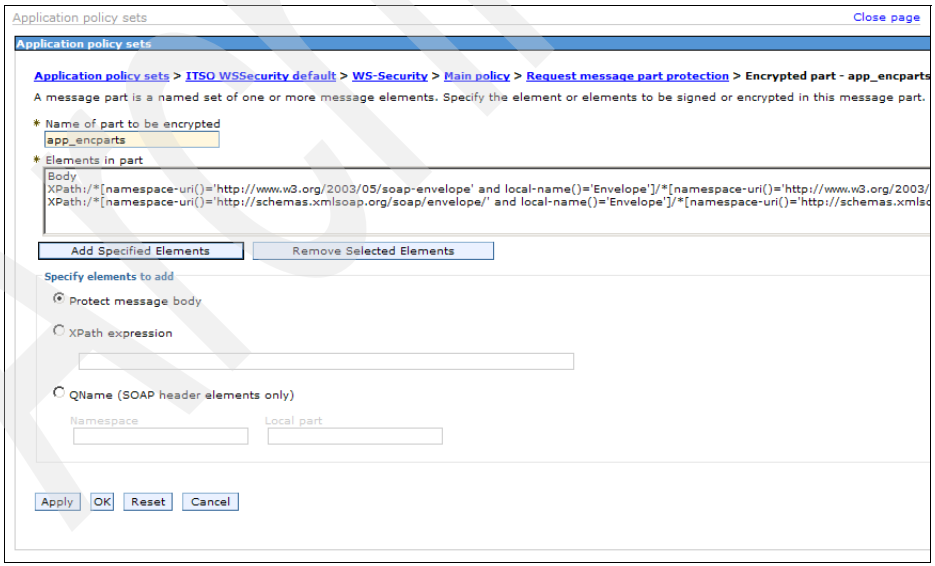


Figure 8-22 Edit encrypted parts

We encrypt the body only, so we have to remove the UsernameToken encryption in the SOAP header.

- Under the Elements in part section, remove the two XPath statements by selecting each XPath expression → **Remove Selected Elements** (Figure 8-23) → **Apply** → **Save**.

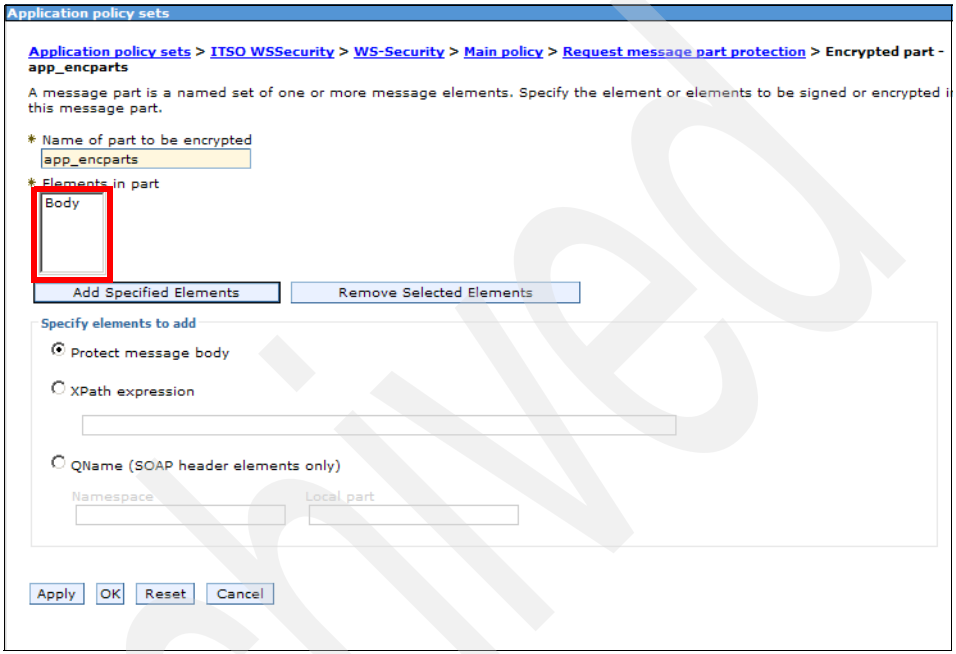


Figure 8-23 Encrypt the message body

- Click **Main policy** in the navigation path to return to the Main policy page → **Response message part protection**.

  The response message does not have to be secured.

- Under the Confidentiality section → **app_encparts** → **Delete**.
- Under the Integrity protection section → **app_signparts** → **Delete** → **Save**.

  We have created the customized Policy set. Next we attach the customized Policy set to the Web service and client.

Attach the custom Policy set to the Web service

In this section, you attach the ITSO WSSecurity Policy set to the EchoService service and client (In the sample application WSSample..., not EchoServiceEAR if you created the example from Chapter 7, “JAX-WS programming model” on page 175.)

To assign the Policy set to EchoService, do the following tasks:

- ☐ In the left pane, click **Services** → **Service providers**.
- ☐ Click **EchoService** → check **EchoService** → **Attach** → Click **ITSO WSSecurity** from the drop-down list.
- ☐ Check **EchoService** → **Assign** Binding → **New** (Figure 8-24).

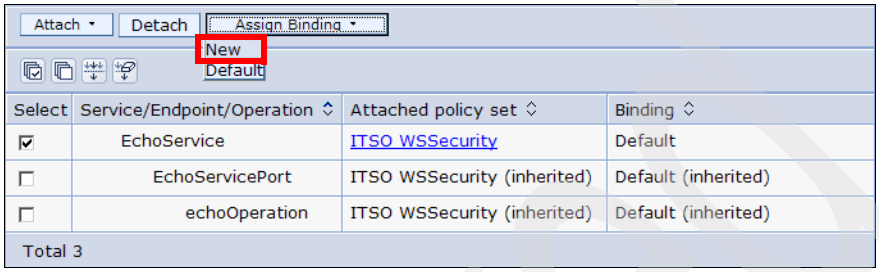


Figure 8-24 Assign binding

- ☐ Type `ITSO-service` as the Bindings configuration name → **Add** → **WS-Security** → **Save** (Figure 8-25).

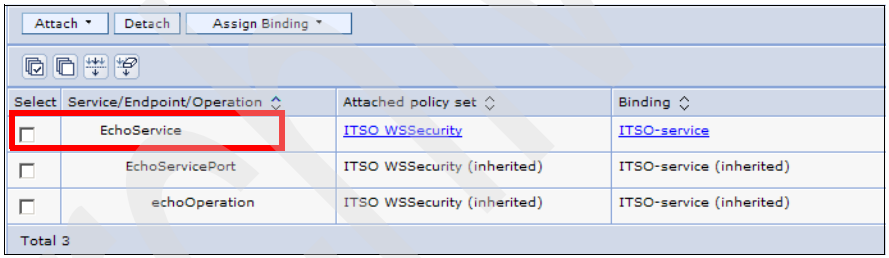


Figure 8-25 Binding is assigned

We have attached the Policy set and binding to the Web service.

Assign the custom Policy set to the Web service client

In this section, you attach the custom Policy set to the Web service client:

- ☐ In the left pane, click **Services** → **Service clients**.
- ☐ Click **EchoService** → check **EchoService** → **Attach** → click **ITSO WSSecurity** from the drop-down list.
- ☐ Check **EchoService** → **Assign Binding** → **New**.

- ☐ Type `ITSO-client` as the name → **Add** → **WS-Security** → **Save** (Figure 8-26).

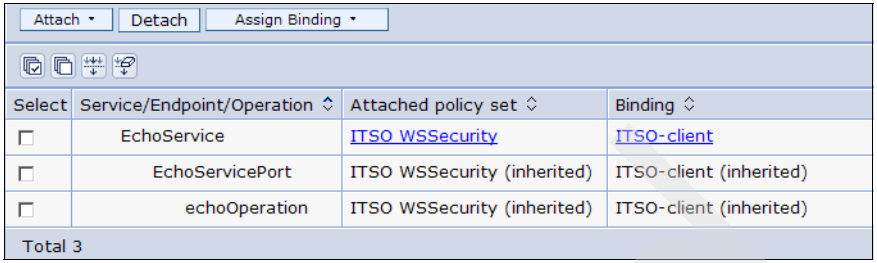


Figure 8-26 Bindings assigned to EchoService client.

You have attached the Policy set and binding to your Web service client.

8.3.5 Configure the custom binding

In this section, we configure the custom binding for the EchoService service. We configure XML encryption for the request message. On the client side, we use Mark's public key to encrypt the SOAP request. On the service side, we use Mark's private key to decrypt the inbound message (Figure 8-19 on page 267).

Configure the Web service binding

In this section, you configure the custom binding for the EchoService service. You configure the XML encryption to decrypt the request message using Mark's private key. First of all, we configure the Protection token to encrypt the request message, and then we configure way the message is encrypted.

Configure the server protection token

- ☐ In the left pane, click **Services** → **Service providers**.
- ☐ Click **EchoService** → **ITSO-service** binding → **WS-Security** (Figure 8-27).

Figure 8-27 WS-Security page

☐ Click **Authentication and protection** (Figure 8-28).

Service providers

Service providers > EchoService > ITSO-service > WS-Security > **Authentication and protection**

Configure custom bindings for tokens and message parts that are required by the policy set.

Protection tokens

Unconfigure

Select	Protection token name	Protection token type	Usage	Status
	AsymmetricBindingInitiatorEncryptionToken0	X509V3 Token v1.0	Asymmetric encryption generator	Not configured
	AsymmetricBindingInitiatorSignatureToken0	X509V3 Token v1.0	Asymmetric signature consumer	Not configured
	AsymmetricBindingRecipientEncryptionToken0	X509V3 Token v1.0	Asymmetric encryption consumer	Not configured
	AsymmetricBindingRecipientSignatureToken0	X509V3 Token v1.0	Asymmetric signature generator	Not configured

Total 4

Authentication tokens

Unconfigure

Select	Security token reference	Authentication token type	Usage	Status
None				

Total 0

Request message signature and encryption protection

Unconfigure

Select	Request message part reference	Protection	Status
	request:app_encparts	Encryption	Not configured

Total 1

Response message signature and encryption protection

Unconfigure Move Up Move Down

Select	Response message part reference	Protection	Order	Status
None				

Total 0

Figure 8-28 Authentication and protection page

Because the Web services provider only has to decrypt the SOAP request, we only have to configure Asymmetric encryption consumer.

- Click **AsymmetricBindingRecipientEncryptionToken0**.
- Verify that the JAAS login is **wss.consume.x509** → **Apply** to generate a callback handler binding (Figure 8-29).

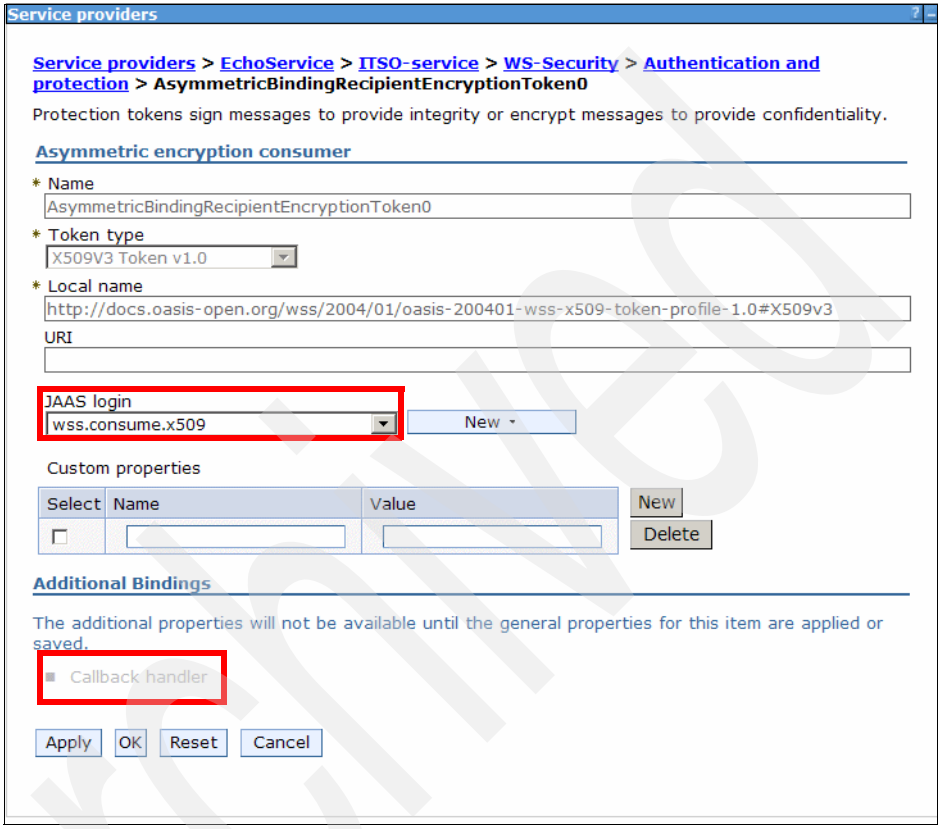


Figure 8-29 Generate callback handler binding

- Click **Callback handler** (which is now ungreyed) → Under Certificates, make sure **Trust any certificate** is selected.
- In the Keystore section, select **Custom** as the name, then click **Custom keystore configuration**.
- In the custom keystore configuration page, in the **Keystore** section, enter the **Full path** name for the receiver.jks keystore (supplied in the book's additional materials — see Appendix B, "Additional material" on page 561). For example:

  `C:\My Documents\ITSO7618\PolicySet\receiver.jks`

- Select **JKS** as type → For **Password**, enter `itso` → For **Confirm password**, enter `itso`.

- Under **Key**, enter `cn=server,o=IBM, C=US` in the **Name** field → Enter `mark` in the **Alias** field → For Key **Password**, enter `itso` → For **Confirm password**, enter `itso` (Figure 8-30) → **OK** → **OK** → **OK** → **Save**.

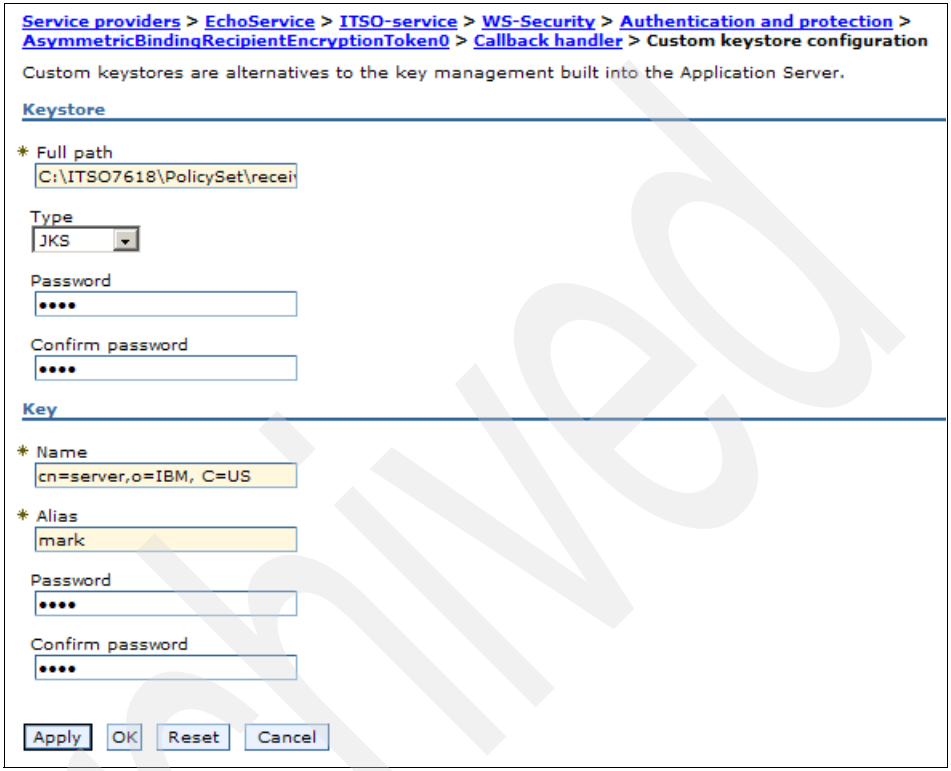


Figure 8-30 Custom keystore configuration

The AsymmetricBindingRecipientEncryptionToken0 is now configured. For the other three protection tokens, we do not have adjust their configuration — but we do have to change their status to configured.

- Click **AsymmetricBindingInitiatorEncryptionToken0** → **OK**.
- Click **AsymmetricBindingInitiatorSignatureToken0** → **OK**.
- Click **AsymmetricBindingRecipientSignatureToken0** → **OK** → **Save**.

Configure server Request message encryption protection

Now return to Figure 8-28 on page 273 and see under **Request message signature and encryption protection** that request:app encparts still has to be configured.

- Click **request:app encparts** → In the Name field, type `req-enc-part` → **Apply**.

- ☐ Under Key information, click **New**...
- ☐ Enter `req-enc-keyinfo` for the name → Ensure that **AsymmetricBindingRecipientEncryptionToken0** is selected for Token generator or consumer name → **OK**.
- ☐ Under Key information, select **req-enc-keyinfo** → **Add** → **OK** → **Save** (Figure 8-31).

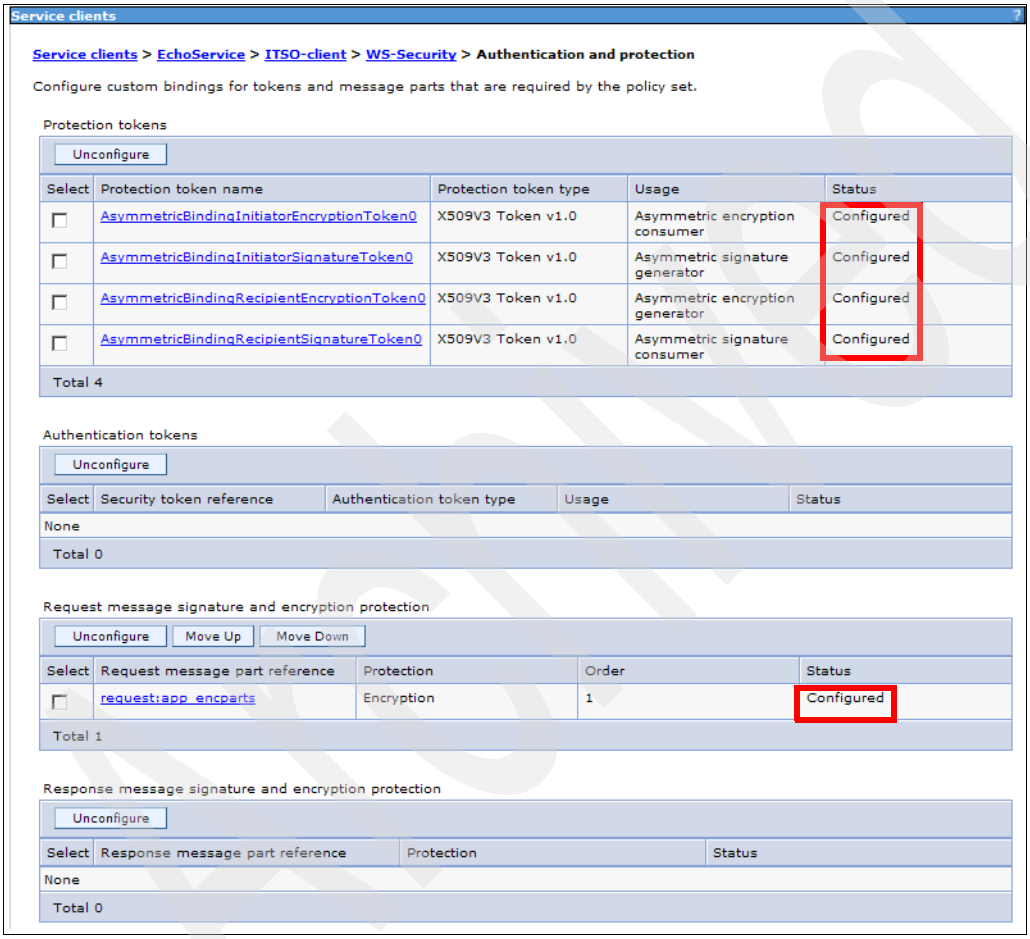


Figure 8-31 WS-Security is configured for the service

We have now configured the binding for the Web services.

Configure the EchoService client bindings

In this section, we configure the custom binding for the EchoService client. We configure the XML encryption to encrypt the request message using Mark's public key. Again, there are two parts to the configuration: first, the Protection token; and second, the message configuration.

Configure the client protection token

- ☐ In the left pane, click **Services** → **Service clients**.
- ☐ Click **EchoService** → **ITSO-client** → **WS-Security** (Figure 8-32)

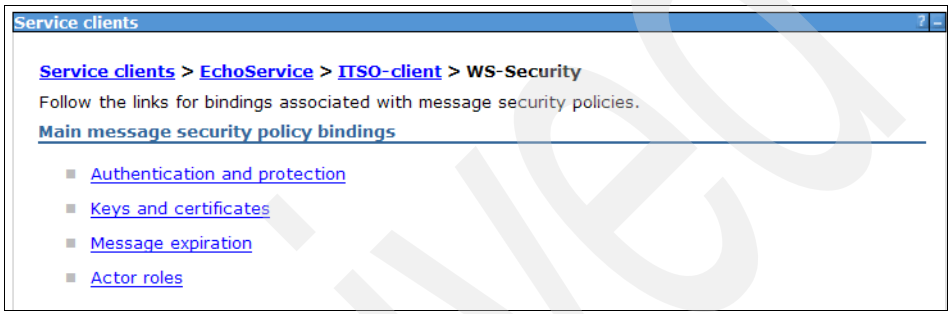


Figure 8-32 Client WS-Security configuration

- ☐ Click **Authentication and protection** → **AsymmetricBindingRecipientEncryptionToken0** → Verify that JAAS login is **wss.generate.x509** → **Apply** to generate a callback handler binding.
- ☐ Click **Callback handler** → In the Keystore section → **Custom** as the name, then select **Custom keystore configuration**.

In the custom keystore configuration page, enter the following data:

- ☐ Enter the full path name for the sender.jks keystore. For example:

```
C:\ITSO7618\PolicySet\sender.jks
```

- ☐ Select **JKS** as type → For **Password**, type `itso` → for **Confirm password** → type `itso` → Under **Key**, type `cn=client,o=IBM, C=US` in the **Name** field → type `mark` in the **Alias** field → **OK** → **OK** → **OK** → **Save**.

The AsymmetricBindingRecipientEncryptionToken0 is now configured. For the other three protection tokens, we do not have adjust their configuration — but we do have to change their status to configured.

- ☐ Click **AsymmetricBindingInitiatorEncryptionToken0** → **OK**.
- ☐ Click **AsymmetricBindingInitiatorSignatureToken0** → **OK**.
- ☐ Click **AsymmetricBindingRecipientSignatureToken0** → **OK** → **Save**.

Configure client Request message encryption protection

- ☐ Under Request message signature and encryption protection, click **request:app encparts** → In the **Name** field, enter `req-enc-part` → **Apply**.
- ☐ Under **Key** information → **New**.
- ☐ Enter `req-enc-keyinfo` for the **Name** → Under **Type**, select **Key identifier** → Ensure that **AsymmetricBindingRecipientEncryptionToken0** is selected for Token generator or consumer name → **OK** → **OK** → **Save**.

Restart the client and server

We must restart the service and the client to pick up the configuration changes:

- ☐ In the left pane, select **Applications** → **Enterprise Applications**. The applications are listed.
- ☐ Check **WSSampleServicesSei** and **WSSampleClientSei** → **Stop** → **Start** to restart the applications.

The *ITSO WSSecurity* custom Policy set has been applied to the Web service and its client. In order to make sure that the Policy set has been taken into effect, we have to examine the SOAP traffic to see if WS-Security information is added to the SOAP message.

Monitor the SOAP traffic

- ☐ You use the TCP/IP monitor shipped with Rational Application Developer to watch the SOAP traffic. In this section, you first monitor the SOAP traffic for the two-way synchronous message exchange pattern. Then, you monitor the SOAP traffic for the two-way asynchronous message exchange pattern. To monitor the SOAP traffic, do the following tasks:
- ☐ Start Rational Application Developer if it is not already running → Click **Window** → **Preferences** → **Run/Debug** → **TCP/IP Monitor** → Select the monitor we created in the last section, → **Start** if the status is stopped.

Test a two-way synchronous message exchange pattern

- ☐ Open a Web browser and open the Feature pack for Web services Sample Demo page using a URL with the host name and port set correctly, as follows:

```
http://<hostname>:port/wssamplesei/demo
```

For example, if your server is running at port 9081, then the URL is:

```
http://localhost:9081/wssamplesei/demo
```

- Select **Synchronous Echo** as the Message Type → Type a Message String, `Hello World` → Type `http://localhost:9089`, for the **Service URI**, which points to port that the TCP/IP monitor is listening to → Make sure Use SOAP 1.2 is unchecked → **Send Message** (Figure 8-33).

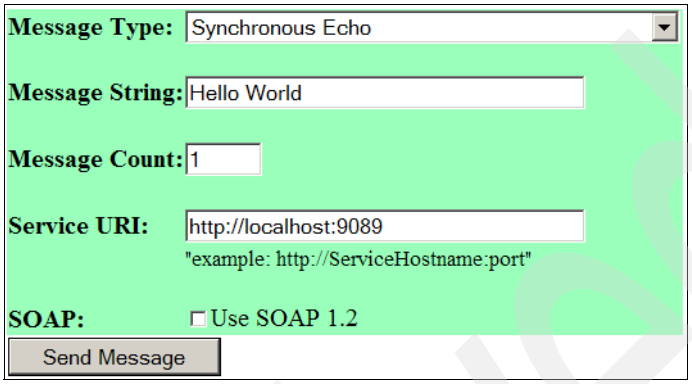


Figure 8-33 Synchronous Echo parameters

The results are shown in Figure 8-34.

```
Connecting to... http://localhost:9089

Message Request:
Hello World


Message Response:
JAX-WS==>>Hello World
```

Figure 8-34 Synchronous Echo results

The TCP/IP monitor window should have opened automatically; if not, display it (see page 264), and display the XML view of the request and response messages.

The (formatted) SOAP request is listed in Example 8-2.

> **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 on the contents → **Format** → **Document**.

Example 8-2 Encrypted SOAP request message

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:wsa="http://www.w3.org/2005/08/addressing">
  <soapenv:Header>
     <wsse:Security xmlns:wsse="uri[2]" soapenv:mustUnderstand="1">
         <wsu:Timestamp xmlns:wsu="uri[3]">
             <wsu:Created>2008-04-16T09:38:35.15Z</wsu:Created>
         </wsu:Timestamp>
         <EncryptedKey xmlns="http://www.w3.org/2001/04/xmlenc#">
             <EncryptionMethodAlgorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5" />
             <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
                <wsse:SecurityTokenReference>
                    <wsse:KeyIdentifier
                        EncodingType=uri[4]#Base64Binary"ValueType="uri[5]#X509SubjectKeyIdntifier">
                        vmFljXX4Bpp4awaZeM3pL9nPf3s=
                    </wsse:KeyIdentifier>
                </wsse:SecurityTokenReference>
             </ds:KeyInfo>
             <CipherData><CipherValue>TJ2+T3nva ... DpiEq2iY=</CipherValue></CipherData>
             <ReferenceList><DataReference URI="#wssecurity_encryption_id_1" /></ReferenceList>
         </EncryptedKey>
     </wsse:Security>
     <wsa:To>http://localhost:9089/WSSampleSei/EchoService</wsa:To>
     <wsa:MessageID>urn:uuid:5A402A6C19BE1D55CD1208343869325</wsa:MessageID>
     <wsa:Action>echoOperation</wsa:Action>
  </soapenv:Header>
  <soapenv:Body>
     <EncryptedData xmlns="http://www.w3.org/2001/04/xmlenc#"
         Id="wssecurity_encryption_id_1"
         Type="http://www.w3.org/2001/04/xmlenc#Content">
         <EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#aes128-cbc" />
         <CipherData><CipherValue>CjHEzM ... VCixwW</CipherValue></CipherData>
     </EncryptedData>
  </soapenv:Body>
</soapenv:Envelope>
```

The WS-Addressing information is emphasized in **blue**. The encrypted SOAP body is emphasized in **red**. Therefore, the `ITSO WSSecurity` custom Policy set has been successfully applied.

Look at the unencrypted response message Example 8-3.

[2] uri is http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd
[3] uri is http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd
[4] uri is http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0
[5] uri is http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0

Example 8-3 SOAP response

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope
   xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:wsa="http://www.w3.org/2005/08/addressing">
   <soapenv:Header>
      <wsse:Security
xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd"
         soapenv:mustUnderstand="1">
         <wsu:Timestamp
xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd">
            <wsu:Created>2008-04-16T11:04:28.281Z</wsu:Created>
         </wsu:Timestamp>
      </wsse:Security>
      <wsa:Action>
         http://com/ibm/was/wssample/sei/echo/EchoServicePortType/echoOperationResponse
      </wsa:Action>
      <wsa:RelatesTo>urn:uuid:5A402A6C19BE1D55CD1208343869325</wsa:RelatesTo>
   </soapenv:Header>
   <soapenv:Body>
      <ns2:echoStringResponse
         xmlns:ns2="http://com/ibm/was/wssample/sei/echo/">
         <echoResponse>JAX-WS==&gt;&gt;Hello World</echoResponse>
      </ns2:echoStringResponse>
   </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 these are not used.

Test a two-way asynchronous message exchange pattern

- Return to the browser and select **Asynchronous Echo with Async Communication** as the Message Type. Complete the rest of the panel as before (Figure 8-35).

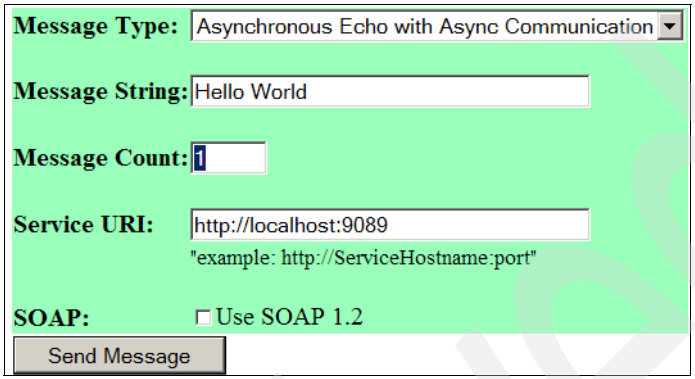


Figure 8-35 Asynchronous 2-way request

The SOAP request is listed in the TCP/IP Monitor,(Example 8-4) There is neither a response back to the browser, not a response listed in the TCP/IP monitor. Why?

Looking at the SOAP request (Example 8-4), you notice that it is slightly different from the request in (Example 8-3 on page 281). The asynchronous over-the-wire request message contains a *ReplyTo* address to the service, and the client will be listening for the response at this endpoint reference.

- Change the address of the **Service URI** in the browser to point to the Web service and not to the monitor → Type a new **Message string,** `Hello Asynch World` → **Send message**

After a longer delay, the response is displayed in the browser.

The problem lies with the TCP/IP monitor. The SOAP response routing is taking place above the TCP/IP stack (the Web service is setting the destination address). The TCP/IP monitor does not see the response, and worse, when the request is routed through the TCP/IP monitor, the Web service routes the response back to the Web service client. There can be other monitors to work around this problem, or, as we suggest in “Feature pack for Web services implementation of wire async” on page 204, use API tracing.

Example 8-4 SOAP request two-way asynchronous message exchange pattern

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope
   xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:wsa="http://www.w3.org/2005/08/addressing">
   <soapenv:Header>
      <wsse:Security
         xmlns:wsse="uri[6]" soapenv:mustUnderstand="1">
         <wsu:Timestamp xmlns:wsu="uri[7]">
            <wsu:Created>2008-04-16T11:37:12.234Z</wsu:Created>
         </wsu:Timestamp>
         <EncryptedKey xmlns="http://www.w3.org/2001/04/xmlenc#">
            <EncryptionMethod
               Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5" />
            <ds:KeyInfo
               xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
               <wsse:SecurityTokenReference>
                  <wsse:KeyIdentifier
                     EncodingType="uri[8]#Base64Binary" ValueType="uri[9]#X509SubjectKeyIdentifier">
                     vmFljXX4Bpp4awaZeM3pL9nPf3s=
                  </wsse:KeyIdentifier>
               </wsse:SecurityTokenReference>
            </ds:KeyInfo>
            <CipherData><CipherValue>V9SdOQ...2OOVOa4=</CipherValue></CipherData>
            <ReferenceList><DataReference URI="#wssecurity_encryption_id_1" /></ReferenceList>
         </EncryptedKey>
      </wsse:Security>
      <wsa:To>http://localhost:9089/WSSampleSei/EchoService</wsa:To>
      <wsa:ReplyTo><wsa:Address>See footnote[10]</wsa:Address></wsa:ReplyTo>
      <wsa:MessageID>urn:uuid:5A402A6C19BE1D55CD1208345822545</wsa:MessageID>
      <wsa:Action>echoOperation</wsa:Action>
   </soapenv:Header>
   <soapenv:Body>
      <EncryptedData xmlns="http://www.w3.org/2001/04/xmlenc#"
         Id="wssecurity_encryption_id_1" Type="http://www.w3.org/2001/04/xmlenc#Content">
         <EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#aes128-cbc" />
         <CipherData><CipherValue>7+jSDK...plslE</CipherValue></CipherData>
      </EncryptedData>
   </soapenv:Body>
</soapenv:Envelope>
```

[6] uri is http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd
[7] uri is http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd
[8] uri is http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0
[9] uri is http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0
[10] Response address is http://192.168.182.129:9081/IBM_WS_SYS_RESPONSESERVLET/WSSampleClientSei:SampleClientSei.war/services/EchoService.EchoServicePort

The service initiates a connection to the ReplyTo endpoint reference to send a response. As the request and response use different HTTP channels for the two-way asynchronous message exchange pattern, the TCP/IP monitor can only intercept the SOAP request, not the SOAP response. Therefore, the SOAP response pane in the TCP/IP monitor is empty.

You have applied the custom Policy set and custom binding to the sample application. In the next section, you will use Rational Application Developer to work with Policy set.

8.4 Tools support

Rational Application Developer works with Policy sets. Policy sets are imported in the preferences view under qualities of service. This allows a developer to work with Policy sets exported from a production environment.

To import the Policy set, go to **Window** → **Preferences** → **Web services** → **Qualities of Service**, as seen in Figure 8-36.

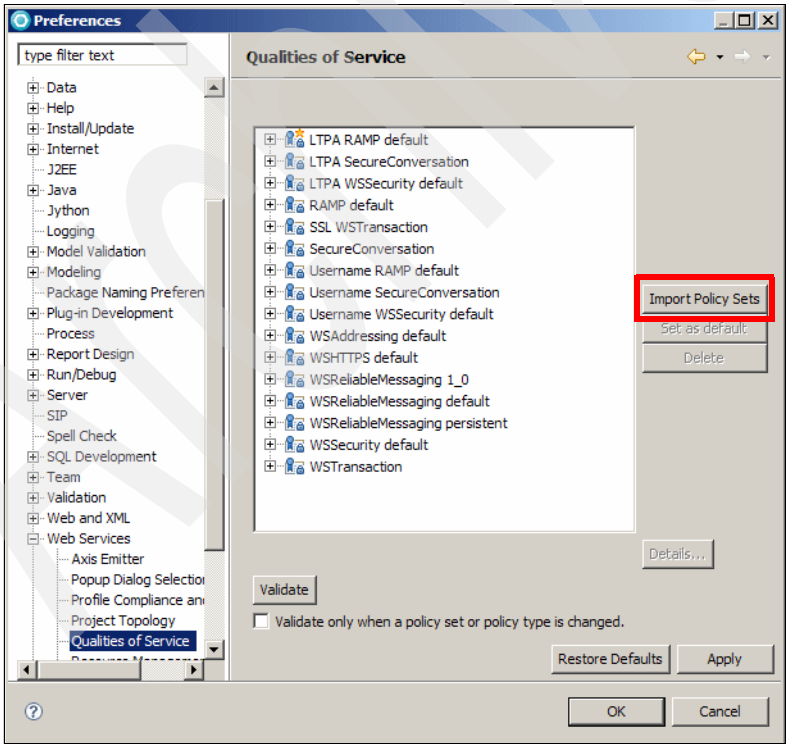


Figure 8-36 Qualities of Service in the Preferences page

Using the Policy sets that are included with Rational Application Developer simplifies configuring the qualities of service for Web services and clients. Several Policy sets are included in the WebSphere Application Server Feature pack for Web services, which are available in the workbench.

Policy sets are attached to Web Services and clients using a wizard. All Policy sets attached to the service are listed, and more can be added or removed. Policy sets are modifiable using Rational Application Developers as well.

In this section, we show how to use the tools shipped with Rational Application Developer to apply Policy set to Web services. We have prepared a simple Calculator Web service application to demonstrate applying the WS-Security Policy set using Rational Application Developer.

8.4.1 Prepare for the sample

The sample includes a Web service, which is a J2EE application, and a Web service client, which is a stand-alone Java client in a Java project.

Import the calculator project

- Start Rational Application Developer → In the J2EE perspective **File** → **Import** → **Other** → **Project Interchange** → **Browse** to navigate to and select **Calculator.zip** from the **C:\ITSO7618\PolicySet** → **Open** → **Select All**, as seen in Figure 8-37 → **Finish**.

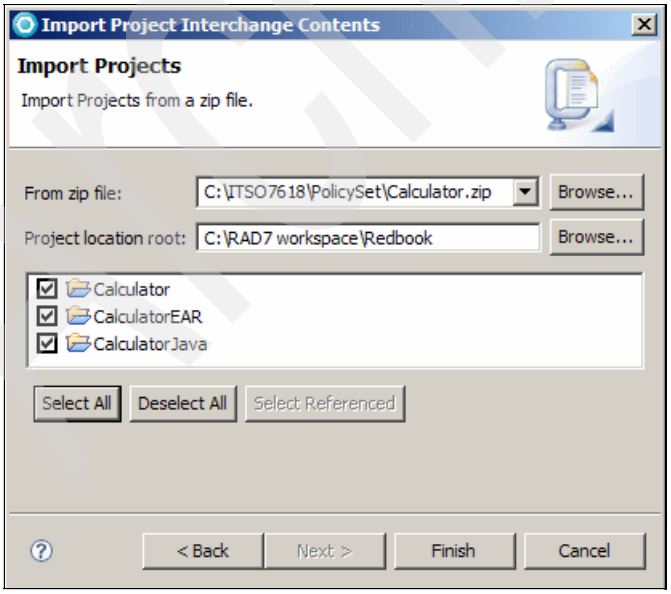


Figure 8-37 Import project interchange

Compile the calculator project

You might get compilation errors, because the installation directory of your Rational Application Developer might be different.

- ☐ Right-click the **CalculatorJava** project → **Java Build Path** → **Libraries** →
  - ☐ Select **com.ibm.jaxws.thinclient_6.1.0.jar** → **Edit...**, This jar should be in your `%WebSphere Installation directory%\runtimes` folder.
  - ☐ Select WebSphere Application Server v6.1 → **Edit...** This should be your WebSphere Application Server v6.1 server and not the stub.
- ☐ (Optional) In the Servers view, double-click your WebSphere Application Server v6.1 to open the server editor. In the server editor, under the **Publishing** section, check **Run server with resources on Server** (Figure 8-38)

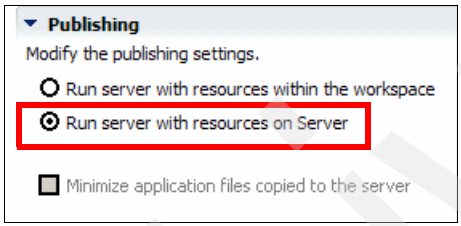


Figure 8-38 Server editor

> **Note:** The option, *Run server with resources on Server*, installs and copies the full application and its server-specific configuration from the workbench into the directories of the server.
>
> If we use the option, *Run server with resources within the workspace*, the application still works after applying the Policy set, however, we do not see the Policy set attached to the application in the WebSphere administrative console.

Publish the calculator

- ☐ Right-click WebSphere Application Server v6.1 → **Add and remove projects**. → select **CalculatorEAR** → **Add** → **Finish**.

  Alternatively, drag Calculator.ear from the Project explorer and drop it on the server.

Test the calculator

- ☐ In the Package Explorer → expand the project **CalculatorJava** → **src** → **com.ibm**.
- ☐ Right-click **TestCalculator.java** → **Run As** → **Java Application**.

The console output should look similar to the following lines:

```
27-Feb-2008 11:04:01 AM com.ibm.ws.ssl.config.SSLConfigManager
INFO: ssl.disable.url.hostname.verification.CWPKI0027I
2.0
0.0
```

If you get the error, `Connection refused`, this means that the SOAP request was not sent to the correct port. Open **TestCalculator.java** in your **CalculatorJava** project, add the following line right after you create CalculatorPortProxy object:

```
proxy._getDescriptor().setEndpoint("http://localhost:xxxx/Calculator/Ca
lculatorService");
```

Replace xxxx with the port number that your WebSphere Application Server is running on and you should get the correct result.

Work around: There is a static URL in the generated client stub, and the code is called before you get a chance to change the URL. It is in the **CalculatorService** class. You have to edit this URL to match your service. You might also be getting a compile error, because this same static URL is in the WsdlLocation attribute of the @WebServiceClient annotation. Change both URLs to the same value.

Change **localhost:9081** to the correct value in:

```
static {
        URL url = null;
        try {
            url = new
URL("http://localhost:9081/Calculator/CalculatorService/calculatorservice.wsdl");
        } catch (MalformedURLException e) {
            e.printStackTrace();
        }
        CALCULATORSERVICE_WSDL_LOCATION = url;
    }
```

8.4.2 Attach a Policy set to Web services

In this section, you attach the WS-Security default Policy set to a Web service using Rational Application Developer.

Keystores for the sample

Because the tooling does not support the Web service side custom binding configuration, we use the sample keystore and certificate files in WebSphere Application Server v6.1 for this example. Support for the server side custom binding configuration from the tools is planned to be available in a future release of Rational Application Developer.

Figure 8-39 shows how the client uses two different keystores to sign, encrypt, and decrypt the message on the client side and the server uses two keystores to decrypt, verify, encrypt, and sign the response. As the service side uses the default binding, we only have to configure the client side binding.

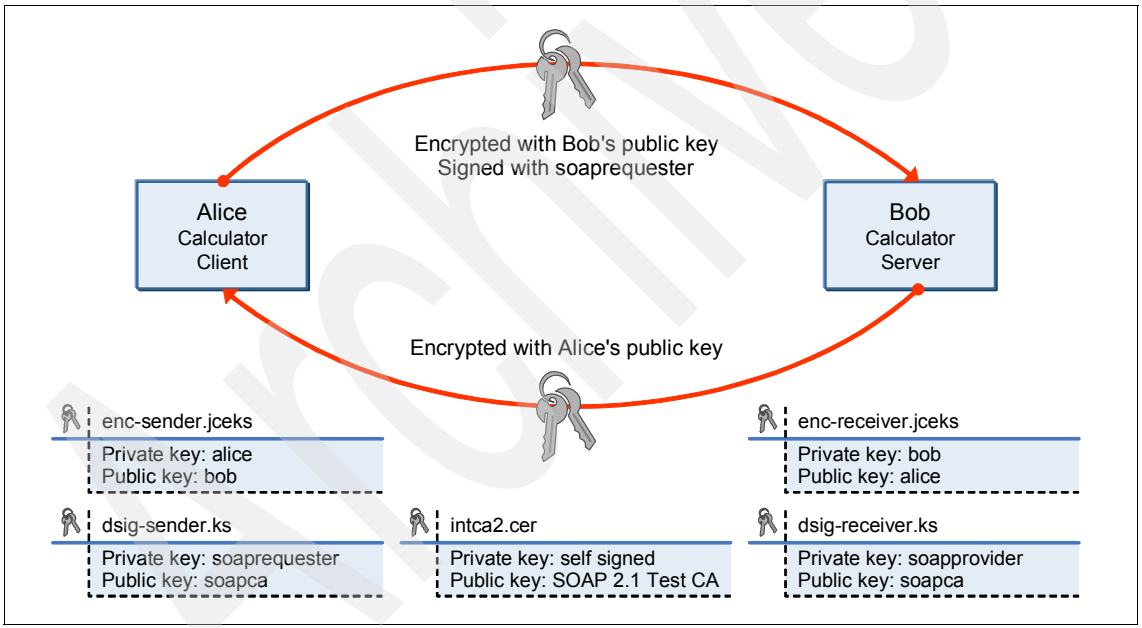


Figure 8-39 Security configuration for Calculator

Table 8-2 lists the keystore and keys for the Calculator client to *encrypt the outbound* and *decrypt the inbound* SOAP messages from the client. The table also shows the corresponding values used for encryption and decryption by the server.

Table 8-2 Encryption/decryption keystore configuration information

	Calculator client	Calculator server
Who am I?	Alice	Bob
What is my encryption keystore called?	<%WAS_HOME%>/ etc/ws-security/samples/ enc-sender.jceks pw = storepass	<%WAS_HOME%>/ etc/ws-security/samples/ enc-receiver.jceks pw = storepass
What is my private encryption key?	alias = alice CN=Alice, O=IBM, C=US pw = keypass	alias = bob CN=Bob, O=IBM, C=US pw = keypass
What public keys do I have for decryption?	alias = bob CN=Bob, O=IBM, C=US	alias = alice CN=Alice, O=IBM, C=US

Table 8-3 shows the keystores to sign and verify the outbound and inbound messages. Rather than verify the inbound message to the client, we decide to trust all inbound messages by providing the following parameter to the JVM that we use to run the Calculator client:

```
-Djava.security.auth.login.config={USER_INSTALL_ROOT}/profiles\WS01\pr
operties\wsjaas_client.conf
```

Table 8-3 Signature keystore configuration information

	Calculator client	Calculator server
Who am I?	soaprequester	soapprovider
What is my signing keystore called?	<%WAS_HOME%>/ etc/ws-security/samples/ dsig-sender.ks[a] pw = client	<%WAS_HOME%>/ etc/ws-security/samples/ dsig-receiver.ks pw = server
What is my private signing key?	alias = soaprequester CN=Alice, O=IBM, C=US pw = client	alias = soapprovider CN=Bob, O=IBM, C=US pw = server
What public keys do I have for verification?	soapca CN=SOAP 1.2 Test CA OU=TRL, O=IBM, l=Yamato, S=Kanagawa, C=JP	soapca CN=SOAP 1.2 Test CA OU=TRL, O=IBM, l=Yamato, S=Kanagawa, C=JP

a. .ks is a synonym for jceks

We use the client values in this table to configure the client in 8.4.3, “Attach the WS Security Policy set to the Web Service Client” on page 296. We use the default configuration for the server.

Validating the default keystore in the server

> **Attention:** There have been problems reported with using the default keystores. The usual symptom is that the key identifier in the SOAP message does not match the key identifier in the keystore. We have experienced this problem on some machines and not others, as have other users. It is under investigation, but as yet it is not clear what the cause of the problem is and how to reproduce it. If you have this problem when running the secured calculator, refer to this section for help in debugging.

Table 8-2 shows the configuration of keystores how they are used by the Calculator application.

Run `%WAS_INSTALL_DIRECTORY%\bin\ikeyman.bat` to inspect the contents of the keystores.

- Navigate to **Services** → **Policy sets** → **Default Policy set bindings** → **WS-Security** → **Authentication and protection** to verify the key token information. To access the custom keystore configuration for each token listed in Table 8-4, click on the token name in the table in the Administrator Console → **Additional bindings** → **Callback handler** → **Custom keystore configuration** (Figure 8-40 shows an example for `gen-encx509token`).

  The settings should be as shown in Table 8-4.

Table 8-4 Server token values

Token name	Path	Key	Commentary
gen_ encx509token	<%WAS_HOME%>/ etc/ws-security/samples/ enc-receiver.jceks pw=storepass	Alias=alice CN=Alice, O=IBM, C=US	Used to asymmetrically encrypt messages to Alice using alice's public key in enc-receiver.jceks
con_ encx509token		Alias=bob CN=Bob, O=IBM, C=US	Used to asymmetrically decrypt messages from Bob using Bob's private key in enc-receiver.jceks
gen_ signx509token	<%WAS_HOME%>/ etc/ws-security/samples/ dsig-receiver.ks pw=server	Alias=soapprovider CN=SOAPProvider, OU=TRL, O=IBM, ST=Kanagawa, C=JP	Used to asymmetrically sign (encrypt) messages as soapprovider using soapprovider's private key in dsig-receiver.ks signed by soapca

Token name	Path	Key	Commentary
con_ signx509token	<%WAS_HOME%>/ etc/ws-security/samples/ intca2.cer	Certificate store = DigSigCertStore	Used to asymmetrically verify (decrypt) messages using public keys in DigSigCertStore
	<%WAS_HOME%>/ etc/ws-security/samples/ dsig-receiver.ks pw=server	Trust anchor = DigSigTrustAnchor (self signed)	Used to asymmetrically verify (decrypt) messages using public keys in DigSigCertStore

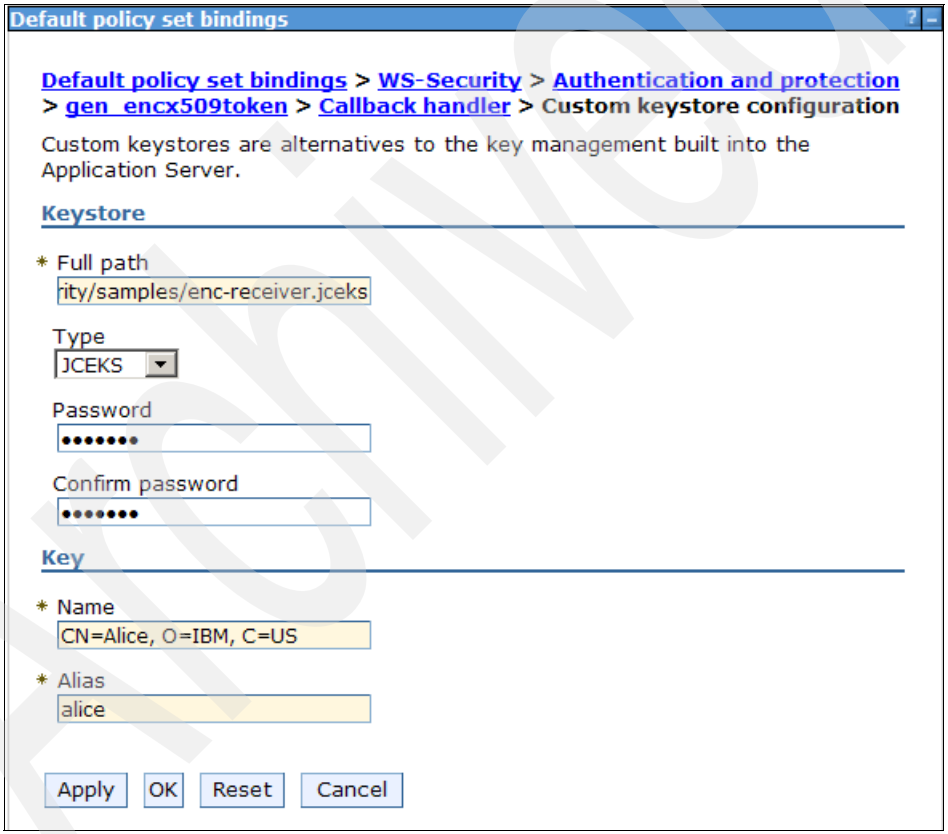


Figure 8-40 Example of configuration of a keystore token

Attach the WS Security Policy set to the Web service

We apply a Policy set at the service, port, or operation level. 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. The calculator service has two operations: add and subtract. For this example we apply the Policy set to both operations.

- In the Project Explorer view, expand the **JAX-WS Web Services** folder → expand the **Services** folder. Under this folder, find the entry for the **Calculator** service.

> **Note:** If you cannot see your Web service under the JAX-WS Web Services folder, restarting your workbench should resolve the problem, or you can right-click **JAX-WS Web services** → **Refresh**.

- Right-click on the **Calculator** → **Manage Policy set Attachment…**. (Figure 8-41).

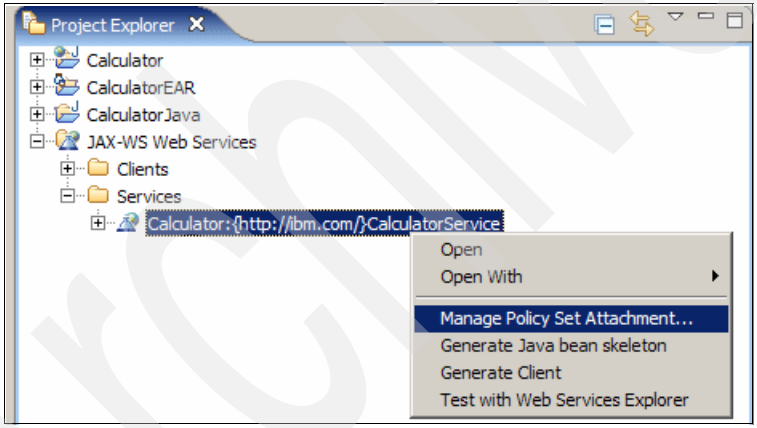


Figure 8-41 Manage policy set attachment

This invokes the Service Side Policy set Attachment Wizard

- In the **Add Policy set Attachment to Service** → Select **CalculatorEAR** Project from the drop-down list if it is not already selected (Figure 8-42).

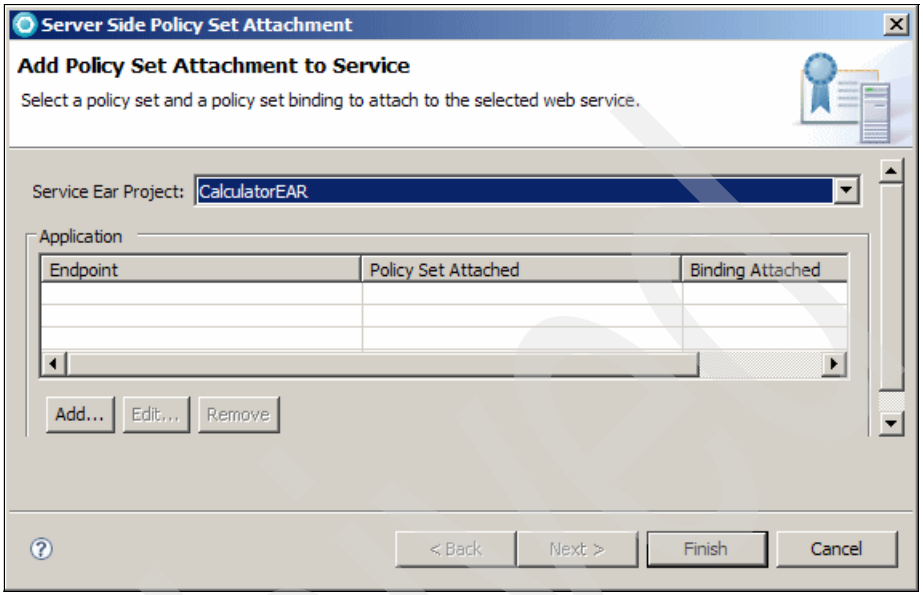


Figure 8-42 Add Policy set attachment to service

The Application Group lists all endpoints for this Web Service that have been already attached to a Policy set. This table is currently empty because we have not attached any service endpoint to any Policy set yet.

- Click **Add...** under the table inside the Application group.

This opens the End Point Definition Dialog, which lets us select the service endpoint to attach a Policy set to. We can attach the Policy set to the whole service, to a specific endpoint, or to a specific operation. For this example, we attach the Policy set to the whole service, so accept the default. For the Policy set, select the **WSSecurity default Policy set** from the drop-down list and leave the **Default** Binding unchanged, because the tools do not support configuring Custom binding for the Service Side in this release (see Figure 8-43). → **OK**.

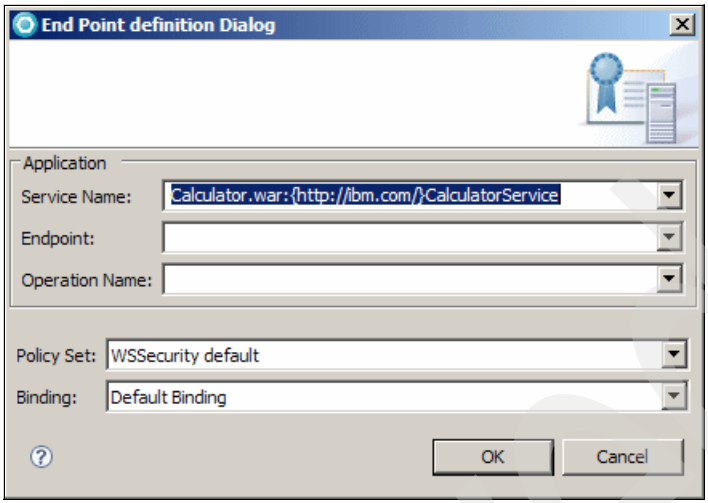


Figure 8-43 End point definition dialog

This takes us back to the **Add Policy set Attachment to Service** page. You can now see the entry that you just added in the Application table (Figure 8-44)

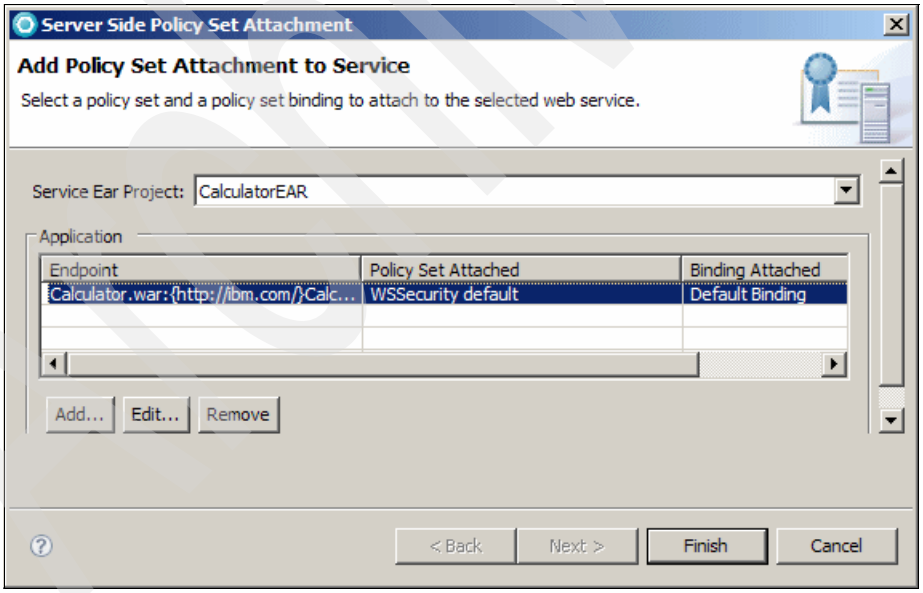


Figure 8-44 Policy set is applied to the service

- Click **Finish**.

  After a Policy set has been attached to a Web service, a **policyAttachments.xml** file is generated in the CalculatorEAR\META-INF folder (Example 8-5). This file will be appended for each additional Policy set setting added to any service within the EAR.

Example 8-5 policyAttachments.xml

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<PolicySetAttachment
   xmlns="http://www.ibm.com/xmlns/prod/websphere/200605/policysetattachment">
   <PolicySetReference name="WSSecurity default"
      id="com.ibm.ast.ws.policyset.ui.qos.WSSecuritydefault">
      <Resource pattern="WebService:/Calculator.war:{http://ibm.com/}CalculatorService" />
   </PolicySetReference>
</PolicySetAttachment>
```

- Check **Run server with resources on Server** in the Server editor to see the Policy set being attached to your application in the administrative console.

  Tip: You might have to undeploy Calculator, restart the server, and redeploy if you do not get the same results.

- In the Servers view, right-click your WebSphere Application Server, select **Run administrative console** → Log in to the administrative console.
- Select **Services** → **Service providers** → **CalculatorService**.

  In Figure 8-45 we see that the **WSSecurity default Policy set** is attached to the service and the **default** binding is assigned.

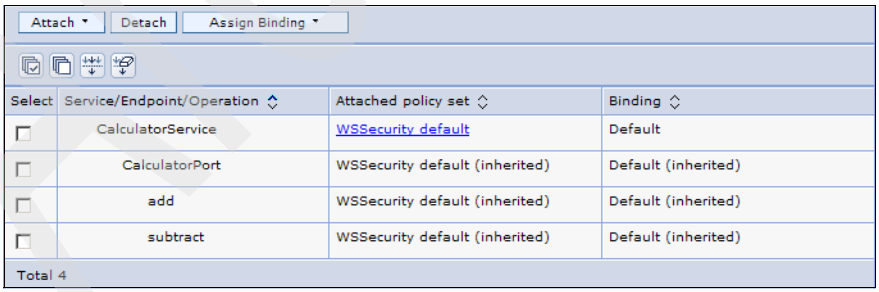


Figure 8-45 Policy set and binding for CalculatorService

8.4.3 Attach the WS Security Policy set to the Web Service Client

We attach Policy sets to the Web service client using a wizard. Each attachment specifies an endpoint, a Policy set, and a binding. Because each configuration is specific to an application and a user, we must configure a binding for certain policy types. For a given Web service and a client of that service, the Policy sets and bindings configuration of the client must match the Policy sets of the service to function correctly.

In previous versions of WebSphere Application Server, it was not possible to apply WS-Security to the Java thin client. In Feature pack for Web services, we can use WS-Security in a thin client by applying a Policy set.

> **Note:** Another way to apply WS-Security to Java thin client is to use the Web services Security application programming interfaces (WSS API) in the Feature pack for Web services. We show how to use WSS API to configure WS-Security in "Web services security API" on page 315.

To attach the WS Security Policy set to the Web service client:

- In the Project Explorer view, expand the **JAX-WS Web Services** folder → expand the **Clients** folder.

  Under this folder you see the entry **CalculatorJava**, which is the Java thin client for our Web services.

- Right-click **CalculatorJava** → **Manage Policy set Attachment...** (Figure 8-46).

  This invokes the Client Side Policy set Attachment Wizard.

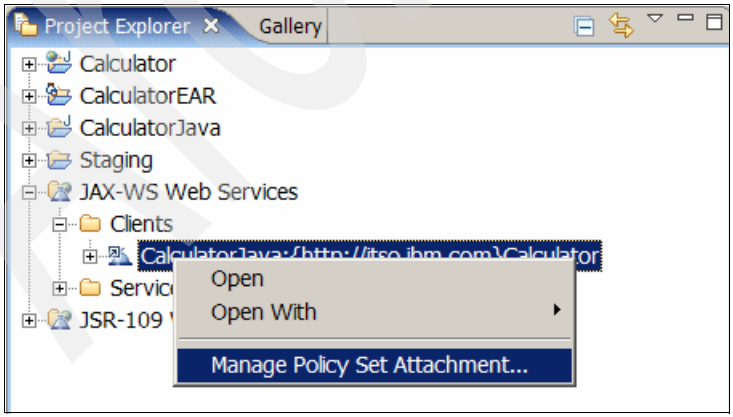


Figure 8-46 Attach Policy set to the client

- In the **Add Policy set Attachment to Web Service Client** page → select **CalculatorClient** from the Client project drop-down list if it is not already selected → **Add…**

  This opens the End Point Definition Dialog. The Policy sets and bindings configuration for our Web service client must match the service to function correctly. So we should apply the WS-Security default Policy set to the client.

- From the drop-down list, select the **WS Security default** Policy set.
- In the **Binding** field, type `ClientBinding` for the binding that you want to associate with your Policy set → **OK** → **Cancel** (Figure 8-47).

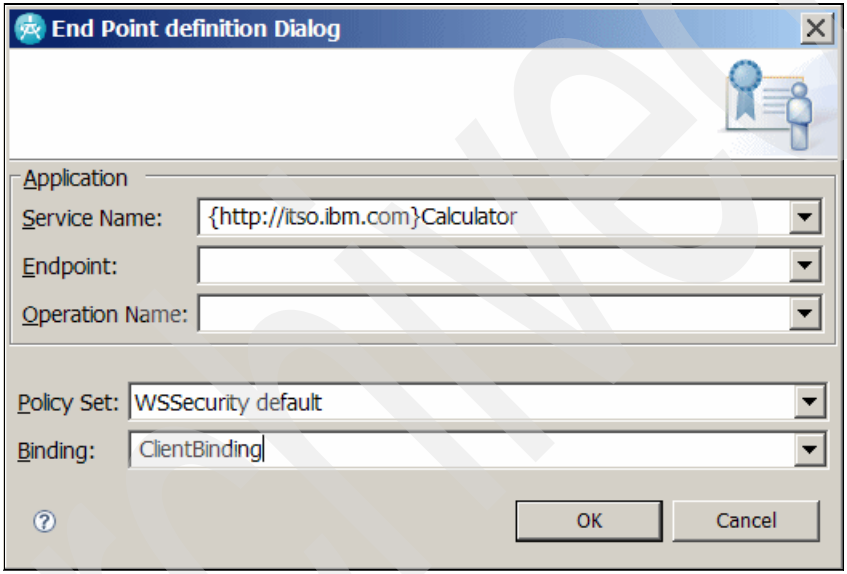


Figure 8-47 End point definition dialog for the client

> **Limitation:** At this time, the End Point Configuration dialog box does not dismiss when we click **OK**. After cancelling, only the WS-Addressing binding configuration is available, and not WS-Security.

Because the information that a binding contains is unique to a given environment or platform, we can use each binding with only one Policy set. The binding is located in the workspace so that the Java thin client can access the WS-Security binding information.

The policy types contained by the Policy set are listed in the Bindings Configuration table of the Add Policy set Attachment to Web Service Client dialog. Any of these policies that require additional configuration information are marked. For the WS-Security default Policy set, the WSAddressing policy does not have to be configured, while the WSSecurity policy does. Therefore, this dialog gives us an error message because we have not configured the client side binding (Figure 8-48).

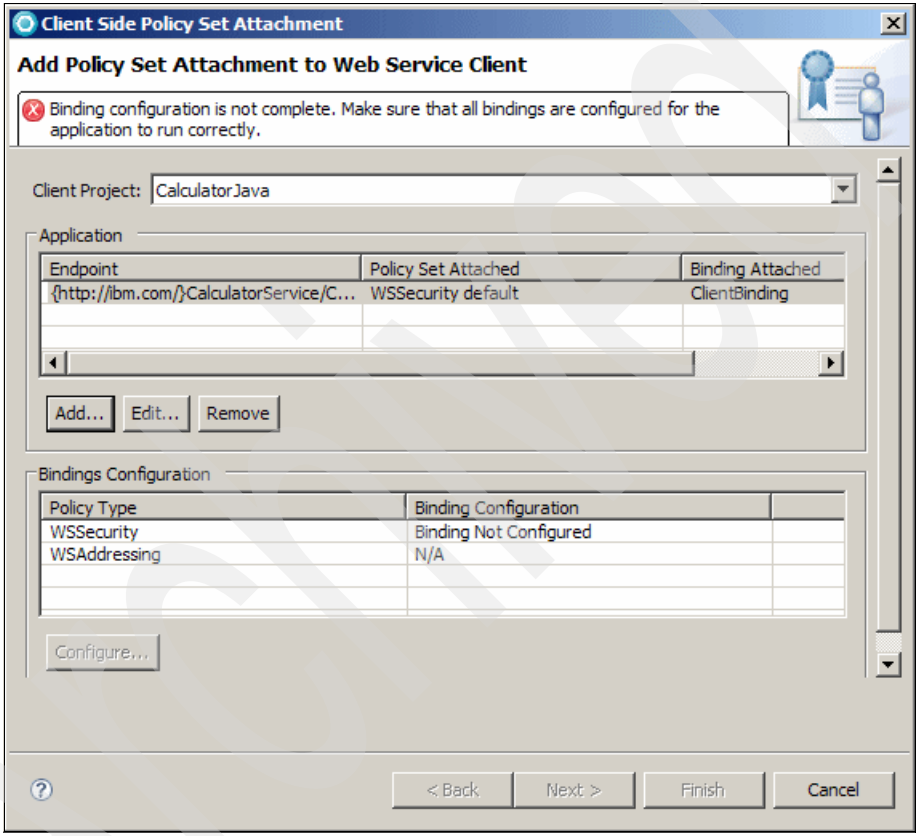


Figure 8-48 Add Policy set attachment to Web service client

Configure the Web service client custom binding

As discussed, there are no default bindings on the client side. We have to go through the task of configuring our own client bindings as custom bindings. To configure the client custom binding, complete the following steps:

- Select the **WS-Security** Policy type inside the Binding Configuration group → **Configure…**.(Figure 8-49).

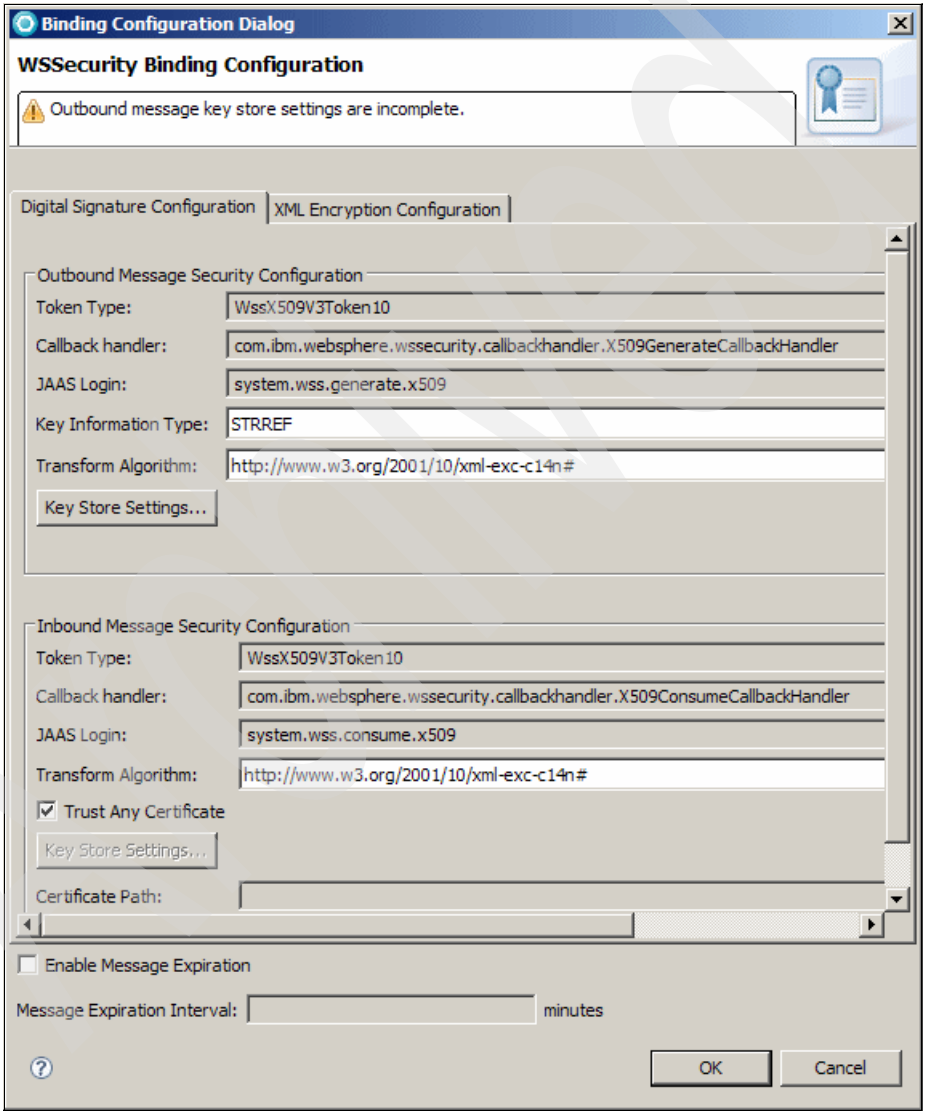


Figure 8-49 WSSecurity Binding Configuration

This Dialog has two tabs: **Digital Signature Configuration** and **XML Encryption Configuration.** Digital signature ensures message integrity, and XML Encryption ensures message confidentiality. The Warning message in the banner, `Outbound Message Key store settings are incomplete`, indicates missing information. We use the sample security files included with WebSphere Application Server to configure WS-Security.

On the Digital Signature Configuration tab, the outbound message security configuration is for the Web service client's request to the server.

- ☐ Click **Key Store Settings**... → Complete the fields as shown in Figure 8-50, substituting the correct Keystore Path for your installation[11] → **OK**.

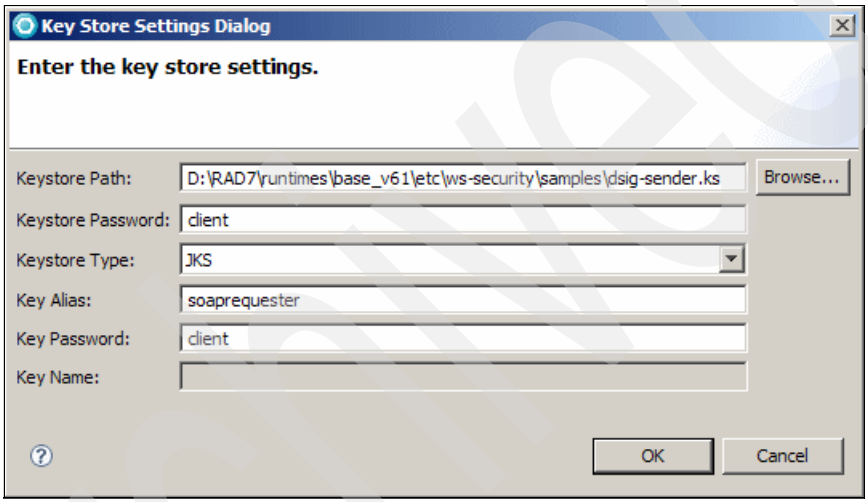


Figure 8-50 Keystore setting for outbound digital signature

[11] If you are using a stand-alone WebSphere Application Server, the samples are found at: %WebSphere Install Directory%etc\ws-security\samples\dsig-sender.ks

► Switch to the **XML Encryption Configuration** tab (Figure 8-51).

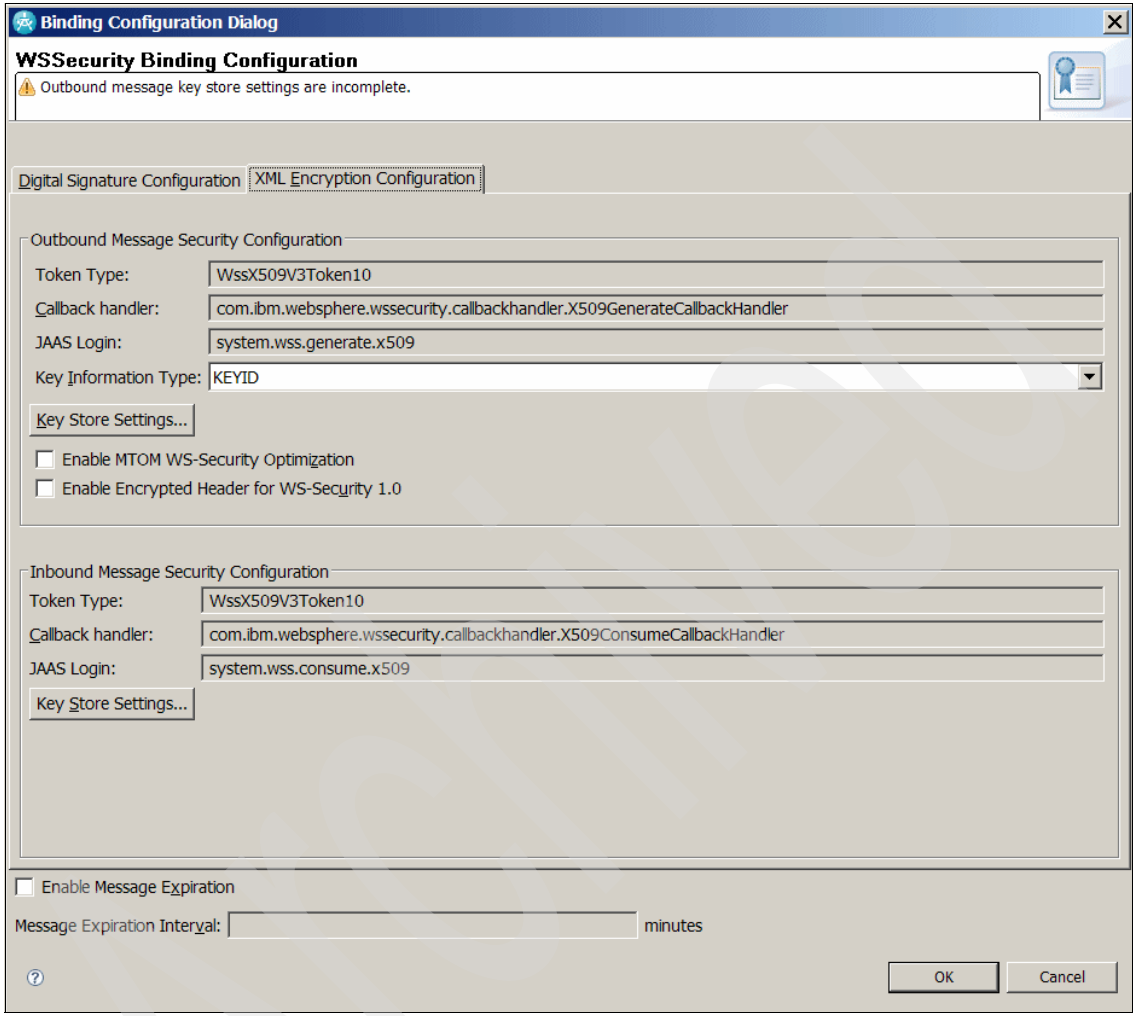


Figure 8-51 XML Encryption Configuration tab

On the WS-Security Binding Configuration tab, the outbound message security configuration is for the Web service client's request to the server.

☐ Click **Key Store Settings**... → Complete the fields as shown in Figure 8-52, substituting the correct Keystore Path for your installation[12] → **OK.**

[12] If you are using a stand-alone WebSphere Application Server, the samples are found at: %WebSphere Install Directory%etc\ws-security\samples\enc-sender.jceks

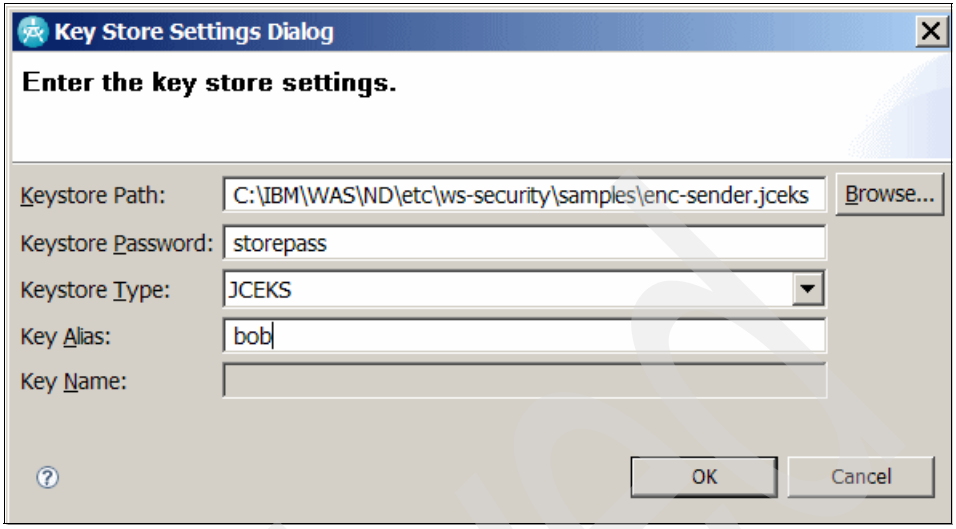


Figure 8-52 Outbound message encryption keystore settings — using Bob's public key

On the WS-Security Binding Configuration tab, the inbound message security configuration is for the Web service client's request to the server.

- Click **Key Store Settings...** → Complete the fields as shown in Figure 8-53, substituting the correct Keystore Path for your installation[13] → **OK**.

  Notice the Key Name field (circled with a dotted line) in this figure[14].

[13] If you are using a stand-alone WebSphere Application Server, the samples are found at %WebSphere Install Directory%etc\ws-security\samples\dsig-sender.ks

[14] The Key Name is generated from the Key Alias when the configuration is finished. Reopen the Key store settings dialog to see the Key Name.

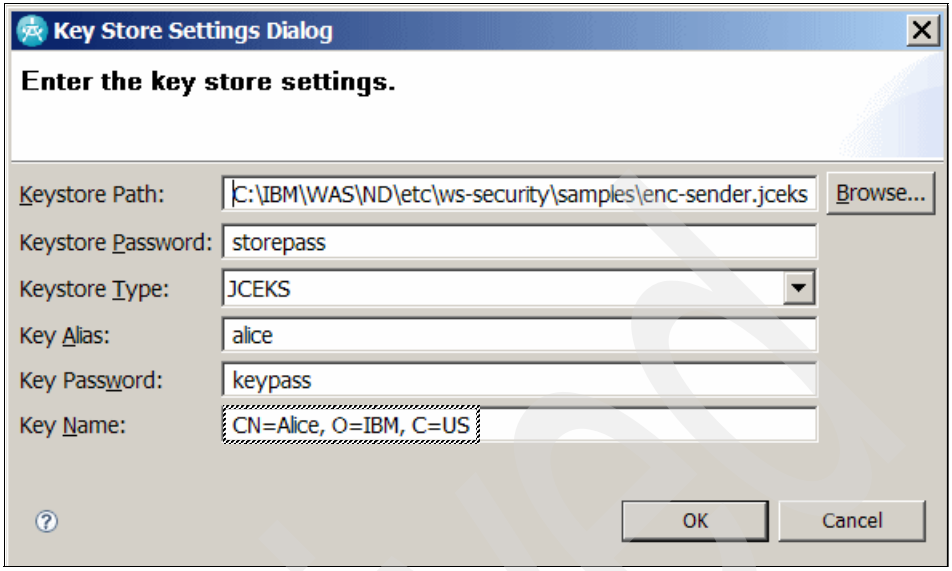


Figure 8-53 inbound message encryption keystore settings — using Alice's private key

Now there are no more warnings in the WS Security Binding Configuration Dialog. This means that we have finished entering all the required fields.

- Click **OK**.

This takes us back to the Add Policy set Attachment to Web Service Client Wizard. The WS Security shows up in the Binding Configurations table as Binding Configured. And the error message that we had before is not there any more.

- Click **Finish**.

The clientPolicyAttachments.xml file is generated and stored in the Client Application EAR, in the META-INF folder. This file contains the Policy set attached, and the name of the binding used.

Invoke Web services from Java thin client

Once we have applied the Policy set to the Web service and the Web service client, you are ready to test it:

- In the Package Explorer, expand the project **CalculatorJava** → **src** → **com.ibm**.
- Right-click **TestCalculator.java**, and choose **Run As** → **Run...**.

- ☐ In the left pane, double-click **Java Application** to create a configuration.
- ☐ In the right pane, click the **Arguments** tab and then put in a **VM argument**:

```
-Djava.security.auth.login.config=%WebSphere Install Root%\profiles\
%Feature Pack for Web services enabled profile%\properties\wsjaas_cl
ient.conf
```

- ☐ **Apply** → **Run** (Figure 8-53).

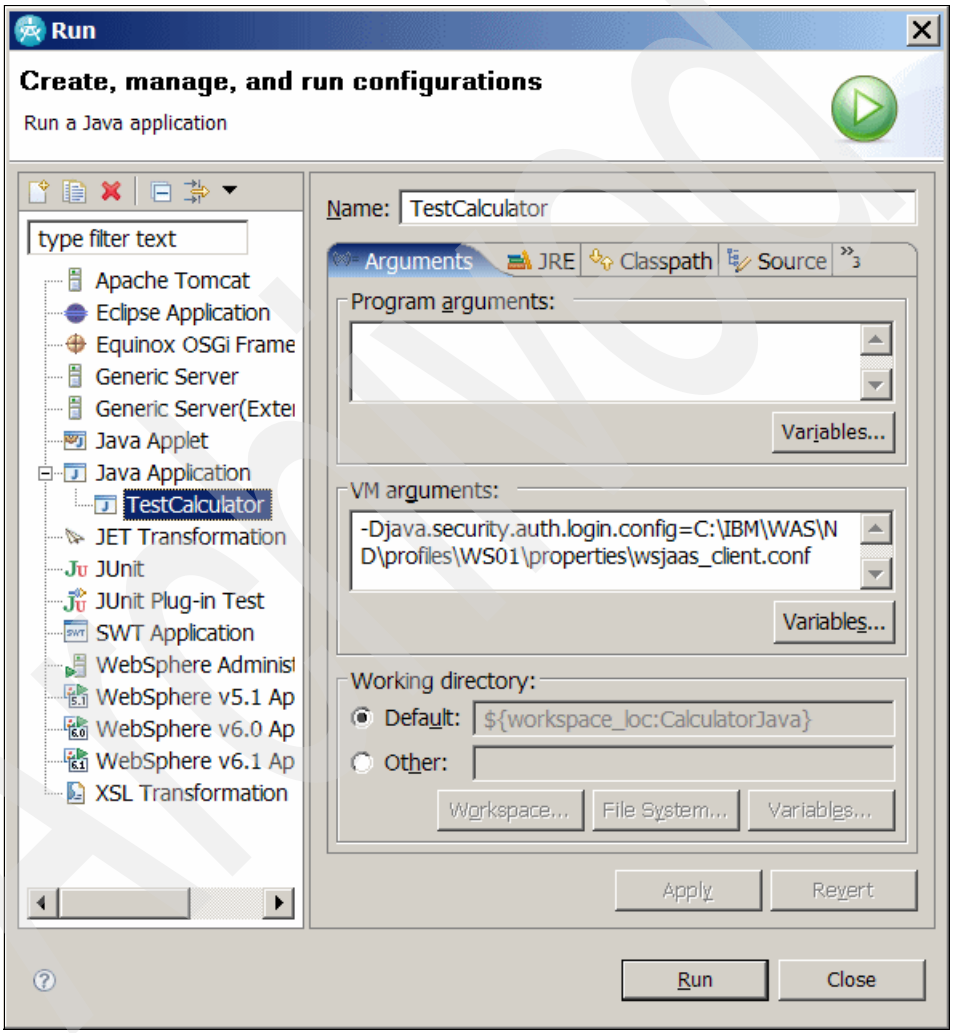


Figure 8-54 Run configuration

The correct result is displayed in the console as shown in Example 8-6.

Example 8-6 Results of calculator shown in console

```
24-Apr-2008 16:17:28 com.ibm.ws.ssl.config.SSLConfigManager
INFO: ssl.disable.url.hostname.verification.CWPKI0027I
Calculator client 1 + 1 = 2.0
Calculator client 1 - 1 = 0.0
```

If you do not get the results shown, check in the server console log, to see if the service ran (Example 8-7).

Example 8-7 Server log

```
[24/04/08 16:17:33:859 BST] 00000026 SystemOut O Calculator service 1.0 + 1.0 = 2.0
[24/04/08 16:17:34:046 BST] 00000026 SystemOut O Calculator service 1.0 - 1.0 = 0.0
```

Monitor the SOAP traffic

To confirm that the WS Security policy is successfully applied to the Web services and the client, monitor the SOAP traffic using the TCP/IP monitor. First we have to modify the Java thin client to send the SOAP request to the port that the TCP/IP monitor is listening to:

- Open **TestCalculator.java** in your **CalculatorJava** project and add this line right after creating the CalculatorPortProxy object (supposing that the TCP/IP Monitor is listening to 9089):

  ```
  proxy._getDescriptor().setEndpoint("http://localhost:9089/Calculator/Ca
  lculator");
  ```

- Make sure that your TCP/IP monitor is started.

  Run the Java application again. The SOAP messages appear in the TCP/IP monitor. Examine the SOAP request and response. We can see that the WS-Addressing is enabled and the message is signed and encrypted (Example 8-8).

Example 8-8 SOAP request

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope
   xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:wsa="http://www.w3.org/2005/08/addressing">
   <soapenv:Header>
      <wsse:Security soapenv:mustUnderstand="1" xmlns:wsse="uri 1[15]">
         <wsu:Timestamp wsu:Id="wssecurity_signature_id_9" xmlns:wsu="uri 2[16]">
            <wsu:Created>2008-04-24T15:17:34.15Z</wsu:Created>
         </wsu:Timestamp>
         <wsse:BinarySecurityToken wsu:Id="x509bst_13" EncodingType="uri 3[17]"
            ValueType="uri 4[18]" xmlns:wsu="uri 5[19]">...WRyQvO9voY...</wsse:BinarySecurityToken>
         <EncryptedKey xmlns="http://www.w3.org/2001/04/xmlenc#">
            <EncryptionMethod
               Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5">
            </EncryptionMethod>
            <ds:KeyInfo
               xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
               <wsse:SecurityTokenReference>
                  <wsse:KeyIdentifier EncodingType="uri 6[20]" ValueType="uri 7[21]">
                     /62wXObED7z6c1yX7QkvN1thQdY=
                  </wsse:KeyIdentifier>
               </wsse:SecurityTokenReference>
            </ds:KeyInfo>
            <CipherData><CipherValue>...Josqyx/vrXPeOHk=...</CipherValue></CipherData>
            <ReferenceList>
               <DataReference URI="#wssecurity_encryption_id_14"></DataReference>
               <DataReference URI="#wssecurity_encryption_id_15"></DataReference>
            </ReferenceList>
         </EncryptedKey>
         <EncryptedData Id="wssecurity_encryption_id_15"
            Type="http://www.w3.org/2001/04/xmlenc#Element"
            xmlns="http://www.w3.org/2001/04/xmlenc#">
            <EncryptionMethod
               Algorithm="http://www.w3.org/2001/04/xmlenc#aes128-cbc">
            </EncryptionMethod>
            <CipherData><CipherValue>...OQkFxF5RUrVEj/FR3t9SOs= ...</CipherValue></CipherData>
         </EncryptedData>
```

[15] uri 1 = http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd
[16] uri 2 = http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd
[17] uri 3 = http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#Base64Binary
[18] uri 4 = http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509v3
[19] uri 5 = http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd
[20] uri 6 = http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#Base64Binary
[21] uri 7 = http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509SubjectKeyIdentifier

```
      </wsse:Security>
      <wsa:To wsu:Id="wssecurity_signature_id_10" xmlns:wsu="uri 8[22]">
         http://localhost:9089/Calculator/Calculator
      </wsa:To>
      <wsa:MessageID wsu:Id="wssecurity_signature_id_11" xmlns:wsu="uri 9[23]">
         urn:uuid:C3BA517140B549C0481209050254259
      </wsa:MessageID>
      <wsa:Action wsu:Id="wssecurity_signature_id_12" xmlns:wsu="uri 10[24]">
         http://itso.ibm.com/subtract
      </wsa:Action>
   </soapenv:Header>
   <soapenv:Body wsu:Id="wssecurity_signature_id_8" xmlns:wsu="uri 12[25]">
      <EncryptedData Id="wssecurity_encryption_id_14"
         Type="http://www.w3.org/2001/04/xmlenc#Content"
         xmlns="http://www.w3.org/2001/04/xmlenc#">
         <EncryptionMethod
            Algorithm="http://www.w3.org/2001/04/xmlenc#aes128-cbc">
         </EncryptionMethod>
         <CipherData><CipherValue>...qW8fy1Xi9h...</CipherValue></CipherData>
      </EncryptedData>
   </soapenv:Body>
</soapenv:Envelope>
```

8.5 More information

The WebSphere Application Server Application Version 6.1 Feature Pack for the Web Services Information Center has a large amount of detail on Policy sets, for example:

http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/topic/com.ibm.websphere.wsfep.multiplatform.doc/info/ae/ae/cwbs_wsspsps.html

For complete instructions on 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”:

http://www.ibm.com/developerworks/websphere/library/techarticles/0712_levay/0712_levay.html

[22] uri 8 = http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd
[23] uri 9 = http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd
[24] uri 10 = http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd
[25] uri 12 = http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd

The developerWorks 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”, also shows a variety of WS-Security configurations using the Policy set:

http://www.ibm.com/developerworks/websphere/library/techarticles/0801_zeitouni/0801_zeitouni.html

9

Secure conversation

This chapter introduces the WS-SecureConversation support in Feature pack for Web services.

We review the basic concepts in WS-Security and give you an overview of WS-Trust. Then, we explore the motivation and mechanism of WS-SecureConversation. We also introduce the WS-SecureConversation scenarios. Finally, we provide two examples to apply WS-SecureConversation to a Web services application using WebSphere Application Server and Rational Application Developer.

9.1 WS-Security review

WS-SecureConversation is built on top of the WS-Security and WS-Trust models to provide secure communication across one or more messages. Let us first review the WS-Security concepts.

9.1.1 Message-level security versus transport-level security

Web services is secured using two mechanisms: transport-level security and message-level security.

Traditionally Web services 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 there are network intermediaries and multiple session to connect the client with the server, there are separate trust relationships associated with the separate point-to-point communications. The trust relationship is between the requester and the intermediary, and between the intermediary and the Web service (Figure 9-1).

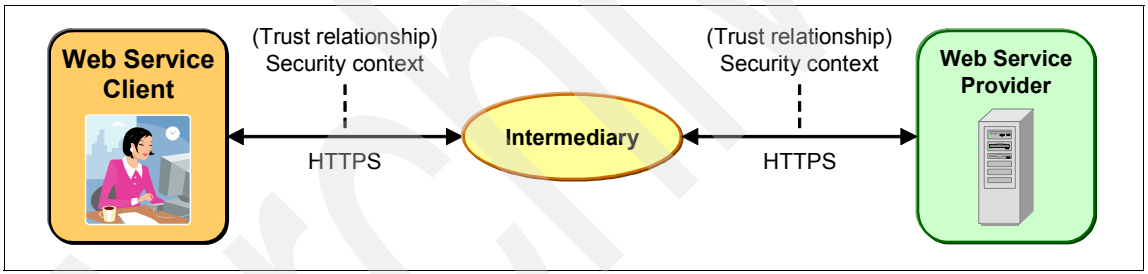


Figure 9-1 Transport-level security

This is fine in some scenarios, but might not be acceptable in others. With intermediaries, the entire message has to be decrypted to access the routing information, which would 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 is done through a combination of message integrity, confidentiality, and use of security tokens to verify messages.

The trust relationship is established between the requester and the target Web service, even when there is an intermediary involved in the message flow, as seen in Figure 9-2. Sometimes intermediaries have to 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.

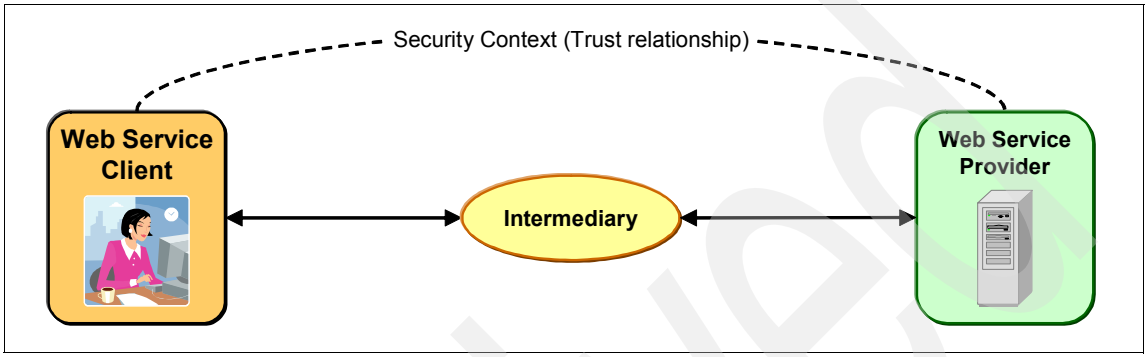


Figure 9-2 Transport-level security

9.1.2 Three major issues in WS-Security

WS-Security addressed three major issues involved in securing SOAP message exchanges: authentication, message integrity, and message confidentiality.

Authentication

Authentication is used to ensure that parties within a business transaction are really who they claim to be; thus proof of identity is required. This proof can be claimed in various ways:

1. One simple way is by presenting a user identifier and a password. This is referred to as a username token in WS-Security domain.
2. A more complex way is to use an X.509 certificate issued by a trusted Certificate Authority.

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 itself and a separate piece of information that is digitally signed 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

In order to validate a message has not been tampered with or corrupted during its transmission over the Internet, the message can be digitally signed 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 their private key, and the sender is able to validate the response has not been tampered with or corrupted 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.

9.1.3 Digital signature and XML encryption

WS-Security uses digital signature to provide message integrity, XML encryption to provide XML encryption, and security tokens to authenticate the client. Let us take a closer look at digital signature and XML encryption technologies.

Digital signature

A digital signature provides the message integrity. Here is how it works:

1. The sender first creates a hash of the SOAP message to be sent using a certain algorithm. A hash function takes the SOAP message as input and produces a fixed length string as output, sometimes termed 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) 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 using the *sender's public key*. The result is the value of the hash the sender put in the signature. Then, the receiver run 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 had changed during transmission, these two hash values will not match. The whole process is shown in Figure 9-3.

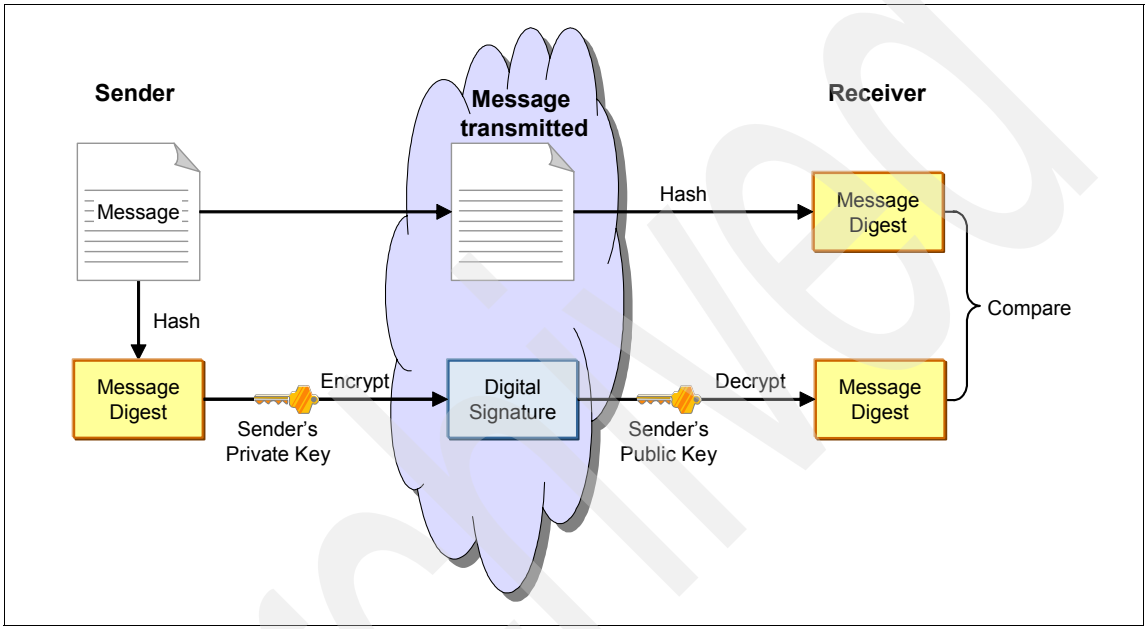


Figure 9-3 Digital signature

XML encryption

There are two popular algorithms in modern encryption methods. One is the symmetric key algorithm, and the other 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 using the shared key and the receiver must use the same key to decrypt the message.

In a asymmetric key algorithm, there are a pair of keys in a certificate: 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 the keys 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.

Symmetric key algorithm is more efficient than 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 get the best of both worlds, XML encryption usually uses two-phase process using both symmetric and asymmetric algorithms:

1. The sender first generates a symmetric key. This key will only be used for one communication session, so it is referred to as the session key. The sender uses it to encrypt the message.
2. The sender encrypts session key itself using the *public key of the message recipient*, and attaches the encrypted key to the message. By encrypting the session key instead of the whole chunk of SOAP message using asymmetric key algorithm, XML encryption provides a more efficient way to encrypt the SOAP message.
3. The recipient gets the message, and uses *its own private key* to decrypt the session key and then use that to decrypt the message. Because the session key is only used once, even if someone managed to discover it, they would only be able to decrypt one message. The whole process is shown in Figure 9-4.

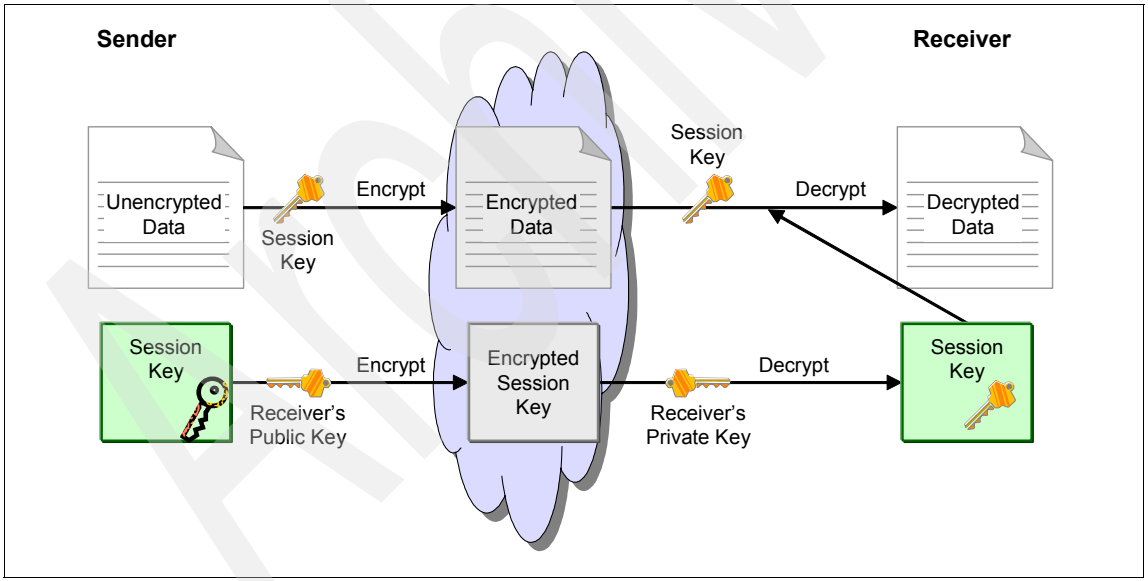


Figure 9-4 XML encryption

XML encryption provides both a *relatively* efficient solution and one that is easy to manage 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. It would be nice if you were 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 "WS-SecureConversation" on page 322.

9.1.4 WS-Security support in Feature pack for Web services

The Feature pack for Web services provides full support for the following OASIS specifications and WS-I profiles:

- OASIS: WS-Security: SOAP Message Security 1.0 (WS-Security 2004)
- OASIS: WS-Security: UsernameToken Profile 1.0
- OASIS: WS-Security X.509 Certificate Token Profile 1.0
- WS-I Basic Security Profile (WS-I BSP) 1.0

The Feature pack for Web services also supports the following specifications:

- OASIS: WS-Security: Username Token Profile 1.1
- OASIS: WS-Security X.509 Certificate Token Profile 1.1, which includes support for the Thumbprint type of security token reference.
- Several key functions of WS-Security: SOAP Message Security 1.1 (WS-Security 2004), including: signature confirmation, the ability to encrypt SOAP headers, and the ability to add a thumbprint that references an X.509 certificate.
- OASIS: WS-Trust Version 1.1 submission draft.
- OASIS: WS-SecureConversationVersion 1.1 submission draft

9.1.5 Web services security API

In previous releases of WebSphere Application Server, Web services security had to be configured using the deployment descriptor. There was no application programming interfaces (API) support. The deployment descriptor approach meant that the client had to be a JAX-RPC managed client. The Feature pack for Web services includes a Web services security API (WSS API) that is used to add WS-Security on Web Services clients, either a J2EE client, or a J2SE™ client.

The configuration model for Web services has also been redesigned from a deployment descriptor model to a policy set model. The configuration programming model is based on configuring policy sets using a security policy to specify security constraints. The functions provided by the policy set configurations are the same as the functions supported by the WSS API for the Web Service Security runtime. Web Service Security is enabled by either using a policy set that is configured using the administrative console, or by using the WSS API for configuration.

The security policy that is defined using policy sets has a higher priority over the WSS API. When the WSS API and the policy set are both used in the application, the default behavior is for the security policy from the policy set to be enforced and the WSS API to be ignored. To use the WSS API in the application, make sure that there is no policy set attached to the application or to the application resources, or make sure there is no security policy in the attached policy set.

The WSS API uses default values for most of the parameters. These defaults are overridden where needed. The WSS API supports the following specifications:

- WS-Security 1.0 and 1.1 specifications
- WS-SecureConversation
- Username Token Profile 1.0 and 1.1.
- X.509 Certificate Token Profile 1.0 and 1.1.

Configure Web service security using the WSS APIs example

In 8.4.3, “Attach the WS Security Policy set to the Web Service Client” on page 296, you applied WS-Security policy set to Calculator Web service application. In this section, you use the same Calculator Web service application, and you figure Web services security using the WSS API instead. To complete this, do the following tasks:

- ☐ Follow “Prepare for the sample” on page 257 to import the Calculator Web service application. If you have imported the application and configured it using the Policy Set, delete the existing application and import it again.
- ☐ Apply the WSSecurity default Policy Set to your Web service, as documented on “Attach the WS Security Policy set to the Web service” on page 292.
- ☐ In the Package Explorer, expand the project **CalculatorJava** → **src** → **com.ibm.itso** double-click **TestCalculator.java** to open it.

  The complete code listing for *TestCalculator.java* is in Example A-13 on page 547. We will not list the complete code here. Let us quickly review the highlights of the sample code.
- ☐ Get the message context to apply WS-Security to the SOAP request and response. Example 9-1 shows how to get the message context.

Example 9-1 Get the message context

```
CalculatorSOAPProxy proxy = new CalculatorSOAPProxy(
      new URL(endpointURL + "?wsdl"),
      new QName("http://itso.ibm.com", "Calculator"));
Calculator port = proxy._getDescriptor().getProxy();
BindingProvider bp = (BindingProvider) port;
Map<String, Object> requestContext = bp.getRequestContext();
Map<String, Object> responseContext = bp.getRequestContext();
```

- Example 9-2 shows how to add a digital signature to sign your outcoming SOAP request.

Example 9-2 Add a digital signature to sign the SOAP request

```
// Generate the WSSFactory instance (step: a)
WSSFactory factory = WSSFactory.getInstance();
// Generate the WSSGenerationContext instance (step: b)
WSSGenerationContext gencont = factory.newWSSGenerationContext();
X509GenerateCallbackHandler callbackHandler = new X509GenerateCallbackHandler(
      "", KEY_PATH + "dsig-sender.ks", "jks", "client".toCharArray(),
      "soaprequester", "client".toCharArray(),
      "CN=SOAPRequester, OU=TRL, O=IBM, ST=Kanagawa, C=JP", null);
// Generate the security token to be used for the signature (step:c)
SecurityToken token = factory.newSecurityToken(X509Token.class, callbackHandler);
// Generate the WSSSignature instance (step: d)
WSSSignature sig = factory.newWSSSignature(token);
// Add the WSSSignature to WSSGenerationContext (step: e)
gencont.add(sig);
```

- Example 9-3 shows how to encrypt your outcoming SOAP request.

  The default key encryption algorithm for WSS API is *key wrap RSA OAEP*, while the default key encryption algorithm for *WSSecurity default Policy Set* is *key wrap RSA 1.5*. The WSS API provides the flexibility to configure advanced options to set the key encryption method, as highlighted in the example.

Example 9-3 Encrypt the outcoming SOAP request

```
// Generate the callback handler
X509GenerateCallbackHandler callbackHandler2 = new X509GenerateCallbackHandler(
      "", KEY_PATH + "enc-sender.jceks", "jceks", "storepass".toCharArray(),
      "bob", null, "CN=Bob, O=IBM, C=US",null);
// Generate the security token used for encryption (step: c)
SecurityToken token2 = factory.newSecurityToken(X509Token.class, callbackHandler2);
// Generate WSSEncryption instance (step: d)
WSSEncryption enc = factory.newWSSEncryption(token2);
// DEFAULT: WSSEncryption.KW_RSA_OAEP
enc.setKeyEncryptionMethod(WSSEncryption.KW_RSA15);
gencont.add(enc);
gencont.process(requestContext);
```

- Example 9-4 how to decrypt the incoming SOAP response and verify the signature of the incoming SOAP response.

Example 9-4 Decrypt the incoming SOAP response

```
WSSConsumingContext concont = factory.newWSSConsumingContext();
X509ConsumeCallbackHandler callbackHandler3 = new X509ConsumeCallbackHandler(
      "", KEY_PATH + "enc-sender.jceks", "jceks", "storepass".toCharArray(),
      "alice", "keypass".toCharArray(), "CN=Alice, O=IBM, C=US");
// Generate the WSSDecryption instance (step: d)
WSSDecryption dec =
   factory.newWSSDecryption(X509Token.class, callbackHandler3);
dec.addAllowedKeyEncryptionMethod(WSSEncryption.KW_RSA15);
concont.add(dec);
```

When we run the *TestCalculator.java*, we have to add a VM argument, as documented in “Invoke Web services from Java thin client” on page 303.

The results at the server and client are:

```
[28/04/08 13:03:48:812 BST] 00000027 SystemOut     O Calculator service 1.0 + 1.0 = 2.0
[28/04/08 13:03:49:171 BST] 00000027 SystemOut     O Calculator service 1.0 - 1.0 = 0.0
```

```
28-Apr-2008 13:03:41 com.ibm.ws.ssl.config.SSLConfigManager
INFO: ssl.disable.url.hostname.verification.CWPKI0027I
Calculator client 1 + 1 = 2.0
Calculator client 1 - 1 = 0.0
```

The WSS API provides a simple programming model to configure WS-Security. From this example, you can see the common usage scenario, like signing and encrypting the SOAP body content, only takes about five lines of code as it provides intelligent defaults. It also provides the flexibility and fine-grained control for advanced functions.

9.2 WS-Trust

WS-Security describes enhancements to SOAP messaging to provide quality of protection through message integrity, message confidentiality, and single message authentication. One important class of WS-Security is to define mechanisms for signing and encrypting SOAP messages using security tokens. Security tokens are a collection of claims 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 use security tokens, it does not address how security tokens are issued and exchanged.

The WS-Trust specification builds on 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.

9.2.1 Security Token Service

The key concept in WS-Trust is a Security Token Service (STS). An STS 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, an STS requires proof, such as a signature to prove knowledge of a security token or set of security tokens.

The STS 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 STS to issue a security token with its own trust statement. In many cases it is used to issue the same assertions but in different formats. For example, an STS 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, as seen in Figure 9-5.

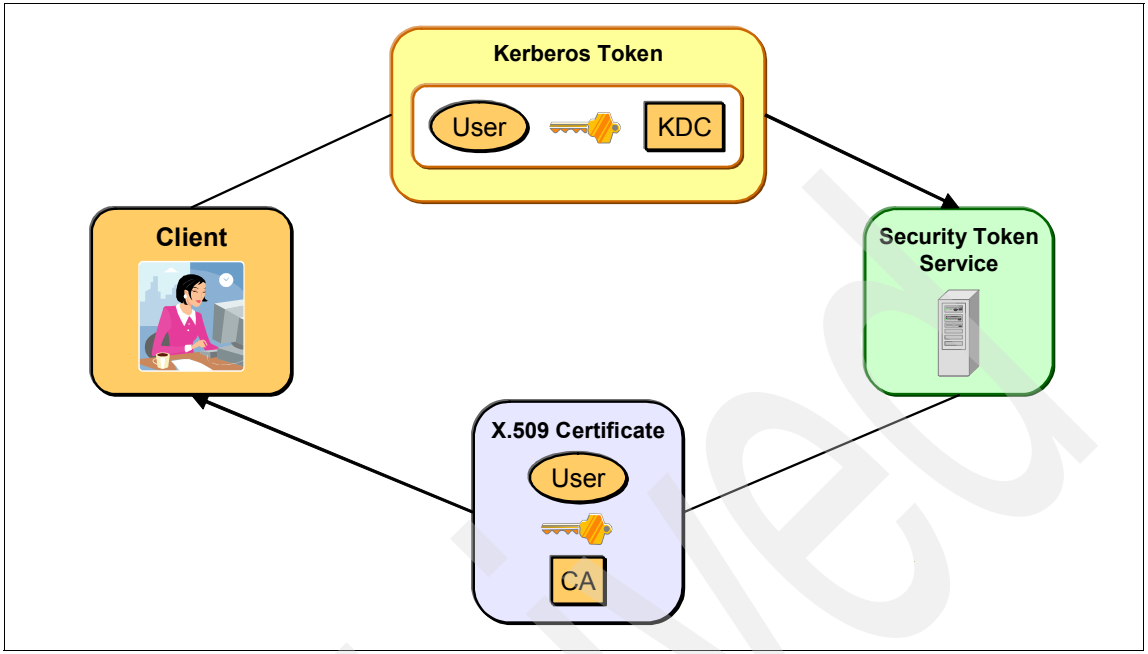


Figure 9-5 Issuing security tokens in different formats by Security Token Service

This 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.

9.2.2 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 requestor does not have the necessary token(s) to prove required claims to a service, it contacts the Security Token Service and requests the tokens with the proper claims. The Security Token Service can, in turn, 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 with the Web service and the other with the Web service client. The WS-Trust model is illustrated in Figure 9-6.

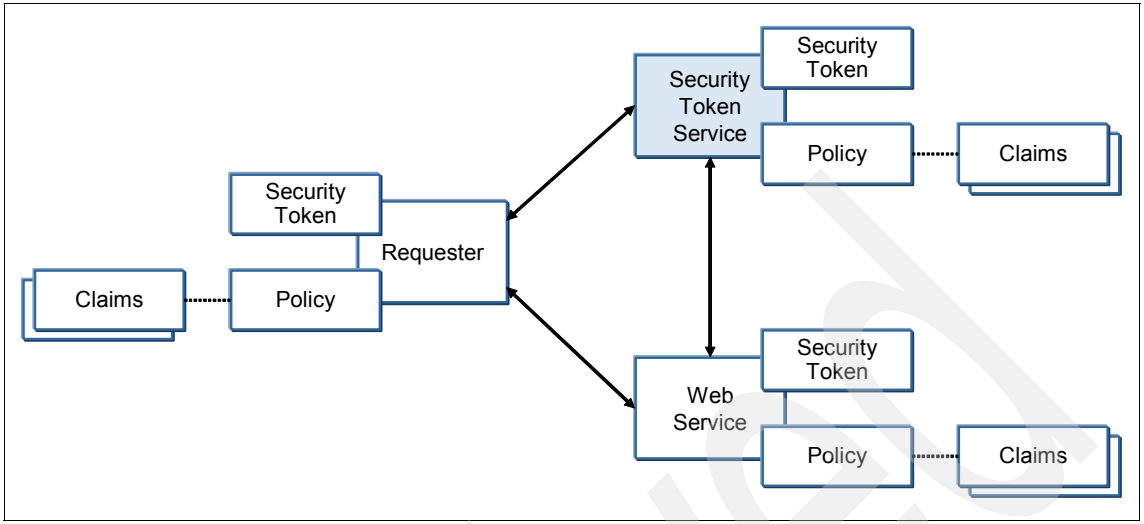


Figure 9-6 WS-Trust Model

9.2.3 Security Token Service framework

The Security Token Service defines a framework for token issuance. A requestor sends a request, and if the policy permits and the recipient's requirements are met, then the requestor receives a security token response. This process uses the `<wst:RequestSecurityToken>` element to send the request and `<wst:RequestSecurityTokenResponse>` element to receive the new or renewed security token.

Requesting a security token

The `<wst:RequestSecurityToken>` element (RST) is used to request a security token. This element is signed by the requestor, using tokens contained/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

The syntax for this element is as follows (Example 9-5).

Example 9-5 <wst:RequestSecurityToken> element

```
<wst:RequestSecurityToken Context="...">
<wst:TokenType>...</wst:TokenType>
```

```
<wst:RequestType>...</wst:RequestType>
...
</wst:RequestSecurityToken>
```

Returning a security token

The `<wst:RequestSecurityTokenResponse>` element (RSTR) is used to return a security token or response. The security token is used in subsequent SOAP messages and referred to based on the mechanisms defined by WS-Security. The syntax for this element is as follows: (Example 9-6)

Example 9-6 `<wst:RequestSecurityTokenResponse> element`

```
<wst:RequestSecurityTokenResponse Context="...">
<wst:TokenType>...</wst:TokenType>
<wst:RequestedSecurityToken>...</wst:RequestedSecurityToken>
...
</wst:RequestSecurityTokenResponse>
```

WS-Trust in Feature pack for Web services

Feature pack for Web services supports the OASIS Version 1.1 submission draft, which became available in February 2005. However, the Feature pack for Web services has not implemented all the contents of the WS-Trust draft specification. The implementation of WS-Trust establishes a Security Context Token for WS-SecureConversation using the four requests to obtain and manage the Security Context Token: issue, renew, validate, and cancel. The implementation by the Feature pack for Web services is called the trust service. The trust service uses the secure messaging mechanisms of WS-Trust to define additional extensions for the issuance, exchange, and validation of security tokens. For more details, refer to the IBM Infocenter, found at:

http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/index.jsp?topic=/com.ibm.websphere.wsfep.multiplatform.doc/info/ae/ae/cwbs_wstrust.html

9.3 WS-SecureConversation

The WS-Security (WS-Security) Version 1.0 standard from OASIS[1] defined how to digitally sign and encrypt the SOAP message to provide message level protection. The standard also defined how to attach and reference a security token for digital signature and encryption. But it did not provide session-based protection when a long series of related messages were exchanged.

[1] http://specs.xmlsoap.org/ws/2005/02/sc/WS-SecureConversation.pdf

The WS-SecureConversation standard is a building block that is used in conjunction with the other Web service and application-specific protocols such as WS-Security and WS-Trust to accommodate a wide variety of security models and technologies. WS-SecureConversationis built on top of the WS-Security and WS-Trust models to provide secure communication across one or more messages. The WS-SecureConversationdraft specification describes how to establish a Security Context Token between two parties, and it uses WS-Trust specification to issue and exchange Security Context Tokens.

9.3.1 Motivation

There are two major driving forces for WS-SecureConversation; one is from the performance perspective, the other is from the security requirements of WS-ReliableMessaging.

Improve 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, multi-message conversations between the Web services. WS-Security also supports this model, but the solution is not optimal.

There are two sub-optimal usages of WS-Security in these scenarios:

- Repeated use of computationally expensive cryptographic operations such as public key validation.
- Sending and receiving many messages using the same cryptographic keys, providing more information that allows brute force attacks to “break the code”.

For these reasons, protocols like HTTP/S use public keys to perform a simple negotiation that defines conversation specific keys. This key exchange allows more efficient security implementations and also 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”, and use WS-SecureConversation to agree on session specific keys for signing and encrypting information.

Protect the sequence of reliable messaging

Reliable messaging in Web services allows applications to send and receive messages simply, reliably, and efficiently even in the face of application, platform, or network failure. It 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, and exposes 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 lifecycle or traffic messages. Imagine there are two valid clients each with a sequence. Both are authorized at the service level, but one of the clients is actually a hacker, and he wants to attack the other sequence. If he can guess the sequence identifier, then he can create a fake TerminateSequence message that references the target Sequence and sends this message to the appropriate RM Destination, as seen in Figure 9-7.

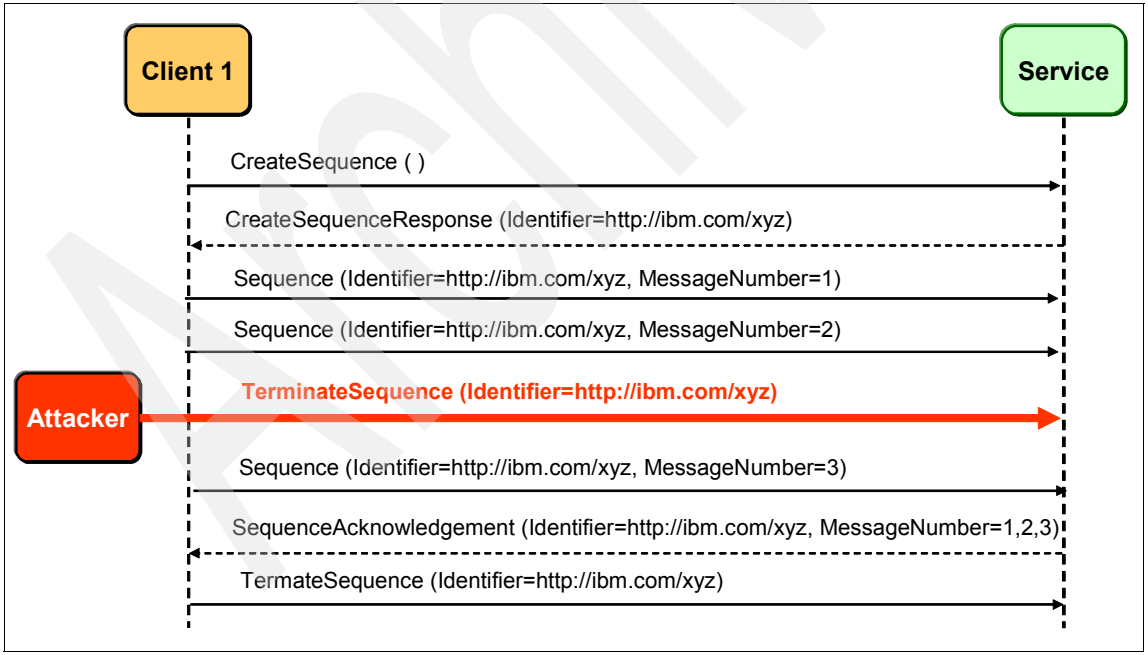


Figure 9-7 Sequence attack

WS-SecureConversation provides a mechanism for protecting sequences. We explain the mechanism in 9.3.4, “Secure Conversation with Reliable Messaging scenario”.

9.3.2 Key concepts

In this section, we introduce two key concepts in WS-SecureConversation: Security Context Token and 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; 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-SecureConversationspecification, a security context is represented by the `<wsc:SecurityContextToken>` security token. The following URI represents the Security Context Token type that is required to establish a secure conversation:

`http://schemas.xmlsoap.org/ws/2005/02/sc/sct`

To request a Security Context Token, a Request Security Token (RST), which is a message sent to request a security token, 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 requestor as if it was generated by the endpoint service.

Example 9-7 shows an RST request to issue a security token.

Example 9-7 Security Context Token Request Example

```
<wst:RequestSecurityToken Context="http://www.ibm.com/login/">
<wst:RequestType>http://schemas.xmlsoap.org/ws/2005/02/trust/Issue</wst:RequestType>
      <wst:TokenType>http://schemas.xmlsoap.org/ws/2005/02/sc/sct</wst:TokenType>
          <wst:Entropy>
```

```
            <wst:BinarySecret>swYVsjsi75fB+RksmDdWKQ==</wst:BinarySecret>
        </wst:Entropy>
      <wsp:AppliesTo xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy">
                <wsa:EndpointReference>
                     <wsa:Address>WSSample/services/EchoService</wsa:Address>
                </wsa:EndpointReference>
      </wsp:AppliesTo>
</wst:RequestSecurityToken>
```

Example 9-8 shows a RSTR response to issue a security token:

Example 9-8 Request Security Token Response Example

```
<wst:RequestedSecurityToken
           xmlns:wst="http://schemas.xmlsoap.org/ws/2005/02/trust">
        <wsc:SecurityContextToken
             xmlns:wsc="http://schemas.xmlsoap.org/ws/2005/02/sc"
             xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/
                        oasis-200401-wss-wssecurity-utility-1.0.xsd"
             wsu:Id="uuid:617A2281DAD3C3EC211179342073467">
             <wsc:Identifier xmlns:wsc="http://schemas.xmlsoap.org/ws/2005/02/sc">
                          uuid:617A2281DAD3C3EC211179342073466
             </wsc:Identifier>
             <wsc:Instance xmlns:wsc="http://schemas.xmlsoap.org/ws/2005/02/sc">
                    uuid:617A2281DAD3C3EC211179342073465
             </wsc:Instance>
        </wsc:SecurityContextToken>
     </wst:RequestedSecurityToken>
<wst:RequestedProofToken xmlns:wst="http://schemas.xmlsoap.org/ws/2005/02/trust">
           <wst:ComputedKey xmlns:wst="http://schemas.xmlsoap.org/ws/2005/02/trust">
                   http://schemas.xmlsoap.org/ws/2005/02/trust/CK/PSHA1
           </wst:ComputedKey>
</wst:RequestedProofToken>
```

Derived Key

A Security Context Token implies or contains a shared secret. This secret could be used for signing and encrypting messages, but 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 using the same key in multi-message conversations is also considered bad practice because it gives too much data to attackers to analyze. We recommend that Derived Keys are used for signing and encrypting messages associated only with the security context.

WS-SecureConversation provides a secured session for long running message exchanges and leveraging of the symmetric cryptographic algorithm using coordinated Derived Keys.

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 seen in Figure 9-8.

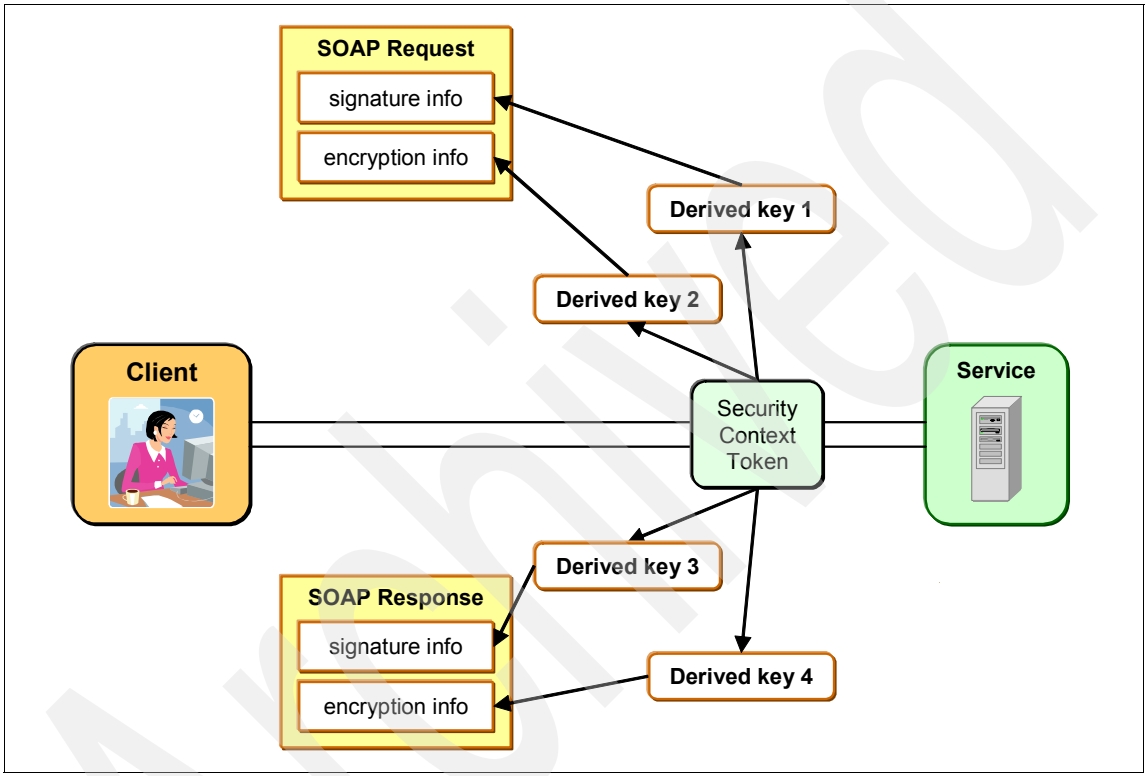


Figure 9-8 use Derived Keys to sign and encrypt SOAP messages

In order to keep the keys fresh, subsequent derivations can be used in multi-message 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, we can then leverage symmetric cryptography to sign and encrypt the message. A symmetric cryptographic algorithm is less CPU intensive than the asymmetric cryptography, and provides better performance and throughput when compared to the asymmetric cryptographic algorithms.

9.3.3 Secure Conversation scenario

This section describes the overall flow on the message exchanges in WS-SecureConversation. During the secure conversation session, the initiator establishes the Security Context Token (SCT) 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 9-9 describes the flow that is required to establish a secured context and to use session-based security.

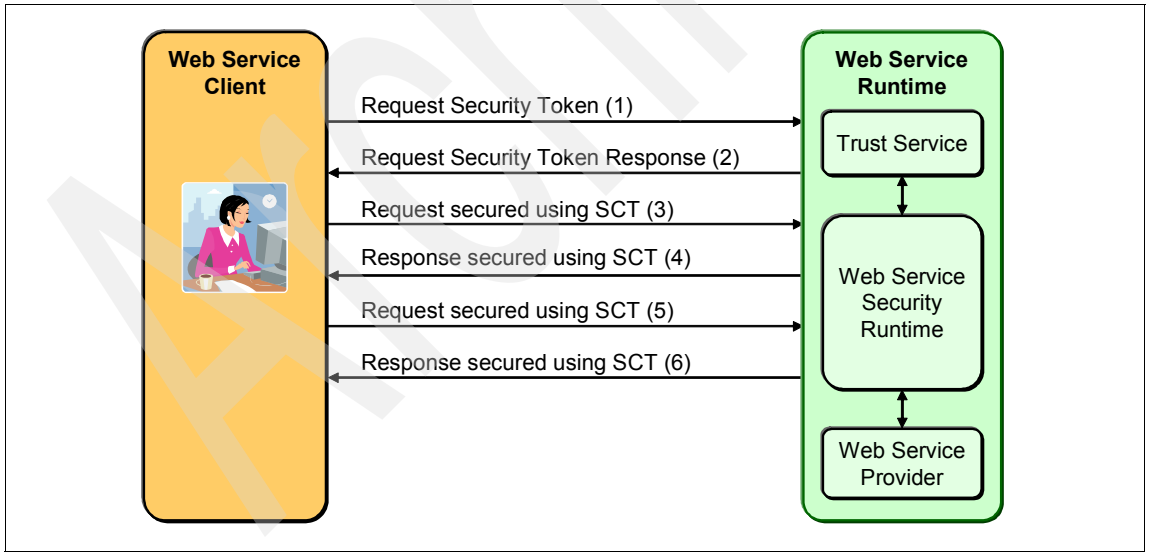


Figure 9-9 Message overflow in secure conversation

To use secure conversation, the following steps are involved:

1. The client sends a Request Security Token (RST) for a Security Context Token to an application endpoint. The RST is encrypted and signed using WS-Security information defined in the bootstrap security policy. The bootstrap policy is used by the initiator to acquire security token from 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 using Request Security Token Response (RSTR). The RSTR is also signed and encrypted to ensure 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 RequestSecurityTokenResponse is trusted, the client secures (signs and encrypts) the subsequent application messages by using the Derived Keys. The Derived Keys are derived from secret of 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, repeat step 3 and 4 until the communications session is finished.

Deep dive

To help understand WS-SecureConversationbetter, let us take an in-depth look into the two stages of WS-SecureConversation; the establishment of a Security Context Token, and the usage of Derived Keys.

Establish Security Context Token

Figure 9-10 on page 330 shows how the messages are exchanged between the initiator and the recipient to establish the Security Context Token in WebSphere Application Server.

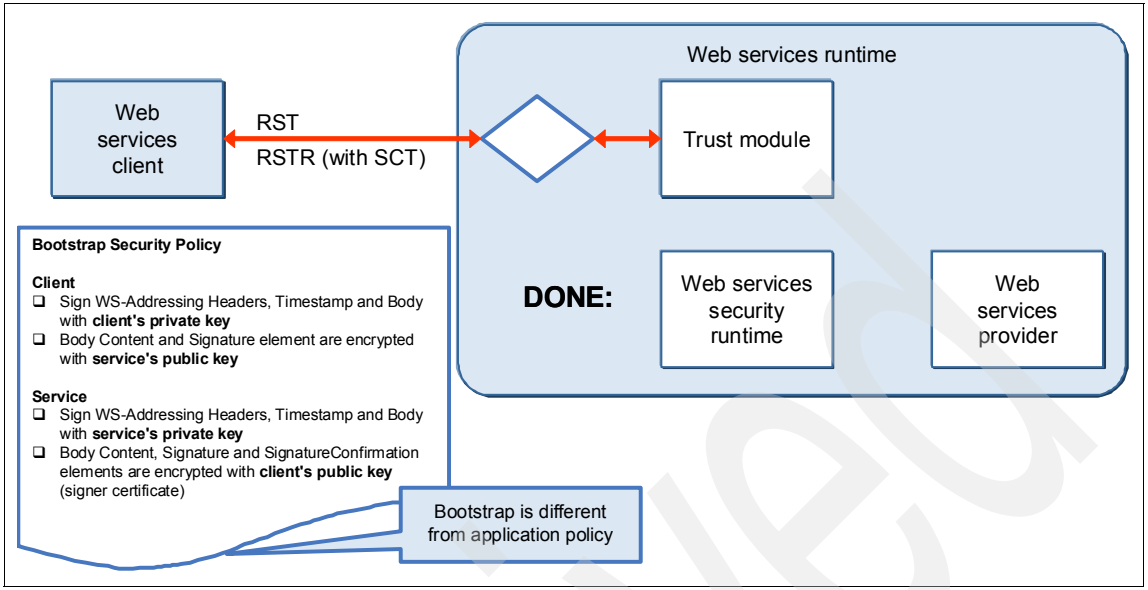


Figure 9-10 Establish Security Context Token between initiator and recipient

The Web services client sends a Request Security Token (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 using the trust service's public key.

The Trust service that processed the request is trusted based on the bootstrap policy. After the request has been validated, the trust service returns the Security Context Token using Request Security Token Response (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 using the client's public key. This process uses Asymmetric cryptography algorithm.

Note that the bootstrap policy, which is used to secure the RST and validate the RSTR request, is different from the application security policy.

Use Derived Key

After the Security Context Token is established, the application messages are secured using Derived Key based on the Security Context Token, as seen in Figure 9-11.

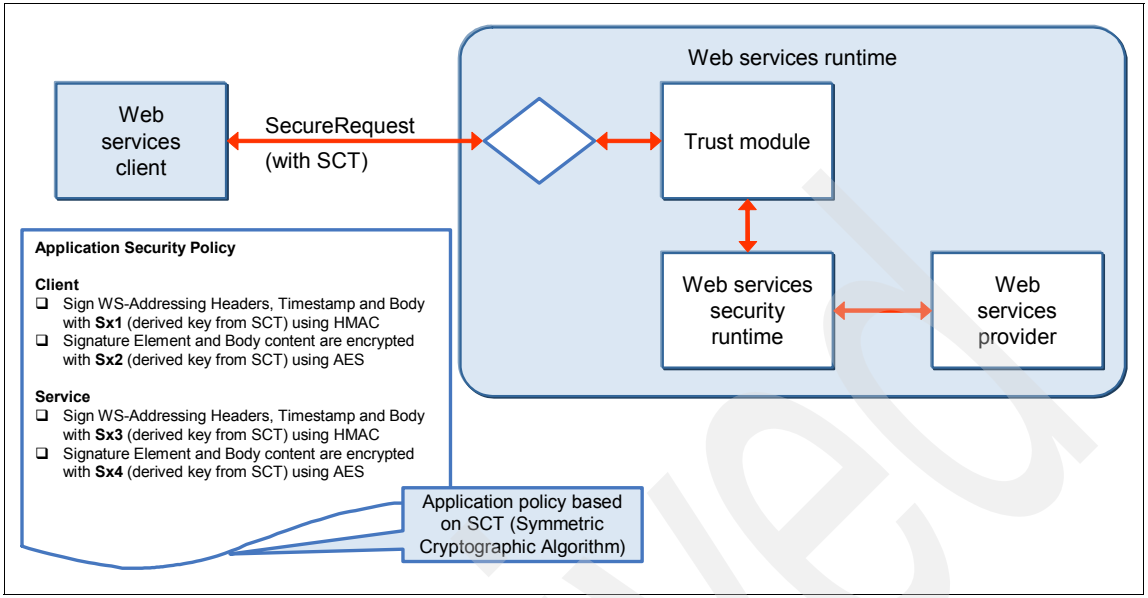


Figure 9-11 Derived Key is used to secure the application messages

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 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 Symmetric cryptography algorithm to communicate, which is more efficient than Asymmetric cryptography algorithm.

Configure trust service using System Policy Sets

In Chapter 8, “Policy sets” on page 247, we discussed how to use 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 to issue, cancel, renew, and validate security tokens. Because the trust service is a Web service itself, 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 System Policy Set and Application Policy Set is that System Policy Set is not available for application resources.

- ☐ In the admin console, expand **Services** → **Policy sets** → **System policy sets**.

The default System Policy Sets shipped with WebSphere Application Server are displayed (Figure 9-12).

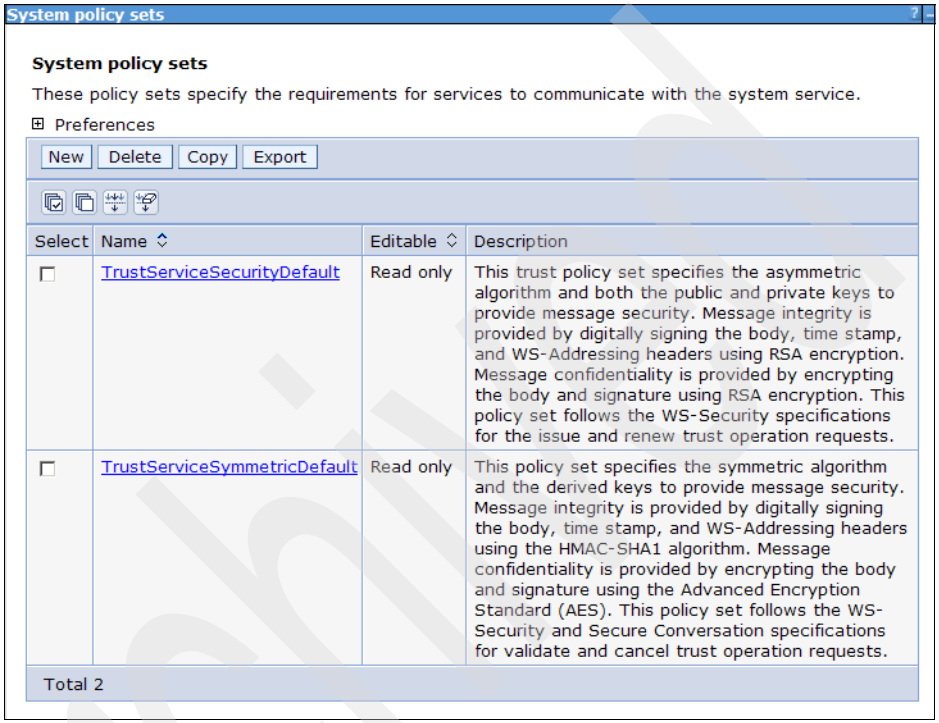


Figure 9-12 System Policy set

WebSphere Application Server provides two system policy sets for the security trust service: TrustServiceSecurityDefault and TrustServiceSymmetricDefault. The description for these two default Policy Sets is given in Figure 9-12.

We can create our 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 (STR) 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 using secure conversation, you might have to create a custom System Policy Set to use a key identifier.

You can also update the Security Context Token provider configuration for the trust service

- Click **Services** → **Trust service** → **Token providers** → **Security Context Token** (Figure 9-13).

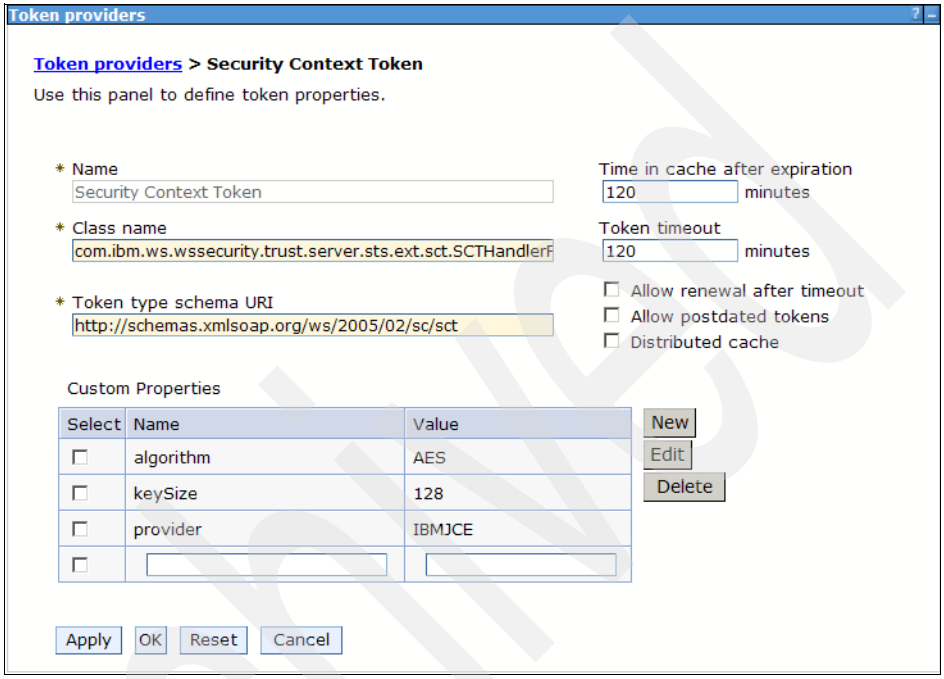


Figure 9-13 Update the Security Context Token provider configuration

- Click **Services** → **Trust service** → **Targets**. The configuring trust service endpoint targets panel is opened, as seen in Figure 9-14.

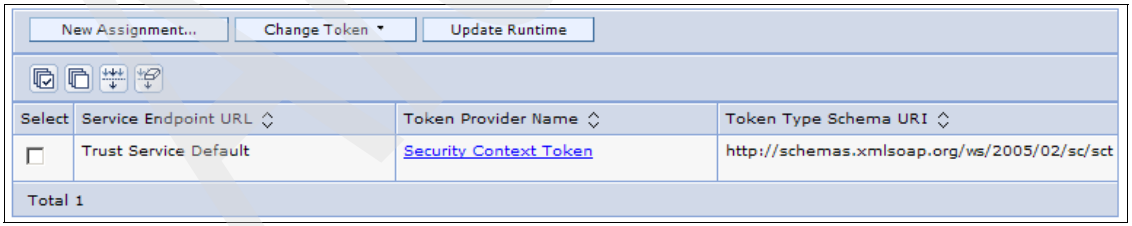


Figure 9-14 Configure trust service endpoint targets

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.

- Click **Services** → **Trust service** → **Trust service attachments**. The trust service attachments page is opened, as seen in Figure 9-15.

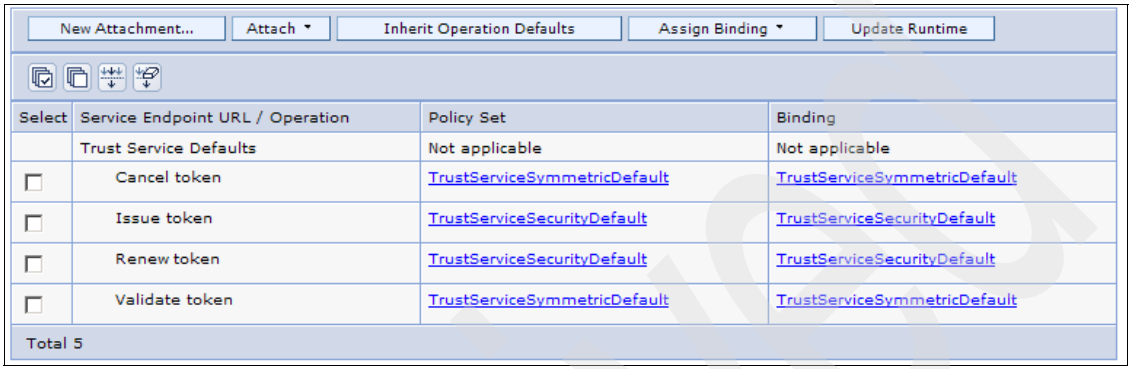


Figure 9-15 Trust service attachment

We attach the trust service operations for a service endpoint to a system policy set and binding. Each new endpoint that is specified initially has the following four operations: issue, renew, cancel, and 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.

9.3.4 Secure Conversation with Reliable Messaging scenario

As discussed, 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 possible sequence attack.

In the Secure Conversation with Reliable Messaging scenario, the Security Context Token is used to secure the Reliable Messaging sequence. Figure 9-16 shows the message flows that are required to establish a Security Context Token to secure reliable messaging.

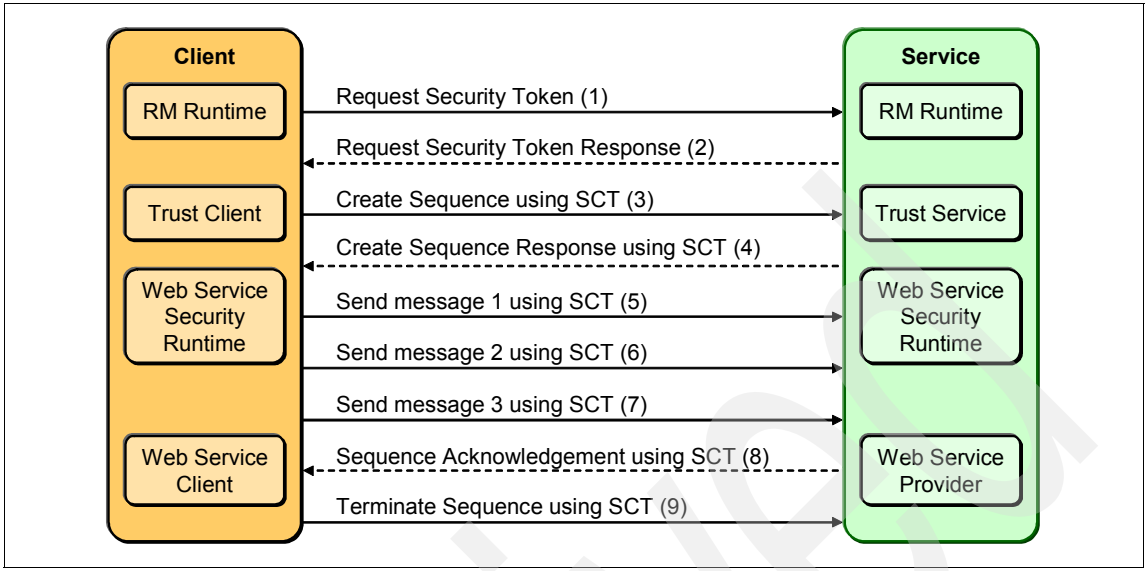


Figure 9-16 Message overflow for secure conversation with reliable messaging

To use secure conversation with reliable messaging, the following steps are involved:

1. The WS-ReliableMessaging runtime calls WS-Security APIs to get the Universally Unique Identifier (UUID) of the Security Context Token for the session. If there is already a Security Context Token established, the UUID of the existing Security Context Token is returned to WS-ReliableMessaging. If there is no Security Context Token established, the WS-Security runtime initiates a call to the recipient to establish the Security Context Token.
2. After the WS-ReliableMessaging runtime acquires the UUID of the Security Context Token, the WS-ReliableMessaging runtime scopes the CreateSequence message to the Security Context Token by using the SecurityTokenReference (STR) argument in the CreateSequence message and responds with the CreateSequenceResponse message.
3. The exchange of the application messages is similar to the WS-SecureConversationscenario. The messages are secured by Security Context Token.
4. The WS-ReliableMessaging runtime responds with the CreateSequenceResponse message.
5. The WS-ReliableMessaging runtime sends a SequenceAcknowledgement message to acknowledge that the message is properly delivered and the message is secured by the Security Context Token.

6. Finally, the WS-ReliableMessaging runtime sends a TerminateSequence message to terminate the sequence and the message is secured by the Security Context Token.

9.4 Secure conversation example

In this section, we provide two examples to show how to apply secure conversation to our Web services applications using Rational Application Developer. In the first example, we apply WS-SecureConversation to our Web service. In the second example, we apply the Reliable Asynchronous Message Profile (RAMP) to our Web services. We also monitor the SOAP traffic to see the message flows of secure conversation.

9.4.1 Apply secure conversation to Web services

The first example is to apply secure conversation to Web services. The Feature pack for Web services ships with the SecureConversation Policy Set ready to use out of the box. In order to see the SOAP traffic, we have to create a custom policy set to also include the HTTPTransport policy. We use TCPMon shipped with WebSphere Application Server to monitor the SOAP traffic.

Note: The TCP/IP Monitor shipped with Rational Application Developer is not capable of handling WS-SecureConversationrelated traffic, which includes multiple messages and SOAP envelopes in requests and responses. Due to this, we chose TCPMon shipped with WebSphere Application Server as the monitoring tool.

To use the TCPMon to monitor the WS-SecureConversationrelated traffic, we also have to add the HTTPTransport policy, because the proxy capability provided by HTTPTransport policy offers a convenient mechanism in order to get TCPMon configured to capture both the application request/response and the trust service request/response.

Rational Application Developer ships a JAX-WS address book sample, which allows you to store and retrieve information found in an address book. You apply the custom policy set to this sample.

Prepare for the sample

- ☐ Start the **WebSphere Application Server** that uses the Feature pack for Web services enabled profile.

- Create a new workspace, and add the WebSphere Application Server you started to the server view.
- To install the JAX-WS address sample from Rational Application Developer, Click **Help** → **Samples Gallery** → **Technology samples** → **Web services** → **WebSphere JAX-WS address book Web service sample** → **Import the sample**.
- Open **jwsAddressBookClient** → **WebContent** → **WEB_INF** → **wsdl** → **AddressBook.wsdl** and check the SOAP address matches your configuration → **Save**.
- Right-click **jwsAddressBookClient** → **WebContent** → **SampleAddressBookProxy** → **jwsAddressBookClient/WebContent/sampleAddressBookPortProxy/TestClient.jsp** → **Run As...** → **Run on Server** (Figure 9-17).

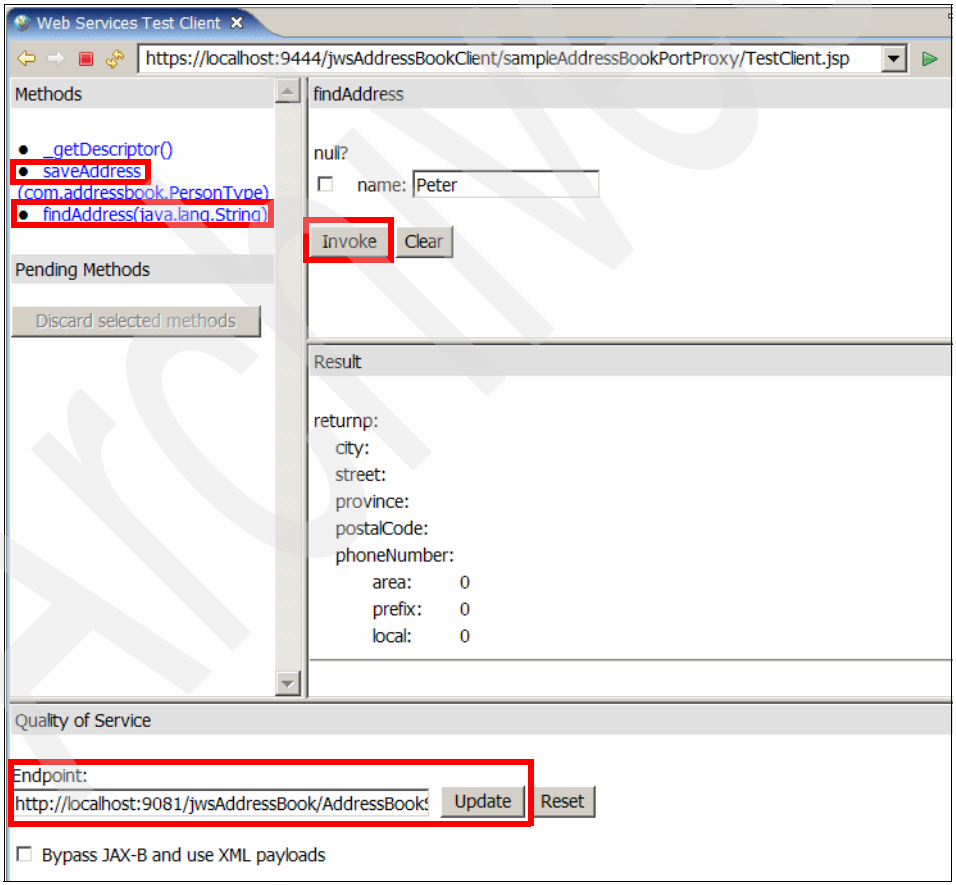


Figure 9-17 AddressBook test client

You might be prompted to associate the client with a server in the workspace. This also prompts you to deploy the .ear file.

- ☐ Check the **Endpoint** and update it if necessary.
- ☐ Select the **saveAddress** method → Type a name in the **Name** field. All other fields are optional (do not check the boxes) → **Invoke**...
- ☐ Select the **findAddress** method → Retype the **Name** again → **Invoke...**

Create the custom Policy Set

In this section, we create a custom Policy Set to include the SecureConversation policy and HTTPTransport policy. we then export the custom policy set to our local directory to import it into our Rational Application Developer development environment. To do this, complete the following steps:

- ☐ Start the **WebSphere Administrative Console** → **Log in**.
- ☐ In the left pane, select **Services** → **Policy Sets** → **Application Policy Sets**.
- ☐ In the right pane, check **SecureConversation** (Figure 9-18) → Click **Copy** at the top of the page.

☑	SecureConversation	Read only	This policy set provides message integrity by digitally signing the body, time stamp, and WS-Addressing headers. Message confidentiality is provided by encrypting the body and the signature. This policy set follows the WS-SecureConversation and WS-Security specifications.

Figure 9-18 Check SecureConversation → Copy

- Back in the Application policy sets panel → Type **ITSO SecureConversation** into the Name field (Figure 9-19) → **OK**.

Figure 9-19 name the custom Policy set

- Click **ITSO SecureConversation** Policy set → **Add**, and then click **HTTP transport** (Figure 9-20) → **Save**.

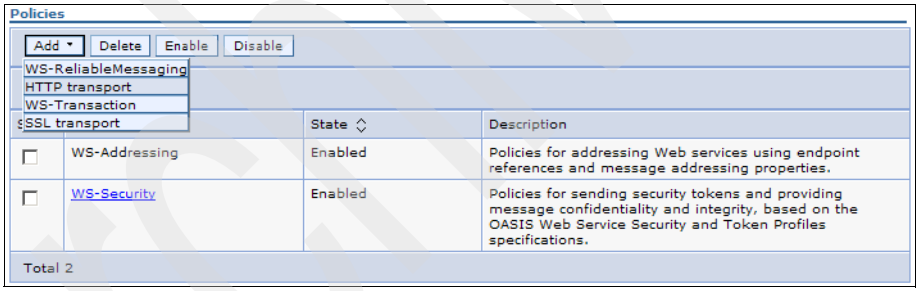


Figure 9-20 Add HTTP transport policy

- Check **ITSO SecureConversation** Policy Set → **Export**.
- Click **ITSO SecureConversation.zip** → save it to your local directory.

You have now created the custom policy set.

Apply policy set to Web service and client

In this section, we import the ITSO SecureConversation Policy Set into our Rational Application Developer development environment. Then, we apply the ITSO SecureConversation Policy Set to our Web service and client.

- In Rational Application Developer → **Window** → **Preferences** → **Web services** → **Qualities of Service** → **Import Policy Sets** → Browse to the exported **ITSO SecureConversation.zip** → **Open**. The **ITSO SecureConversation** Policy Set is imported (Figure 9-21).

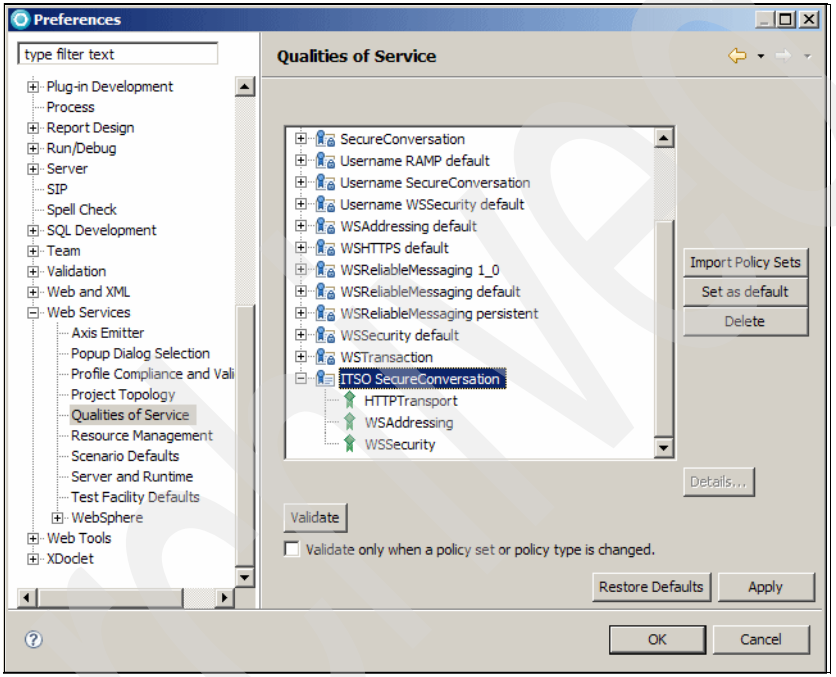


Figure 9-21 Import ITSO SecureConversation Policy set

- In the Project Explorer vie → Expand JAX-WS Web Services → **Services**.

  Under this folder there is an entry for the **jwsAddressBook** service.

- Right-click on the **jwsAddressBook** → **Manage Policy Set Attachment…** → **Add…** (Figure 9-22)
- Select the **ITSO SecureConversation** Policy set from the drop-down list and leave the **Default** Binding unchanged. Click **OK** and then click **Finish**.

  The ITSO SecureConversation Policy set is now applied to your service.

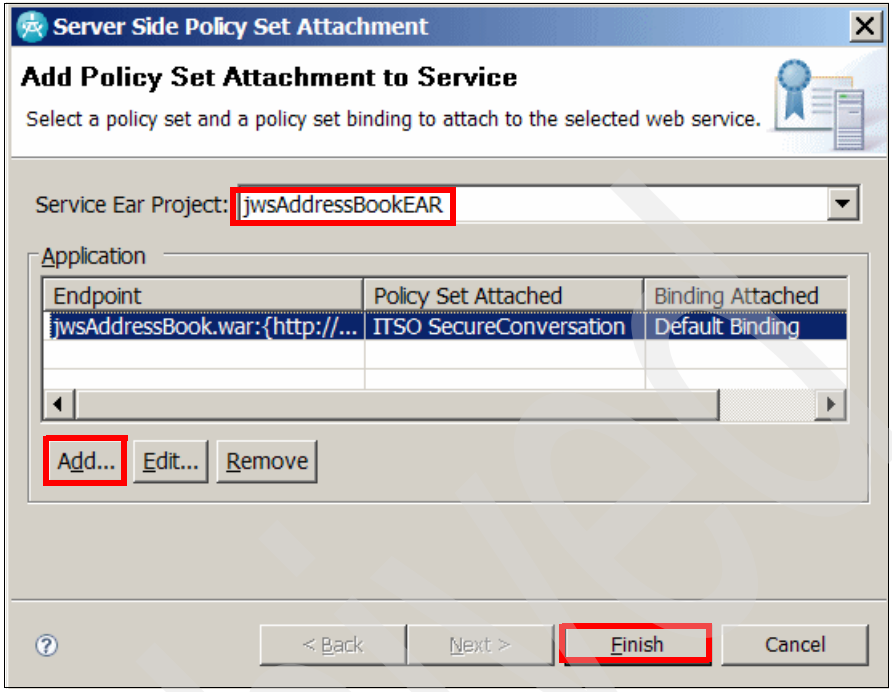


Figure 9-22 Applying ITSOSecureConversation Policy set to the service

To apply this Policy set to your client, do the following tasks:

- ☐ In the Project Explorer view → expand JAX-WS Web Services → **Clients** folder.

  Under this folder there is an entry for the **jwsAddressBook** client.

- ☐ Right-click on the **jwsAddressBook** → **Manage Policy Set Attachment…** → **Add…**
- ☐ Select **ITSO SecureConversation** Policy Set → In the Binding field, type **SCBinding** → **OK** (Figure 9-23).

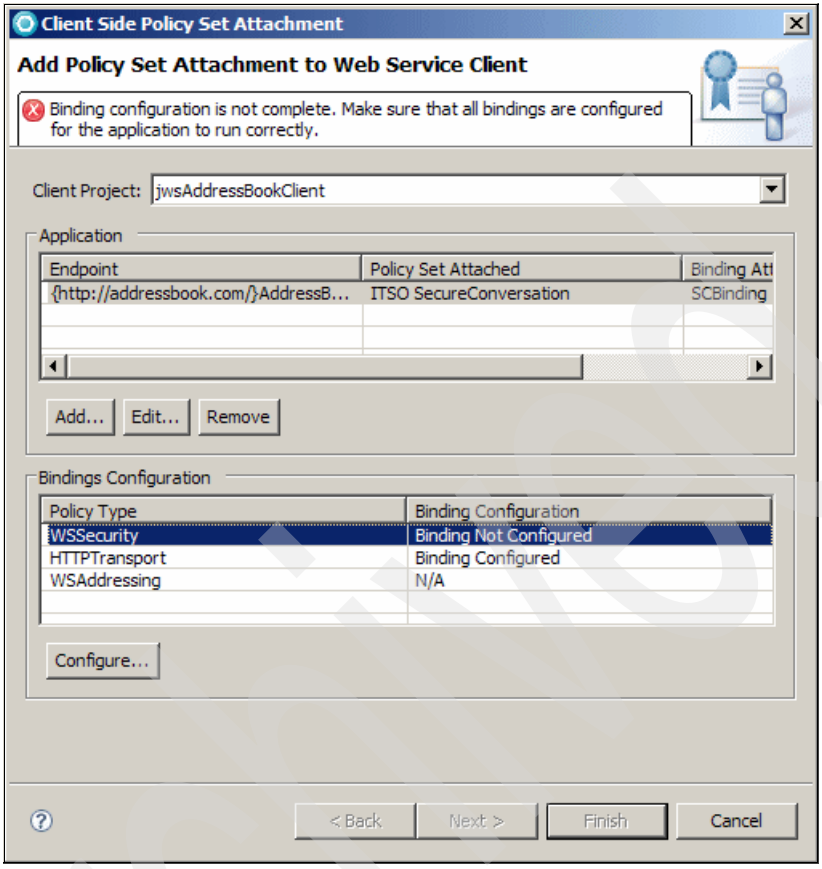


Figure 9-23 Attach ITSO SecureConversation Policy set to client

- Click **HTTPTransport** inside the Binding Configuration group → **Configure...** → Type `localhost` for the Host, → `9089` for the Port (Figure 9-24 on page 343) → **OK**.

  The reason that we choose port 9089 is that 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 will be directed to the proxy server running at http://localhost:9089. The TCPMon will be started in proxy mode and listen to port 9089. Using the proxy capability of the HTTPTransport policy was a convenient mechanism in order to get TCPMon configured to capture both the application request/response and the trust service request/response.

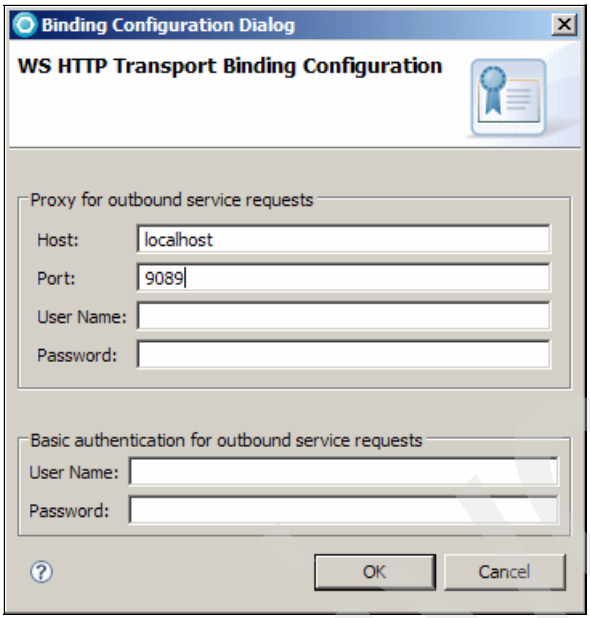


Figure 9-24 WS HTTP Transport Binding Configuration

The WSSecurity Policy Type is not configured yet. The values (in Table 9-1) that we use are the same as in Table 8-2 and Table 8-3 on page 289.

Table 9-1 Security settings for Addressbook

	Encryption client	**Signature client**
What is my identity?	Alice	soaprequester
What is my keystore called?	<%WAS_HOME%>/ etc/ws-security/samples/ enc-sender.jceks pw = storepass	<%WAS_HOME%>/ etc/ws-security/samples/ dsig-sender.ks[a] pw = client
What is my private encryption key?	alias = alice CN=Alice, O=IBM, C=US pw = keypass	alias = soaprequester CN=Alice, O=IBM, C=US pw = client
What public keys do I have for decryption?	alias = bob CN=Bob, O=IBM, C=US	soapca CN=SOAP 1.2 Test CA OU=TRL, O=IBM, l=Yamato, S=Kanagawa, C=JP

a. .ks is a synonym for jceks

- Select the **WSSecurity** Policy type inside the Binding Configuration group and click **Configure…**. (Figure 8-49 on page 299).
- Click **Key Store Settings** on the Outbound Signature tab → Complete the fields using the values in the Signature client column of Table 9-1 on page 343. The result is shown in Figure 8-50 on page 300.
- Switch to the **XML Encryption Configuration** tab → Under Outbound Message Security Configuration, click **Key Store Settings** → Type the values from the Encryption client column of Table 9-1 on page 343. The result is shown in Figure 8-52 on page 302.
- Repeat this for the Inbound Message Security Configuration (see Figure 8-53 on page 303) → **OK** → **Finish**.

Validate the Trust service bindings

We have found the Trust service bindings are similarly wrongly configured on some servers, such as the bindings to the default keystores (see "Validating the default keystore in the server" on page 290).

The Trust service bindings token for encryption (gen_enctgen) callback handler custom keystore configuration should appear as in Figure 9-25. If yours references the enc_sender.jceks keystore, and bob, then it would be trying to use Alice's keystore rather than its own, and trying to encrypt using its own public key. It should be using Alice's public key from its own enc-receiver.jceks keystore.

The symptom we observe using the shipped configuration is:

```
[29/04/08 09:46:25:176 BST] 00000035 EncryptionCon E   CWWSS5601E: The
following exception occured while decrypting the message:
java.lang.RuntimeException: javax.crypto.BadPaddingException: Not
PKCS#1 block type 2 or Zero padding
```

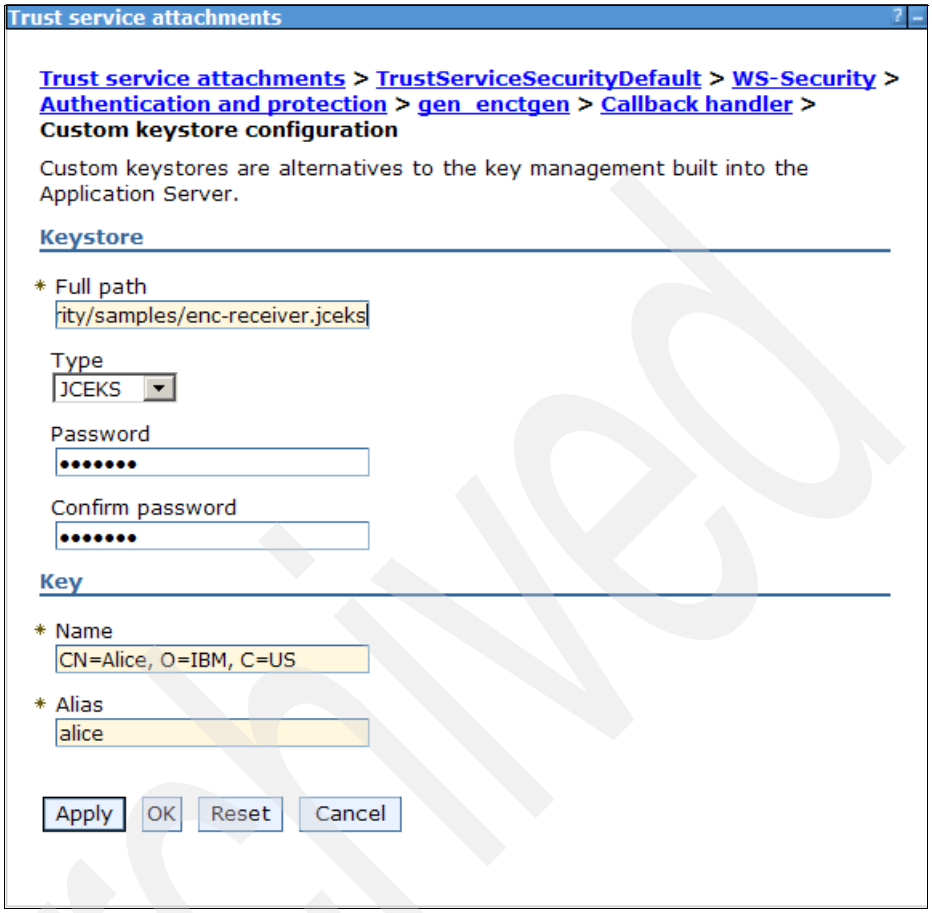


Figure 9-25 Correct configuration of the gen_enctgen custom keystore

Monitor the SOAP traffic in WS-SecureConversation

As discussed, you use TCPMon shipped with WebSphere Application Server to monitor the SOAP traffic, as the TCP/IP Monitor shipped with Rational Application Developer is not capable of handling WS-SecureConversation related traffic.

- ☐ To start TCPMon, Open a **Command Prompt** → Navigate to the `<%WAS_HOME%>\bin` folder
- ☐ Type the commands shown in Example 9-9. TCPMon is opened.

Example 9-9 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_6.1.0.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 the TCPMon is to download the TCPMon from the Apache Software Foundation website:

http://ws.apache.org/commons/tcpmon/download.cgi

Select **Binary Distribution** from this page. After the download and extraction, simply run and then extract the double-click **tcpmon.bat** in `tcpmon-1.0-bin\build` folder to start the TCPMon.

Notice that the TCPMon from Apache has more functions than the one shipped with WebSphere Application Server.

- Enter **9089** as the Listen Port # → check **Proxy** (Figure 9-26) → **Add**...

  Port 9089 is read for the SOAP traffic

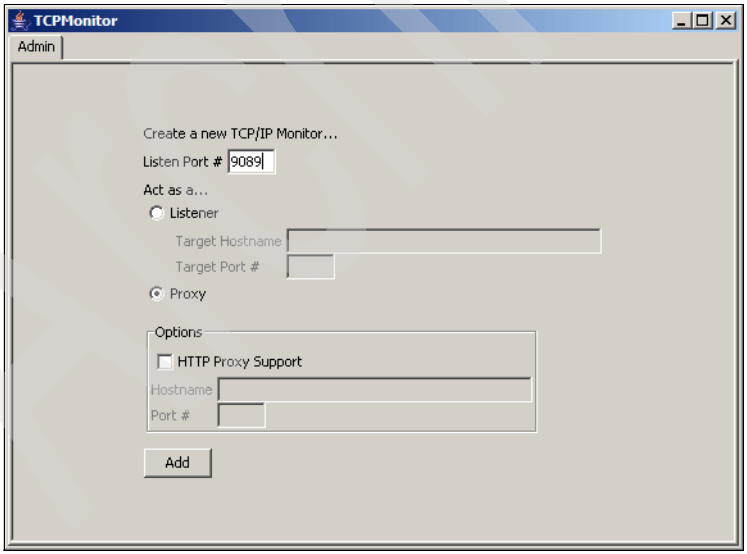


Figure 9-26 TCPMon

- ☐ Return to Rational Application Developer. In the Servers view, expand WebSphere Application Server v6.1 → **jwsAddressBookEAR** → **Restart jwsAddressBookEAR**.
- ☐ In the Project Explorer, expand **jwsAddressBookClient** → **WebContent** → **sampleAddressBookPortProxy** → Right-click **TestClient.jsp** → **Run As** → **Run on Server**.
- ☐ On the TestClient.jsp panel, find the **Quality of Service** window and set the **Endpoint** to `http://localhost:9081/jwsAddressBook/AddressBookService` → **Update**[2].
- ☐ Check the Endpoint by clicking **_getDescriptor** in the Method window.
- ☐ Click the **saveAddress** method → Type a name and a city → **Invoke**.
- ☐ In the TestClient.jsp panel, the result "trust" is displayed.
- ☐ The SOAP traffic is displayed in TCPMon, as seen in Figure 9-27.

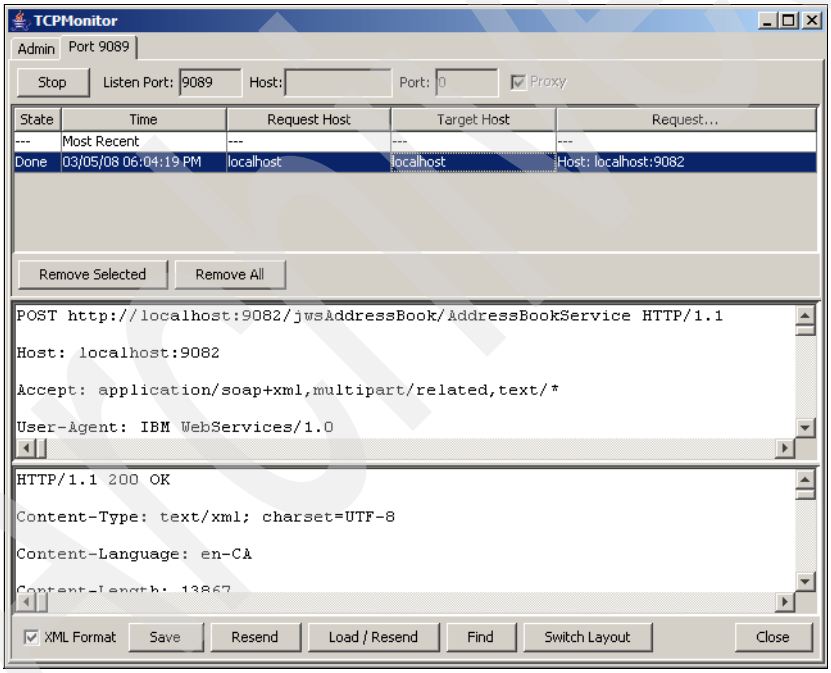


Figure 9-27 SOAP traffic in TCPMon

[2] Set to the true address of the SOAP service — the HTTP policy maps the address to the TCPMON address.

The SOAP request is shown in the upper pane, and the SOAP response is shown in the bottom pane. The request pane has two SOAP requests, and the response pane also has two SOAP responses. This reflects the nature of secure conversation. As discussed in “Secure Conversation scenario” on page 328, the secure conversation has two stages:

1. In the first stage, the Web service client sends a Request Security Token (RST) for a Security Context Token to an application endpoint.

   The RST is encrypted and signed using WS-Security information defined in the bootstrap security policy. 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 requestor as if it was generated by the endpoint service. Because the Security Context Token is encrypted, you might not be able to find anything very interesting in the first request/response pair in the TCPMon.

2. In the second stage, after the Security Context Token is established, the application messages are secured using a Derived Key based on the Security Context Token.

Take a closer look at the second request/response pair. Example 9-10 shows the interesting part of the second SOAP request:

Example 9-10 SOAP request snippet

```
<wsc:SecurityContextToken wsu:Id="sct_21" xmlns:wsc="uri 1[3]" xmlns:wsu="uri 2[4]">
   <wsc:Identifier>
      uuid:137277A5CC55391E861204761863247
   </wsc:Identifier>
</wsc:SecurityContextToken>
<wsc:DerivedKeyToken wsu:Id="dkt_25" xmlns:wsc="uri 1" xmlns:wsu="uri 2">
   <wsse:SecurityTokenReference>
      <wsse:Reference URI="#sct_21"
         ValueType="uri 3[5]">
      </wsse:Reference>
   </wsse:SecurityTokenReference>
   <wsc:Length>16</wsc:Length>
   <wsc:Nonce>... ...</wsc:Nonce>
</wsc:DerivedKeyToken>
<wsc:DerivedKeyToken wsu:Id="dkt_22" xmlns:wsc="uri 1" xmlns:wsu="uri 2">
   <wsse:SecurityTokenReference>
      <wsse:Reference URI="#sct_21" ValueType="uri 3">
      </wsse:Reference>
   </wsse:SecurityTokenReference>
   <wsc:Length>20</wsc:Length>
   <wsc:Nonce>... ...</wsc:Nonce>
</wsc:DerivedKeyToken>
```

The Security Context Token contains a Universally Unique Identifier (UUID), which is used as identification of a shared secret. DerivedKeyToken refers to the Security Context Token. Two Derived Key tokens are used to secure the SOAP message; one for signing and one for encrypting.

- 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.

Looking at the TCPMon, we see that the subsequent message invocations use the Security Context Token to encrypt and sign the messages.

[3] uri 1 = http://schemas.xmlsoap.org/ws/2005/02/sc
[4] uri 2 = http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd
[5] uri 3 = http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd

9.4.2 Apply secure conversation and reliable messaging to Web services

The second example is to apply Secure Conversation and Reliable Messaging to Web services. The Feature pack for Web services ships with the Reliable Asynchronous Message Profile (RAMP) Policy set that can be used out of the box. 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 using the WS-SecureConversationand WS-Security specifications.
- Confidentiality by encryption that includes encrypting the body, signature and signature confirmation elements, using the WS-SecureConversation and WS-Security specifications.

In order to see the SOAP traffic, we have to create a custom policy set to include the HTTPTransport policy. We use TCPMon shipped with WebSphere Application Server to monitor the SOAP traffic. The steps to apply Secure Conversation and Reliable Messaging are very similar to those in “Apply secure conversation to Web services” on page 336. So we just outline the steps here:

- ☐ Remove the Policy set that you applied to JAX-WS address book Web service sample, because you intend to apply a new Policy set.
- ☐ Create a custom Policy Set named ITSO RAMP to include the RAMP policy and HTTPTransport policy.
- ☐ Import the ITSO RAMP Policy Set into the Rational Application Developer development environment.
- ☐ Apply the ITSO RAMP Policy Set to the Web service and client.
- ☐ Configure the HTTPTransport policy type and WSSecurity policy set.
- ☐ Monitor the SOAP traffic using TCPMon.

After these steps are completed, study the SOAP traffic shown in TCPMon.

1. The first SOAP request/response pair shows that the Web service client sends a Request Security Token (RST), and the trust service processes the RST and responds with a Request Security Token Response (RSTR). This process is signed and encrypted by WS-Security.

2. In the second SOAP request, after the Reliable Messaging runtime acquires the UUID of the Security Context Token, it scopes the CreateSequence message to the Security Context Token by using the Security Token Reference (STR) argument.
   `<wsrm:UsesSequenceSTR soap:mustUnderstand='1'/>` forces the Reliable Messaging destination to ensure that the sequence is secured using the supplied STR. The SOAP request is shown in Example 9-11.

Example 9-11 Second SOAP request snippet

```
<wsc:SecurityContextToken wsu:Id="sct_23" xmlns:wsc="uri 1[6]" xmlns:wsu="uri 2[7]">
   <wsc:Identifier>
      uuid:2B493AC174AA79CF741204777157407
   </wsc:Identifier>
</wsc:SecurityContextToken>
<wsc:DerivedKeyToken wsu:Id="dkt_27" xmlns:wsc="uri 1" xmlns:wsu="uri 2">
   <wsse:SecurityTokenReference>
      <wsse:Reference URI="#sct_23" ValueType="uri 3[8]" />
   </wsse:SecurityTokenReference>
   <wsc:Length>16</wsc:Length>
   <wsc:Nonce>......</wsc:Nonce>
</wsc:DerivedKeyToken>
<wsc:DerivedKeyToken wsu:Id="dkt_24" xmlns:wsc="uri 1" xmlns:wsu="uri 2">
   <wsse:SecurityTokenReference>
      <wsse:Reference URI="#sct_23" ValueType="uri 3" />
   </wsse:SecurityTokenReference>
   <wsc:Length>20</wsc:Length>
   <wsc:Nonce>..../wsc:Nonce></wsc:Nonce>
</wsc:DerivedKeyToken>
<wsrm:UsesSequenceSTR soapenv:mustUnderstand="1"
   wsu:Id="wssecurity_signature_id_16" xmlns:wsrm="uri 4[9]" xmlns:wsu="uri 2" />
</wsrm:UsesSequenceSTR>
....
<wsa:Action wsu:Id="wssecurity_signature_id_20" xmlns:wsu="uri 2">
   http://docs.oasis-open.org/ws-rx/wsrm/200702/CreateSequence
</wsa:Action>
```

[6] uri 1 = http://schemas.xmlsoap.org/ws/2005/02/sc
[7] uri 2 = http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd
[8] uri 3 = http://schemas.xmlsoap.org/ws/2005/02/sc/sct
[9] uri 4 = http://docs.oasis-open.org/ws-rx/wsrm/200702

3. The Reliable Messaging destination responds with the CreateSequenceResponse message. Because the SOAP body is encrypted and signed, we do not see the returned sequence identifier. The SOAP message is seen in Example 9-12.

Example 9-12 Second response snippet

```
<wsa:Action wsu:Id="wssecurity_signature_id_32" xmlns:wsu="uri 1[10]">
   http://docs.oasis-open.org/ws-rx/wsrm/200702/CreateSequenceResponse
</wsa:Action>
<soapenv:Body wsu:Id="wssecurity_signature_id_29" xmlns:wsu="uri 1">
   <EncryptedData Id="wssecurity_encryption_id_37" Type="uri 2[11]" xmlns="uri 3[12]">
      ......
      <CipherData>
         <CipherValue>Sequence identifier is encrypted here</CipherValue>
      </CipherData>
   </EncryptedData>
</soapenv:Body>
```

4. The client sends the message to the service secured by Security Context Token, as seen in Example 9-13:

Example 9-13 Third request snippet

```
<wsrm:Sequence soapenv:mustUnderstand="1"
   wsu:Id="wssecurity_signature_id_40" xmlns:wsrm="uri 1[13]" xmlns:wsu="uri 2[14]">
   <wsrm:Identifier>urn:uuid:2B493AC174AA79CF741204777158962</wsrm:Identifier>
   <wsrm:MessageNumber>1</wsrm:MessageNumber>
</wsrm:Sequence>
```

The goal of RAMP policy is to ensure the integrity of the Reliable Messaging elements. This is accomplished by digital signing all the WS-ReliableMessaging header elements such as `<wsrm:Sequence>` and `<wsrm:SequenceAcknowledgement>`. It is hard to tell from looking at the messages sent over the wire that this has happened because the `<wsse:Signature>` element has been encrypted. But one artifact of the signing operation that you can see is the addition of the wsu:Id attribute added to the WS-RM header elements of the form wsu:Id="wssecurity_signature_id_N". In the above example, it is `wsu:Id="wssecurity_signature_id_40"`. Both the `<wsrm:Sequence>` elements and the 2 child elements, `<wsrm:Identifier>` and `<wsrm:MessageNumber>`, are signed.

[10] uri 1 = http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd
[11] uri 2 = http://www.w3.org/2001/04/xmlenc#Content
[12] uri 3 = http://www.w3.org/2001/04/xmlenc#
[13] uri 1 = http://docs.oasis-open.org/ws-rx/wsrm/200702
[14] uri 2 = http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd

5. The Reliable Messaging runtime sends a SequenceAcknowledgement message to acknowledge that the message is properly delivered and secured by the Security Context Token, as seen in Example 9-14.

Example 9-14 Third response snippet

```
<wsrm:SequenceAcknowledgement soapenv:mustUnderstand="1"
   wsu:Id="wssecurity_signature_id_100" xmlns:wsrm="uri 1[15]" xmlns:wsu="uri 2[16]">
   <wsrm:Identifier>urn:uuid:2B493AC174AA79CF741204777158962</wsrm:Identifier>
   <wsrm:AcknowledgementRange Lower="1" Upper="1"></wsrm:AcknowledgementRange>
</wsrm:SequenceAcknowledgement>
<wsrm:Sequence soapenv:mustUnderstand="1"
   wsu:Id="wssecurity_signature_id_101" xmlns:wsrm="uri 1" xmlns:wsu="uri 2">
   <wsrm:Identifier>urn:uuid:2B493AC174AA79CF741204777163059</wsrm:Identifier>
   <wsrm:MessageNumber>1</wsrm:MessageNumber>
</wsrm:Sequence>
```

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.

9.5 More information

The WS-Trust specification is available at:

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

The WS-SecureConversation specification is available at:

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

The WebSphere Application Server Application Version 6.1 Feature pack for Web services information center has a great amount of detail on WS-SecureConversation:

http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/topic/com.ibm.websphere.wsfep.multiplatform.doc/info/ae/ae/cwbs_wssecureconvstd.html

[15] uri 1 = http://docs.oasis-open.org/ws-rx/wsrm/200702
[16] uri 2 = http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd

10

Reliable messaging

In this chapter we introduce the details of the functionality provided by WS-ReliableMessaging in the Feature pack for Web services, describing the various qualities of service available and how to choose them. We also link the technology to the patterns outlined in Chapter 4, “Facets and patterns” on page 55, and walk you through some simple examples showing you how to implement those patterns.

10.1 Overview of WS-ReliableMessaging

At a fundamental level, WS-ReliableMessaging provides an interoperable SOAP based protocol for ensuring that SOAP messages are delivered reliably between endpoints. It is dependent on URLs for defining the endpoints, SOAP for communicating between the endpoints, and it creates and uses message sequence numbers to detect and repair missing, duplicate, and out-of-order messages. The WS-ReliableMessaging protocol is independent of the underlying transport.

The standard itself allows for four different levels of delivery assurance:

- AtLeastOnce
- AtMostOnce
- ExactlyOnce
- InOrder

The Feature pack for Web services provides support only for ExactlyOnce, and InOrder.

The ExactlyOnce delivery assurance means that a Web service endpoint receives a message sent by a client once, and only once. Senders retransmit unacknowledged messages to ensure that the message is received by the receiving WS-ReliableMessaging runtime at least once. Duplicates are filtered by the WS-ReliableMessaging implementation.

The InOrder assurance ensures that a Web service endpoint is invoked by messages in the same order that a client sends them. In this mode, the WS-ReliableMessaging implementation stores messages that are received out of order until the messages which preceded them are received and used to invoke the endpoint. This assurance type is optional in the Feature pack for Web services, and is configured as part of the WS-ReliableMessaging policy type.

Scope of WS-ReliableMessaging

The WS-ReliableMessaging specifications are focused entirely on defining a SOAP based protocol for reliable message exchange and do not include any definition of qualities of service provided by endpoints such as transactionality or persistence. Nor does the WS-ReliableMessaging specification deal with defining policies to be shared between a client and server that are needed to establish agreed qualities of service. This latter aspect is being addressed, firstly by RAMP, and latterly by the RSP being developed by Web Services Interoperability organization.

These might seem like surprising omissions, but only if you are expecting the goal of WS-ReliableMessaging to provide a “standards based WebSphere MQ”. We discuss WS-ReliableMessaging and MOM in 10.6.1, “Messaging Oriented Middleware” on page 382. The point to note here is: WS-ReliableMessaging is not a product for building reliable distributed applications, such as WebSphere MQ, or WebSphere Application Server. WS-ReliableMessaging is simply a SOAP protocol that is designed to be composable with other Web service specifications and packaged as part of a platform for building interoperable reliable distributed applications by different vendors. It is up to vendors, such IBM with the Feature pack for Web services, to build a product that realizes the potential of WS-ReliableMessaging. In focusing on the protocol, rather than on the productization, WS-ReliableMessaging is following in the successful pattern that has been established by the TCP/IP protocol itself.

Reliability on the wire might add sufficient value to some scenarios to encourage the use of WS-ReliableMessaging by itself, but it seems unlikely. In Chapter 3, “Business scenarios” on page 35 and Chapter 4, “Facets and patterns” on page 55 we explored some business scenarios, and those scenarios that used WS-ReliableMessaging drew on additional functionality provided by the Feature pack for Web services to build a solution, such as support for RAMP, and support for persistent and non-persistent qualities of service, and in some cases transactional send and receive.

In this chapter our goal is not only to explain WS-ReliableMessaging, but also how the Feature pack for Web services uses it in conjunction with other Web service specifications and WebSphere Application Server capabilities.

10.1.1 Applying WS-ReliableMessaging

A significant benefit of the WS-ReliableMessaging implementation in the Feature pack for Web services is making existing JAX-WS clients and servers reliable without modifying any code. The WS-ReliableMessaging implementation is designed so that it is easily applied to clients and services using the Policy sets mechanism.

That said, there are a number of WS-ReliableMessaging limitations and interoperability best practices noted in this book that might impact of the code or design of clients or services. Thus, if reliability is a probable future requirement for you, consider these best practices prior to implementing your application to forestall late design changes.

Additionally, there is a WS-ReliableMessaging SPI which you can use when the WS-ReliableMessaging policy type is used. The WS-ReliableMessaging SPI gives programmatic control over some of the WS-ReliableMessaging implementation.

10.1.2 Versions

The Feature pack for Web services supports both WS-ReliableMessaging 1.0 and WS-ReliableMessaging 1.1.

For interoperability with WCF, use WS-ReliableMessaging 1.0. For other scenarios, consider the capabilities of the WS-ReliableMessaging implementations and the technical details of the versions outlined in this chapter.

10.1.3 How WS-ReliableMessaging works

The WS-ReliableMessaging standards define a number of SOAP messages and SOAP headers which are used to implement the reliability protocol. The protocol is based around the concept of a “Sequence” which is applied to application messages. A single sequence, taken from the WS-ReliableMessaging specification, is illustrated in Figure 10-1.

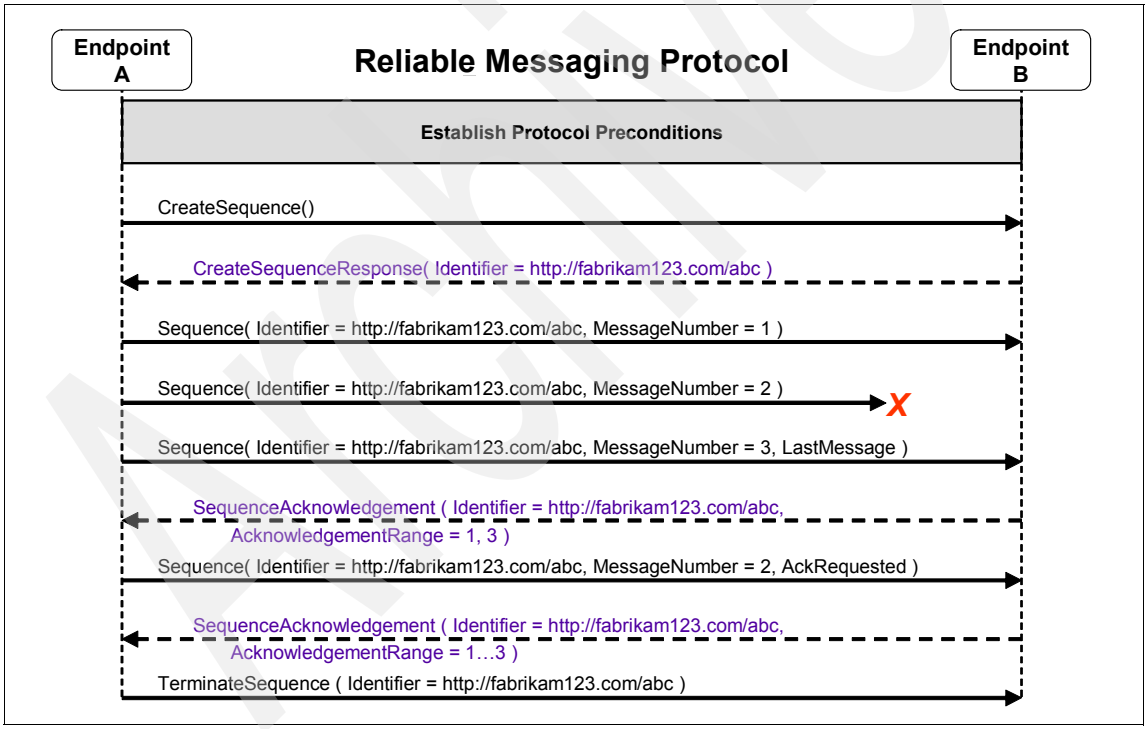


Figure 10-1 Example message exchange (from WS-RM)

The important thing to note at the abstract level is that the protocol consists of "out-of-band" messages such as "CreateSequence" and "TerminateSequence," which are SOAP messages not defined as part of the application WSDL, and SOAP Headers, for example, "Sequence" and "SequenceAcknowledgement," which are added to the application messages to ensure reliable delivery.

When a two-way operation is invoked reliably, there are two sequences in use. One is used to transmit messages reliably from client to server, and the other to transmit messages reliably from server to client.

The exact flows will depend on network reliability, invocation style, WS-ReliableMessaging version, and some implementation details. More details can be found in "Message flows" on page 363.

10.1.4 Sequence lifecycle

In the Feature pack for Web services implementation of WS-ReliableMessaging, a sequence is created and managed by the feature pack, on behalf of the application. The Feature pack for Web services creates a new sequence between two endpoints if one does not exist, or uses an existing sequence if one is in progress.

The behavior of applications should not depend on the behavior of WS-ReliableMessaging sequences. And of course this is true, by definition, of existing applications which start to use WS-ReliableMessaging by having a new Policy set attached to them.

A sequence remains active until it times-out or is explicitly terminated by either the sender or receiver. During this time the sequence consumes resources on both client and server.

Between use and being discarded there is an intermediate "closed" state for a sequence in which the sequence still exists but cannot be used for any new messages. This state can be used by WS-ReliableMessaging implementations, and is of little interest to application programmers. See Figure 10-2.

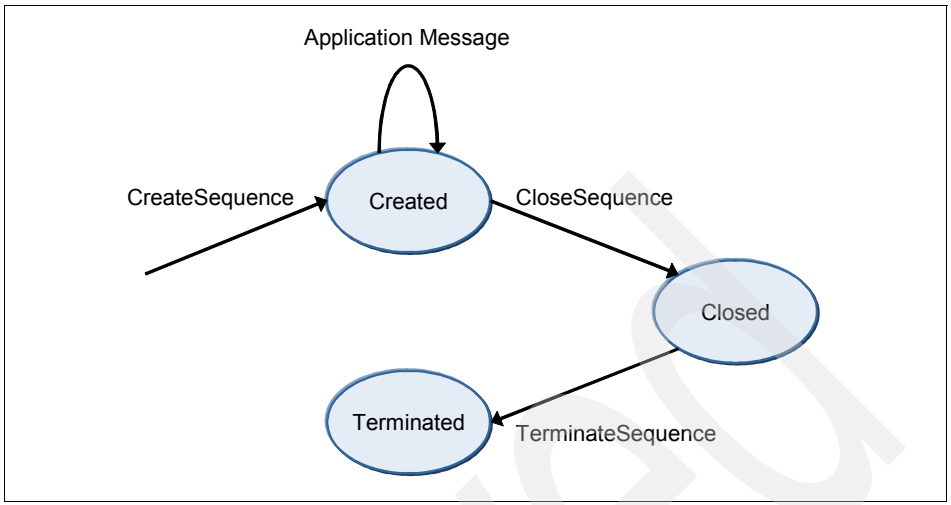


Figure 10-2 Sequence lifecycle state diagram

Sequence timeout

The Feature pack for Web services is designed with the long running sequences of a B2B scenario in mind. B2B scenarios commonly use a long running sequence to funnel many messages reliably between two hosts. For this reason, sequences created by the Feature pack for Web services have a 24 hour inactivity time-out. This means that they will only be discarded after 24 hours without any activity associated with the sequence, unless the sequence is terminated explicitly by the client sending a TerminateSequence message or an administrator taking action.

In comparison, WCF is designed around short running sequences and has a default inactivity time-out of 10 minutes. It is important that sequences are not simply left to time-out for at least two reasons:

- Resources are being wasted somewhere in the system.
- It is liable to leave a participant in a confused state.

  For example, when WCF terminates a sequence after a 10 minute time-out, the Feature pack for Web services does not know, and attempts to use that sequence for further messages. This results in a fault back from WCF, and leaves the Feature pack for Web services client requiring manual or programmatic intervention.

You should consider the frequency and length of interactions within your application and plan sequence management accordingly. If your pattern matches the Feature pack for Web services design, then increasing the inactivity time-out in WCF to match is a good approach. If, you have no control over the WCF

configuration, you should either ensure that the sequence does not hit its inactivity time-out because of frequent application messages, or you should use the WS-ReliableMessaging SPI to create and close sequences within the inactivity time-out.

If your pattern is more closely aligned with short, infrequent reliable interactions then you should be careful to manage the sequence lifecycle from Feature pack for Web services clients using the WS-ReliableMessaging SPI outlined in “Using the SPI” on page 385. You should ensure that sequences are terminated before clients exit so that resources are not consumed on the server until the 24 hour inactivity time-out occurs.

10.1.5 InOrder delivery

The WS-ReliableMessaging message flows can deliver messages to the destination WS-ReliableMessaging implementation in a different order than the way that messages were sent by the client code. This is particularly likely to happen if messages are lost and have to be resent. The presence of a message number in the Sequence header allows the destination to determine the correct order and, if configured, ensure that the JAX-WS endpoint code is invoked by messages in the order they were sent. In the Feature pack for Web services, this is configured as part of the Policy set applied to the receiving service.

There can be a number of motivating factors for enabling InOrder delivery, but you should be aware of the following implications.

Security: Denial of service attack

A client starts a sequence and sends a series of large messages omitting the first message. This requires the server to store these messages, either in memory or on-disk, consuming resources until the sequence either times out or is removed administratively.

Performance: Sequential processing

Inevitably, because the messages have to be invoked sequentially, performance degrades in line with the time spent invoking the endpoint.

Stability: Out of order effects are non-linear

In the fair weather case, InOrder WS-ReliableMessaging performs similarly to not InOrder WS-ReliableMessaging because messages tend to be delivered in the correct order. However, when a message is lost or messages are received out of order, the impact of the performance and throughput of your system is much greater. This is because more resources are consumed storing the out of order messages than would normally be the case.

Also, the retransmission time in the client is set to minimize retransmissions, however this means that with a high throughput system, either in terms of number of messages or size of messages, the size of the out-of order window can be large. It is also worth considering the impact that a single sequence that is dealing with failure and writing many out of order messages to the store might have on other sequences that share the same store (that is, does it slow down receipt of messages enough to push other sequences into an out-of-order state).

These situations are manageable with the correct clustering and capacity planning but it depends heavily on the pattern of use: number of sequences, size of messages, messages per second, and so on. For this reason you are encouraged to test failure cases with your specific use case in mind to determine your capacity requirements.

10.1.6 WS-ReliableMessaging and the underlying transport

The WS-ReliableMessaging protocol is independent of the underlying transport. Currently the Feature pack for Web services only implements WS-ReliableMessaging over SOAP/HTTP using TCP/IP as the underlying transport.

WS-ReliableMessaging over SOAP/HTTP

TCP/IP is itself known as a reliable protocol, in distinction to IP, which is thought of as unreliable. Indeed there are great similarities between the protocol TCP/IP uses to maintain reliability over IP, and the protocol WS-ReliableMessaging uses to maintain reliability over SOAP. Both persist information in the client and server and use sequence numbering to detect missing, duplicate, and out of order data packets and then correct any transmission errors by retransmitting missing packets (Figure 10-3).

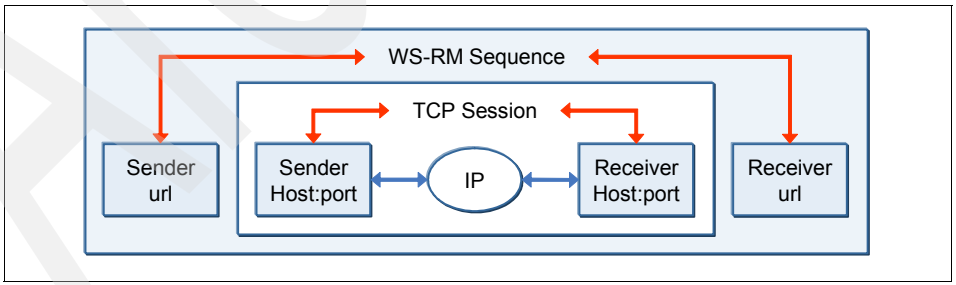


Figure 10-3 Comparison between WS-ReliableMessaging over SOAP/Http and TCP/IP

When WS-ReliableMessaging is running over TCP/IP, the WS-ReliableMessaging sequence spans as many TCP/IP sessions as necessary. So if a TCP/IP session is closed and the WS-ReliableMessaging sequence is incomplete, when a connection is next established between the Sender and Receiver, WS-ReliableMessaging resynchronizes on the existing sequence using the new TCP/IP session. The resynchronization succeeds as long as both the Sender and Receiver know the state of the current sequence — and that depends on how the session was lost (was it just the TCP/IP session that was lost, or did a server go down?), and whether the quality of service for the connection includes persisting the message and sequence information.

10.2 Message flows

The WS-ReliableMessaging protocol introduces additional out-of-band SOAP messages over and above the application messages defined in the WSDL. This section introduces you to some of these messages and the SOAP Headers added to application messages. The exact messages sent vary depending on message frequency and performance considerations. This section outlines the different patterns. For full details of all of the possible messages, look at the WS-ReliableMessaging specifications found at:

http://specs.xmlsoap.org/ws/2005/02/rm/ws-reliablemessaging.pdf.
http://docs.oasis-open.org/ws-rx/wsrm/200702/wsrm-1.1-spec-os-01.pdf

A very important consideration when architecting your solution is to decide whether to use synchronous, or asynchronous message flows (see 7.5.3, “Asynchrony on the wire” on page 202). First we look at asynchronous flows, because they map to the needs of WS-ReliableMessaging more simply, and then at synchronous flows that have changed between WS-ReliableMessaging 1.0 and 1.1. Finally, in 10.2.5, “Decision: synchronous versus asynchronous for two-way” we discuss how to decide which model to use.

10.2.1 One-way Asynchronous Message Flow with lost message

Figure 10-4 shows the message flows when one-way asynchronous WS-ReliableMessaging is used and one of the application messages is lost.

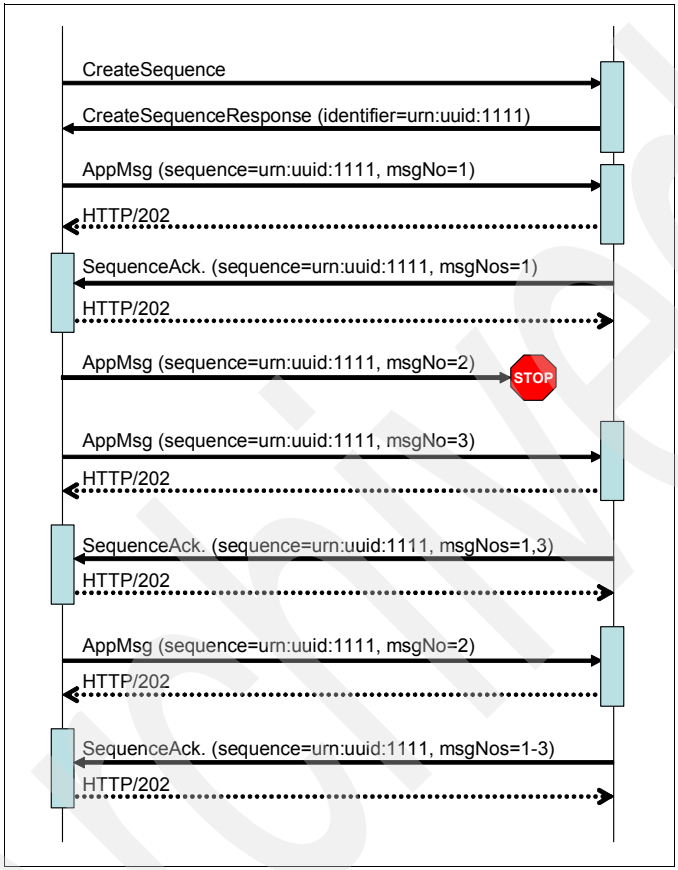


Figure 10-4 One-way asynchronous WS-ReliableMessaging flow

The initial message in the flow is a CreateSequence message to the destination, to which the destination responds by sending a CreateSequenceResponse message to the sender, including a sequence identifier. The CreateSequence message includes an AcksTo header, which defines where acknowledgements should be sent.

The first application message is sent, with the sequence identifier and a message number included in the SOAP Header. The destination processes that header and sends a SequenceAcknowledgement message to the AcksTo containing the message number 1, allowing the sender to discard message 1.

The second and third application messages are then sent, with message 2 being lost. The destination processes message 3 and sends a SequenceAcknowledgement message to the sender acknowledging messages 1 and 3. After a time-out interval in which the sender has not received an acknowledgement for message 2, the sender retransmits message 2. This time the message is received by the destination, and a SequenceAcknowledgement is sent to the sender acknowledging messages 1, 2, and 3.

Once the client has received acknowledgements for all the application messages it terminates the sequence by sending a TerminateSequence message to the destination. The destination responds with a TerminateSequenceResponse message which simply confirms to the sender the state of the sequence as it was terminated.

10.2.2 Two-way Asynchronous Message Flow

A two-way asynchronous message flow consists of two sequences, one from the client to the server, and one from the server to the client, each carrying requests or responses respectively.

Sequences are created, used, and terminated in the same way. The sequence used to carry the response message is created by sending a CreateSequence message to the endpoint identified by the ReplyTo in the application request message. This occurs when the first response message targeted at that destination is ready to be sent.

It is possible for a WS-ReliableMessaging implementation to “piggyback” acknowledgements on messages by adding a SequenceAcknowledgement header to the application messages flowing in either direction. The implementation in the Feature pack for Web services regularly does this, piggybacking acknowledgements of request messages on response messages and acknowledgements of response messages on subsequence request messages. For this reason, depending on the frequency of messages in a sequence, you rarely or never see an out-of-band SequenceAcknowledgement message.

10.2.3 WS-ReliableMessaging 1.0 Sync two-way message flow

A synchronous two-way message flow is substantially different because the server cannot open a connection to send a message to the client. The WS-ReliableMessaging 1.0 specification does not directly address this problem, and to fill that gap, the loosely defined ‘replay model’ is used. In this model, the client retransmits application request messages until it has received the response message, even if the application request message has been acknowledged.

Additionally, “piggybacking” is used extensively to transmit acknowledgements, and the “offer” mechanism is used to create the response sequence. The offer mechanism allows the client to offer a sequence identifier of a response sequence to the server in the CreateSequence for the request sequence. This allows both sequences to be created with a single request response flow, rather than requiring a message to be sent from the server to the client which is not possible in this scenario. The flow is as follows (Figure 10-5).

Figure 10-5 WSRM 1.0 Sync two-way with failure

The client sends a CreateSequence message to the server, which includes an “offer” section identifying a sequence that the server can use to send response messages. The server responds with a CreateSequenceResponse message which includes the sequence identifier the client should use for application response messages and an acceptance of the offered response sequence. The client then sends the first application message with the outbound sequence identifier attached. The response message is sent back to the client on the HTTP back channel. This message has the “offered” sequence identifier in its SOAP Header in addition to the acknowledgement for the request message. This acknowledgement allows the client to discard application message 1 from its store.

A second application message is sent as part of the outbound sequence, including an acknowledgement for the response to application message 1. The message is processed but the HTTP connection fails before the response is received by the client. A third application message is sent, again associated with the outbound sequence and acknowledging the first application response message. The application response includes an acknowledgement of application requests 1 through 3.

Though application request message 2 has now been acknowledged, the client cannot discard it because it needs to resend it to retrieve the response message. The client discards application request message 3 from its store. The client then retransmits application request message 2. The server detects the duplicate transmission and retrieves the response message from its store and returns it on the HTTP back channel. The client receives this response and detects that application request message 2 is no longer required and is removed from the client store.

The sequences are now in a fully acknowledged state and the client terminates the sequence by sending a message to the server. This terminates both the outbound and offered sequences. The server responds with a terminate sequence response message confirming the terminated state.

10.2.4 WS-RM 1.1 and WS-MC 1.1 sync two-way message flow

Finally, there is the WS-ReliableMessaging 1.1 and WS-Make Connection pattern. WS-Make Connection is a response to the problem of Asynchronous messaging when one party is behind a firewall and therefore cannot receive wire-asynchronous responses. It defines a simple message and selection criteria which allows a client to poll a server for messages which the server wishes to send to the client. To aid this, the WS-Make Connection specification defines an address pattern to tell Web services infrastructure that a message is to be sent to a WS-Make Connection enabled client. For more detailed information, see the WS-Make Connection standard, found at:

http://docs.oasis-open.org/ws-rx/wsmc/200702/wsmc-1.0-spec-os-01.pdf

Despite the fact that it is now a separate standard, WS-Make Connection is part of the WS-ReliableMessaging implementation in the Feature pack for Web services and is not supported for direct customer use (Figure 10-5).

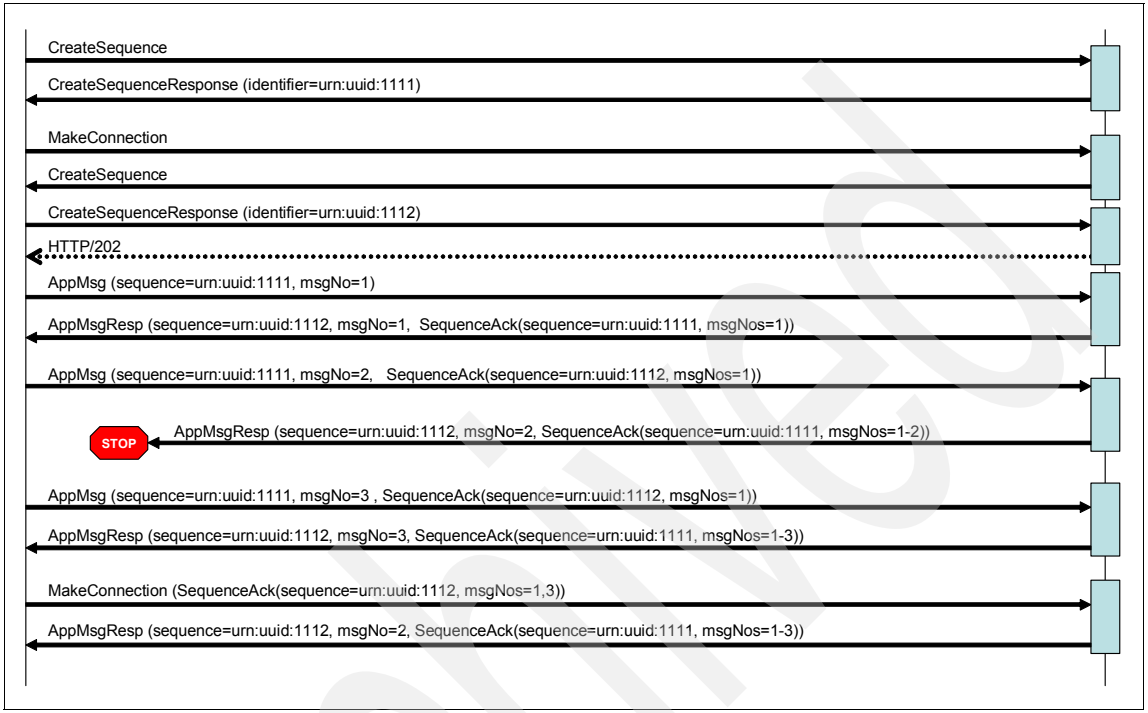


Figure 10-6 WSRM 1.1 w/WS-MC 1.1 and message loss

The flow is initiated by a CreateSequence for the request message from the client to the server, which responds with a CreateSequenceResponse. This is followed by the first application message, attached to the outbound sequence. The client then begins polling with WS-Make Connection. The server responds to the WS-Make Connection message with a CreateSequence for the response sequence. The client responds by sending the CreateSequenceResponse to the service. The application response is then sent, attached to the response sequence which has been created.

As with our WS-ReliableMessaging 1.0 example, message 2 is sent, received, and the response message lost. Message 3 is then sent, and the response includes an acknowledgement for message 2, which is then removed from the store. At a polling interval, a WS-Make Connection message is sent to the service, and the response to message 2 retransmitted on it, and received by the client. All three application messages have now been successfully acknowledged, so the client terminates the request sequence by sending a TerminateSequence message to the service, which response with a TerminateSequenceResponse.

Note: Responses to WS-MakeConnection messages can include a MessagePending header that tells the client implementation that there are more messages waiting for a MakeConnection message. This will trigger further WS-Make Connection messages from the client to the server without waiting for the polling interval.

10.2.5 Decision: synchronous versus asynchronous for two-way

As summarized in Figure 10-7, there are substantial differences in the message flows between synchronous and asynchronous invocations. Because the WS-ReliableMessaging protocol in inherently asynchronous, and because the “Replay model” is undefined and WS-Make Connection is new, the asynchronous pattern is more interoperable. As with unreliable Web services, asynchronous is also more appropriate for long running services. For WS-ReliableMessaging 1.0, it is also more appropriate for large request messages as the request message does not have to be retransmitted in the event of a failure, and it can be removed from the store as soon as it has been acknowledged.

Despite these advantages, the asynchronous model requires that asynchronous Web services are supported by the endpoint, which might be problematic depending on the network architecture

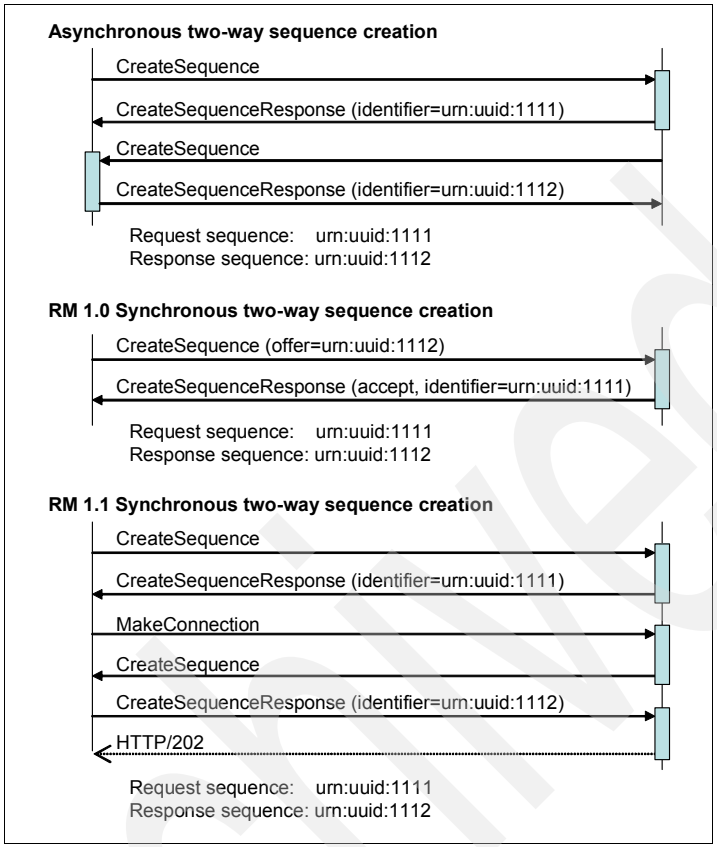


Figure 10-7 All three two-way sequence creation models

10.3 Persistence and transactionality

Persistence and transactionality and are different but related concepts. Persistence refers to hardening data so that it remains available, even after a failure. Transactionality refers to coordinating access to resources, such as data in a database and data in a message, so that the results obtained by an application are repeatable. An aspect of transactionality is durability (maintaining repeatability after system or application failure), so transactionality and persistence are related. In addition simple transactionality requires that the transaction has only two states, initial and result, and to the outside observer there are no intervening states — for example, if a message is sent as part of a transaction, the message is either sent, or it is not sent. It cannot be sent, and later withdrawn if an error is detected.

Persistence

When we talk about persistent WS-ReliableMessaging, we mean that a message is written to a persistent store by the WS-ReliableMessaging implementation before being sent. This allows the message to be resent if there is a communications failure and the client fails before resending. There are also cases when the WS-ReliableMessaging might write a message to a persistent store when it receives a message. This is done to allow messages to be received out of order and survive a server failure before the missing messages are received. When the missing messages are received, the persisted messages are retrieved and the endpoint invoked.

Transactionality

When we talk about transactionality with WS-ReliableMessaging, we mean that a message being sent is added to the store under a transaction and is not sent until the transaction is committed. It also applies when a message is received and delivered to an endpoint under a transaction. The removal of the message from the receiver's store is included in the transaction, allowing for re-delivery to the endpoint if the transaction rolls back.

Note: It is not possible to have a request/response interaction on a client within the same transaction; the request is not sent until the transaction commits and therefore there is no way for the service to receive the request and send a response.

On the server it is different. It is quite possible to receive the incoming request, create a response, and send it back to the client under the same transaction. All these actions are coordinated within a single transaction the scope of which can be extended to include other resources, such as a database, under extended transaction control.

Support for persistence and transactionality

The Feature pack for Web services exploits the freedom in the WS-ReliableMessaging specifications to provide three qualities of service for senders and receivers.

- Unmanaged Non-Persistent
  - The WS-ReliableMessaging state is held in memory within the client or server JVM.
- Managed Non-Persistent
  - The WS-ReliableMessaging state is held in a non-persistent manner in a messaging engine, allowing for failure of servers in a cluster sharing the messaging engine, but not of the messaging engine itself.

- Managed Persistent
  - The WS-ReliableMessaging state is held in a persistent messaging engine, allowing for failure and restart of the messaging engine.

Figure 10-8 shows how the qualities of service map to the transactionality of senders and receivers.

			Non Transactional	Transactional	
				Send	Receive
Managed	In order	Persistent	Yes[1]		
		Not persistent	n/a		
	Not in order	Persistent	Yes[1]		n/a
		Not persistent	Yes		
Unmanaged	In order	Persistent	n/a		
		Not persistent			
	Not in order	Persistent			n/a
		Not persistent	Yes[2]		

1 - One way recommended in preference to two way messaging

2 - Not recommended on z/OS. Use managed non-persistent instead

Figure 10-8 Qualities of service and transactionality

The managed levels of durability rely on a WebSphere messaging engine as the WS-ReliableMessaging store which is identified in the Policy set binding and applied to the client or server.

10.3.1 One-way transactional persistent messaging

One of the sweet spots for the WS-ReliableMessaging implementation in the Feature pack for Web services is one-way transactional persistent messaging. Exemplified in the Check clearing scenario and design pattern, in this case a message is sent and received transactionally, and the message is protected against both communications and endpoint failure.

To enable this scenario, apply a Policy set containing WS-ReliableMessaging configured for persistence and InOrder delivery to the service. The client must also have a Policy set containing persistent WS-ReliableMessaging and the client must set the JAX-WS Request Context property `enableTransactionalOneWay` to `TRUE`.

In the case of client failure, when the client is restarted, the unacknowledged message is not resent to the endpoint until another message is sent to the same endpoint — then the unacknowledged message is resent. In a scenario where there is regular communication between servers, this limitation is not a problem. Where the endpoints involved only ever exchange one message, when the client is restarted, manual intervention via the administrative console is required to resend the unacknowledged message.

In the case of server failure, when the server restarts, any undelivered messages found in the persistent store are delivered as if just received.

10.3.2 Two-way transactional persistent messaging

The JAX-WS 2.0 client request-response APIs do not support a transactional two-way model, and neither does the Feature pack for Web services. Figure 10-9 illustrates the components that are involved in a two-way transactional model.

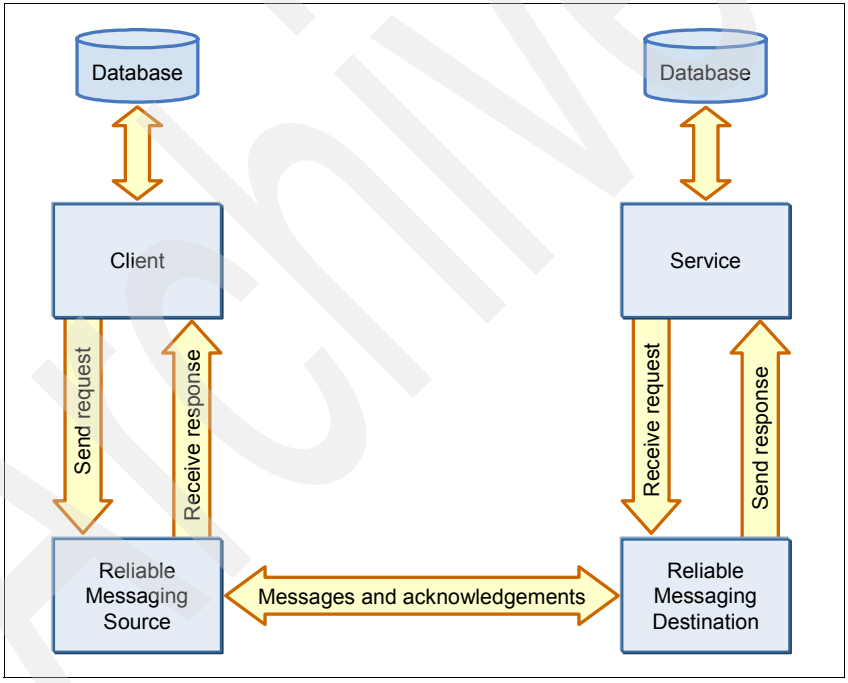


Figure 10-9 Components involved in a two-way transaction

To see why, consider what the scope of the transaction would be. The client programming model is a single request/response service invocation. The client, nor any other applications outside the transaction scope, should see any intervening state between the client issuing the request and receiving the response back from the server. In “Single transaction model with WS-AtomicTransaction” on page 377, we show how to implement a tightly coupled transaction model that preserves this invariant.

But to implement this transactional model and at the same time preserve the benefits of a loosely coupled, messaging model is not possible. To be loosely coupled means that the intervening states between creating the request and receiving the response are visible outside the transaction scope, which contradicts the assumption the client’s request/reply operation is atomic. Another way this is sometimes illustrated is by saying that the client can only deliver its request message when it commits its response back. But the service cannot return a response until it has received the request.

You should be careful that you do not use a managed WS-ReliableMessaging two-way client from within a transaction because you do not receive a warning and your application simply hangs.

A further problem with the model is that the client would have to be restartable. Unlike a traditional flat transaction, the client, rather than closing the transaction scope with a commit and passing responsibility for completing the transaction to the transaction system, closes the transaction expecting a response back. The client, as well as the transaction system, must be restartable to complete the transaction should a failure occur.

The same problem of the restartable client exists for the two-way persistent, but non-transactional model. If you do apply a persistent Policy set to a two-way client, the request will be retransmitted as with the one-way client, however the client application never receives the response if the client application is restarted.

10.3.3 Alternative transactional models

While a two-way transactional request response WS-ReliableMessaging client is not supported, there are alternatives that you can consider, depending on the situation.

Three transaction model with WS-ReliableMessaging

One option is to use a three transaction model. This consists of two transactions on the client: one for sending and one for receiving, and one transaction on the server to process the inbound message and send the response (Figure 10-10).

The model is based on a wire-asynchronous interaction, and is most appropriate when the service is loosely coupled or long-running in nature; for example, submitting a form for human approval.

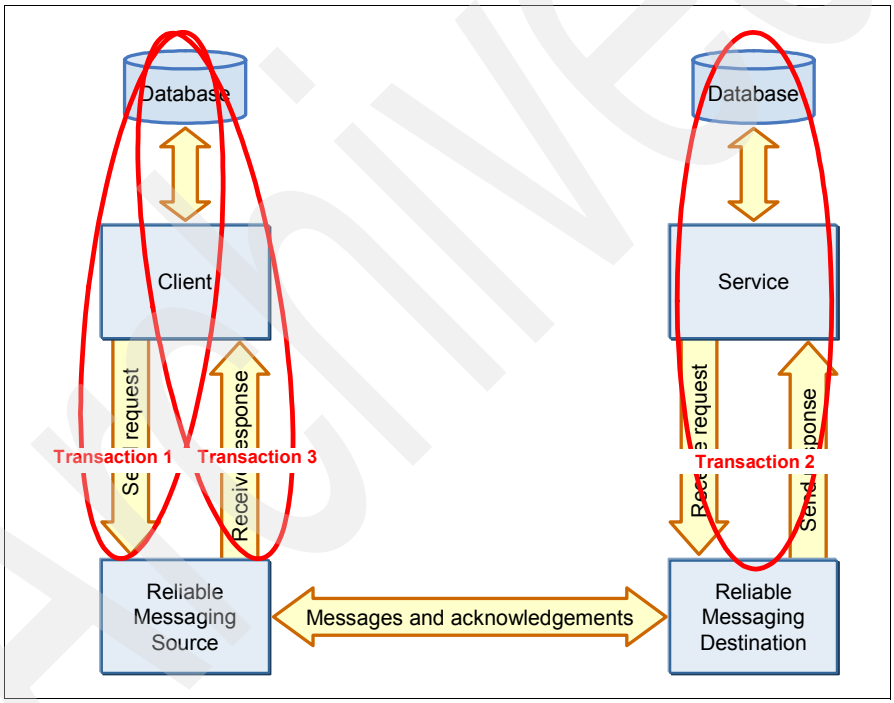


Figure 10-10 Three legged transaction model

The model is based on a wire-asynchronous interaction, and is most appropriate when the service is loosely coupled or long-running in nature; for example, submitting a form for human approval.

The model can be implemented in two different ways. The first, illustrated in Figure 10-11, models the request-response pattern as two one-way flows, with two clients and two services.

Three transaction model: Variation one

The client, or requesting server, hosts a one-way client and a one-way service. The client sends the request message reliably and transactionally to the service on the service hosting server. This service receives the message under a transaction and, using target information sent in the message, uses a one-way client to reliably and transactionally send the response message to the service hosted on the requesting server. The response message is then received transactionally by the response service. In this scenario, all correlation must be done by the application.

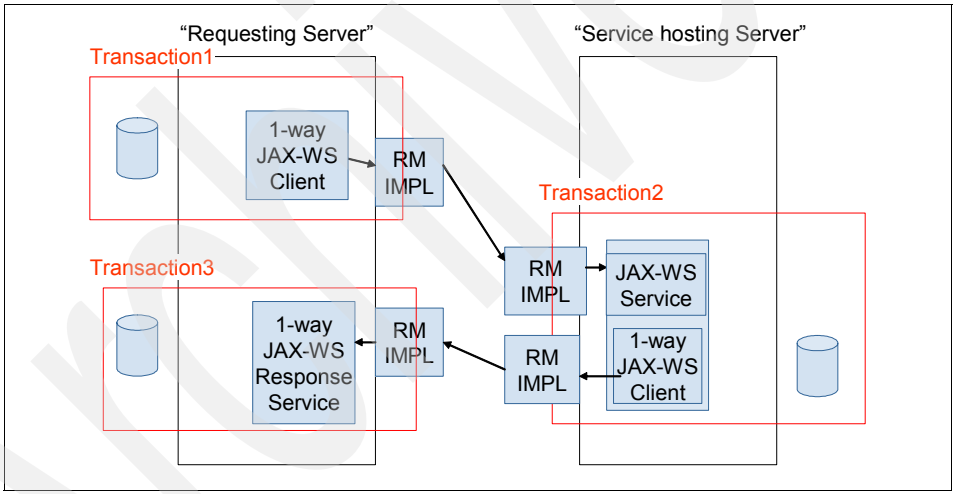


Figure 10-11 Three transaction model with WS-RM and a one-way service

Three transaction model: Variation two

The second pattern to implement a three transaction model is shown in Figure 10-12, and involves advanced use of WS-Addressing. This model again involves a one-way client and service on the requesting server, but this time uses a request-response service in the service hosting server.

The client sends a one-way message to the service, with a WS-Addressing ReplyTo header pointing to the response service. This can be done using the WebSphere Application Server WS-Addressing SPI. The service processing occurs under a transaction, and the Web services engine automatically sends the response message to the ReplyTo. The response service receives the response message under a transaction, and completes the processing. This scenario can exploit the WS-Addressing correlation functionality to simplify the correlation task.

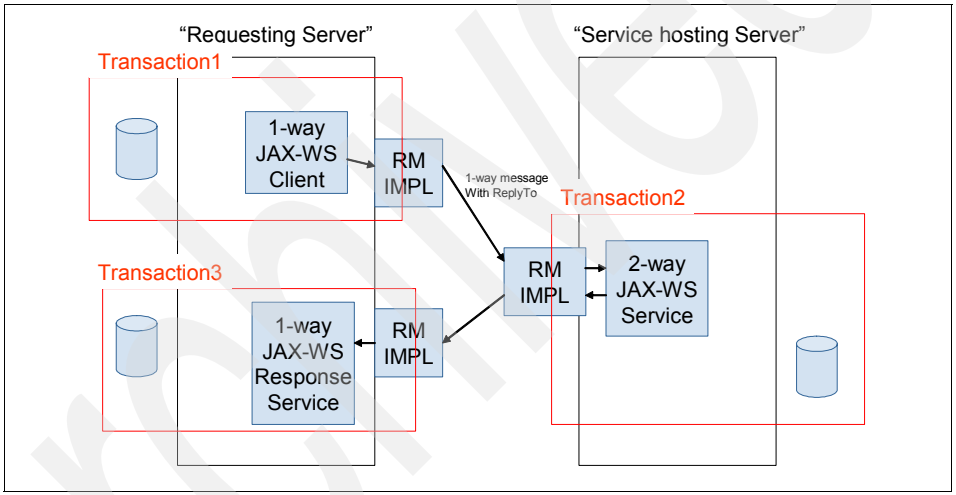


Figure 10-12 Three transaction model with WS-RM and a two-way service

Single transaction model with WS-AtomicTransaction

If the requesting client and service are willing to be tightly coupled, and the service invocaton is short, then using the WS-AtomicTransaction support can be an appropriate approach. With WS-AtomicTransaction (supported in the Feature pack for Web services, but not new), the transaction from the requesting server is propagated to the service hosting server, and the service logic run under logically the same transaction. Then, once the call has returned and the client transaction commits, so does the work done on the server.

This tightly couples the client and server, allowing the client to control when locks held by the server are committed or rolled back. Therefore, the decision to use this model is both technical and organizational.

This model does require that the client has some “retry” logic of its own, to cover the case where the transaction is rolled back. However, this logic should simply be the same as the original sending logic i.e. triggering a message based on the contents of a transactional store or queue (Figure 10-13).

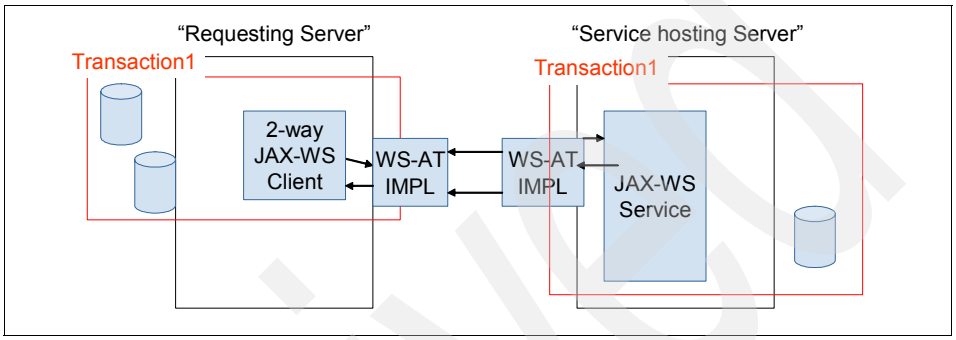


Figure 10-13 Single transaction model with WS-AT

10.3.4 z/OS specific requirements

WebSphere Application Server for z/OS has an architecture that uses a number of potentially transient servant instances to satisfy application requests. This model does not fit with the WS-ReliableMessaging unmanaged nonpersistent mechanism, as when a servant is terminated, all the sequence state would be lost. Additionally, the state would not be shared between servant regions leading to unrecognized sequence faults because of a lack of servant affinity. For these reasons, for WS-ReliableMessaging, you should treat a WebSphere Application Server for z/OS instance as an equivalent of a distributed cluster, and use one of the managed qualities of service and the WebSphere Application Server Proxy.

10.3.5 WS-AtomicTransaction with WS-ReliableMessaging

As mentioned in “Two-way transactional persistent messaging”, the Feature pack for Web services supports WS-AtomicTransaction, which allows for distributed transactions that are propagated over Web services. At both conceptual and implementation levels, WS-AtomicTransaction and WS-ReliableMessaging are mutually exclusive. They must not be used together.

10.4 Availability

As most Web services are stateless, to create a highly available Web service, it is ordinarily enough to cluster a number of server instances behind an HTTP workload balancer and to ensure that there are no single points of failure on the route messages taken to and from your cluster. However WS-ReliableMessaging introduces state into the picture. The WS-ReliableMessaging sequence has state that needs to be managed in a highly available environment to allow multiple server instances to participate in a given sequence, and to take over message processing or retransmission if a server fails (Figure 10-14).

In the Feature pack for Web services implementation, this is supported using the managed persistent and managed non-persistent qualities of service which store state in the shared messaging engine in a WebSphere Application Server Network Deployment cluster.

The state in the initial sequence creation handshake is not stored because it is not recoverable. For this reason, a WebSphere Application Server Proxy which correctly routes CreateSequenceResponse messages must be used in place of a generic HTTP load balancer.

Attention: Clustering of a WS-RM application is dependent on use of the WebSphere Application Server Proxy to correctly route CreateSequenceResponse messages to the server creating the response sequence. The Feature pack for Web services does not upgrade the WAS Proxy, thus only SOAP 1.1 is supported.

While turning off the Web services function in the WAS Proxy allows SOAP 1.2 WS-ReliableMessaging messages to flow, they are not correctly routed and errors appear in the server logs.

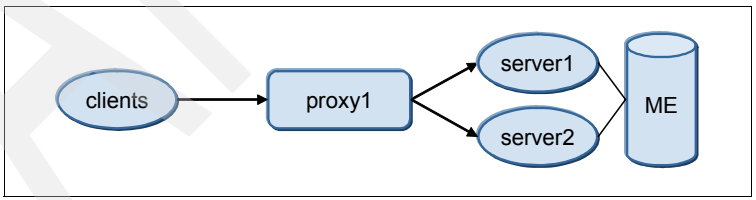


Figure 10-14 Simple clustered topology

10.4.1 High availability topologies

There are two levels of high availability possible with the Feature pack for Web services.

Highly Available WS-ReliableMessaging endpoint

In this case when a messaging engine fails, the WS-ReliableMessaging sequences it hosts and any undelivered messages associated with those sequences are lost. New sequences created are hosted in the failed over messaging engine. This corresponds to the managed non-persistent quality of service using a clustered messaging engine.

Highly Available WS-ReliableMessaging Sequence

In this case, all sequence state is stored durably in the messaging engine. When the messaging engine fails over undelivered messages associated with the sequence will be delivered and new messages associated with the sequence will be accepted. This corresponds to the managed persistent quality of service using a clustered messaging engine.

Example

Figure 10-15 shows a possible deployment configuration supporting highly available WS-ReliableMessaging sequences. This example uses a highly available messaging engine implemented using a highly available database and the managed persistent form of WS-ReliableMessaging. Given the complexity of this deployment, it is important to consider whether highly available sequences are actually required. Because of the asynchronous nature of WS-ReliableMessaging, in scenarios such as “Check clearing” on page 48 might not require high availability, but persistence and transactionality.

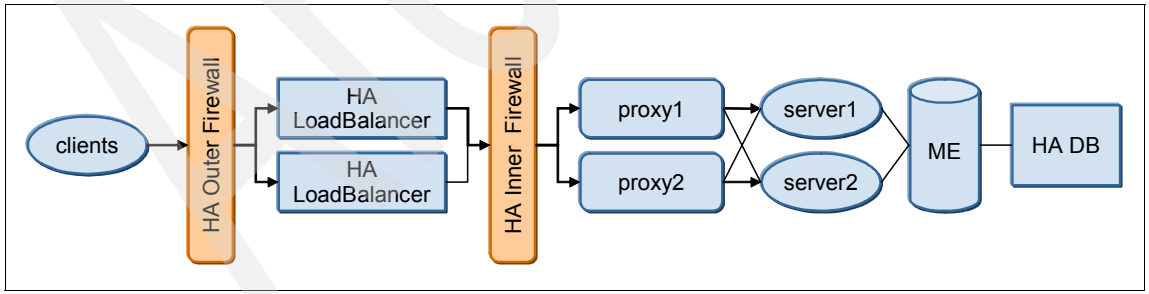


Figure 10-15 Example deployment of highly available WS-ReliableMessaging

It is important to note that to support this; all paths through the network have to be highly available, including the firewalls and load balancer. For more information about building a highly available architecture, see the Redbooks publication, *"WebSphere Application Server Network Deployment V6: High Availability Solutions"*, found at:

http://www.redbooks.ibm.com/abstracts/sg246688.html

10.5 Scalability and performance

The clustered WS-ReliableMessaging implementation in the Feature pack for Web services was designed with high availability as the focus. Clustering for scalability was not a design focus and is discouraged.

10.5.1 Messaging engines

In general terms, the performance of your application will improve with the number of messaging engines in use, with a maximum possible of one per service. However, "remote" messaging engines result in very poor performance, and should be avoided. This means that you should attempt to co-locate the service and messaging engine on a single server.

One way to do this and support high availability is to have a two server cluster, with a primary and standby server. Weight the primary server heavily such that it processes the vast majority of the requests and also hosts the clustered messaging engine. If the primary server fails, the messaging engine will fail over to the standby server and all requests will be processed by it with a local messaging engine, retaining a similar level of performance. The messaging engine should be configured to fail-back to the primary server when it becomes available.

It is also important not to configure the same messaging engine for a client, server pair as this will lead to very poor performance and possible functional failures.

Performance of persistent WS-ReliableMessaging

The WS-ReliableMessaging use of the messaging engine with a persistent configuration is similar to that when "Assured persistent" JMS is used, so you should look at standard messaging engine advice for that configuration. See the Redbooks publication, *"WebSphere Application Server V6 Scalability and Performance Handbook"*, found at:

http://www.redbooks.ibm.com/abstracts/sg246392.html

Certainly if you are using a filestore messaging engine, disk access becomes a dominant factor in determining the throughput of your service, so a fast disk or Network Attached Storage should be used.

10.6 MOM and WS-ReliableMessaging

Many enterprise IT professionals are familiar with Messaging Oriented Middleware (MOM) such as WebSphere MQ. At a high level, it might appear that WS-ReliableMessaging provides a similar capability. Both aim to provide a reliable, loosely coupled, interoperable messaging capability. This section examines the similarities and differences between them. There are three points to bear in mind when making a comparison:

- These technologies are not alternative ways of doing the same thing. They are different ways of doing different things, and there is no simple comparison between them. In practice, to decide which technology to use (and you might well end up with both), you have to start from specific scenarios you have to support.
- Neither technology is static, and MOM implementations such as WebSphere MQ as it is older and more varied, has capabilities that, while not core to the MOM model, improve its fit to the Internet world: things like Http tunnelling, URI queue naming, RESTful, and thin client support.
- Obviously, Web services and in particular WS-ReliableMessaging are a new technology in relation to MOM, and this influences the choice of technology depending on your attitude to the risk of newer technology versus older.

10.6.1 Messaging Oriented Middleware

Messaging Oriented Middleware (MOM) is a segment of the middleware market focused on providing a messaging infrastructure that is reliable, manageable, and simple to use. It is commonly queue based with a queue based API, and is exemplified by WebSphere MQ and APIs like JMS. Other examples of MOM include MSMQ from Microsoft, SonicMQ from Progress and Open MQ from the Glassfish open source project.

Queues, APIs, and infrastructure

The queue-centric nature of most MOM systems provides a strong asynchronous paradigm and extremely simple programming model. It isolates application code from the network, from deployment details such as server addresses, from management issues such as availability, performance, monitoring, and tuning, and it provides a reliability model, with the level of reliability and transactionality defined in terms of the queue and the interactions with it.

MOM vendors provide a common API and deployment infrastructure and require their own networking agents at both ends of a connection. This has contributed to MOMs reliability and ease of deployment into a managed environment. With only one vendor's software involved at both ends of a network connection, hiding the intricacies of network programming, you have a good expectation that both ends of the connection will work together, especially important when both ends of the connection might be running on different hardware and operating systems, and if they do not, you only have one vendor to call.

10.6.2 Web services

The history of Web services has a lot of different threads. One of the threads is for Web services to be the Internet protocol for programs. Taking a leaf out of the TCP/IP and HTTP recipe for success, Web services strategy for widespread adoption and interoperability is by defining a protocol on the wire (SOAP), and leaving different vendors to find the programming models and deployment mechanisms that best meet their requirements. This strategy suits the unmanaged environment of the Internet well, and is becoming of much greater significance as programs make use of resources on the Internet rather than being locked into fixed client/server or other fixed endpoint patterns.

Programs being a lot less intuitive than human beings, Web services also define how these protocols are used and what the contents mean. The success of Web services so far has been down to the interoperability achievements of SOAP, and the success of WSDL and .XSD Schemas in defining how the SOAP protocol is to be used and the meaning of the messages contained within.

10.6.3 Comparison

Corresponding to the queues, APIs and infrastructure of MOM, we have the URIs, protocols and composability of Web services.

URIs and queues

There is a lot of similarity between queues and URIs, not least because now in WebSphere MQ v7.0, queue destinations are cast as URIs, and in the Feature pack for Web services URIs correspond to message queues used internally to move service requests and responses between senders and receivers. The major difference is that the queue is a pivotal part of the MOM programming and deployment model, whereas for WS-ReliableMessaging it is hidden from the application and deployment model.

The great virtue of URIs is they are a universal feature of the Internet and do not require an additional look up or deployment mechanism, unlike the destinations of JMS or the queues of WebSphere MQ. From the perspective of Web services, the network model is very simple. It is a set of URIs.

This is the virtue of queues: They acknowledge that the network is a source of problems and provide a mechanism to isolate programs from the network, delegating many of the tasks necessary to make a networked program work reliably to an administrator, thus simplifying the programming task, and resulting in an overall improvement in reliability and availability.

In a nutshell, URIs are simple to administer, and for programs with less demanding reliability and availability requirements, simple to program to. On the other hand, it is easier to achieve a high degree of reliability and availability by using queues to share responsibility between programmers and administrators, than it is to program and configure reliability and availability into a Web service.

Protocols and APIs

A common API or a common protocol are both ways to achieve isolation from underlying technologies and thus interoperability. An API can be tied to an underlying implementation (like WebSphere MQ), or it can be decoupled from its implementation, (like JMS). But whether the API and implementation are coupled or not, the common API approach couples the implementation at either end of the wire, unless there is an agreed common protocol. Both approaches are very successful: protocols are particularly found in network layers and APIs in presentation layers.

The strengths of the common protocol approach compared to a common API are twofold:

- Agnostic about the implementation at each end of a connection.
- Composable and extendable with other protocols and services at and between the endpoints.

The composability allows WS-ReliableMessaging to be applied to existing services with no change to the applications, and to be supplemented by new capabilities such as authentication, encryption and signature, again without changing the underlying services.

The common API approach can claim to score more highly on:

- Simplicity, at least for anything beyond the simplest requirement
- Interoperability between existing applications

Infrastructure and composeability

MOM is sold as a package, complete with infrastructure and a large ecosystem of supporting projects and services; whereas WS-ReliableMessaging is embedded into a platform and makes use of its services, as well as being composed with other Web services services. In comparing MOM and WS-ReliableMessaging, you are not making a like for like comparison. In practice, the comparison is not between the technology that is WS-ReliableMessaging and the technology that is MOM — the comparison is between MOM platforms against the platforms WS-ReliableMessaging is embedded into. You can really only make that comparison with specific scenarios in mind.

10.6.4 Conclusion

When should you choose MOM and when WS-ReliableMessaging, or both? Of course it depends on your business and the scenarios you intend to implement. And that is the main point — look at the broad business picture, rather than compare only the features of the technologies. Evaluate which approach is better for each scenario in your context.

Today, it seems to us, there are two sweet spots for WS-ReliableMessaging: Providing session reliability across wireless connections (“Insurance underwriter” on page 40) and reliable and persistent store and forward messaging, especially for federations of applications (“Check clearing” on page 48).

10.7 Using the SPI

There are situations where simply applying a Policy set including WS-ReliableMessaging is not enough to build a well behaved WS-ReliableMessaging application. An example is that of a thin client JVM process that must not end until the message it sent has been acknowledged and the sequence terminated. In these situations we use the WS-ReliableMessaging SPI to gain more control over the WS-ReliableMessaging sequence in use.

> **Note:** Always apply a Policy set including WS-ReliableMessaging, otherwise you receive an exception when trying to use the SPI.

The SPI operates on a JAX-WS client, and has the methods shown in Table 10-1.

Table 10-1 SPI methods

Method	Action	Comments
Create sequence	Create a new sequence associated with this client.	Create sequence is not necessarily complete when the method returns. This method throws an exception if there is an existing sequence associated with this client.
Send acknowledgement request[a]	Sends a sequence acknowledgement request to the destination which responds with an acknowledgement indicating which messages have been successfully received and which the client store can discard.	While most current WS-ReliableMessaging implementation send acknowledgements frequently, this can be useful if sending large messages to one which does not, and consequently allows the client storage to fill up.
Close sequence	Instruct the destination to stop accepting new messages in the sequence but retain sequence state to allow tidy-up.	Do not do this before a wait for completion as it can prevent messages being received by the destination, hence prevent them being acknowledged, hence the sequence does not complete.
Terminate sequence	Instruct the destination that the sequence is no longer of use and can be discarded.	You use this when you are finished with a sequence and do not wish to ensure that all messages were received e.g. shutting down in a hurry.
Wait for completion	Wait until all the sent messages have been acknowledged. It then terminates the sequence. There are two flavors of this method, one with a time-out, and one without.	If you are waiting for completion on sequences for a large number of clients and it is taking too long, you might want to send an acknowledgement request for each first before calling waitUntilSequenceCompleted. As discussed in “Sequence lifecycle”, it is important to terminate sequences to free up sequence state without having to wait for the 24 hour inactivity time-out. This is a particularly common issue when using the thin client where the JVM can simply exit.

a. Not supported with WS-ReliableMessaging 1.0.

Attention: With the thin client, it is also important to note that because of the asynchronous nature of the WS-ReliableMessaging implementation, it is common for the Web service method invocation to return before the message is actually sent. If the next step in the program is for the JVM to exit, then the message might never be sent.

You should use “waitUntilSequenceCompleted“ to ensure that all messages have been sent and that the sequence has been terminated before your client JVM exits.

10.7.1 What you cannot do with the SPI

The SPI is not part of the WS-ReliableMessaging specification. The specification only deals with protocols, not with implementations. So programs written using the SPI are not so easy to migrate to a different platform. But of course, the protocols generated by the SPI are conformant with the WS-ReliableMessaging specifications.

Our software engineering experience, and experience with writing loosely coupled applications, suggests that you should not mix business logic with network management logic, and you should not need finer grained control. Moreover, as Table 10-1 shows, there are limitations with the SPI and it is not easy to program correctly. You want to be able to trust that the WS-ReliableMessaging implementation works and gets your message to its destination.

Our recommendation is to aim to deal with failures administratively and not programmatically within the application. In order to do that, sequence state is viewable and modifiable in the administrative console. It is possible to export messages which are stuck in a sequence that is unable to complete successfully.

These options are also available via JMX™ MBeans, which opens up the possibility of writing your own management applications or purchasing management solutions from a vendor, letting your programmers focus on providing functionality to meet business requirements, and your infrastructure team managing the reliability, availability, and performance of the solutions.

10.8 Examples

In this section we apply persistent and non-persistent Policy sets and learn how to use the WS-ReliableMessaging SPI to gain more control over the WS-ReliableMessaging sequences.

10.8.1 Applying an unmanaged quality of service

The unmanaged non-persistent QoS is very simple to apply clients and services using one of the supplied Policy sets. First we have to select which version of WS-ReliableMessaging to use. If we select WS-ReliableMessaging 1.0, then we use the `WSReliableMessaging 1_0` Policy set for this example instead of the `WSReliableMessaging default` Policy set.

You can follow the instructions for applying the `WSAddressing default` Policy set in 8.3.3, “Attach a Policy set to a Web service” on page 257, except use the `WSReliableMessaging 1_0` or `WSReliableMessaging default` and apply it not to the PingService but to **EchoService** in the **WSSampleServicesSei** and **WSSSampleClientSei** application (Figure 10-16).

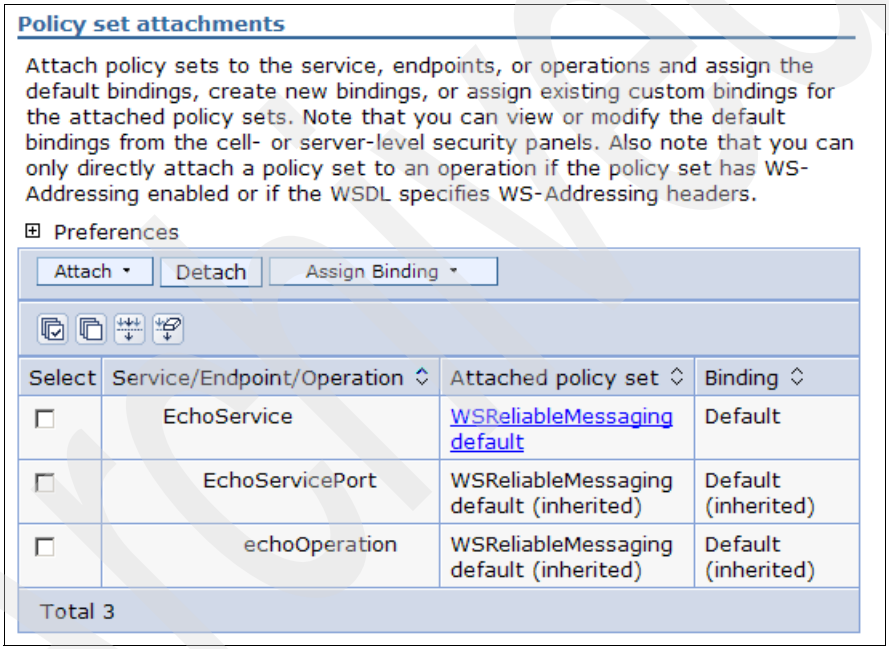


Figure 10-16 Applied WSReliableMessaging default to EchoService (client and server)

> **Tip:** If you did the examples in Chapter 8, “Policy sets” on page 247, you have to detach the WS-Security Policy set first from both the client and server.

Ensure that you have applied the same Policy set to both the client and the server and *stop and restart* both the client and server to pick up the new Policy sets. You might want to run the test program before and after restarting the applications (Figure 10-18). The result is shown in Figure 10-17.

```
Connecting to... http://localhost:9081

Message Request:
Hello Reliable World


Message Response:
JAX-WS==>>Hello Reliable World
```

Figure 10-17 Result of running demo program

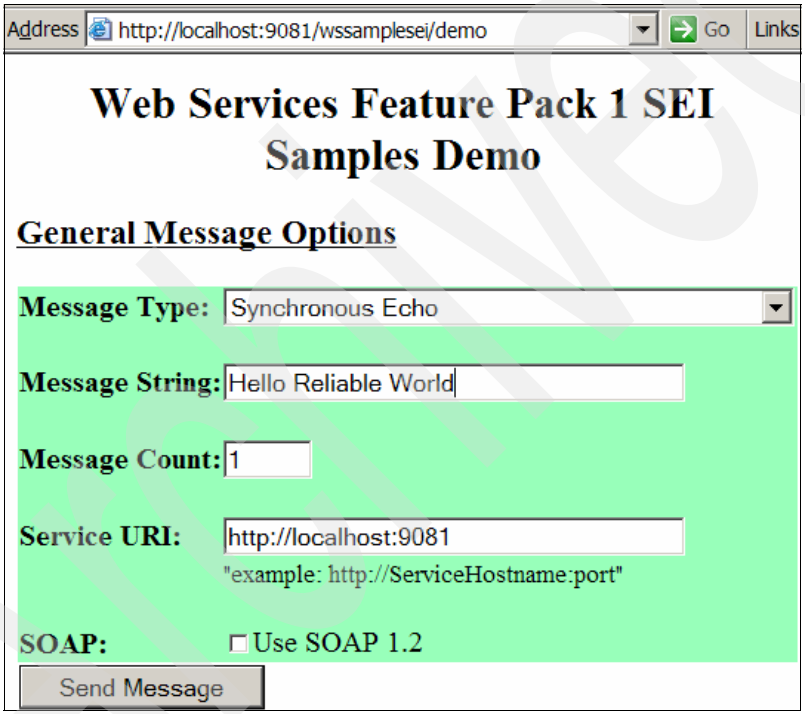


Figure 10-18 Testing reliable messaging using demo client

View WS-ReliableMessaging flows

To confirm that we have applied the Policy set correctly, and to further understand the message flows, we view the messages using the TCP/IP Monitor bundled in Rational Application Developer following the instructions found in “Monitor the SOAP traffic” on page 264.

For the single application message that you send using the sample Web client, you will see some messages in the TCP/IP monitor similar to the message flow illustrated in Figure 10-6 on page 368. Depending on whether this is the first request in a sequence, or a subsequent one, we will see different messages in the TCP/IP monitor. It is simplest to start by looking at subsequent messages.

Subsequent flows

Figure 10-19 shows the salient features of the messages. Here we see there are two sequences established, 78 from the client to the server, and 36 from the server to the client (we have abbreviated the actual uuids). The EchoResponse flow, as well as providing the response, also piggybacks the acknowledgement to all the messages in the sequence that have flowed from the client to the server in sequence 78. The final flow is in one direction only, and acknowledges the echoOperationResponse message sequence.

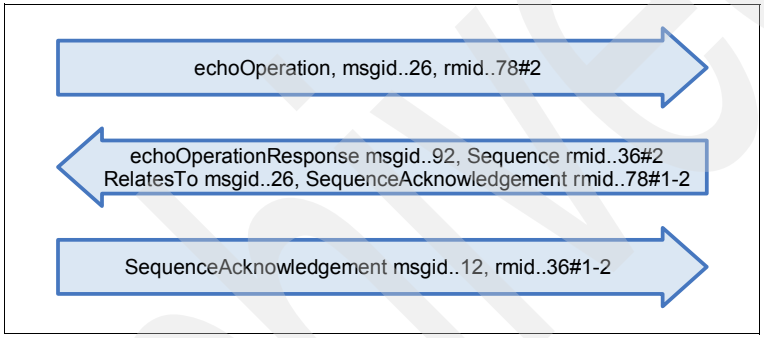


Figure 10-19 Second EchoOperation request

Initial flow

To see the initial flows that start the two sequences we deleted all the sequences from the server using the Administrator Console. We will look at the results first, and then show you how to delete sequences using the Administrator Console.

Figure 10-20 is the complex series of flows that initiated the sequence. The flows start off straightforwardly enough. The client requests a sequence, gets a response with the sequence identifier, and then sends the echoOperation request.

It now gets more complicated, not least because the order of messages in the TCP/IP monitor does not correspond to the order the messages were sent, and because the timestamps in the messages are not of sufficient resolution to determine the true order. The message numbers are the order of the messages as they appeared in the TCP/IP monitor, whereas the ordering in the diagram is the order that we have inferred, and it can vary from run to run.

The reason the order is indeterminate, is that we are using a synchronous wire protocol, and the client uses the MakeConnection flow to poll the server from time to time to see if it has anything to tell the client.

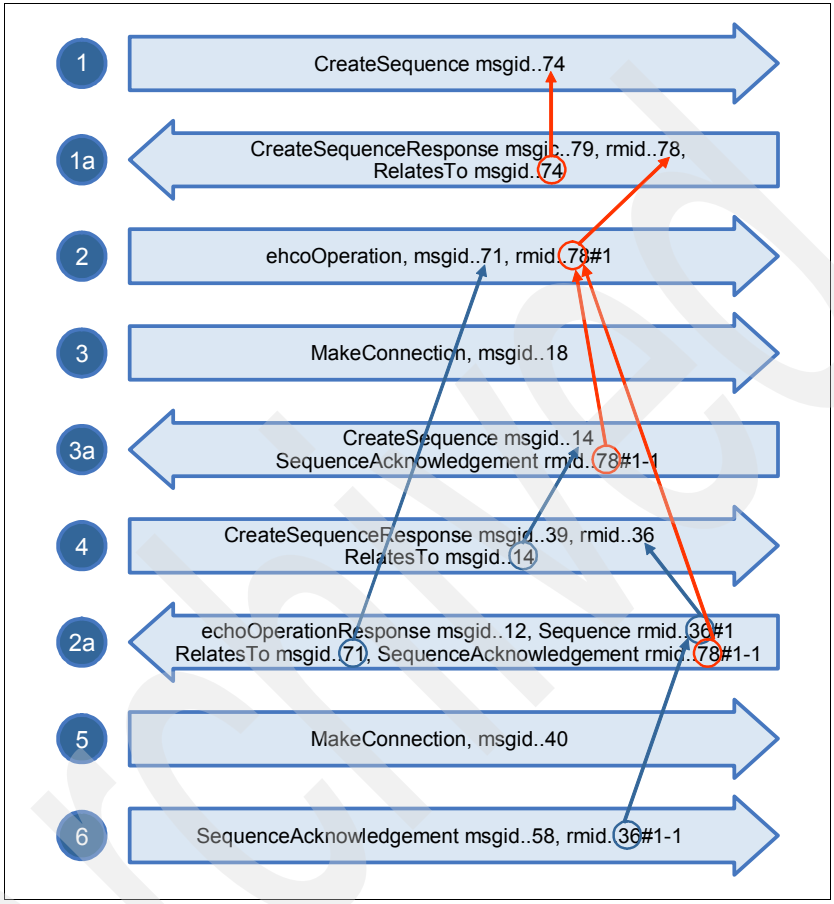


Figure 10-20 Initial echoOperation message sequence

We know that sequence 36 needs to be created before it is used, so probably the flow sequence is as illustrated, which would also go some way to explaining why sequence 78 is acknowledged twice, once by the thread that was responding to the make connection with creating a new server to client sequence, which saw it could piggyback an acknowledgement to the echoOperation request message, and once by the thread which responded to the echoOperation requests with the echoOperation response.

10.8.2 Managing message sequences

In this section we look at how to manage message sequences using the Administrator Console.

- Open the Administrator Console and select **Servers** → **Application servers** from the left hand navigation bar.
- Under the Configuration tab, find Additional Properties at the bottom of the page → **Reliable messaging state**.

If we select **Message store**, we can see our inbound and outbound non-persistent message stores (Figure 10-21).

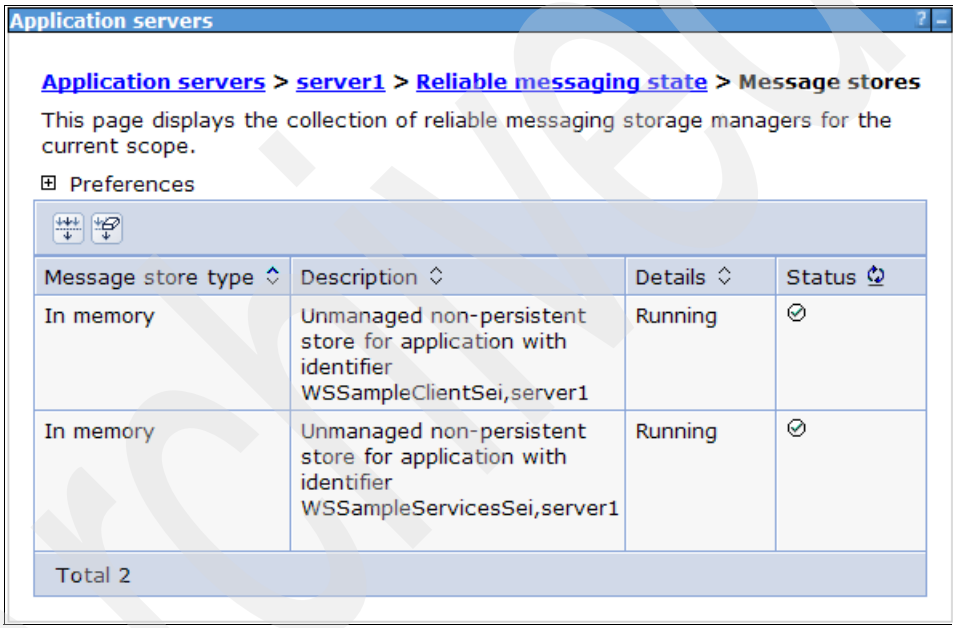


Figure 10-21 Message stores in use

If we select Inbound sequences, we can see the store message sequences (Figure 10-22), and their inbound state for both the client and server (note that the last two digits of the sequence identifiers match the numbers used in Figure 10-19 on page 390 and Figure 10-20 on page 391 — these are the flows we traced). We can also look at the outbound sequence state.

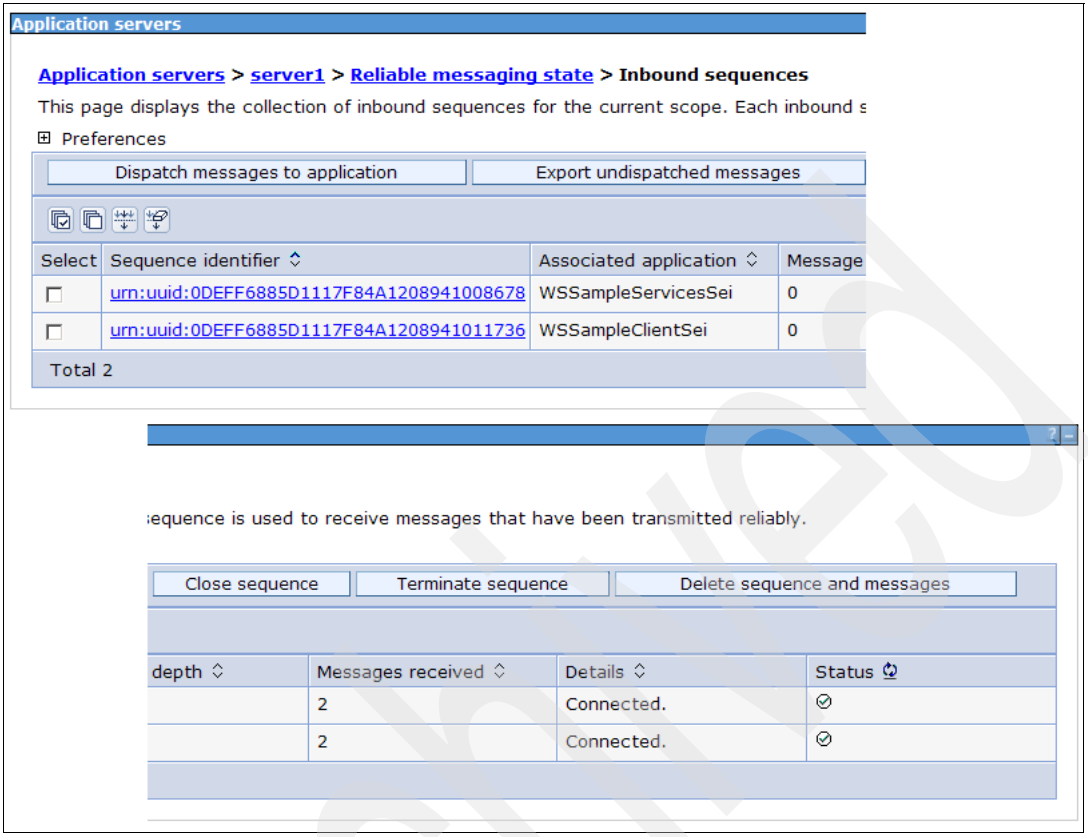


Figure 10-22 Inbound messaging sequence state

We can drill down further to see the properties of each sequence, any unacknowledged messages, the acknowledgement state, and other information.

Clearing the message state

For the non-persistent messages we are looking at, restarting the server is a quick way to clear the messaging state:

- Drill down to the inbound sequences panel (Figure 10-22) → Check all the sequences to be cleared → **Delete sequence and messages** → **OK**.
- Similarly in the outbound sequences panel → Check all the sequences to be cleared → **Delete messages** → **OK**.

10.8.3 Applying a managed persistent QoS

In this example we apply the `WSReliableMessaging persistent` Policy set to EchoService. It involves creating Service Integration Bus (SIB) artifacts used to persist the messages. This will be a simplistic process, and you should give more thought to designing your SIB infrastructure for a production system.

Create the Service Integration Bus artifacts

- In the Administrator Console navigator → **Service Integration** → **Buses**.
- Click New → Type `itsoBus` as the bus name, and for simplicity of this example, toggle off Bus security (Figure 10-23) → **Next** → **Finish** → **Save**.

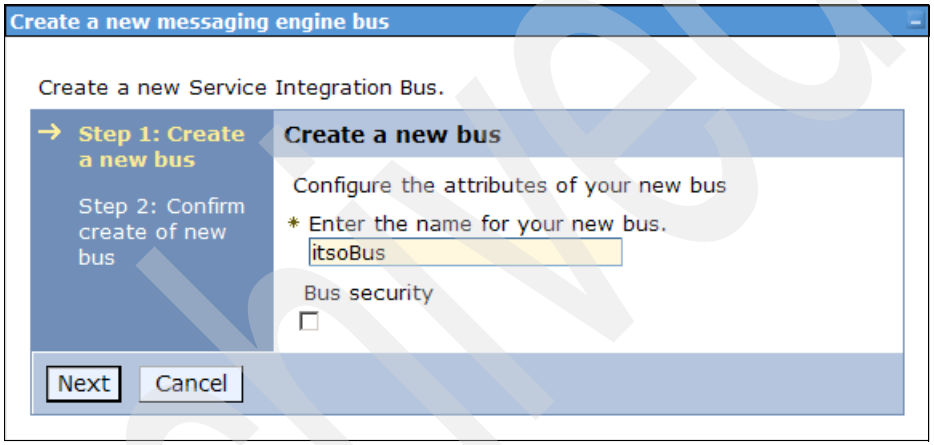


Figure 10-23 Create a service integration bus without security enabled

☐ Click itsoBus to navigate to the itsoBus main panel (Figure 10-24).

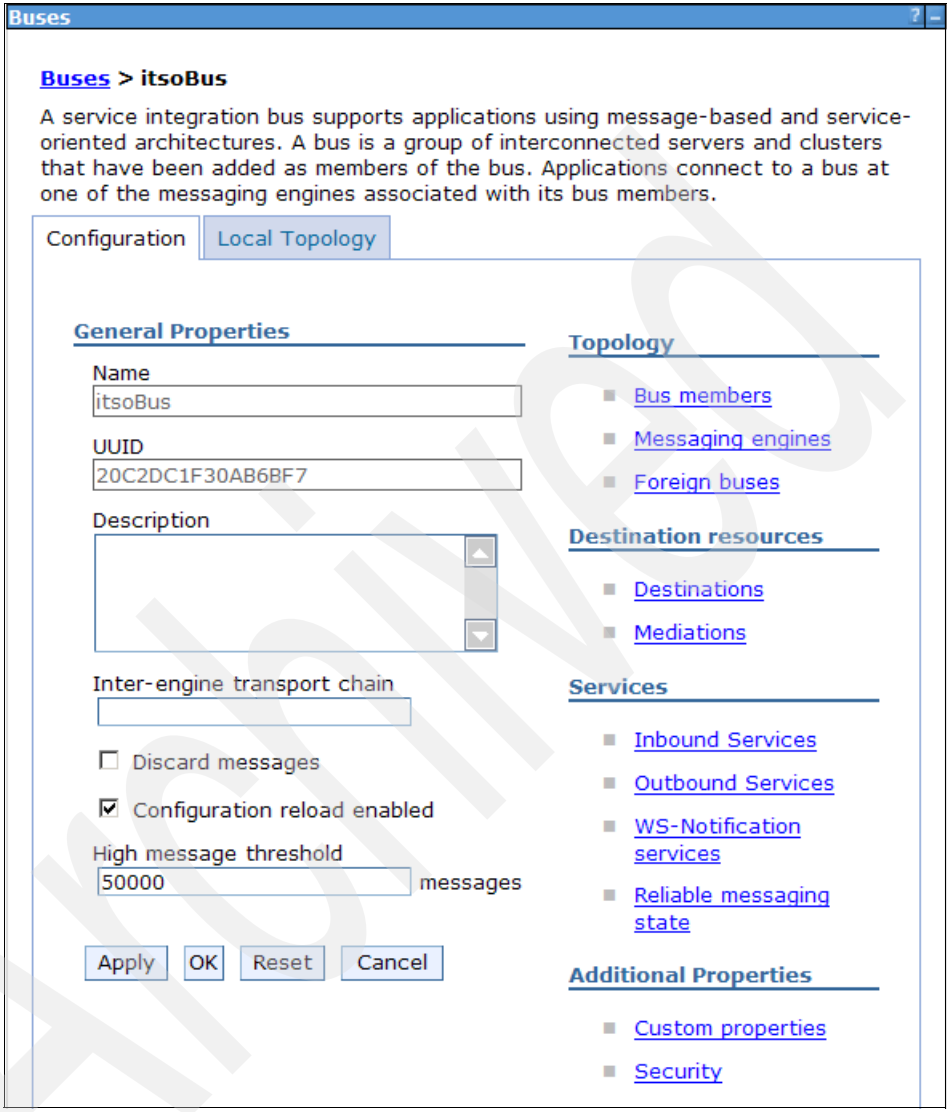


Figure 10-24 The itsoBus general administrative panel

☐ Click **Bus members** → **Add** to add a messaging engine to the bus (Figure 10-25).

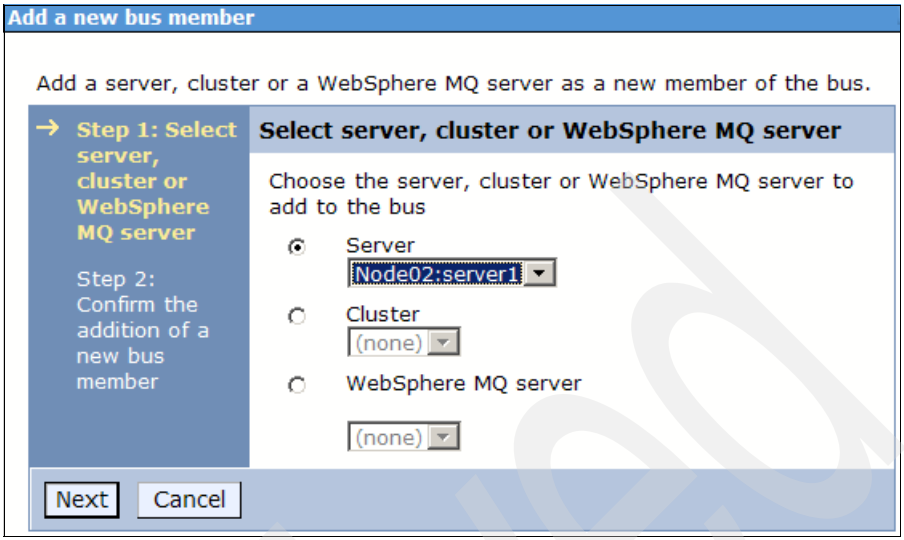


Figure 10-25 Select the server to add the messaging engine to

Click through the panels and save. This will create a filestore based messaging engine well suited to single server experimentation. (Figure 10-26).

- ☐ **Next** → **Next** → **Next** → **Finish** → **Save**.

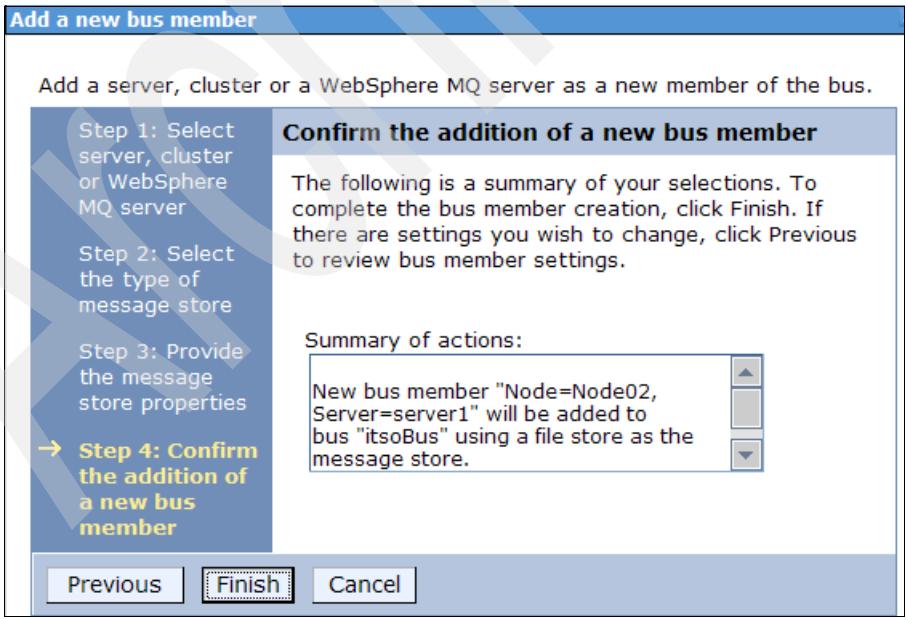


Figure 10-26 Complete creation of the messaging engine by confirming the configuration

Modify the default binding

We now have to let the WS-ReliableMessaging implementation know which messaging engine to use.

- ☐ Navigate to **Services** → **Policy sets** → **Default Policy set bindings** shown in Figure 10-27 → **WS-ReliableMessaging (**Figure 10-27).

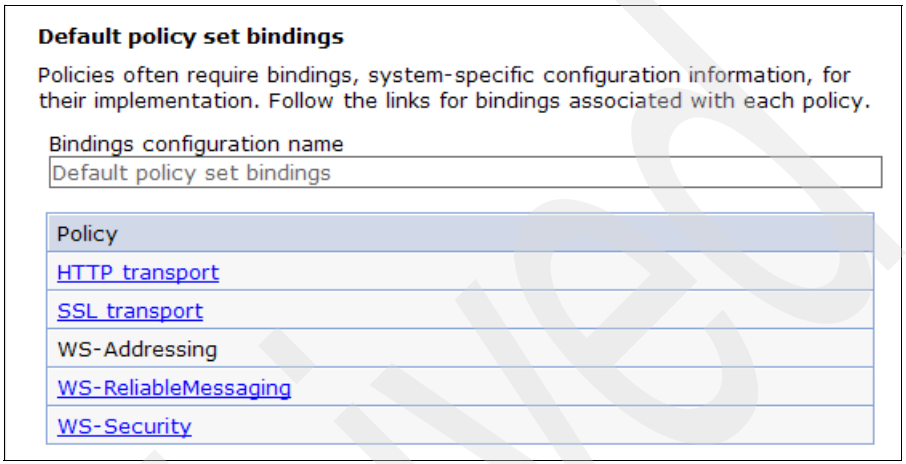


Figure 10-27 Default Policy set bindings panel

- ☐ Select the bus and messaging engine created in Figure 10-28 → **OK** → **Save**.

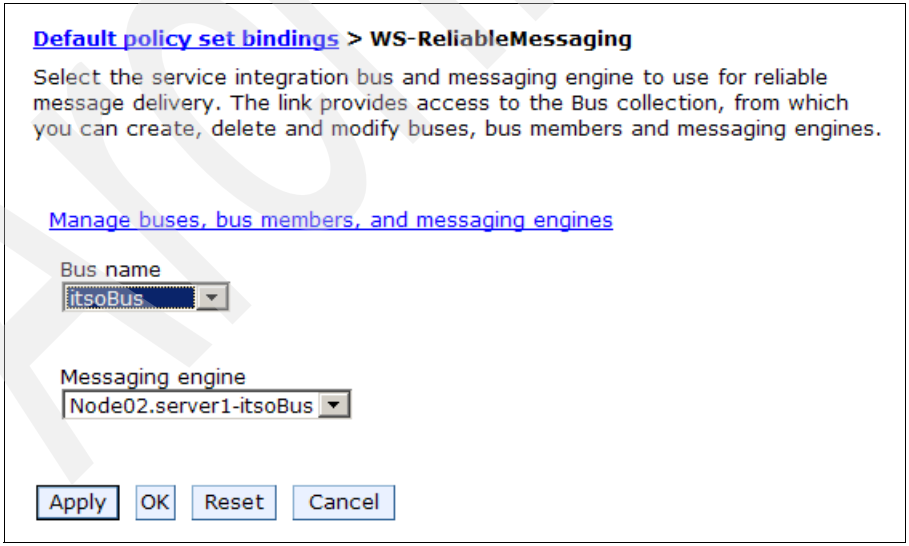


Figure 10-28 Select the Bus and Messaging engine

Apply the WSRM persistent Policy set

In this section we remove the WS-ReliableMessaging default Policy set that we applied to the EchoService in 10.8.1, “Applying an unmanaged quality of service” on page 388, and instead apply the WS-ReliableMessaging persistent Policy set. Ensure that the WS-ReliableMessaging default Policy set is applied to the EchoService client.

> **Note:** The WS-ReliableMessaging persistent Policy set uses WS-ReliableMessaging 1.1. If you wish to use persistent WS-ReliableMessaging 1.0 you have to create a new Policy set.

- Navigate to **Service providers** → **EchoService** (in the WSSampleServicesSei application) → check **EchoService** → **Detach** → **Save** (Figure 10-29).

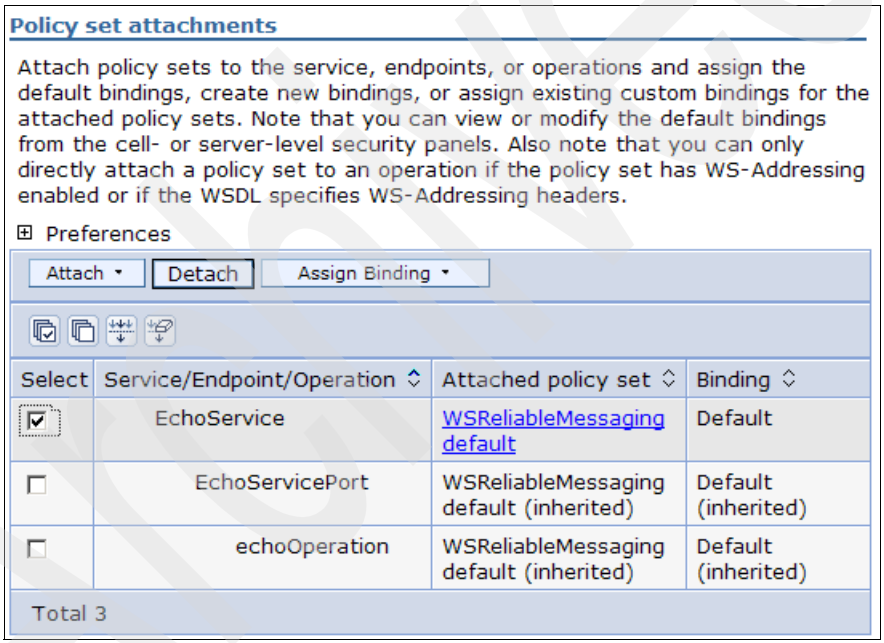


Figure 10-29 Detach WS-ReliableMessaging default

- Returning to the same screen, select **EchoService** again → **Attach** → **WS-ReliableMessaging persistent** → **Save** (Figure 10-30).

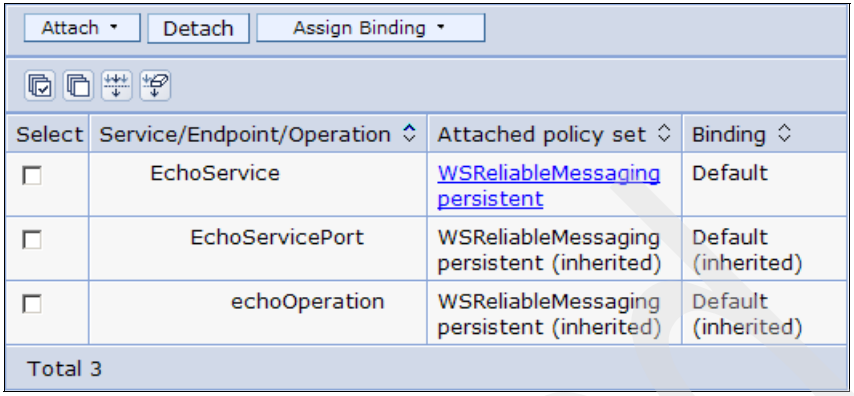


Figure 10-30 Attach WS-ReliableMessaging persistent to EchoService

- Restart the EchoService service to pick up the Policy set changes.

Try it out

To demonstrate that the configuration works, we use the sample Web client to invoke the EchoService (Figure 10-31).

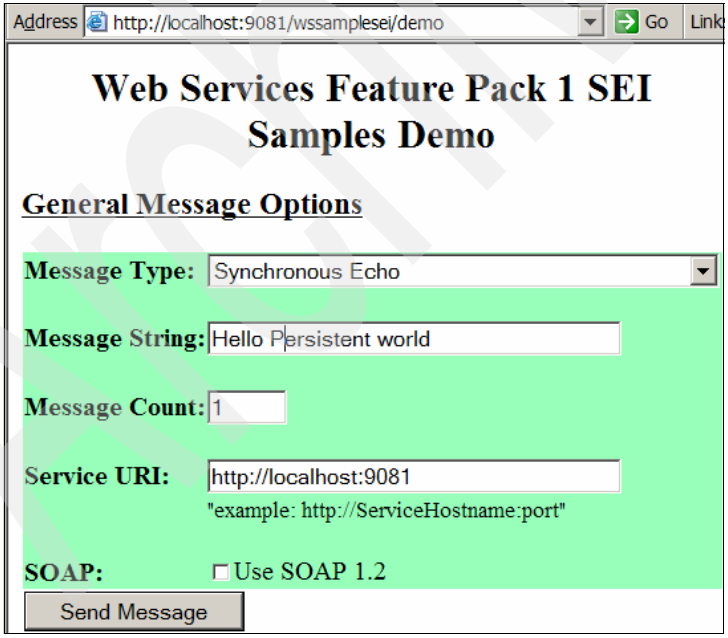


Figure 10-31 Invoke EchoService with the Web samples demo

Once this has successfully returned, open the Administrator Console and view the Console (Example 10-1).

Example 10-1 console output running EchoService

```
SystemOut     O >> SERVLET: Request count = 1
SystemOut     O >> SERVLET: Request index: 1
SystemOut     O >> CLIENT: SEI Echo to http://localhost:9089/WSSampleSei/EchoService
WSChannelFram A   CHFW0019I: The Transport Channel Service has started chain HttpOutboundChain:localhost:9089.
SystemOut     O >> SERVICE: SEI Echo JAX-WS Service: Request received.
SystemOut     O >> SERVICE: SEI Echo Input String 'Hello Persistent world'
SystemOut     O >> CLIENT: SEI Echo invocation complete.
SystemOut     O >> CLIENT: SEI Echo response is: JAX-WS==>>Hello Persistent world
```

- ☐ Navigate to **Buses** → **itsoBus** → **Reliable messaging state** → **inbound sequences** (Figure 10-32) and to **outbound sequences** (Figure 10-33).

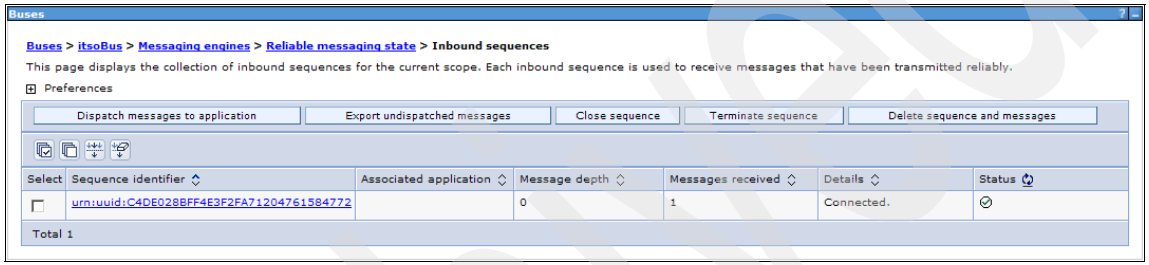


Figure 10-32 View of inbound sequence stored in a messaging engine

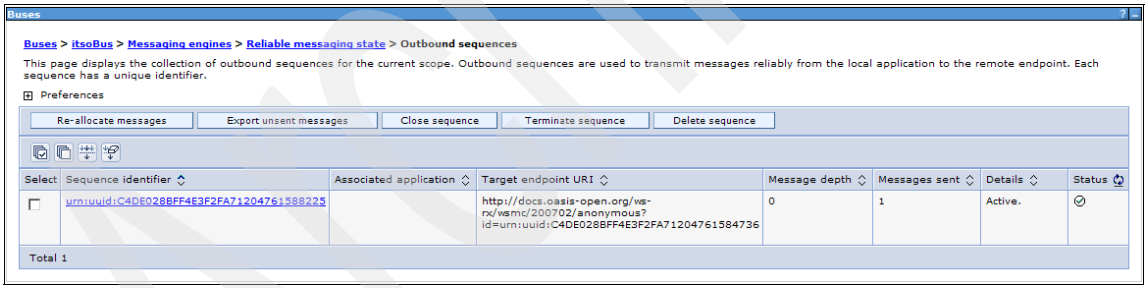


Figure 10-33 View of outbound sequence stored in a messaging engine

10.8.4 SPI Sample code

The most common use of the WS-ReliableMessaging SPI is ensuring that a sequence completes before a thin client JVM exits. This is important for the reasons outlined in “Sequence lifecycle” on page 359 and very straightforward to do. Add the following line of code to your thin client before it exits the main() method (Example 10-2).

Example 10-2 Prevent the JVM exiting before messages are sent and acknowledged

```
WSRMSequenceManagerFactory factory = WSRMSequenceManagerFactory.getInstance();
WSRMSequenceManager manager = factory.createWSRMSequenceManager();
manager.waitUntilSequenceCompleted(clientObject, portQName, endPointUri);
```

11

Interoperability

Web Services have revolutionized the software industry with the promise of interoperability. However, despite new and improved Web Services products, cross-platform interoperability has been problematic. As Web services standards increase in size and complexity, vendor products vary in their interpretation of fine-grained implementation details. Therefore there is an increased risk that interoperability across platforms or applications might fall short.

As a result, the Web Services Interoperability organization, or WS-I, was created to mitigate such issues. In this chapter, we briefly discuss the role of WS-I and its deliverables that accelerate interoperability. Among WS-I's deliverables, the WS-I Profiles have most significance. WS-I profiles provide a tangible mechanism to establish interoperable Web services applications. The Feature pack for Web services uses WS-I Profiles to assert compliance with WS-I interoperability standards and guidelines. In this chapter, we discuss the enhancements added in the Feature pack for Web services runtime and tools to support WS-I profiles.

We also give guidance about how to develop Web services that interoperate between J2EE and the Microsoft .Net Framework. We use a sample Insurance Underwriter application to demonstrate various levels of interoperability between the Feature pack for Web services and the Microsoft .Net Framework. Finally, we provide a list of best practices that simplify developing interoperable Web services applications.

11.1 Understanding Interoperability

According to IEEE, interoperability is defined as follows:

> **Definition:** The ability of a system or a product to work with other systems or products without special effort on the part of the customer. Interoperability is made possible by the implementation of standards.

Platform vendors and solution providers market products and services that enable the development of conformant and interoperable Web services. Standards organizations such as the Open Applications Group have incorporated the WS-I profiles into their own specifications, and others such as OASIS and the W3C are improving how standards are developing with a more critical eye towards true interoperability based on feedback from WS-I and its members.

Increasingly, customers are requiring conformance from their vendors and solution providers and considering interoperability a key criterion for vendor selection. Furthermore, leading industry analyst firms are advising their clients to enforce use of the WS-I profiles as part of a company's core development process to help ensure interoperability and, ultimately, success of their Web services implementations.

It is important to get a basic understanding of Interoperability within distributed systems from a conceptual point of view before looking at practical ways of developing interoperable Web services.

Conceptual Levels of Interoperability model (CLI)

The CLI model, as shown in Figure 11-1, proposes a conceptual layered approach to interoperability.

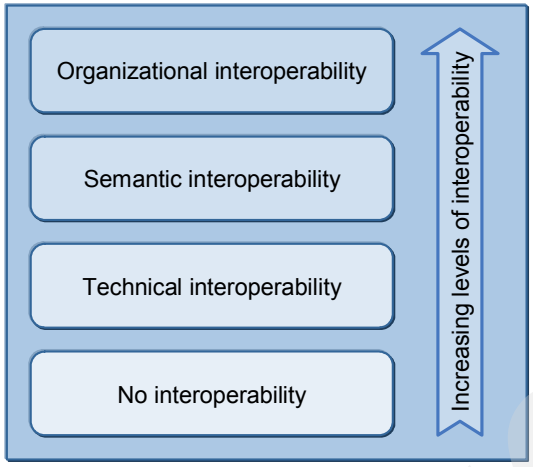


Figure 11-1 Conceptual Levels of Interoperability

Here we describe the levels of the CLI model:

1. No Interoperability

   Stand-alone systems have no interoperability.

2. Technical Interoperability (communication protocols)

   Under Technical interoperability, a communication protocol exists for exchanging data between participating systems. On this level, a communication infrastructure is established allowing systems to exchange bits and bytes, and the underlying networks and protocols are unambiguously defined.

   Consider the following scenario:

   – Messages are exchanged securely and reliably between the sending and receiving infrastructure.

   – The receiving infrastructure is responsible for delivering the message payload to its applications.

3. Semantic Interoperability (application syntax and semantics)

   Semantic interoperability is the ability of two or more computer systems to exchange information and have the meaning of that information accurately and automatically interpreted by the receiving system.

   Semantic interoperability is distinguished from other forms of interoperability by considering whether the information transferred has, in its communicated form, all of the meaning required for the receiving system to interpret it correctly, even when the algorithms used by the receiving system are unknown to the sending system.

Consider the following scenario:

- An application knows the business context to which the payload belongs.
- The payload is valid from an application perspective.
- The application successfully processes payload.

4. Organizational Interoperability (business processes)

   Organizational interoperability (business process interoperability) is a state that exists when a business process meets a specific objective automatically using human labor only exceptionally. Typically, Organizational interoperability is present when a process conforms to standards that enable it to achieve its objective regardless of ownership, location, make, version or design of the computer systems used.

   Consider the following scenario:

   - An application notifies appropriate users that are responsible for verification and approval steps, as well as tracking deadlines.

Figure 11-2 shows how interoperability is applied at different layers within the context of a Web services framework.

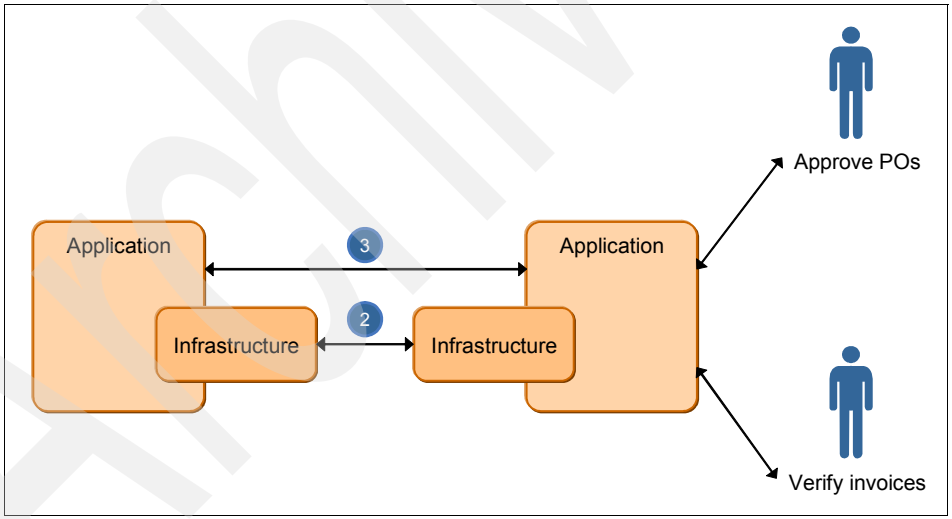


Figure 11-2 Application of the CLI model

11.2 Web Services Interoperability organization

Note: This section is an update and elaboration of Chapter 9, *Web services Interoperability*, and Chapter 23, *Web services interoperability tools and examples*, in the Redbooks publication, *Web services Handbook for WebSphere Application Server v6.1,* SG24-7257, found at:

http://www.redbooks.ibm.com/abstracts/sg247257.html

Most companies realize that Web services interoperability plays a vital role in the adoption of Web services. As a result, an open industry Web Services Interoperability Organization (WS-I) was chartered; governed by a consortium of industry-leading corporations, including IBM. The primary goal of WS-I is to provide direction in promoting Web services interoperability across platforms, operating systems, and programming languages.

The WS-I, or the Web Services Interoperability organization organization, encourages vendor and consumer companies to promote Web Services Interoperability organization through a joint effort. The organization's leading companies provide guidance, recommended practices and supporting resources to assist customers to develop interoperable Web services.

The WS-I organization provides a set of deliverables to accelerate and to encourage the adoption of Web Services. These deliverables primarily include profiles, sample applications and testing tools.

11.2.1 Profiles

Profiles "*provide implementation guidelines for how related Web services specifications should be used together for best interoperability*". Profiles are not yet another set of specifications. They include clarifications, refinements, interpretations and amplifications of those specifications which promote interoperability. Additionally, profiles lay down the guidelines and conventions for using these specifications together in ways that ensure interoperability.

Several profiles have been delivered to date, with newer ones currently available as working drafts. A comprehensive list of final and working drafts has been provided in Table 11-1.

Table 11-1 Working status of WS-I Profiles

Profile name	Finalized versions	Working drafts
Basic profile	Basic SOAP 1.0	Basic Profile 1.1 Errata
	Basic SOAP 1.1	Basic Profile 1.2
	Simple SOAP Binding Profile 1.0	Basic Profile 2.0
Attachments Profile	Attachments Profile 1.0	Attachments Profile 1.0 Errata
Basic Security Profile	Basic Security Profile 1.0	Basic Security Profile 1.1
Reliable Secure Profile		Reliable Secure Profile

Further details on WS-I profiles and related specifications can be obtained from:

http://www.ws-i.org

Basic Profile (BP)

Basic Profile 1.1 uses several technology components in the composition and implementation of Web services, including messaging, description, discovery, and security. Each of these components are supported by specifications and standards, including SOAP 1.1, Extensible Markup Language (XML) 1.0, HTTP 1.1, Web Services Description Language (WSDL) 1.1, and Universal Description, Discovery and Integration (UDDI). The WS-I Basic Profile specifies how these technology components are used together to achieve interoperability, and mandates specific use of each of the technologies when appropriate.

A combined claim of conformance to both the Basic Profile 1.1 and the Simple SOAP Binding Profile 1.0 is roughly equivalent to a claim of conformance to the Basic Profile 1.0 plus published errata, or:

```
BP 1.1 + SSBP 1.0 <=> BP 1.0 + errata
```

Basic Profile 1.1 (BP 1.1), is composed with the Simple SOAP Binding Profile 1.0 (SSBP 1.0) supersedes the Basic Profile 1.0 (BP 1.0). The Attachments Profile 1.0 (AP 1.0) adds support for SOAP with Attachments (SwA), and is intended to be used in combination with Basic Profile 1.1 (BP 1.1).

Notable changes since Basic Profile 1.0

The BP1.1 specification is derived from the BP Version 1.0, and incorporates published errata against that specification. The key changes are as follows:

- MESSAGE conformance target: Some requirements that had a MESSAGE conformance target in BP1.0 now use a new target, ENVELOPE.

This facilitates alternate serializations of the message, such as that described in the Attachments Profile (AP).

- SOAP Binding: Requirements relating to the SOAP binding's serialization of the message have been moved to the Simple SOAP Binding Profile 1.0 (SSBP 1.0) to facilitate other serializations.

The WS-I Basic Profile 1.1 begins with a basis of this set of open standards:

- SOAP V1.1
- WSDL V1.1
- UDDI V2.0
- XML V1.0 (Second Edition)
- XML Schema Part 1: Structures
- XML Schema Part 2: Datatypes
- RFC2246: The Transport Layer Security (TLS) Protocol V1.0
- RFC2459: Internet X.509 Public Key Infrastructure Certificate and CRL Profile
- RFC2616: HyperText Transfer Protocol V1.1
- RFC2818: HTTP over TLS
- RFC2965: HTTP State Management Mechanism
- The Secure Sockets Layer (SSL) Protocol V3.0

It also adds constraints and clarifications to those base specifications with the intent to promote interoperability. Some of the key constraints are that it:

- Precludes the use of SOAP encoding (document/literal or RPC/literal *must* be used)
- Requires the use of SOAP/HTTP binding
- Requires the use of HTTP 500 status response for SOAP fault messages
- Requires the use of HTTP POST method
- Requires the use of WSDL V1.1 to describe the interface
- Precludes the use of solicit-response and notification-style operations
- Requires the use of WSDL V1.1 descriptions for UDDI tModel elements representing a Web service

Additional details on the specification are available at:

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

Simple SOAP Binding Profile (SSBP)

The Simple SOAP Binding Profile (SSBP) is derived from those Basic Profile 1.0 requirements related to the serialization of the envelope and its representation in the message, incorporating any errata to date. These requirements have been factored out of the Basic Profile 1.1 to enable other profiles to be composable with it.

The profile is based on SOAP 1.1 and WSDL 1.1, and includes guidelines about message serialization and bindings. For example, SOAP 1.1 defines an XML structure for transmitting messages, the envelope. The profile mandates the use of that structure, and places the following constraints on its use:

- A message must serialize the envelope as the exclusive payload of the HTTP entity-body.
- A message must serialize the envelope as XML 1.0.
- A message must have a Content-Type HTTP header field.
- A message's Content-Type HTTP header field must have a field-value whose media type is text/xml.

Refer to the following link to obtain a comprehensive list of constraints and details:

http://www.ws-i.org/Profiles/SimpleSoapBindingProfile-1.0.html

Attachments Profile (AP)

Attachments are typically used to send binary data, for example, data that is mapped in Java code to java.awt.Image and javax.activation.DataHandler. The raw data can be sent in the SOAP message, but this approach is inefficient because binary data is nibblized in an XML document, and the XML parser has to scan the data as it parses the message.

The WS-I Attachments Profile 1.0 provides a solution to the limitations that are presented by Web Services Description Language (WSDL) 1.1. Because WSDL 1.1 attachments are not part of the XML schema type space, they are message parts only. As message parts, the attachments are not converted into arrays or properties of Java beans. The profile defines the wsi:swaRef XML schema type. You should use the wsi:swaRef XML schema type to overcome the limitations of WSDL 1.1 attachments.

The wsi:swaRef type is an extension of the xsd:anyURI type, where its value contains the content-ID of the attachment.

Note: SwA is not without its problems, however. The SwA solution breaks the Web services model to a certain extent. Among other issues, interoperability with Microsoft (despite being a founding WS-I member) is not supported on any of its platforms (and therefore the WS-I Attachments Profile V1.0). Because of this fundamental problem with SwA support, W3C is moving in the direction of Message Transmission Optimization Mechanism (MTOM), which is supported by Microsoft and by the Feature pack for Web services.

For more details, refer to:

http://www.ws-i.org/Profiles/AttachmentsProfile-1.0.html

Basic Secure Profile (BSP)

The Basic Security Profile 1.0 provides guidance on the use of WS-Security and the Rights Expression® Language (REL), Kerberos, Security Assertion Markup Language (SAML), UserName, and X.509 security token formats.

The profile is based on many specifications:

- RFC 2818: HTTP over TLS
- RFC 2246: The TLS Protocol Version 1.0
- The SSL Protocol Version 3.0
- Web Services Security: SOAP Message Security 1.0 (WS-Security 2004), OASIS Standard 200401, March 2004
- Web Services Security: SOAP Message Security 1.0 (WS-Security 2004), Errata 1.0 Committee Draft 200512, December 2005
- Basic Profile Version 1.0 (BP1.0)
- Basic Profile Version 1.0 Errata
- Basic Profile Version 1.1 (BP1.1)
- Simple SOAP Binding Profile Version 1.0 (SSBP1.0)
- XML-Signature Syntax and Processing
- XML Encryption Syntax and Processing
- Web Services Security: UsernameToken Profile 1.0, OASIS Standard 200401, March 2004
- Web Services Security: UsernameToken Profile 1.0, Errata 1.0 Committee Draft 200401, September 2004
- Web Services Security: X.509 Certificate Token Profile, OASIS Standard 200401, March 2004
- Web Services Security: X.509 Token Profile 1.0, Errata 1.0 Committee Draft 200512, December 2005
- RFC2459: Internet X.509 Public Key Infrastructure Certificate and CRL Profile
- Information technology "Open Systems Interconnection" The Directory: Public-key and attribute certificate frameworks Technical Corrigendum 1
- Web Services Security: Rights Expression Language (REL) Token Profile 1.0, OASIS Standard: 19 December 2004
- Web Services Security: Kerberos Token Profile 1.1, OASIS Standard Specification, 1 February 2006
- Web Services Security: SAML Token Profile 1.0, OASIS Standard, 01 Dec. 2004
- Attachments Profile Version 1.0 (AP1.0)
- Web Services Security: SOAP Messages with Attachments (SwA) Profile 1.1, OASIS Standard, 1 February 2006

The profile also incorporates several extensibility points for these specifications:

- Transport Layer Mechanisms
- SOAP nodes and Messages
- SecurityHeaders
- Timestamps
- Security Token References
- XML-Signature
- XML Encryption
- Binary Security Tokens
- Username Token
- X.509 Certificate Token
- REL Token
- Kerberos Token
- SAML Token
- Attachment Security
- Security Considerations

For more details, refer to:

http://www.ws-i.org/Profiles/BasicSecurityProfile-1.0.html

Reliable Secure Profile (RSP)

The Reliable Secure Profile specification forms a basis by referencing, individually or a composition of, the following specifications:

- WS-I Basic Profile 1.2
- WS-I Basic Profile 2.0
- WS-I Basic Security Profile 1.0
- WS-I Basic Security Profile 1.1
- WS-ReliableMessaging 1.1
- WS-SecureConversation

At the time of writing, only the usage scenarios document was available from:

http://www.ws-i.org/profiles/rsp-scenarios-1.0.pdf

Refer to the WS-I Web site on the latest progress:

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

11.2.2 WS-I Conformance Claims

The WS-I profiles allow Web service artifacts to include a claim of conformance when they have been tested to conform to a particular WS-I Profile. The standard refers to this mechanism at:

http://www.ws-i.org/Profiles/ConformanceClaims-1.0-2004-11-15.html

Conformance claims are attached to a wsdl:port element in a WSDL 1.1 description as a child of its wsdl:documentation element, using the Conformance Claim Schema (see "Conformance Claim XML Schema").

Such conformance claims indicate that the associated Web service instance exhibits conformant behavior, as determined by the requirements associated with this attachment mechanism by the referenced profile.

A conformance claim attached to a wsdl:port element also indicates that it itself is a conformant XML construct. Additionally, the same claim is made for all elements recursively referenced by it.

Each profile specification includes a section that specifies a standardized mechanism to include Conformance claims within your Web service artifacts. For instance, the Basic Profile 1.0 specification includes details on its conformance claims at:

http://www.ws-i.org/Profiles/BasicProfile-1.0-2004-04-16.html#conformance

Example 11-1 shows WSDL using a WS-I Basic Profile 1.0 conformance claim. Notice the highlighted sections.

Example 11-1 WSDL with a WS-I Conformance claim

```
wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl"
  xmlns:tns="http://example.org/myservice"
  xmlns:soapbind="http://schemas.xmlsoap.org/wsdl/soap"
  xmlns:wsi="http://ws-i.org/schemas/conformanceClaim/"
  targetNamespace="http://example.org/myservice">
  <wsdl:portType name="MyPortType">
      ...
  </wsdl:portType>
  <wsdl:binding name="MyBinding" portType="MyPortType" >
      ...
  </wsdl:binding>
  <wsdl:service name="MyService" >
    <wsdl:port name="MyPort" binding="tns:MyBinding" >
      <wsdl:documentation>
        <wsi:Claim conformsTo="http://ws-i.org/profiles/basic/1.0" />
      </wsdl:documentation>
      <soapbind:address
       location="http://example.org/myservice/myport" />
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

11.2.3 Sample application

WS-I delivers sample applications that demonstrate compliance to the WS-I guidelines. The sample applications implement a common business scenario, such as supply-chain management, and have been developed using multiple platforms, programming languages, and tools. The samples also demonstrate interoperability in action and provide readily usable resources for the Web services developer.

Supply Chain Management (SCM)

The main objectives of the WS-I in developing a sample application are to:

- Demonstrate the wire-level interoperability of messages between applications, developed on platforms from multiple vendors that each conform to the Basic Security Profile
- Discover practical implementation problems associated with developing applications that conform to the Basic Security Profile. These problems can then be provided to the BSP team to assist them in revising and improving the Basic Security Profile

These limited objectives mean that this architecture focuses on showing how Web services that adhere to the WS-I Basic Profile Version 1.1 and the Basic Security Profile 1.0 might be used, rather than demonstrating Web services security best practices or details of a supply chain management application.

The SCM Sample Application architecture is based on use cases for a retailer selling consumer electronics. The retailer manages stock in three warehouses. If Warehouse A cannot fulfill an order, the retailer checks Warehouse B; if Warehouse B cannot fulfill the order, the retailer checks Warehouse C. When a warehouse's inventory of a particular product falls below a defined threshold, the warehouse orders more units from the manufacturer. There are three different manufacturers: manufacturers A, B, and C. Any Warehouse can place an order with any manufacturer.

The architecture consists of three system types, as follows:

1. Web Client Application. A Web-based application that provides an HTML interface. The Web client application is used to choose how Web service providers are obtained, which service providers to obtain, and to order items from the retailer. The Web Client Application also includes the Configurator Web service, which can be used to select Web service implementations, and the Logging Facility Web service.
2. Retailer System. A system that consists of the Retailer Web service and three instances of the Warehouse Web service.

3. Manufacturer System. A system that consist of three Web service instances, one for each manufacturer.

Figure 11-3 provides a graphical view of the functional relationships between the three system types.

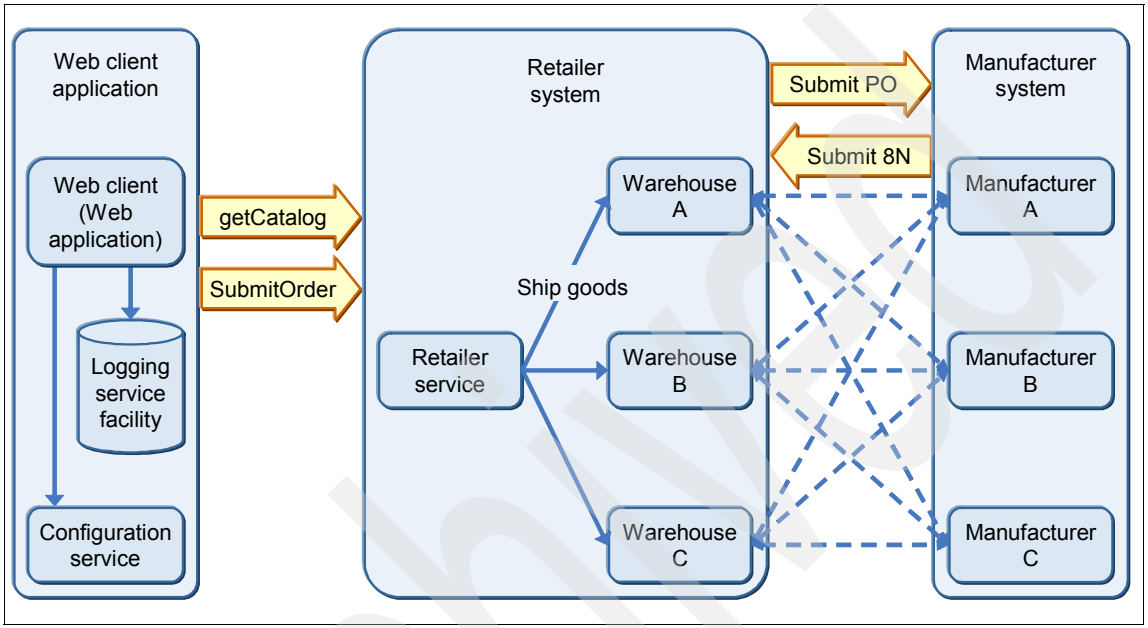


Figure 11-3 WS-I Sample application: functional overview

To date, WS-I has delivered 11 different vendor implementations of the WS-I Sample Application for the Basic Profile 1.1. Additionally, six new implementations from similar vendors are available in a working draft state for Basic Secure Profile 1.0.

Further details and current progress on the WS-I sample application can be obtained from the WS-I Web site.

11.2.4 Testing tools

Testing tools are provided by WS-I to establish conformance with the WS-I guidelines for the messages exchanged between Web services. The tools monitor and analyze messages to determine interoperability issues. The tools are available to developers to run and are aimed at ensuring that their implementations comply with current interoperability guidelines for use with other Web services specifications.

With the help of these tools, usage and implementation errors can be mitigated at a very early stage, thus improving the prospects of interoperability between applications and across platforms.

To date, WS-I has provided tools to verify compliance with the following profiles:

- Basic Profile 1.0
- Basic Profile 1.1
- Attachments Profile 1.0
- Simple SOAP binding profile 1.0

Work for tools to support other WS-I profiles is underway and currently includes a working draft tool for the Basic Secure Profile 1.0. See Figure 11-4.

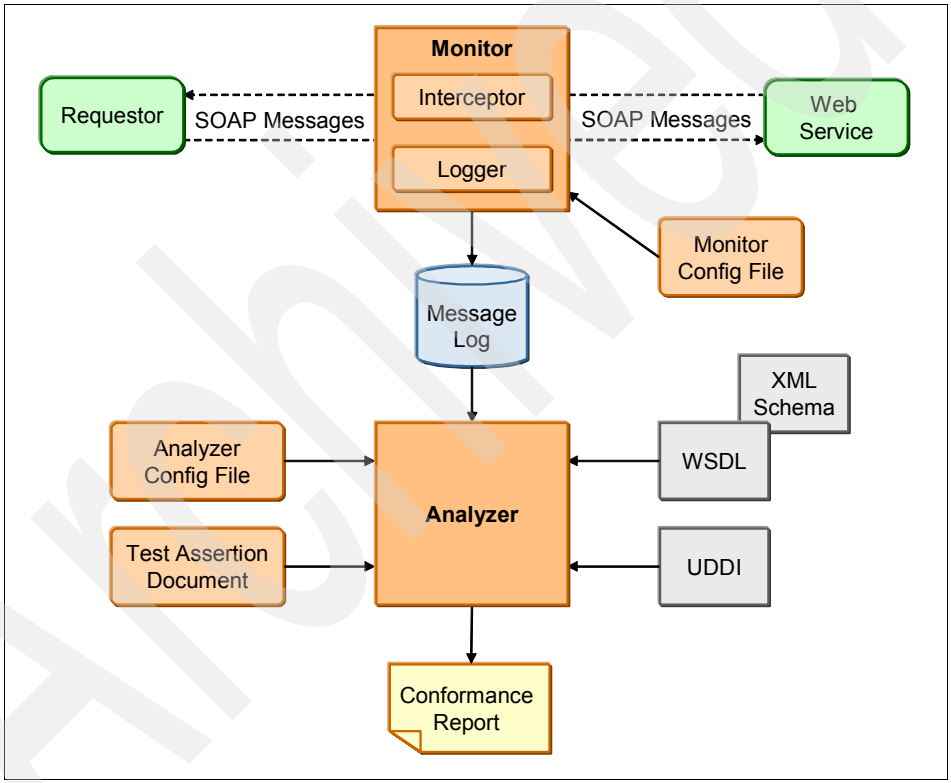


Figure 11-4 WS-I Testing tools architecture

You can download the tools from the WS-I Web site:

http://www.ws-i.org

11.3 Interoperability in Rational Application Developer

The Feature pack for Web services provides mechanisms to assist developers building application that adhere to the WS-I standards, or profiles. Applications built using the Feature pack for Web services can be checked for WS-I compliance by Rational Application Developer automatically, meaning there is a greater assurance of Web Services Interoperability with other platforms and specifications. But bear in mind that a good deal of the functionality in the Feature pack for Web services is ahead of Web Services Interoperability organization profile finalization, and by definition, applications using advanced features are not Web Services Interoperability organization compliant.

Currently the Feature pack for Web services supports five WS-I profiles:

1. Basic Profile (BP)
2. Simple SOAP Binding Profile (SSBP)
3. Attachments Profile (AP)
4. Basic Secure Profile (BSP)
5. Reliable Secure Profile (RSP)

Support for newer WS-I Profiles is an iterative process. You can expect support for most recent WS-I Profiles in later WebSphere Application Server releases shortly after the WS-I Profile specifications are finalized.

Applying WS-I conformance claims to Web service applications and, therefore developing interoperable applications is automated using the Rational Application Developer.

11.3.1 Setting the compliance level

RAD is configured to require, suggest (or warn about non-compliance), or ignore compliance to the WS-I Basic Profile V1.1, Simple SOAP Binding Profile (SSBP) V1.0, and WS-I Attachments Profile (AP) V1.0. Figure 11-5 shows the preferences at the workspace level.

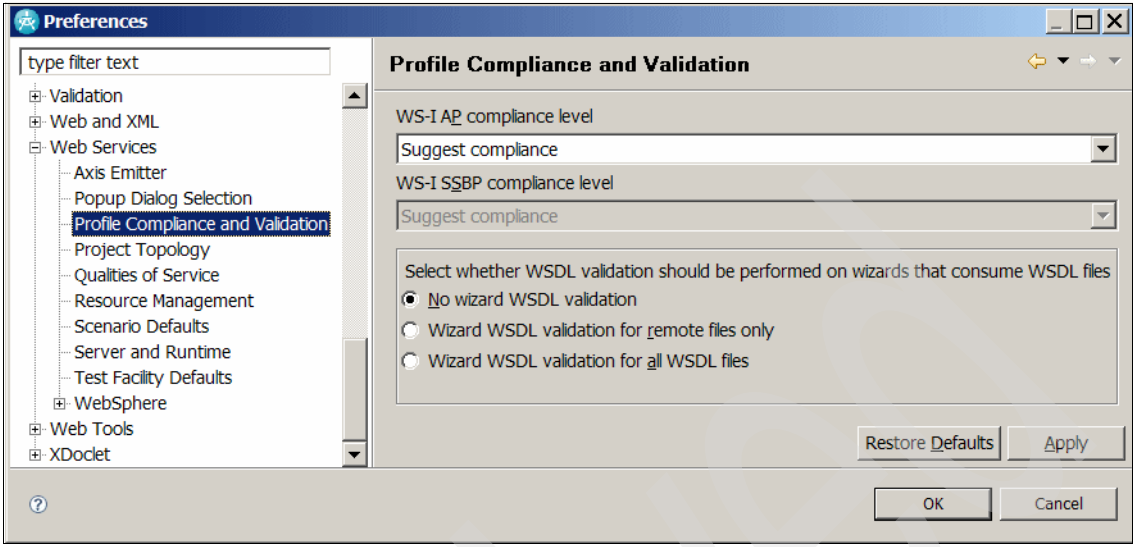


Figure 11-5 WS-I compliance levels preferences: Workspace

These preferences determine the behavior of the WS-I validation tools built into Application Server Toolkit:

- **Require:** The tool does not allow generating a non-compliant Web service.
- **Suggest:** The tool warns when generating a non-compliant Web service, but allows the user to progress.
- **Ignore:** The tool does not warn when generating a non-compliant Web service.

The Simple SOAP Binding Profile compliance level is available if the WS-I Attachment Profile is set to *Ignore*, otherwise it is set to the same level as the Attachment Profile compliance level. The second section specifies whether WSDL files should be validated.

These settings are also specified at the individual *project* level by selecting a project and *Properties* (context). The project settings override the workspace settings. The default for project settings is *Follow Preference*, that is, follow the workspace settings (Figure 11-6).

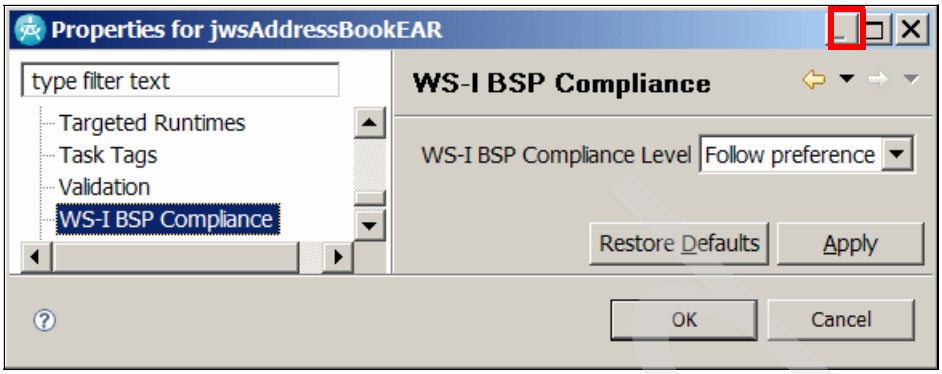


Figure 11-6 WS-I compliance levels preferences: Project

11.3.2 WSDL validator

WSDL files that are created, or are imported into Rational Application Developer, are validated against the W3C Web Services Description Language (WSDL) V1.1 specifications and against the WS-I profiles. WSDL that is generated by Rational Application Developer should be valid, but not necessarily WS-I compliant, depending on your selections (for example, using RPC/encoded or SOAP over JMS bindings is non-compliant).

To validate a WSDL file, select the file and *Run Validation* (context). Validation succeeds or fails, and a pop-up message window opens. Validate is automatic in Web projects. Validation errors appear in the Problems view.

11.3.3 WS-I message validation

To validate a Web service against Web Services Interoperability organization profiles, perform these steps:

- Set up the TCP/IP Monitor and route the Web services calls to the monitor (for example, using port 9089).
- Invoke a method in your Web service sample application to generate traffic through the TCP/IP Monitor.
- To ensure that your Web service is WS-I compliant, generate a log file by clicking the icon (Figure 11-7).

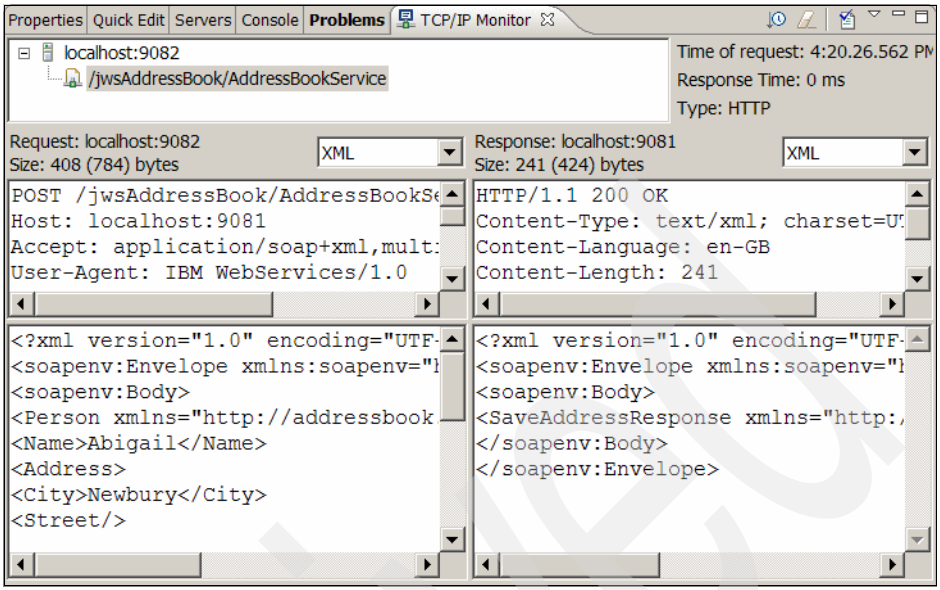


Figure 11-7 Validating SOAP/HTTP messages

- In the Validate WS-I Message Log File dialog, type a name for the log file and select a folder to store it (Figure 11-8) → **Finish**.

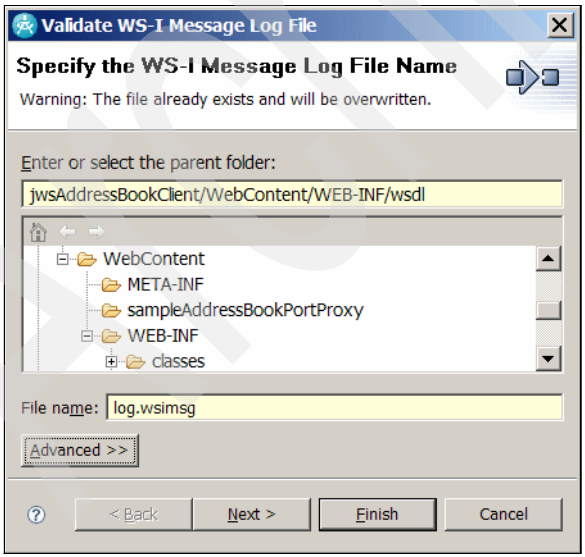


Figure 11-8 SOAP/HTTP message validation result

11.4 WebSphere Interoperability

Table 11-2 shows a comprehensive list of features from earlier WebSphere Web services products that do or do not work with the features incorporated in the Feature pack for Web services.

Table 11-2 Back-level interoperability support

WebSphere Application Server	WebSphere Application Server features	Interoperability with Feature Pack for Web services
WebSphere 5.0	IBM SOAP (based on Apache SOAP)	✓
WebSphere 5.0.2 WebSphere 5.1	WS-I Basic Profile 1.0	✓
	JAX-RPC 1.0	
	JSR-109 1.0	
	SAAJ 1.1	
	UDDI 2.0	
	WS-Security (OASIS Draft 13)	X
	SOAP/JMS	
WebSphere 6.0	WS-I Basic Profile 1.1	✓
	JAX-RPC 1.1	
	JSR109 1.1	
	SAAJ 1.2	
	UDDI 3.0	
	WS-Security 1.0	
	WS-Addressing 1.0	
	WS-Coordination v1.0	
	WS-AtomicTransactions v1.0	
	JAXR	
	Multiple protocol/encodings (SOAP/JMS, SOAP/RMI)	X

WebSphere Application Server	WebSphere Application Server features	Interoperability with Feature Pack for Web services
WebSphere 6.1 (Additional to WebSphere 6.0)	WS-Addressing	✓
	WS-Resource	X
	WS-Business Activity	
	WS-Notification	

Any previous WebSphere releases with features not accounted for in this table are not interoperable with the Feature pack for Web services.

11.5 Interoperability sample: Insurance Underwriter

In this chapter we walk you through building a sample — Insurance Underwriter with an emphasis on Interoperability and distributed deployment. We provide detailed instructions how to configure WebSphere Application Server to interoperate with Windows Communication Foundation using secure, and reliable Web services. We aim to provide enough information about how to deploy and configure Windows Communication Foundation for you to be able to get the example to run, and if you have no familiarity with Windows Communication Foundation, to give you enough information and examples to understand how the example works.

After presenting an overview of the business, we describe the business and technical objectives of the application in the following sections:

- 11.6, “Deploy applications” on page 429
- 11.7, “Configure Policy sets” on page 443
- 11.8, “Attach Policy sets and bindings to applications” on page 454
- 11.9, “Configure resources” on page 464
- 11.10, “Configure bindings” on page 468

11.5.1 Business overview

The Insurance Underwriter sample models a scaled-down version of a real-world insurance underwriter business process, shown in Figure 11-9. It has been adapted to specifically highlight some of the new features in the Feature pack for Web services.

In this insurance underwriter business scenario, an insurance agent acts as the broker for the customer to create a new insurance policy. The agent collects the customer's information and sends an electronic request to the insurance underwriter company's processing center. After receiving the request from the agent, the processing center sends a background check request to the Medical and Credit fulfillment systems. The fulfillment systems reply with an approve, or denied (*true/false*) response. Then the processing center takes the appropriate action by deciding whether, or not to create the insurance policy.

Once the customer's insurance policy is available, the customer's agent or a trusted business partner retrieves the customer's insurance policy based on the customer's social security number.

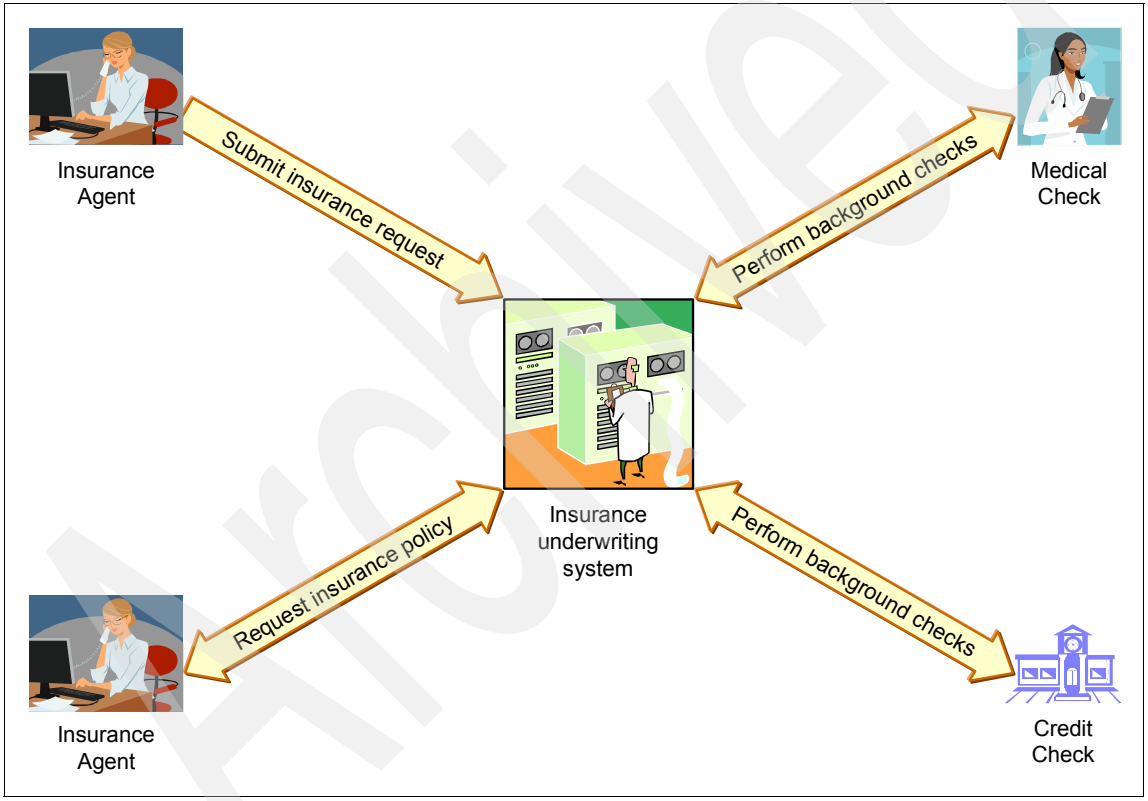


Figure 11-9 Insurance underwriter business scenario

11.5.2 Technical overview

The Insurance Underwriter sample application is designed to demonstrate most of the new features in the Feature pack for Web services. We highlight these new technical features as we describe the functional flow of the insurance underwriter business process. The steps are numbered to correspond with Figure 11-10.

1. The Insurance agent sends a one-way Web service request to the Underwriter system, on behalf of the customer, to create a new insurance policy for the customer.

   The request parameters are data to identify the customer, along with an attachment containing the customer's medical examination results. MTOM/XOP enables the Web service request to include the attachment to deliver it to the destination. The communication between the insurance agent and the underwriter system has assured message delivery by applying the WS-ReliableMessaging policy sets to the Underwriter client and Underwriter service.

2. The Underwriter service sends two concurrent asynchronous requests to the credit check and medical check systems for a background check on the customer.

   The customer information retrieved from the agent's request in step 1, is used to populate the parameters for the requests to the fulfillment systems.

3. The underwriter service polls for the responses for up to 30 seconds. The underwriter requests an insurance policy only when the boolean responses from both services are true. A 14 digit random number is generated and added to a static Hash table with the customer's social security number as the key.

4. Once the insurance policy is created by the Underwriter service, it updates its PolicyStore service with the latest Hash table key/value pair.

5. The customer's agent or the Underwriting company's business partner accesses the customer's insurance policy by connecting using the PolicyStore service thin client packaged within the PolicyStoreThinClient.jar.

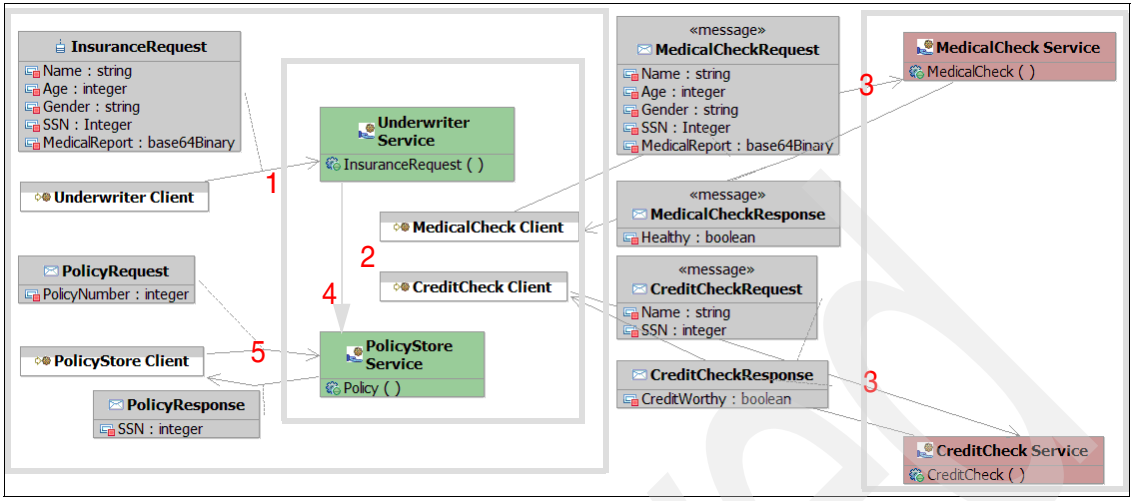


Figure 11-10 Technical Overview

11.5.3 Qualities of service

Figure 11-11 shows the qualities of service and message exchange patterns associated with each of the interactions.

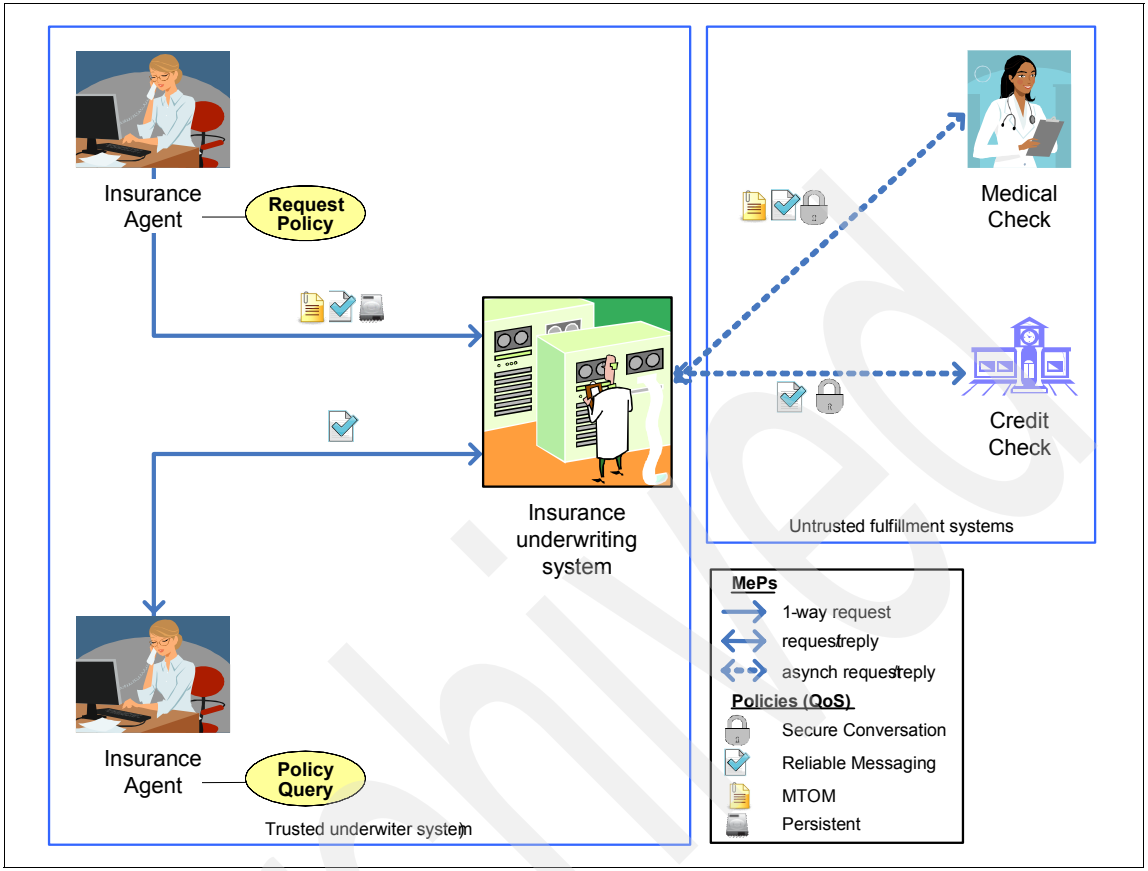


Figure 11-11 Required qualities of service and message exchange patterns

11.5.4 Roles

Recalling 4.3, “Externalization of application QoS” on page 60, particularly Figure 4-4 on page 60, we use the Administrator Console to develop policies and bindings, and resources such as the message bus needed to run the policies. These tasks are performed by security and integration specialists. The job of applying the policies to particular clients and services can be performed by the developer or the integration specialist, using either Rational Application Developer or the Administrator Console.

We chose to develop and deploy the insurance applications using Rational Application Developer, and follow the development approach of importing Policy sets from WebSphere Application Server and the policies and bindings using Rational Application Developer.

The alternative would be to export the .ear files from Rational Application Developer and deploy them directly on WebSphere Application Server using the Administrator Console. We would then apply the policies and bindings using the Administrator Console.

In the Windows Communication Foundation environment, we develop and deploy the applications using Microsoft Visual Studio 2006, and configure the clients and service using the Service Configuration Editor.

> **Note:** Unlike policies, until WebSphere Application Server v7.0 there is no way to export and custom bindings, and then import them into Rational Application Developer. We work around this problem, to keep the example simple, by modifying the default bindings, where we can.

11.5.5 Organization of configuration

The configuration is extensive (Figure 11-12) and takes more than a day to perform. Nonetheless, we thought it useful to provide a step by step guide, as a way to draw some of the strands in the preceding chapter into a single practical example, and also as a tutorial should you have to learn how to perform configuration of reasonably complex policies and bindings. It can also give you some ideas about how to go about organizing configuration tasks.

The configuration of the sample is organized “breadth first”, moving from one kind of configuration to the next, and configuring all the items of each type. So we deploy all the applications, configure the Policy sets, attach the Policy sets and define the bindings, configure the bindings, and so on. The major tasks are listed by heading number below, and within each task there is a further list of sub-tasks. We think this is a good and systematic way to organize configuration that should reduce configuration errors.

The major tasks are as follows:

- 11.6, “Deploy applications” on page 429
- 11.7, “Configure Policy sets” on page 443
- 11.8, “Attach Policy sets and bindings to applications” on page 454
- 11.9, “Configure resources” on page 464
- 11.10, “Configure bindings” on page 468
- 11.11, “Validate WebSphere Application Server sample” on page 517
- 11.12, “Configure and validate Windows sample” on page 525
- 11.13, “Distribute and test the Insurance sample” on page 529.

We also provide a Quickstart configuration based on unzipping our configuration and then doing a few steps of manual installation to make the configuration work. If you want to try out the sample without a lot of configuration, go directly to 11.11, “Validate WebSphere Application Server sample” on page 517, skipping the configuration sections.

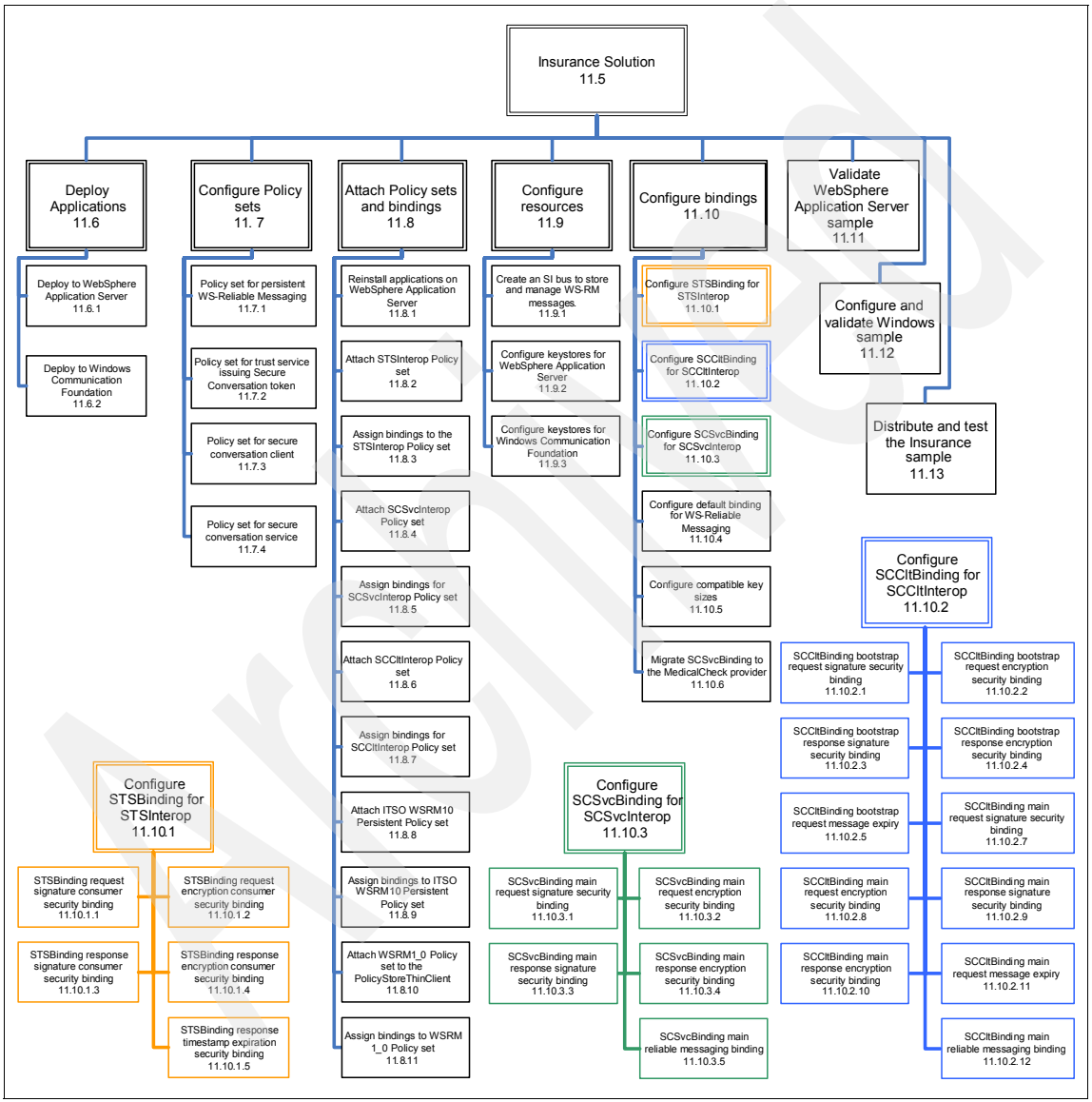


Figure 11-12 Roadmap to configuration of Insurance sample

11.6 Deploy applications

Next, we deploy the applications and verify they work correctly with the default qualities of service:

- 11.6.1, “Deploy application to WebSphere Application Server”
- 11.6.2, “Deploy application to Windows Communication Foundation” on page 434

11.6.1 Deploy application to WebSphere Application Server

In this section, we provide steps to deploy the insurance application on a WebSphere Application Server and verify the installation using the default QoS.

You should start with a fresh workspace in Rational Application Developer, and an installed and running WebSphere Application Server with the Feature pack for Web services. Because we are modifying the default Policy set bindings in this configuration, we recommend that you uninstall any applications on WebSphere Application Server that are using the Feature pack for Web services, including applications developed in this book.

- ☐ **File** → **Import** → **Project Interchange** → **next** → **Browse** to `insurance.zip` in the additional materials → **Select All** → **Finish**.

  There should be only two warnings when the workspace has refreshed.

  > **Note:** If you import .ear files rather than Project Interchange, then you have to restore the Feature pack for Web services project facet using the project preferences to solve numerous compile errors.

- ☐ Right-click the server in the Servers view **Add and remove projects** → **Add all** → **Finish**.

- Open the Insurance Agents application in a Web browser (Figure 11-13). Modify the host name and port if necessary, of course.

  http://localhost:9081/InsuranceAgents/sampleUnderWriterSOAPProxy/TestClient.jsp

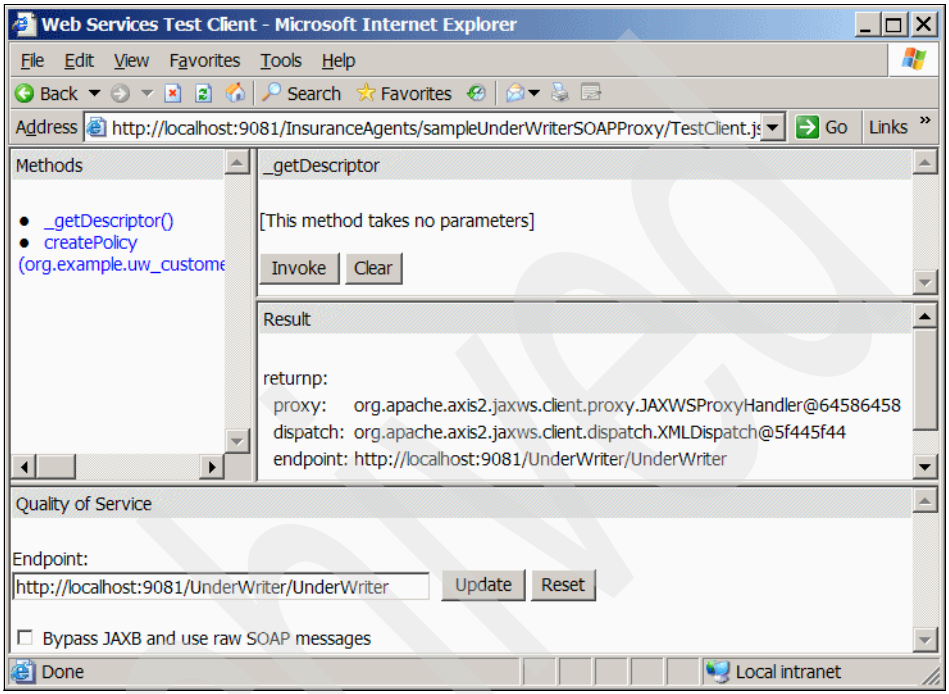


Figure 11-13 InsuranceAgent.jsp

The insurance agent can configure the URL to the Underwriter system endpoint in the **TestClient.jsp** form's *Configuration* panel, as shown in Figure 11-15. The agent can also configure the endpoints to the other services by invoking this URL (Figure 11-14).

`http://localhost:9081/UnderWriter/UpdateEndpoint.html`

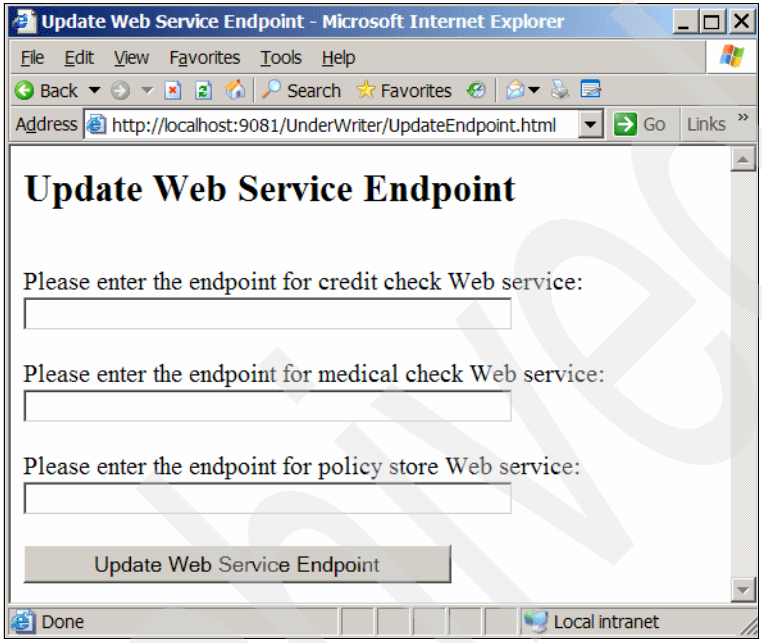


Figure 11-14 Update endpoints of the other services

Note: It is critical that the Underwriter service URL and the PolicyStore service URLs' base addresses match. These services are tightly coupled *(packaged in the same EAR — Underwriter.ear)* so they cannot be distributed.

- Click **createPolicy** (Figure 11-15 on page 432) → Type in the customerInfo and attach their medical examination results *(any *.pdf file will do)* → **Invoke**.

Tip: The sample only needs the name, ssn and attachment. If you type 100 as the ssn, it corresponds to the default ssn used in the PolicyQuery, and simplifies verification.

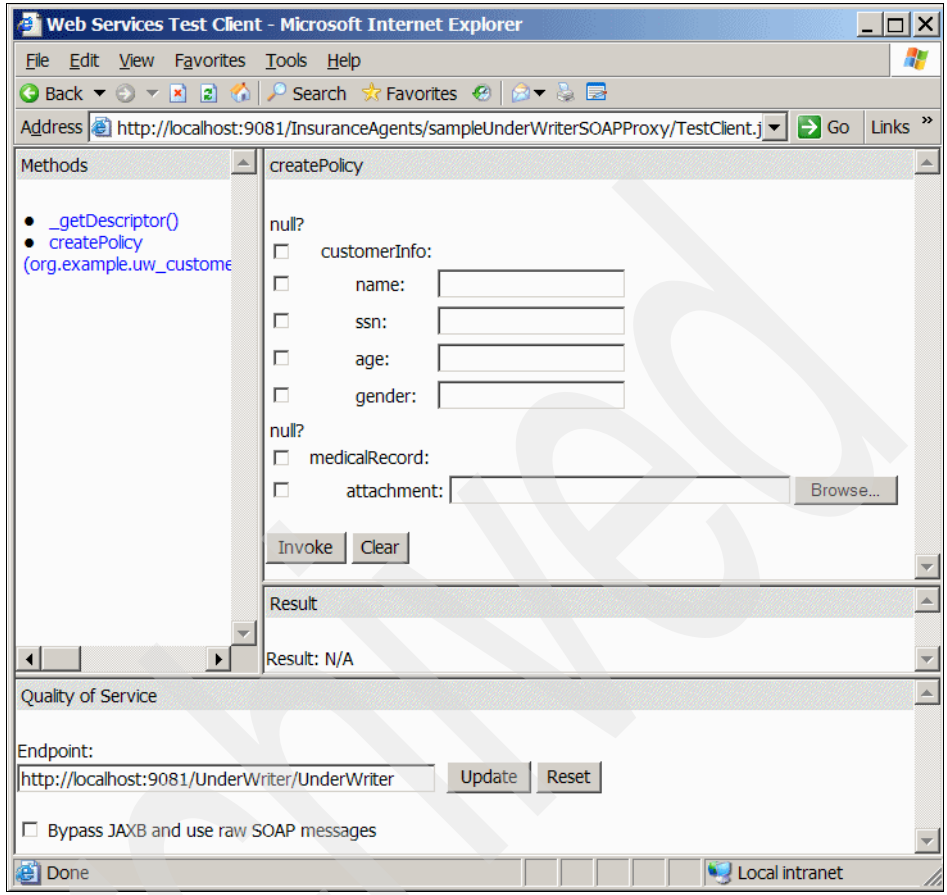


Figure 11-15 createPolicy

Because this is a one-way request, you do not see any response messages. Monitor the WSFP server's SystemOut.log in the console (Example 11-2) to view application messages or exceptions. In this case, no policy was written.

Example 11-2 services responses to Underwriter - invalid SSN

```
[30/04/08 20:36:37:781 BST] 00000070 SystemOut     O The endpoint for credit check
Web service is http://localhost:9081/CreditCheck/CreditCheck
[30/04/08 20:36:38:046 BST] 00000027 SystemOut     O Credit check result is true
[30/04/08 20:36:38:890 BST] 00000070 SystemOut     O The endpoint for medical check
Web service is http://localhost:9081/MedicalCheck/MedicalCheck
[30/04/08 20:36:39:062 BST] 00000027 SystemOut     O Medical check result is false
[30/04/08 20:36:39:906 BST] 00000070 SystemOut     O Policy was denied due to
customer background check failure.
```

- Resubmit the request until two TRUEs are generated. The response is shown in Example 11-3 with a generated policy number. We retrieve the policy in the next step

Example 11-3 Services responses to underwriter - valid SSN

```
[30/04/08 20:40:59:093 BST] 00000070 SystemOut     O The endpoint for credit check
Web service is http://localhost:9081/CreditCheck/CreditCheck
[30/04/08 20:40:59:093 BST] 00000026 SystemOut     O Credit check result is true
[30/04/08 20:41:00:093 BST] 00000070 SystemOut     O The endpoint for medical check
Web service is http://localhost:9081/MedicalCheck/MedicalCheck
[30/04/08 20:41:00:093 BST] 00000026 SystemOut     O Medical check result is true
[30/04/08 20:41:01:093 BST] 00000070 SystemOut     O Generated policy number is
1933414213139622176
```

- Locate PolicyStoreThinJavaClient.java → Run as... → Run and create a new Java Application to store the parameters required (Figure 11-16). The results are shown in Example 11-4.

> **Note:** If you specified ssn = 100 in CustomerInfo, then you do not need any parameters for PolicyStoreThinJavaClient.

Example 11-4 Running PolicyStoreThinClient

```
Social security number is 200
Policy store URL http://localhost:9081/UnderWriter/PolicyStoreService
Policy store QName http://www.example.org/PolicyStore/
Policy store Service PolicyStoreService
01-May-2008 09:58:56 com.ibm.ws.ssl.config.SSLConfigManager
INFO: ssl.disable.url.hostname.verification.CWPKI0027I
The policy number is 7278566345373868670
```

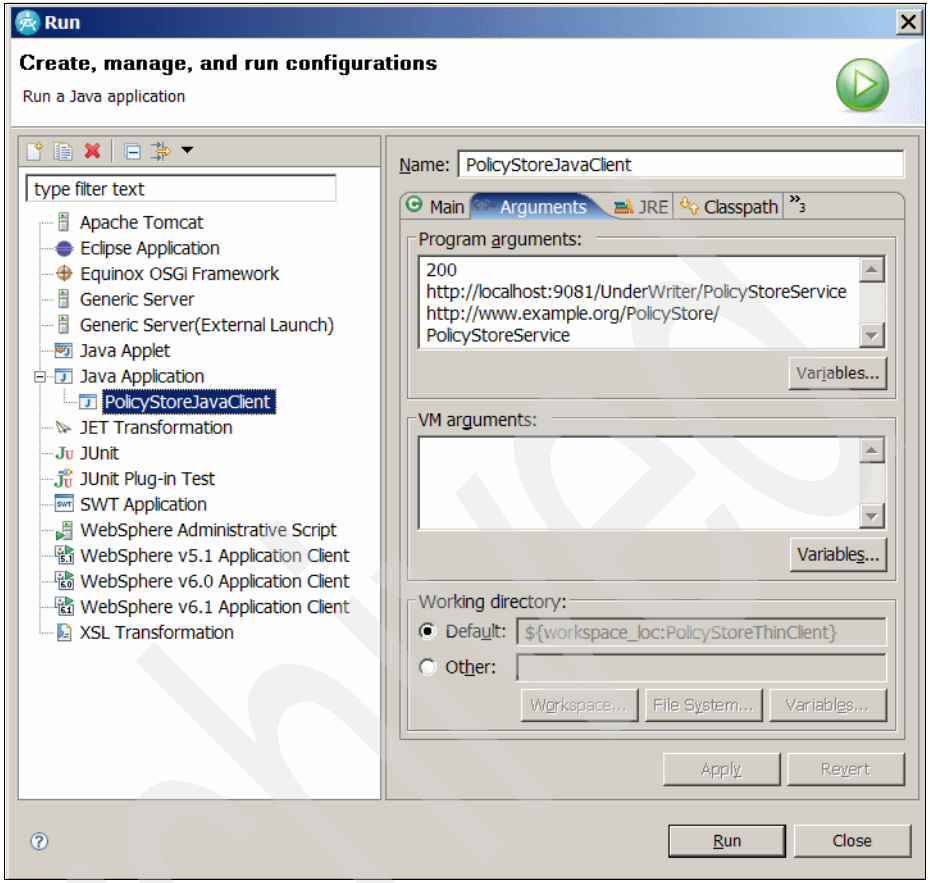


Figure 11-16 Run PolicyStoreThinClient in Rational Application Developer

11.6.2 Deploy application to Windows Communication Foundation

In this section, we provide steps to deploy the insurance application on a Windows Communication Foundation and verify the installation using the default QoS.

We are using Microsoft Visual Studio 2008 (Professional Edition). If you do not have a licensed copy, you should be able to work with the examples by downloading the Windows SDK for Windows Server 2008 and .NET Framework 3.5 from:

http://www.microsoft.com/downloads/details.aspx?FamilyId=E6E1C3DF-A74F-4207-8586-711EBE331CDC&displaylang=en

The SDK comes with tools as well as other features that allow you to edit binding configurations, view traces, and generate Web service artifacts. After installing the SDK, reboot your machine.

Attention: In installing the Microsoft products, or the WCF version of the insurance agent, you are likely to find that the O/S JVM reverts to the Microsoft JVM rather than the JRE™ 1.5 you have to use with Rational Application Developer v.7 and perhaps other software you have installed on your system. You will see Rational Application Developer terminating with a pop-up from Eclipse. There are three possible fixes:

1. Add the parameter -vm <path to your 1.5 JRE>\javaw to the shortcut to start Rational Application Developer, for example:

   ```
   C:\IBM\eclipse\eclipse.exe -vm C:\IBM\Java50\bin\javaw -product
   com.ibm.rational.rsa.product.ide
   ```

2. Modify the system PATH environment variable to include the path to your 1.5 JRE, for example:

   ```
   C:\IBM\Java50;C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\System32\
   Wbem;C:\IBM\SQLLIB\BIN;C:\IBM\SQLLIB\FUNCTION;C:\IBM\SQLLIB\SAMPL
   ES\REPL;C:\Program Files\Microsoft SQL
   Server\80\Tools\Binn\;C:\Program Files\Microsoft SQL
   Server\90\Tools\binn\;
   ```

3. If your version of Windows has the option, go to **Start** → **Set Program Access and Defaults** → **Custom** → **Choose a default virtual machine for Java** and choose an alternative to the Microsoft JVM to use.

☐ Run setup.exe in the Additional Materials WCF folder.

By default, all the files are installed under `C:\Program Files\IBM\SG247618 – Insurance Underwriter Sample`. If the machine is missing the .Net 3.5 runtime environment, the installer automatically prompts the user to download and install it. The machine should be restarted after installing .Net 3.5.

The additional materials also include all the source projects. If you have Visual Studio 2000 open InsuranceSolution.sln in the WFC directory to inspect the code.

Configure and run services

For the purposes of this chapter, we are running our services as managed applications. See the MSDN® article found at:

http://msdn.microsoft.com/en-us/library/ms731758.aspx

Each of the services has a RunXXService.cs component which contains the Main entry point to start the application hosting the service. The application appears as a `.exe` file and a `.exe.config` file in the installation root.

- ☐ Open the installation root of the installed insurance solution (Figure 11-17).

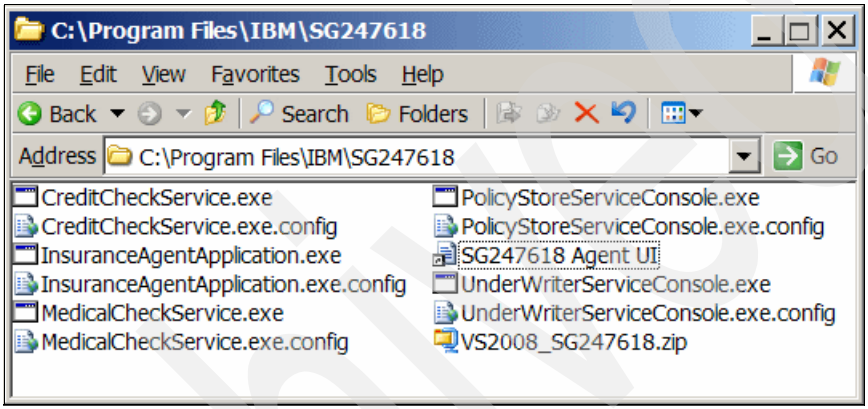


Figure 11-17 installation root of WCF insurance solution

As well as the .exe and .exe.config files, the installation also includes a zip file containing all the source code, and a short cut to the InsuranceAgent client.

The .exe.config file contains the service configuration, including the address. There is a special Service Configuration Editor to view and modify the configuration files, as well as to create new services and clients.

- ☐ Click **Start** → **All Programs** → **Microsoft Windows SDK v6.0A** → **Tools** → **Service Configuration Editor** → **File** → **Open** → **Config file** → Browse to the Insurance sample install root → **Open** UnderWriterServiceConsole.exe.config.
- ☐ Click on Client → Endpoints → CreditCheckEP to view the client binding (Figure 11-18)[1] → **Close**.

[1] The contents of the binding you see might not be identical to the screen capture.

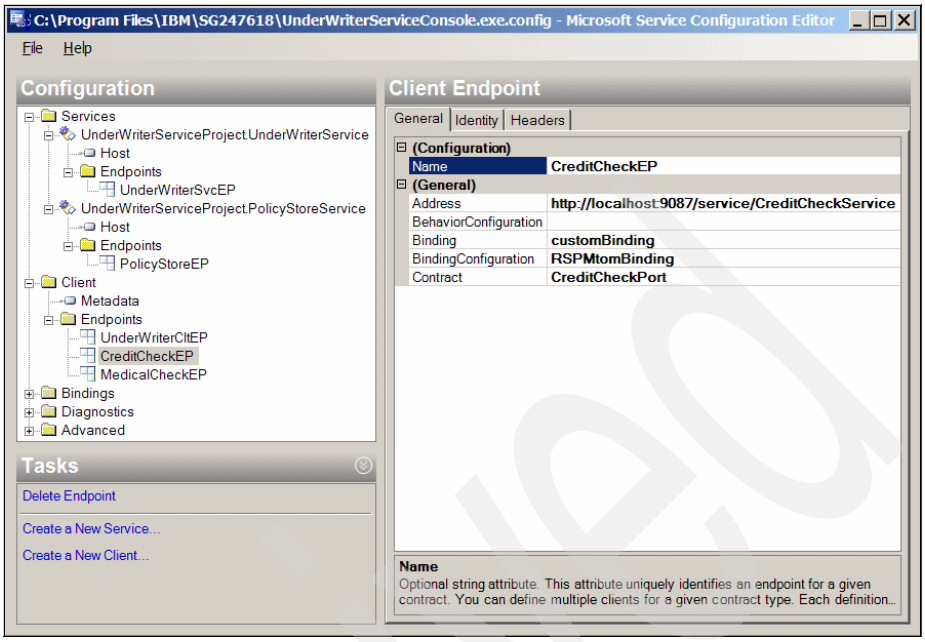


Figure 11-18 Service Configuration Editor

- ☐ Create a directory `Secure Bindings` → **Backup** all the .config files into it.
- ☐ Create a directory `Basic Bindings` → **Unzip** BasicConfigBindings.zip from the WCF directory into it.
- ☐ Replace all the .config files in the install directory with the bindings from `Basic Bindings`

  The address where the Underwriter expects to find the CreditCheck service is:

  `http://localhost:9087/service/CreditCheckService`

- ☐ Run CreditCheckService.exe (Figure 11-19).

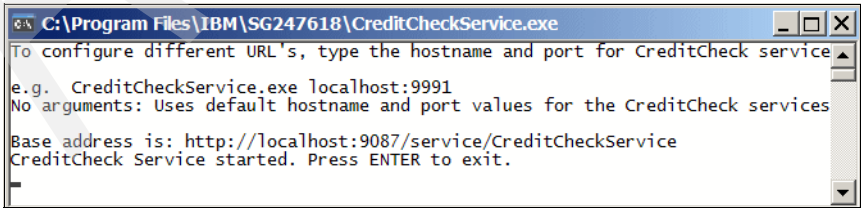


Figure 11-19 Started the CreditCheckService

WCF Service implementation

Skip to “Verify the Insurance Solution” on page 440 if you are not concerned to read about WCF service implementation.

The service address shown in the console must match the address that the Underwriter Service is expecting to use in its CreditCheck client. The runCreditCheck.cs component takes an alternative address as input if you have to change the service address. You must then change the client address to match using the Service Configuration Editor. The code fragment in Example 11-5 (console output removed for brevity) shows how the **CreditCheckService** service is hosted at localhost:9087 by default.

Example 11-5 Hosting CreditCheckService

```
namespace CreditCheckServiceProject {
   public class RunCCService {
      public static void Main(string[] args) {
      string hostport;
      if (args.Length < 1) {
         hostport = "localhost:9087"; }
      else {
         hostport = args[0]; }
      ServiceHost host = new ServiceHost(typeof(CreditCheckService),
         new Uri("http://"+hostport+"/service/CreditCheckService"));
...
      host.Open()
...
```

The methods (without console output) in the CreditCheckService.cs implementation are shown in Example 11-6.

Example 11-6 Fragment of CreditCheckService.cs

```
namespace CreditCheckServiceProject {
   public class CreditCheckService : CreditCheckPort {
      public EvaluateCreditRatingResponse
         EvaluateCreditRating(EvaluateCreditRatingRequest request) {
            EvaluateCreditRatingResponse resp =
               new EvaluateCreditRatingResponse();
            Random choice = new Random();
            resp.result = choice.NextDouble() < 0.2 ? false : true;
            return resp;
      }
...
```

The generated parts of the `EvaluateCreditRatingRequest` and `EvaluateCreditRatingResponse` objects defined in the schemas for the Web service are to be found in the CreditCheck.cs implementation. The generated code looks after the data mappings to and from c# and XML shown in Example 11-7 and Example 11-8.

Note: Partial classes might be a new concept to you. A partial class is a class definition that is split across multiple files, and typically used, as we do here, to separate generated from hand-coded statements.

Example 11-7 EvaluateCreditRatingRequest partial class

```
[System.CodeDom.Compiler.GeneratedCodeAttribute
   ("System.ServiceModel", "3.0.0.0")]
[System.ServiceModel.MessageContractAttribute(
   WrapperName = "EvaluateCreditRating",
   WrapperNamespace = "http://www.example.org/CreditCheck/",
   IsWrapped = true)]
public partial class EvaluateCreditRatingRequest {
   [System.ServiceModel.MessageBodyMemberAttribute(
      Namespace = "http://www.example.org/CreditCheck/", Order = 0)]
   [System.Xml.Serialization.XmlElementAttribute(
      Form = System.Xml.Schema.XmlSchemaForm.Unqualified)]
   public CC_CustomerInfoType customerInfo;
   public EvaluateCreditRatingRequest() { }
   public EvaluateCreditRatingRequest(CC_CustomerInfoType customerInfo)
      {
      this.customerInfo = customerInfo;
   }
}
```

The definition of CC_CustomerInfoType is also generated in the CreditCheck.cs file, but not listed here.

Example 11-8 EvaluateCreditRatingResponse partial class

```
[System.CodeDom.Compiler.GeneratedCodeAttribute
   ("System.ServiceModel", "3.0.0.0")]
[System.ServiceModel.MessageContractAttribute(
   WrapperName = "EvaluateCreditRatingResponse",
   WrapperNamespace = "http://www.example.org/CreditCheck/",
   IsWrapped = true)]
public partial class EvaluateCreditRatingResponse {
   [System.ServiceModel.MessageBodyMemberAttribute(
      Namespace = "http://www.example.org/CreditCheck/", Order = 0)]
        [System.Xml.Serialization.XmlElementAttribute(
         Form = System.Xml.Schema.XmlSchemaForm.Unqualified)]
      public bool result;
      public EvaluateCreditRatingResponse() { }
      public EvaluateCreditRatingResponse(bool result) {
      this.result = result;
   }
}
```

Verify the Insurance Solution

- ☐ Launch the other three services:
  - ☐ MedicalCheckService.exe
  - ☐ UnderwriterService.exe (contains both Underwriter and PolicyStore services)
- ☐ Launch the InsuranceAgent client using the .exe or the short cut (Figure 11-20 on page 441).

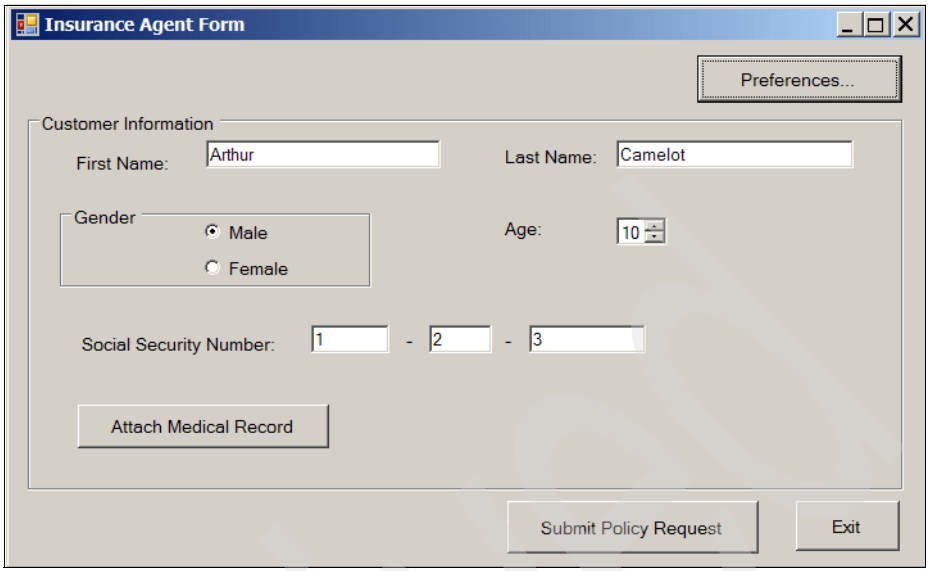


Figure 11-20 Insurance Agent client

Type in any values including attaching a .pdf for the medical record → **Preferences** (Figure 11-21) → **Apply** (the URLs should not have to be changed, but without doing this step, the initial values are not set correctly) → **Submit Policy Request**.

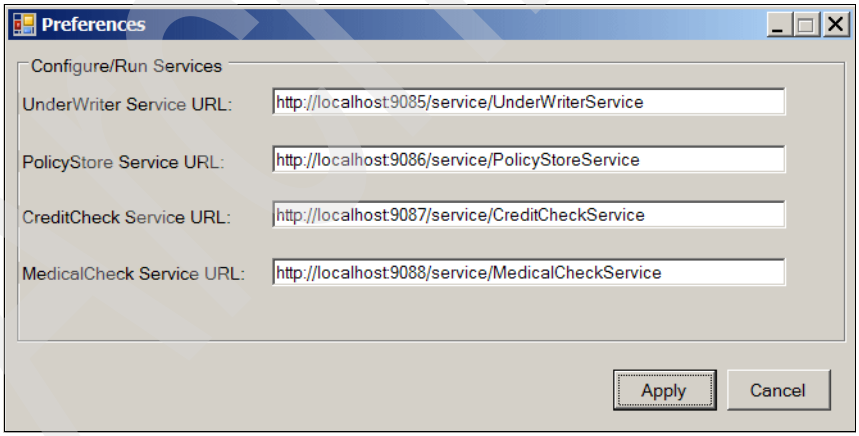


Figure 11-21 Apply URLs

- Click **OK** through the pop-up boxes. Finally, you see a pop-up like either Figure 11-22 or Figure 11-24.

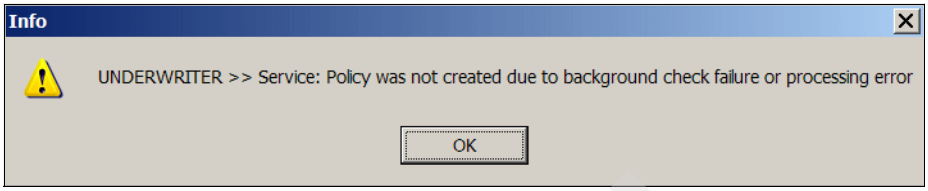


Figure 11-22 Failed to issue policy

Look at the console output for the Underwriter (Figure 11-23). It shows this client had a bad medical report.

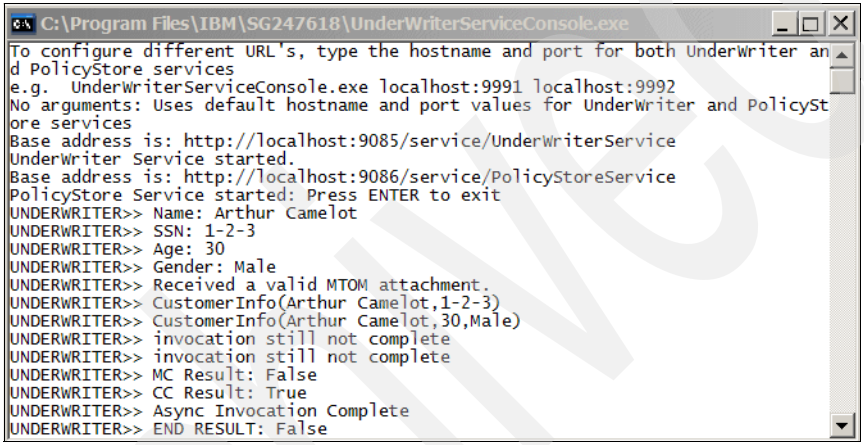


Figure 11-23 |Underwrite console output

Continue to request an insurance policy (changing just the SSN each time) until you get a pop-up as in Figure 11-24.

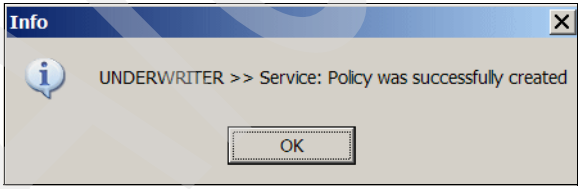


Figure 11-24 Successfully issued policy

- Once we have a policy issued, launch the PolicyStore client from a command window[2] → Type the SSN for the successful insurance client (Figure 11-25). Separate the three sets of digits in the SSN by dashes.

[2] The PolicyStoreClient console application closes directly it receives a response. Running it from a console window allows you to see the response.

```
C:\WINDOWS\system32\CMD.exe
C:\Program Files\IBM\SG247618>PolicyStoreServiceConsole
To configure different URL's, type the hostname and port for PolicyStore service
.
e.g.  PolicyStoreServiceConsole.exe http://localhost:9991/service/PolicyStoreSer
vice
No arguments: Uses default hostname and port values for the PolicyStore services

PolicyStoreService URL is: http://localhost:9086/service/PolicyStoreService
Enter Customer Social Security Number:
4-2-2
Insurance Policy number is: 26105570619043

C:\Program Files\IBM\SG247618>
```

Figure 11-25 Policy Store client

11.7 Configure Policy sets

We have to change a number of the default Policy sets to interoperate with Windows Communication Foundation.

Windows Communication Foundation supports WS-ReliableMessaging 1.0, and there is a WS-ReliableMessaging 1.0 policy provided with the Feature pack for Web services. However, we want to use managed persistent messaging in WebSphere Application Server so we customize the policy to provide persistent messaging.

There is an interoperability problem with secure conversation in the bootstrap message flow used to establish mutual trust in the Security Context Token (SCT) issued by the Security Token Service (STS). The signature confirmation response in the bootstrap procedure needs to be disabled to get around this problem.

There is another signature confirmation problem in the application message flow. The work around is to set Header Layout to lax when signature confirmation is enabled[3].

We also have to configure Windows Communication Foundation and the Feature pack for Web services to use the same key size for signing the derived key used in successive secure conversation transmissions. The default would be to use different sized keys.

[3] A fix for this problem is due to be shipped in an upcoming Fix Pack (> 6.1.0.13)

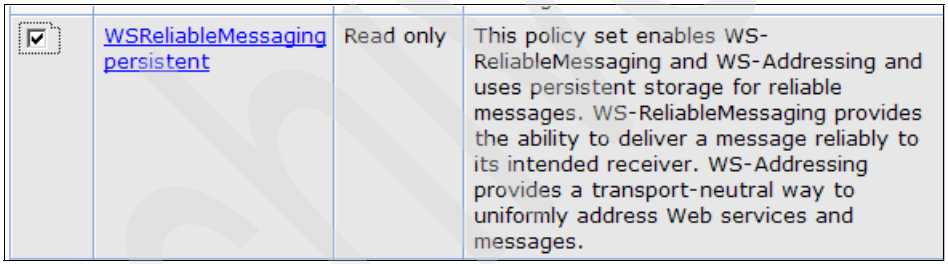
We configure the Policy sets in the following sections:

- 11.7.1, “Policy set for persistent WS-ReliableMessaging 1.0” on page 444
- 11.7.2, “Policy set for trust service issuing Secure Conversation token” on page 446
- 11.7.3, “Policy set for secure conversation client” on page 447
- 11.7.4, “Policy set for a secure conversation service” on page 453

11.7.1 Policy set for persistent WS-ReliableMessaging 1.0

We have to create a policy set based on WS-ReliableMessaging 1.0, but with persistence applied, we take the WS-ReliableMessaging 1.1 persistence set and change it to use the WS-ReliableMessaging 1.0 protocol.

- ☐ Log into the Administrator Console → **Services -> Policy Sets -> Application Policy Sets**.
- ☐ Check **WS-ReliableMessaging persistent** → **Copy** (Figure 11-26)



☑	WSReliableMessaging persistent	Read only	This policy set enables WS-ReliableMessaging and WS-Addressing and uses persistent storage for reliable messages. WS-ReliableMessaging provides the ability to deliver a message reliably to its intended receiver. WS-Addressing provides a transport-neutral way to uniformly address Web services and messages.

Figure 11-26 WS-ReliableMessaging 1.1 persistent policy

- ☐ Type in the name `ITSO WSRM 10 Persistent` → **OK** → **Save**.
- ☐ Click `ITSO WSRM 10 Persistent` → **WS-ReliableMessaging**.

- Select **WS-ReliableMessaging 1.0** from the **Standard** drop-down → Uncheck **MakeConnection** and **In-order** → select Managed persistent → **Apply** → **Save** (Figure 11-27).

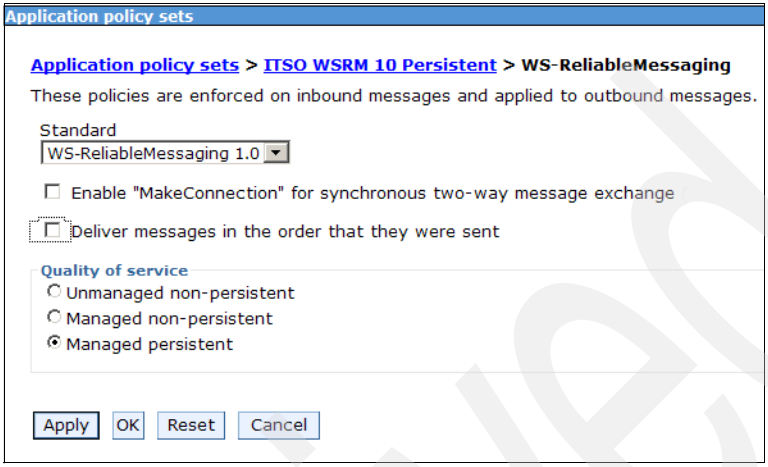


Figure 11-27 Configure WS-ReliableMessaging

11.7.2 Policy set for trust service issuing Secure Conversation token

To disable signature confirmation when creating the Secure Conversation token, we have to alter the Policy set used to issue and renew the token. First we copy the existing Policy set and customize it, then we replace the Policy set used by the Trust service (the STS) to issue and renew tokens.

- In the Administrator Console, navigate to **Services** → **Policy sets** → **System Policy sets** → Check `TrustServiceSecurityDefault` → **Copy** → Name the new Policy set `STSInterop` → Check the box **Attach this Policy set...** → Select **Copy bindings** (Figure 11-28) → **OK** → **Save**.

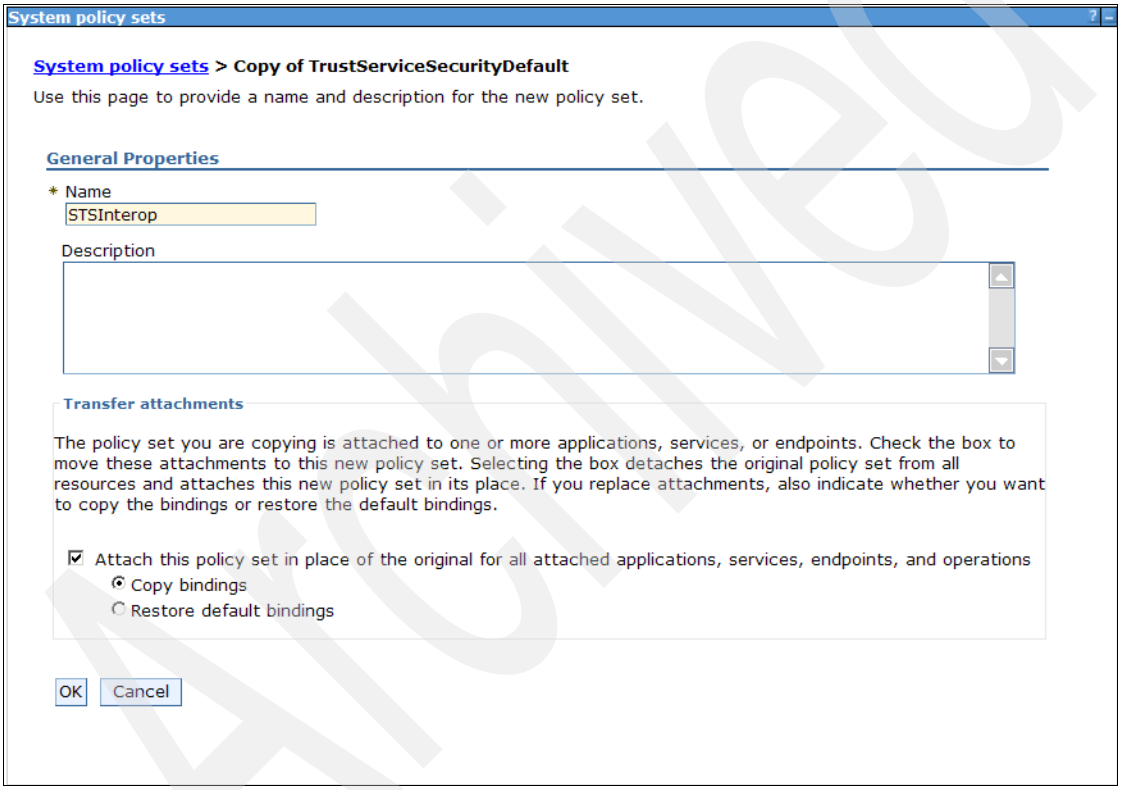


Figure 11-28 Modify TrustServiceSecurityDefault

- Navigate to **Services** → **Policy sets** → **System Policy sets** → Click `STSInterop` → **WS-Security** → **Main policy** → Uncheck **Require signature confirmation** (Figure 11-29) → **OK** → **Save**.

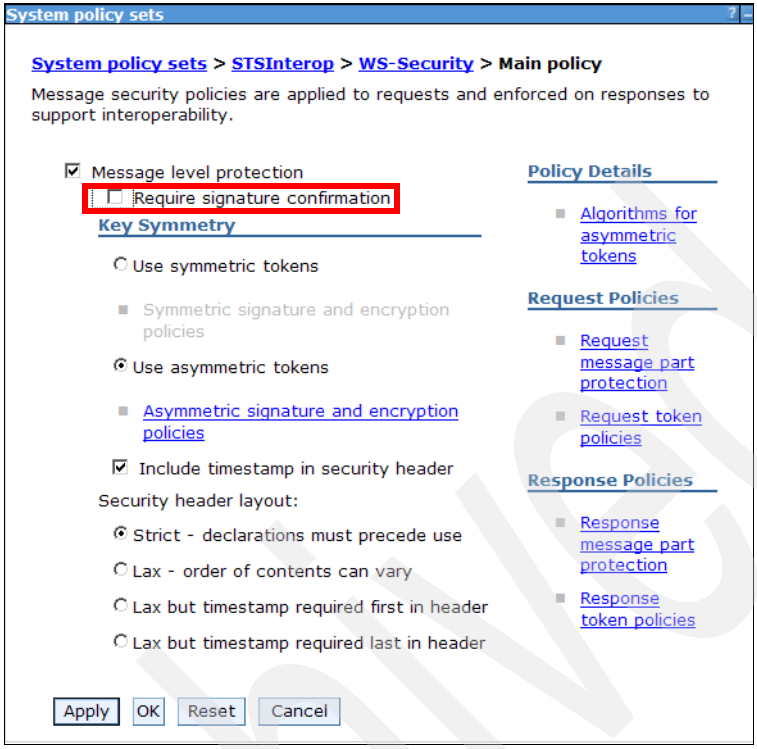


Figure 11-29 Signature confirmation not required

11.7.3 Policy set for secure conversation client

Next we have to configure a Policy set to be used by clients using secure conversation to interoperate with Windows Communication Foundation. There are a number of parts to this operation:

1. Disable signature confirmation in the Bootstrap protocol to match the Policy set we have just created for the Trust service.
2. Specify lax handling of security headers in the application exchange as described in the introduction to 11.7, “Configure Policy sets” on page 443.
3. Add WS-ReliableMessaging to the Policy set and configure it to WS-ReliableMessaging 1.0 with managed persistence.
4. Modify the WS-Security policy type to sign WS-ReliableMessaging messages.

We use the Secure Conversation Policy set shipped with the Feature pack for Web services as the base for making the changes.

- Navigate to **Services** → **Policy sets** → **Application Policy sets** → Check `Secure conversation` → Click **Copy** → Name the new Policy set `SCCltInterop` → **OK** → **Save**.
- Navigate to **Services** → **Policy sets** → **Application Policy sets** → Click `SCCltInterop` → **WS-Security** → **Bootstrap policy** → Uncheck **Require signature confirmation** (Figure 11-30 on page 448) → **OK**.

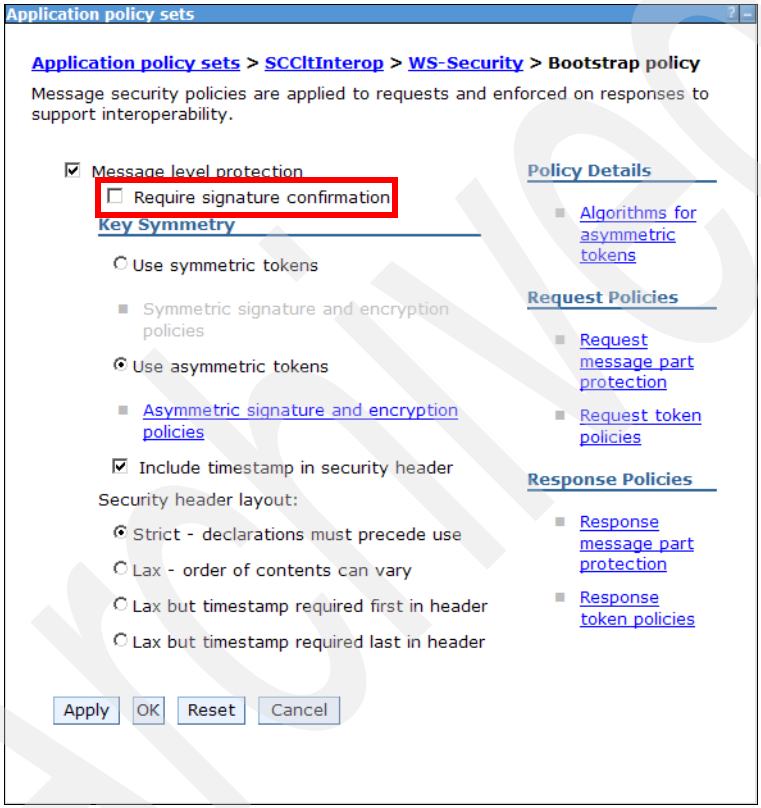


Figure 11-30 Deselect signature confirmation from bootstrap policy

- Click **Main policy** → **Select Lax security header layout** (Figure 11-31) → **OK** → **Save**.

Figure 11-31 Select lax header handling

- Navigate to **Services** → **Policy sets** → **Application Policy sets** → **SCCltInterop** → **Add** → **WS-ReliableMessaging** → **Save** (Figure 11-32).

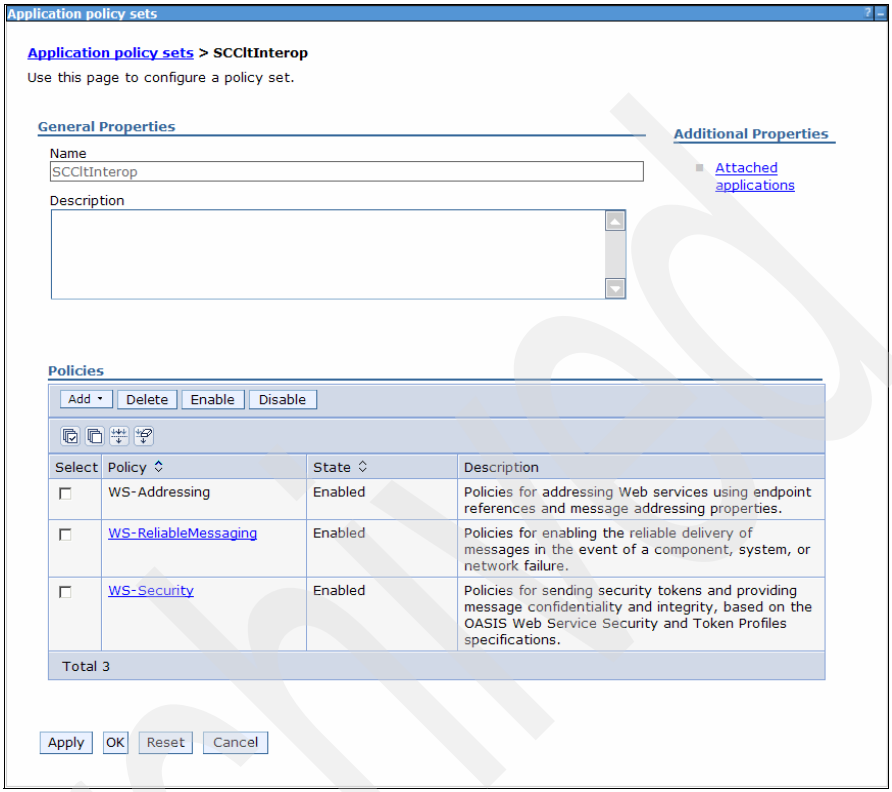


Figure 11-32 Modified Secure Conversation policy set

- Click **WS-ReliableMessaging** (Figure 11-32) → Select **WS-ReliableMessaging 1.0** → Uncheck **Enable "MakeConnection" for ...** → Select **Managed persistent** (Figure 11-33) → **Save**.

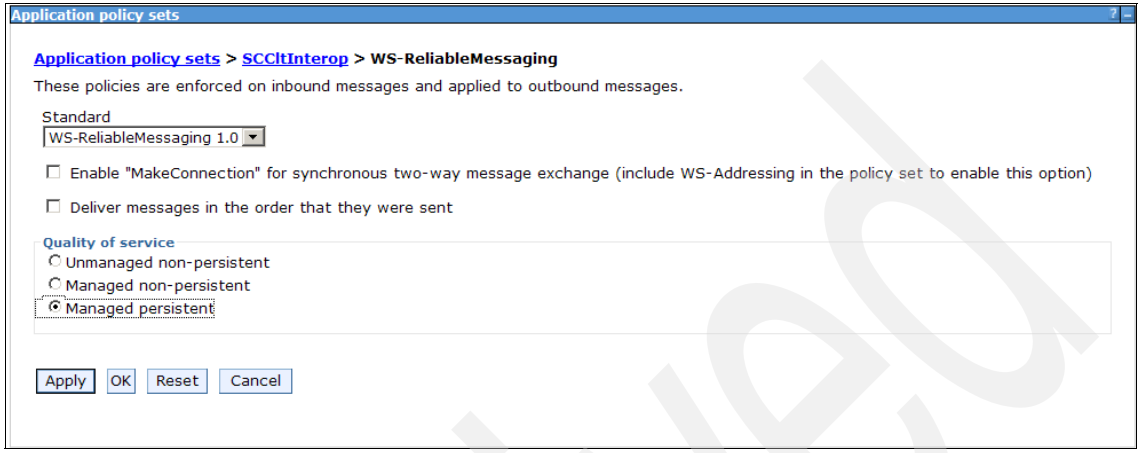


Figure 11-33 Customize WS-ReliableMessaging

- Click **WS-ReliableMessaging** (Figure 11-32 on page 450) → **WS-Security** → **Main policy** → Select **Request message part protection** → Under **Integrity protection** select **app_signparts** → **Edit** → Select **QName** (Figure 11-34).

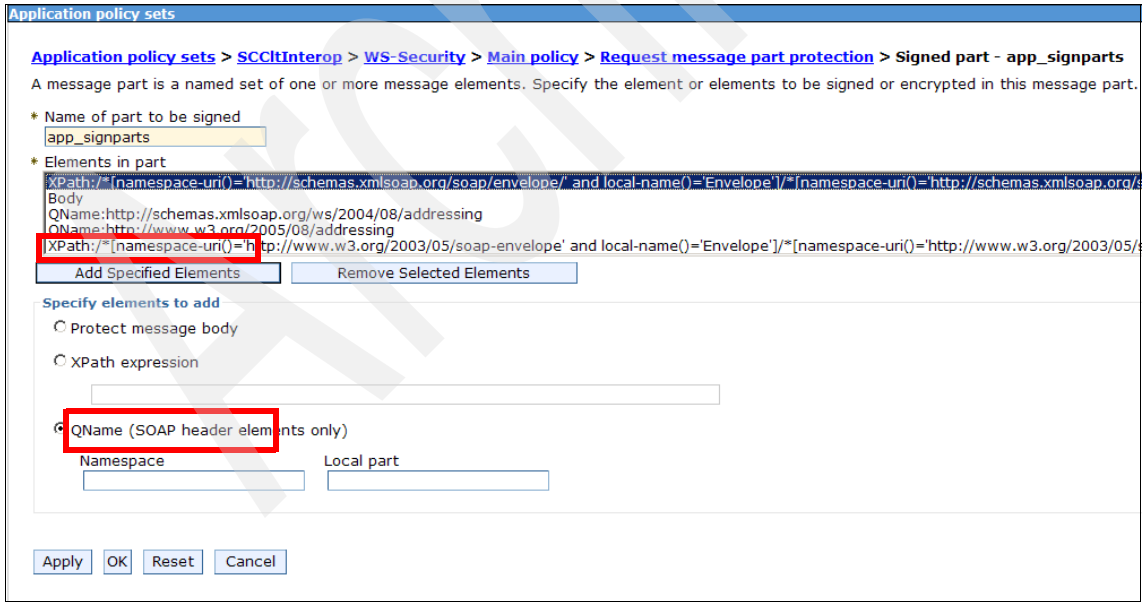


Figure 11-34 Select QName in signed part `app_signparts` of SCCltInterop Policy set

- ☐ Type `http://schemas.xmlsoap.org/ws/2005/02/rm/` into a browser to verify that the URL is correct (Figure 11-35 on page 452) → Paste into the **Namespace** of QName (leave Local part blank) (Figure 11-34) → **Add Specified Elements** → **Apply** → **Save**.

Figure 11-35 Verify `http://schemas.xmlsoap.org/ws/2005/02/rm/`

- ☐ Repeat this procedure to verify and add: `http://docs.oasis-open.org/ws-rx/wsrm/200702` (Figure 11-36) → **Save**.

Figure 11-36 http://docs.oasis-open.org/ws-rx/wsrm/200702

The Elements in part now includes the QNames shown in Figure 11-37.

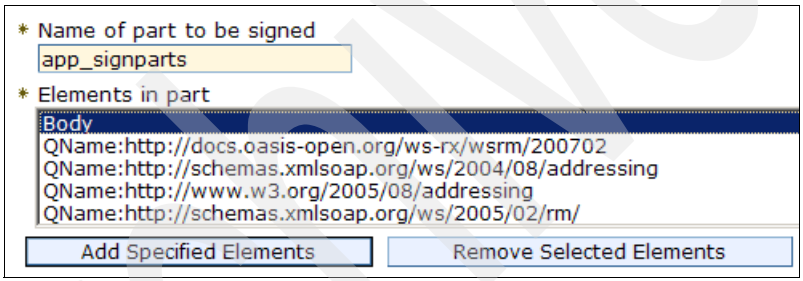


Figure 11-37 QNames in app-signparts

- Click **WS-ReliableMessaging** (Figure 11-32 on page 450) → **WS-Security** → **Main policy** → Select **Response message part protection** → Under **Integrity protection** select **app_signparts** → **Edit** → Select **QName**.
- Repeat the procedure we carried out to add the WS-ReliableMessaging QNames to the request part protection to add the QNames to the response part protection as well. → **Save**.

11.7.4 Policy set for a secure conversation service

Finally, we have to set up a Policy set for a secure conversation service to interoperate with Windows Communication Foundation. We use the Policy set we set up for the client as a template for the service. Unlike the client, we do not need a bootstrap policy, as the client is responsible for obtaining the secure conversation token.

- ☐ Navigate to **Services** → **Policy sets** → **Application Policy sets** → Check `SCCltInterop` → Click **Copy** → Name the new Policy set `SCSvcInterop` → **OK**.
- ☐ Navigate to **Services** → **Policy sets** → **Application Policy sets** → Click `SCSvcInterop` → **WS-Security** → **Remove bootstrap policy** → **OK**.

  The rest of the Policy set matches the client.
- ☐ Verify that the following changes have been made:
  - ☐ Navigate to **Services** → **Policy sets** → **Application Policy sets** → `SCSvcInterop` → **Add** → **WS-ReliableMessaging**.
  - ☐ Click **WS-ReliableMessaging** → Select **WS-ReliableMessaging 1.0** → Uncheck **Enable "MakeConnection" for ...** → Select **Managed persistent** → **Save**.
  - ☐ Click **WS-ReliableMessaging** → **WS-Security** → **Main policy** → Select **Request message part protection** → Under **Integrity protection** select **app_signparts** → **Edit** → Select **QName**.
  - ☐ Type `http://schemas.xmlsoap.org/ws/2005/02/rm/` into a browser to verify the URL is correct (Figure 11-35 on page 452) → Paste into the **Namespace** of QName (leave Local part blank) → **Add Specified Elements** → **Apply** → **Save**.
  - ☐ Repeat this procedure to verify and add `http://docs.oasis-open.org/ws-rx/wsrm/200702` (Figure 11-36 on page 453) → **Save**.
  - ☐ Repeat the procedure we carried out to add the WS-ReliableMessaging QNames to the request part protection to add the QNames to the response part protection as well. → **Save**.

11.8 Attach Policy sets and bindings to applications

With Policy sets defined, we next to attach them to the applications to which they are applied, that is, to the clients and services that use the Policy sets. Slightly confusingly, we also attach the bindings at this point, although we *do not define them* to the next section in the configuration. This is a temporary expedient, due to the fact until WebSphere Application Server v7.0 bindings are resources that are attached to clients or services and are not imported and exported separately. For this reason too, we can only use the default bindings for applications that are deployed from Rational Application Developer, because there is no simple way to import or define custom bindings for services in Rational Application Developer in the current release.

The steps we take are as follows:

- 11.8.1, “Reinstall applications on WebSphere Application Server” on page 455
- 11.8.2, “Attach STSInterop Policy set” on page 457
- 11.8.3, “Assign bindings to the STSInterop Policy set” on page 457
- 11.8.4, “Attach SCSvcInterop Policy set” on page 458
- 11.8.5, “Assign bindings for SCSvcInterop Policy set” on page 459
- 11.8.6, “Attach SCCltInterop Policy set” on page 460
- 11.8.7, “Assign bindings for SCCltInterop Policy set” on page 461
- 11.8.8, “Attach ITSO WSRM 10 Persistent Policy set” on page 462
- 11.8.9, “Assign bindings to ITSO WSRM 10 Persistent Policy set” on page 463
- 11.8.10, “Attach WSRM1_0 Policy set to the PolicyStoreThinClient” on page 463
- 11.8.11, “Assign bindings to WSRM 1_0 Policy set” on page 464

11.8.1 Reinstall applications on WebSphere Application Server

1. Remove projects from the Server.

☐ In Rational Application Developer, in the Servers view → Click **WebSphere Application Server v6.1 Server** → **Add and Remove Projects** → **Select CreditCheckEAR** → **Remove** → **MedicalCheckEAR** → **Remove** → **UnderwriterEAR** → **Remove** → **Finish**

2. Export the CreditCheckEAR files.

☐ Select CreditCheckEAR → **Export** → **EAR** → Browse to a suitable directory to save the .ear file → Check **Export source files** → **Finish**

3. Repeat the export for UnderwriterEAR and MedicalCheckEAR.
4. Install CreditCheckEAR using the Administrator Console.

☐ In the Administrator Console → **Applications** → **Install New Applications** (Figure 11-38) → **Next** → Select **Step 3 Summary** → **Finish** → **Save**.

> **Tip:** Sometimes the installer reports that the application already exists in the application repository, although the application is not listed. Try logging out of the Administrator Console and back into it.

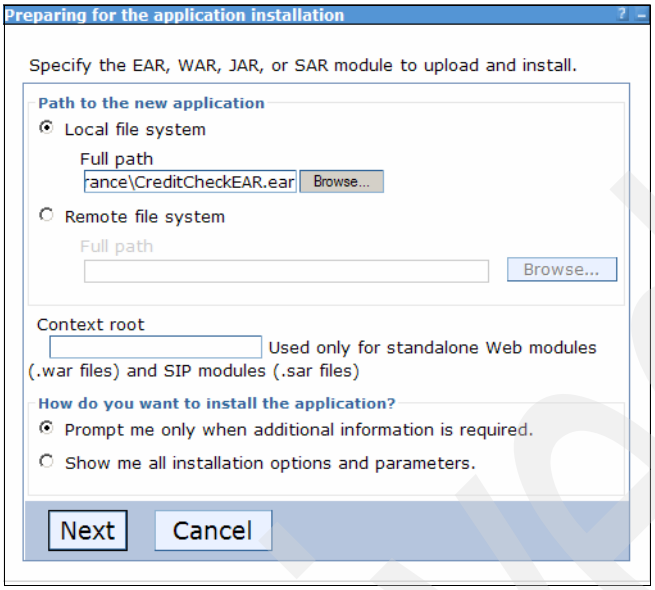


Figure 11-38 Installing CreditCheckEAR

5. Repeat the installation for the MedicalCheckEAR and UnderwriterEAR.

- ☐ In the Administrator Console → **Applications** → **Enterprise Applications** → Check **CreditCheckEAR, MedicalCheckEAR** and **UnderwriterEAR** → **Start**.
- ☐ View **Services** → **Service providers** (Figure 11-39).

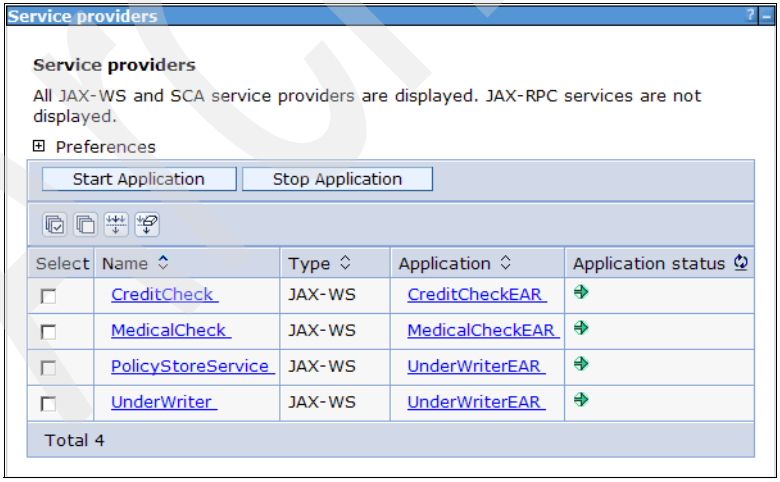


Figure 11-39 Installed and running applications

11.8.2 Attach STSInterop Policy set

In 11.7.2, “Policy set for trust service issuing Secure Conversation token”, we defined the customized Policy set for issuing and renewing the Secure Conversation Token (SCT). Now apply this to the trust service.

- Navigate to **Services** → **Trust service** → **Trust service attachments** → and Check **Issue** and **Renew** token → **Attach** → **STSInterop** (Figure 11-40) → **Save**.

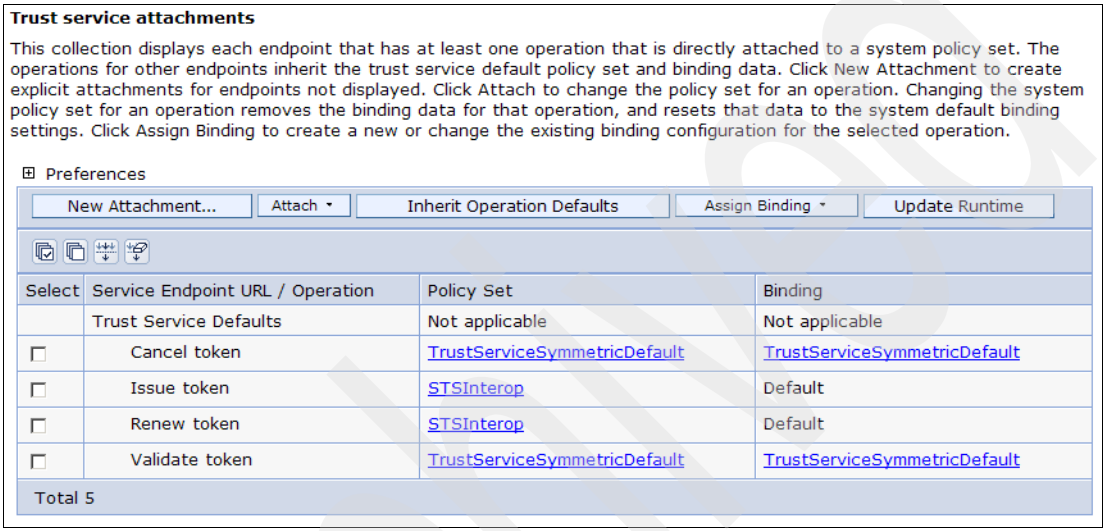


Figure 11-40 Attach STSInterop Policy set

11.8.3 Assign bindings to the STSInterop Policy set

Next we assign a new binding to the issue and renew token operations.

- In the Administrator Console, Navigate to **Services** → **Trust service** → **Trust service attachments** → Check **Issue** and **Renew** token → **Assign binding** → **Create new** → Type `STSBinding` as the Bindings configuration name → **Add** → **WS-Addressing** → **Add** → **WS-Security** → **Save** (Figure 11-41).

Trust service attachments

This collection displays each endpoint that has at least one operation that is directly attached to a system policy set. The operations for other endpoints inherit the trust service default policy set and binding data. Click New Attachment to create explicit attachments for endpoints not displayed. Click Attach to change the policy set for an operation. Changing the system policy set for an operation removes the binding data for that operation, and resets that data to the system default binding settings. Click Assign Binding to create a new or change the existing binding configuration for the selected operation.

⊞ Preferences

New Attachment... | Attach | Inherit Operation Defaults | Assign Binding | Update Runtime

Select	Service Endpoint URL / Operation	Policy Set	Binding
	Trust Service Defaults	Not applicable	Not applicable
☐	Cancel token	TrustServiceSymmetricDefault	TrustServiceSymmetricDefault
☐	Issue token	STSInterop	STSBinding
☐	Renew token	STSInterop	STSBinding
☐	Validate token	TrustServiceSymmetricDefault	TrustServiceSymmetricDefault

Total 5

Figure 11-41 Attached STSbinding to Issue and renew token

11.8.4 Attach SCSvcInterop Policy set

We defined SCSvcInterop in 11.7.4, “Policy set for a secure conversation service” on page 453. Now attach it to the Medical and Credit Check services.

- Click **MedicalCheck** (Figure 11-39 on page 456) → Check **MedicalCheck** → **Attach** → **SCSvcInterop** → **Save** (Figure 11-42)

Policy set attachments

Attach policy sets to the service, endpoints, or operations and assign the default bindings, create new bindings, or assign existing custom bindings for the attached policy sets. Note that you can view or modify the default bindings from the cell- or server-level security panels. Also note that you can only directly attach a policy set to an operation if the policy set has WS-Addressing enabled or if the WSDL specifies WS-Addressing headers.

⊞ Preferences

Attach | Detach | Assign Binding

Select	Service/Endpoint/Operation	Attached policy set	Binding
☐	MedicalCheck	SCSvcInterop	Default
☐	MedicalCheckSOAP	SCSvcInterop (inherited)	Default (inherited)
☐	EvaluateRecord	SCSvcInterop (inherited)	Default (inherited)

Total 3

Figure 11-42 Attached SCSvcInterop to MedicalCheck

Repeat this procedure to attach SCSvcInterop to the CreditCheck service.

11.8.5 Assign bindings for SCSvcInterop Policy set

- Click **MedicalCheck** (Figure 11-39 on page 456) → Check **MedicalCheck** → **Assign binding** → **New** → Type **SCSvcBinding** → **Add** → **WS-Addressing** → **Add** → **WS-Security** → **Add** → **WS-ReliableMessaging** (Figure 11-43).

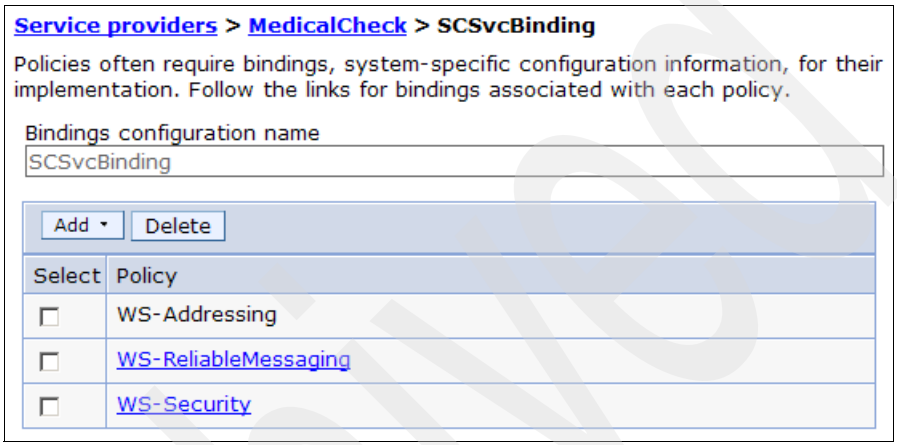


Figure 11-43 Added policies to Medical check SCSvcBinding

- **Save** (Figure 11-44).

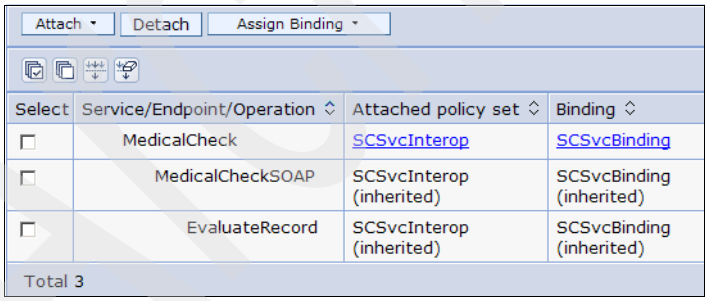


Figure 11-44 SCSvcBinding assigned to MedicalCheck service

- Repeat this procedure to add SCSvcBinding[4] to the CreditCheck service.

[4] We only need attach the binding, as it has the same name we use the same configuration, *but check that the policies have been added.*

11.8.6 Attach SCCltInterop Policy set

We defined SCCltInterop in 11.7.3, “Policy set for secure conversation client” on page 447. Now attach it to the Medical and Credit Check clients.

- View **Services** → **Service client** (Figure 11-45).

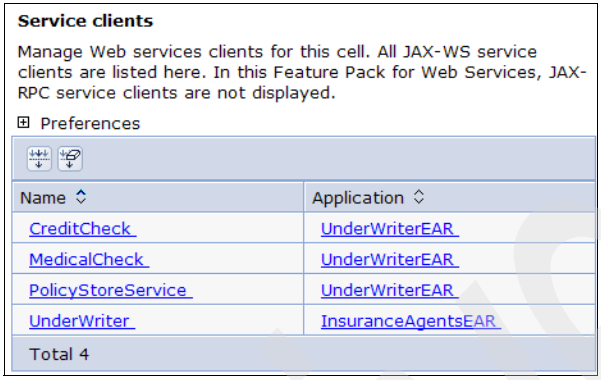


Figure 11-45 Service clients

- Click **MedicalCheck** → Check **MedicalCheck** → **Attach** → **SCCltInterop** → **Save** (Figure 11-46).

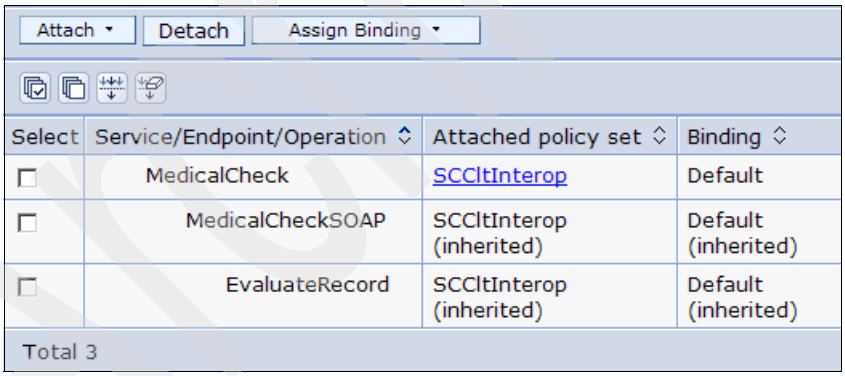


Figure 11-46 Attach SCCltInterop Policy set

- Repeat this procedure to attach SCCltInterop to the CreditCheck client.

11.8.7 Assign bindings for SCCltInterop Policy set

- Click **CreditCheck** (Figure 11-45 on page 460) → Check **CreditCheck** → **Assign binding** → **New** → Type **SCCltBinding** → **Add** → **WS-Addressing** → **Add** → **WS-Security** → **Add** → **WS-ReliableMessaging** (Figure 11-47).

Figure 11-47 Added policies to SCCltbinding

- **Save** (Figure 11-48 on page 461).

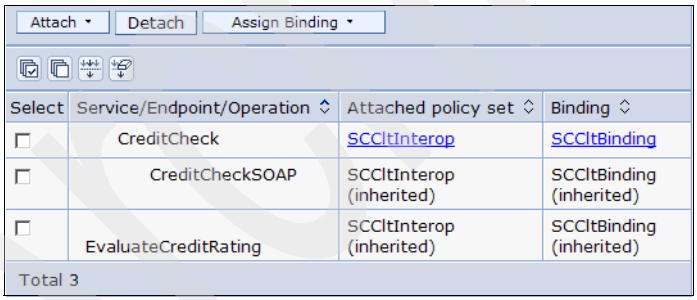


Figure 11-48 Attached SCCltBinding

- Repeat this procedure to attach the SCCltBinding to the Medical check client.

11.8.8 Attach ITSO WSRM 10 Persistent Policy set

The ITSO WSRM 10 persistent Policy set is attached to the Underwriter service and client. Because we are going to modify the default binding for the Policy set, we can import the Policy set into Rational Application Developer and apply it to the underwriter client packaged in the InsuranceAgents application.

1. First we attach the Policy set to the Underwriter service.

☐ In the Administrator Console navigate to **Services** → **Service Providers** → Click **UnderWriter** → Check the **UnderWriter** endpoint → **Attach** → **ITSO WSRM 10 Persistent** (Figure 11-49) → **Save**.

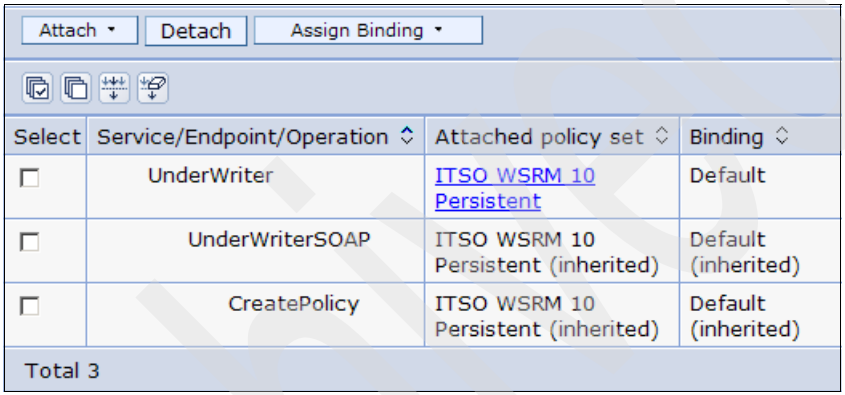


Figure 11-49 Applied ITSO WSRM 10 Persistent policy to Underwriter service

> **Attention:** All services running in the same application must be assigned the same Policy sets and bindings.

2. Next assign the `ITSO WSRM 10 Persistent` Policy set to the PolicyStoreService.

☐ Repeat the procedure to attach the `ITSO WSRM 10 Persistent` Policy set to the PolicyStoreService.

3. Next, export the `ITSO WSRM 10 Persistent` Policy set from WebSphere Application Server and import it into Rational Application Developer.

☐ In the Administrator Console navigate to **Services** → **Application Policy sets** → Check `ITSO WSRM 10 Persistent` → **Export** → **ITSO WSRM 10 Persistent.zip** → **Save** → **Close**.

☐ In Rational Application Developer **Window** → **Preferences** → **Web services** → **Quality of Service** → **Import Policy set** → **Open** `ITSO WSRM 10 Persistent.zip` → **OK**.

4. Finally, attach the Policy set to the InsuranceAgent client.

- In the project Explorer **JAX-WS Web services** → **Clients** → right-click the → Insurance Agents Underwriter client → **Manage Policy set attachment** → **Add...** → **Select** the ITSO WSRM 10 Persistent Policy set → Type IAClientBinding in the Binding field (Figure 11-50) → **OK** → **Finish**.

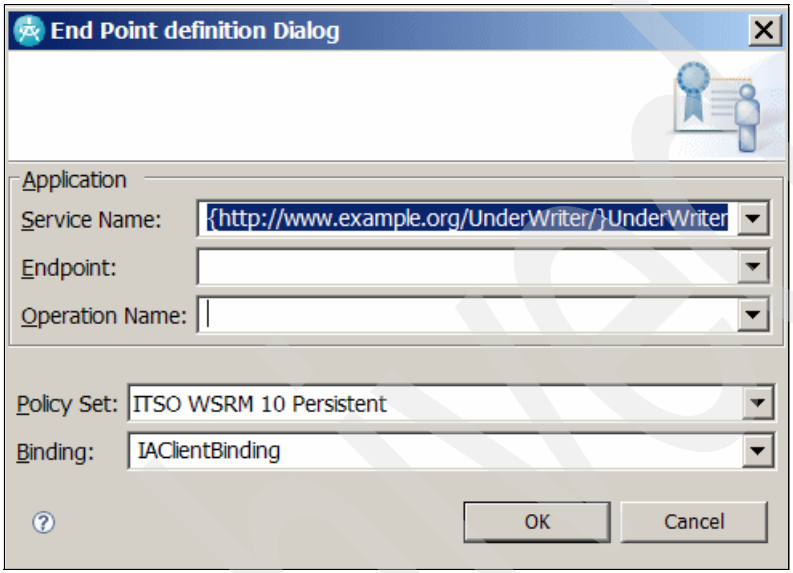


Figure 11-50 Applied ITSO WSRM 10 Persistent Policy set to Underwriter client

11.8.9 Assign bindings to ITSO WSRM 10 Persistent Policy set

There are no bindings to assign as we configure the default bindings for the ITSO WSRM 10 Persistent Policy set.

11.8.10 Attach WSRM1_0 Policy set to the PolicyStoreThinClient

In the Project Explorer in Rational Application Developer → **JAX-WS Web services** → **Clients** → **PolicyStoreThinClient** → **Manage Policy set Attachment** → **Add** → Select **WS-ReliableMessaging 1_0** → Type PSClientBinding as the Binding name → **OK** → **Finish** (Figure 11-51).

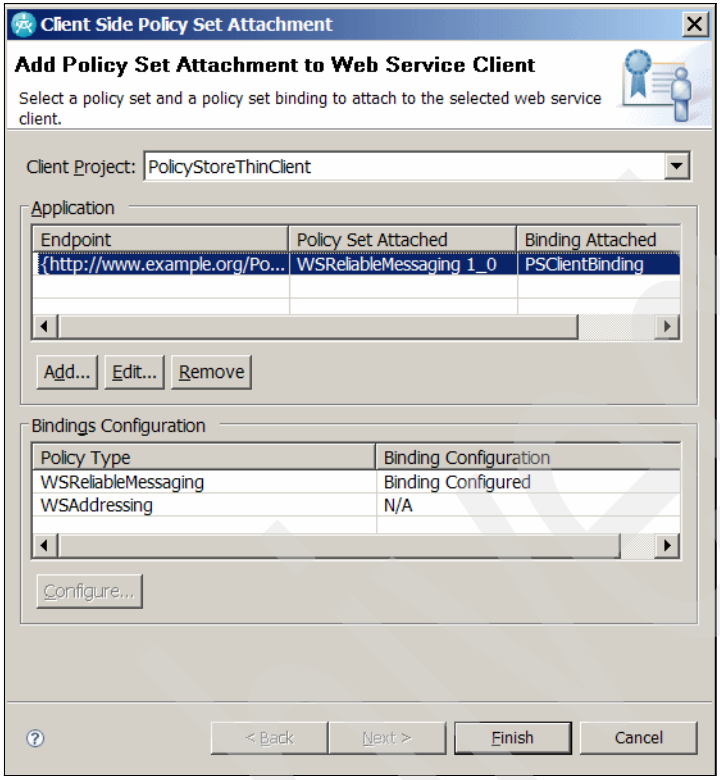


Figure 11-51 Attached WS-ReliableMessaging 1_0 to PolicyStoreThinClient

11.8.11 Assign bindings to WSRM 1_0 Policy set

There are no bindings to assign as we configure the default bindings for the ITSO WS-ReliableMessaging 1_0 Policy set.

11.9 Configure resources

Now that we have the applications deployed and tested and the Policy sets configured, we have to configure the resources necessary to apply the Policy sets to the applications and run the applications with the new qualities of service. The resources we configure are:

- 11.9.1, “Create an SI bus to store and manage WS-RM messages”
- 11.9.2, “Configure keystores for WebSphere Application Server” on page 466
- 11.9.3, “Configure keystores for Windows Communication Foundation” on page 467

11.9.1 Create an SI bus to store and manage WS-RM messages

- ☐ In the left hand navigation pane in the **Administrator Console**, select **Service Integration** → **Buses**.
- ☐ Click **New** → Type in **rmBus** as the bus name → Uncheck **Bus security** → **Next** → **Finish** (Figure 11-52).

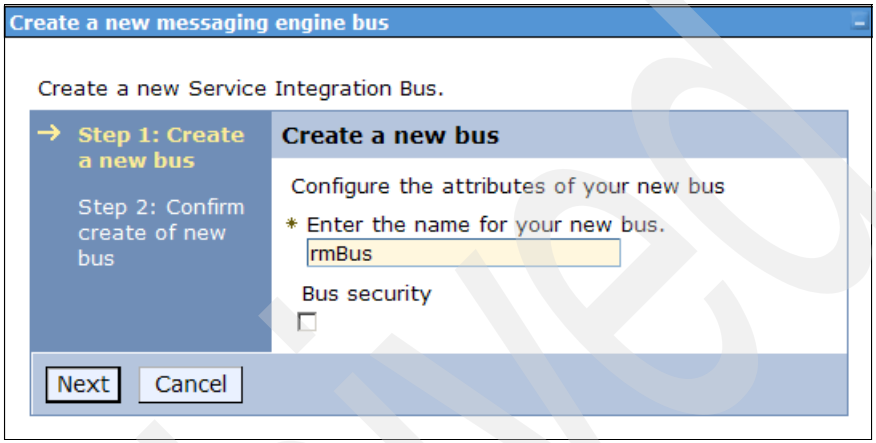


Figure 11-52 Create rmBus

Next we have to allocate our server to the rmBus and configure the kind of message store it is using.

- ☐ Click **rmBus** → **Bus members** → **Add** → select **Server** → **Next** → select **File store** → **Next** → **Next** → **(**Figure 11-53) **Finish** → **Save**.

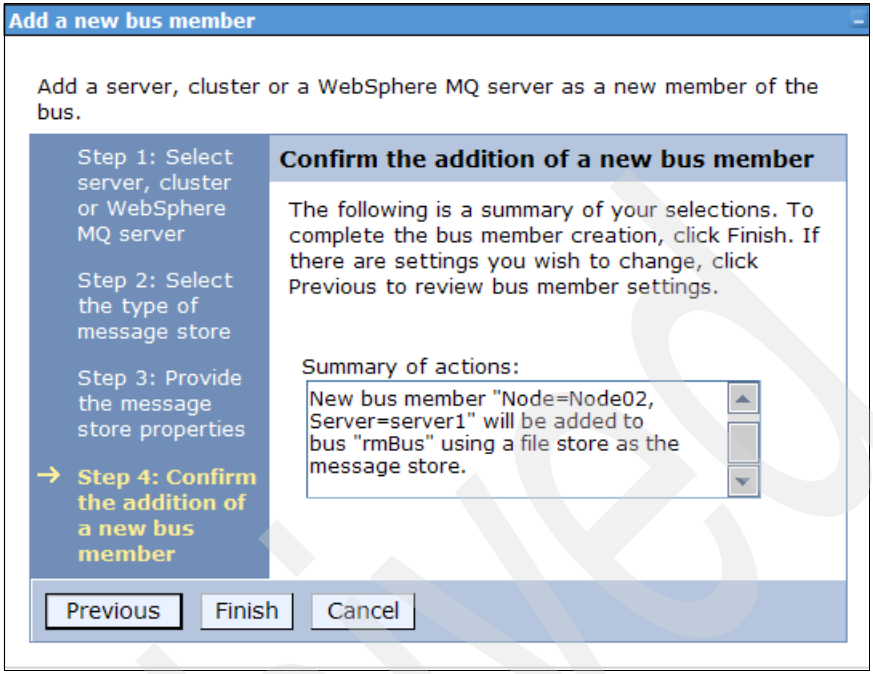


Figure 11-53 Add bus member to messaging engine

11.9.2 Configure keystores for WebSphere Application Server

Table 11-3 shows the keystores and keys preconfigured for the insurance application running in WebSphere Application Server. Copy the keystores into the directory shown. You can use ikeyman in `<WAS_HOME>\bin` to browse the keystores.

Table 11-3 Keystore configuration information

	Client	Service
Who am I?	alice	bob
What is my keystore called?	<WAS_HOME>\etc\ws-security\mySysKeys\sender.jks pw = sampleapp	<WAS_HOME>\etc\ws-security\mySysKeys\receiver.jks pw = sampleapp
What is my private key?	alias = alice CN=alice,O=myco,OU=myou pw = sampleapp	alias = bob CN=bob,O=myco,OU=myou pw = sampleapp

	Client	Service
What public keys do I have?	alias = bob CN=bob,O=myco,OU=myou	alias = alice CN=alice,O=myco,OU=myou
Who signed my key?	CN=myca,O=ca3,OU=sec	

11.9.3 Configure keystores for Windows Communication Foundation

To install the keys that are provided for the insurance application on Windows Communication Foundation (see Table 11-4), we have to run Microsoft Management Console and install the Certificates snap-in for the current user and the local computer.

- ☐ **Start** → **Run...** → Type `mmc` → **OK**.
- ☐ **File** → **Add/Remove snap-in** → (on the Standalone tab) **Add...** → **Certificates** → **Add** → **My user account** → **Finish** → **Certificates** → **Add** → **Computer account** → **Next** → Select **Local computer** → **Finish** → **Close** (Figure 11-54) → **OK**.

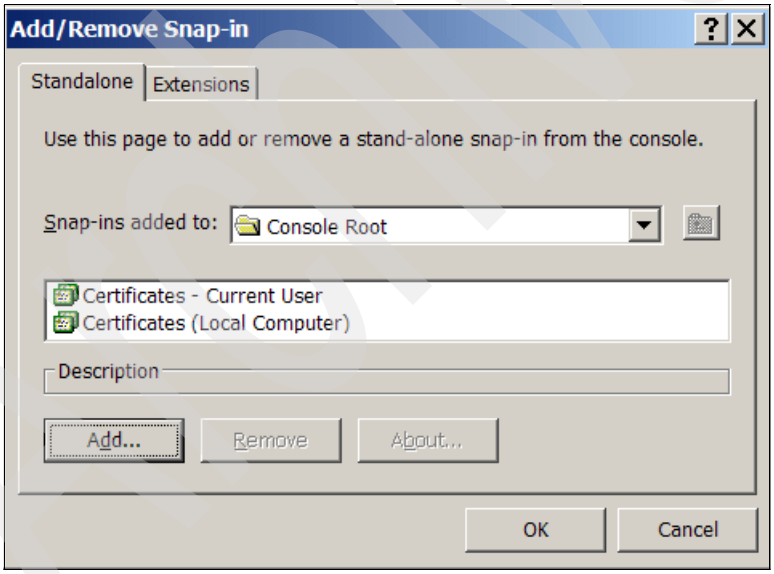


Figure 11-54 Installed two snap-ins to manage certificates

Use the information in Table 11-4 to import the certificates provided in the additional materials into the certificates store. Be extremely careful to import the certificates into the correct certificates store. There are five certificates and six import operations to perform (myca.cer is imported twice).

To import a certificate, do the following tasks (we show the example of adding alice-key.p12 into the Current user personal certificate store.

- Right-click **Certificates - Current User** → **Personal** → **All Tasks** → **Import...** → **Next** → **Browse to** alice-key.p12 → **Open** → **Next** → **Type** the password `sampleapp` → **Check** `Mark this key as exportable` → **Next** → leave `Place all certificates in the following store: Personal` checked → **Next** → **Finish** → **OK**.

Table 11-4 Keystore configuration information

	Client	**Service**
Who am I?	alice	bob
What is my certificate store called? snap-in name	**Current User** pw = sampleapp	**Local Computer** pw = sampleapp
What is my private key? **Personal** folder	file = **alice-key.p12** alias = alice CN=alice,O=myco,OU=myou pw = sampleapp	file = **bob-key.p12** alias = bob CN=bob,O=myco,OU=myou pw = sampleapp
What public keys do I have? **Trusted people** folder	file = **bob-cert.der** alias = bob CN=bob,O=myco,OU=myou	file = **alice-cert.der** alias = alice CN=alice,O=myco,OU=myou
Who signed my key? **Trusted Root Certificate Authority** folder	file = **myca.cer** alias = myca CN=myca,O=ca3,OU=sec	

11.10 Configure bindings

We have attached bindings to the applications in the insurance solution, but thus far we have not configured them. The final step is to apply the resources we configured in 11.9, “Configure resources” to the bindings we attached in 11.8, “Attach Policy sets and bindings to applications”. The following bindings have to be configured using the Administrator Console:

- 11.10.1, “Configure STSBinding for STSInterop” on page 469
- 11.10.2, “Configure SCCltBinding for SCCltInterop” on page 494
- 11.10.3, “Configure SCSvcBinding for SCSvcInterop” on page 508
- 11.10.4, “Configure default binding for WS-ReliableMessaging” on page 514
- 11.10.5, “Configure compatible key sizes” on page 514
- 11.10.6, “Migrate SCSvcBinding to the MedicalCheck provider” on page 516

Because we are going to run both the InsuranceAgents client and the PolicyStoreThinClient from Rational Application Developer, the bindings for these applications would be configured using Rational Application Developer. However, we are using the default bindings, so there is no configuration to perform.

Finally, there is an interoperability issue in the key sizes used by default in WebSphere Application Server and Windows Communication Foundation. We deal with this in 11.10.5, “Configure compatible key sizes”.

11.10.1 Configure STSBinding for STSInterop

The bindings for the STSInterop Policy set attached to the STS service have to:

1. Authenticate the inbound signed request using the Trust Anchor Store we bind to the Security Token service
2. Decrypt the inbound security token request received by the STS
3. Sign the outbound response to the client requesting the secure conversation token
4. Encrypt the response sent to the client
5. Set a time-out value for the validity of the token

The configuration steps are performed in the following sections:

1. We first configure the bindings for the requests received by the Security Token Service (STS):
   a. “STSBinding request signature consumer security bindings” on page 469
   b. “STSBinding request encryption consumer security binding” on page 479
2. Next we configure the bindings for the responses sent by the STS:
   a. “STSBinding response signature generator security binding” on page 483
   b. “STSBinding response encryption generator security binding” on page 488[5]
3. “STSBinding response timestamp expiration security binding” on page 491

STSBinding request signature consumer security bindings

In these steps we configure the information needed to identify and validate the signature on the incoming request to the Security Token Service requesting a new Security Context Token.

[5] WebSphere Application Server supports encrypting responses using the public certificate of the signer of the inbound request. We do not use this support.

The following steps are required:

1. “Link to the Trust Anchor store”.
2. “Identify the signed part, the signing algorithm, and the keystore” on page 473

Link to the Trust Anchor store

- In the Administrator Console navigate to **Services** → **Trust service** → **Trust service attachments** → Select the **STSBinding** attached to the Issue token → **WS-Security** → **Authentication and protection** → **AsymmetricBindingInitiatorSignatureToken0** → Verify wss.consume.x509 is selected for JAAS login → **Apply** (generates a callback handling routine) → **Callback handler** (Figure 11-55).

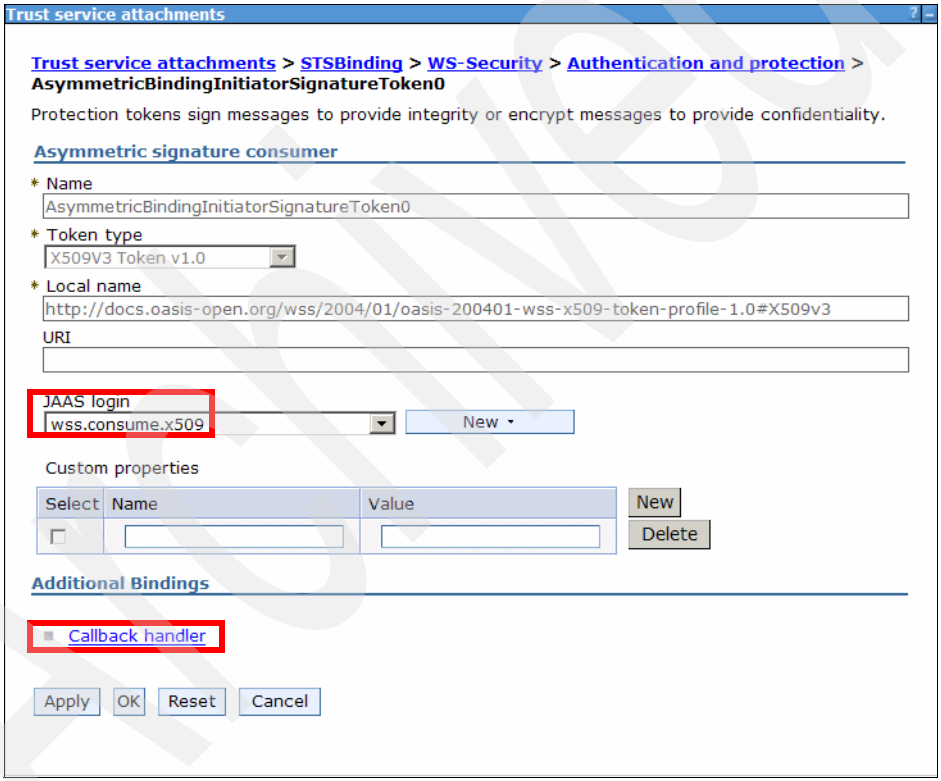


Figure 11-55 Apply Callback handler to JAAS login using wss.consume.x509

- Select **Certificate store** (make sure Name is `None` in the Keystore section) → **Trusted anchor store** → **New ...**(Figure 11-56).

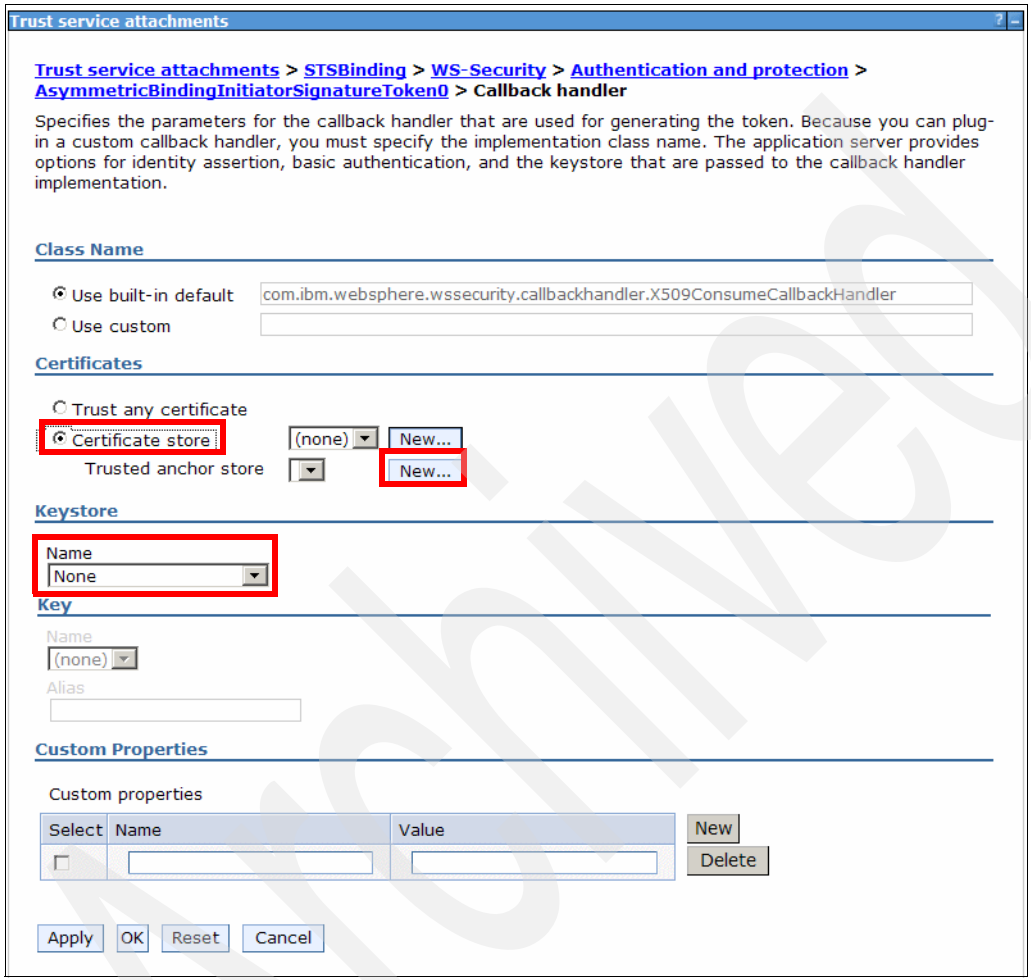


Figure 11-56 Create new trusted certificate store anchor

- Type STSTrustAnchor for the Trusted store name → Select **External Keystore** → Open and explorer window with the path to where `receiver.jks` was unzipped in 11.9.2, “Configure keystores for WebSphere Application Server” on page 466 and paste the path into the Full path field → Set Type to **jks** → Type `sampleapp` as the password (Figure 11-57) → **OK**.

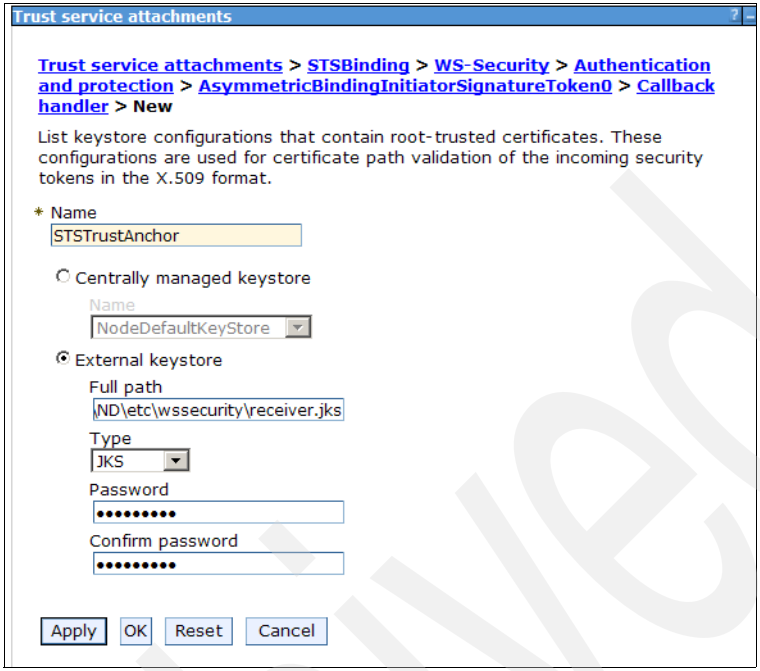


Figure 11-57 Callback handler dialog

- ☐ Verify **STSTrustAnchor** is selected as the Trusted Anchor Store (Figure 11-59) → **OK** → **OK** → **Save** (Figure 11-58).

Figure 11-58 Configuration of AsymmetricBindingInitiatorSignatureToken0

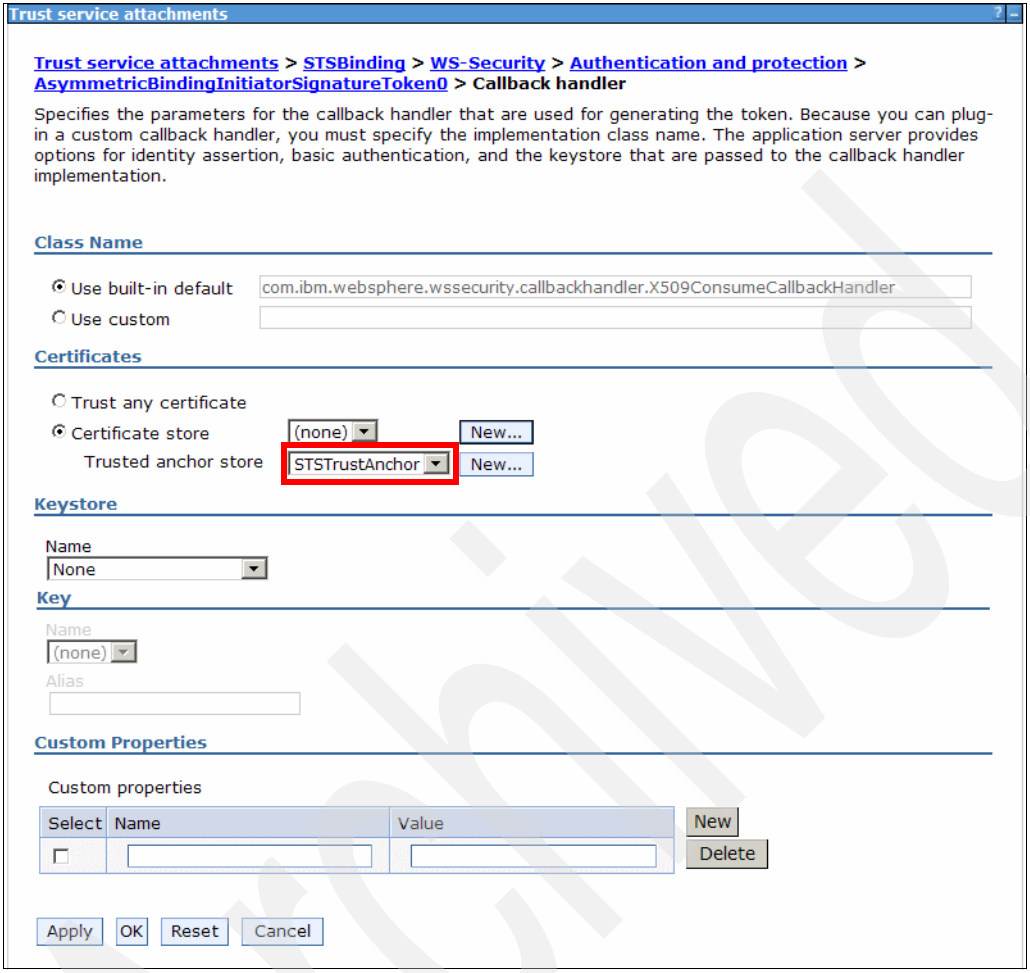


Figure 11-59 Ensure Trusted store anchor set correctly

Identify the signed part, the signing algorithm, and the keystore

- In the Administrator Console navigate to **Services** → **Trust service** → **Trust service attachments** → Select the **STSBinding** attached to the Issue token → **WS-Security** → **Authentication and protection** → Locate the **Request Message signature and encryption protection** section → **request:app_signparts** (Figure 11-60).

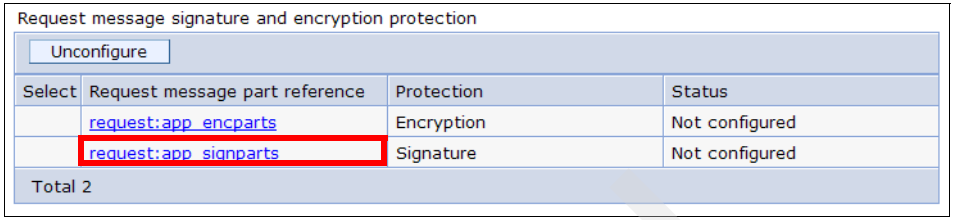


Figure 11-60 request:app_signparts

- In the Name field type `req-sign-msg-part` → **Apply**.

  The Message part reference request:app_signparts should appear in the Assigned box, if not, click the **Add** button to move it from Available to Assigned.

- In **Message part reference** → **Assigned** → Select `request:app_signparts` → **Edit...**(Figure 11-61).

Figure 11-61 Edit request:app_signparts

- Under Transform algorithms (Figure 11-62) → **New**.

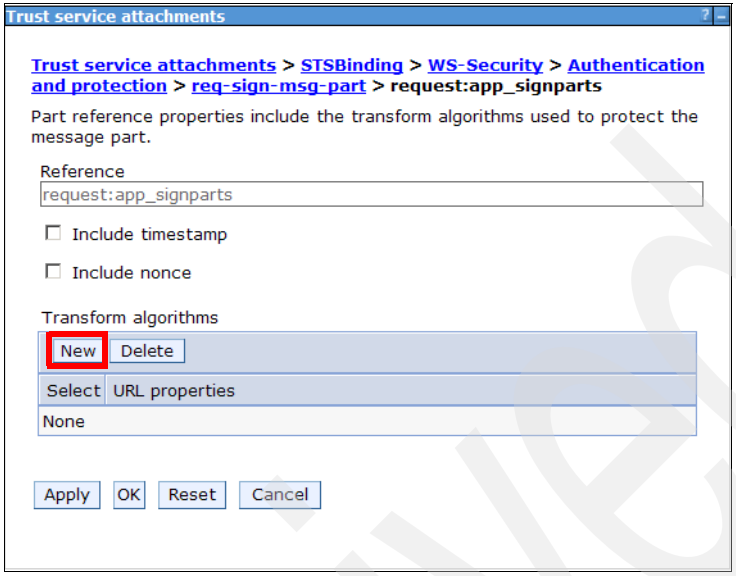


Figure 11-62 New transform algorithm

- Open a browser and type `http://www.w3.org/2001/10/xml-exc-c14n#` → Verify the URL (Figure 11-63).

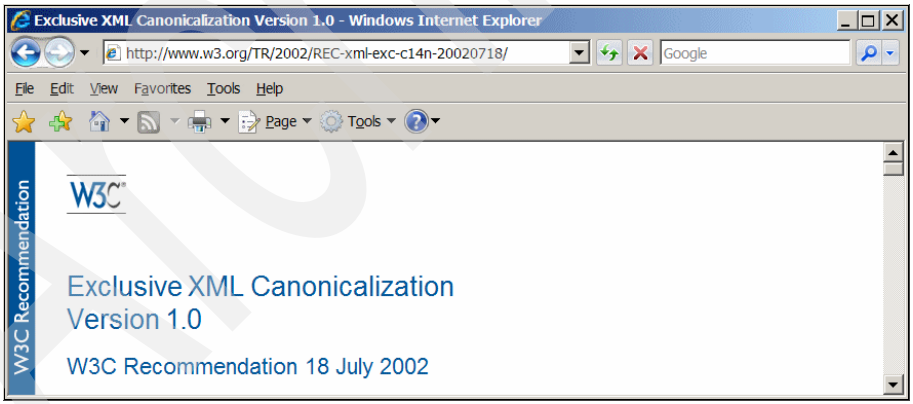


Figure 11-63 Verify the URL, http://www.w3.org/2001/10/xml-exc-c14n#

- Paste it into the URL field (Figure 11-64) → **OK** → **OK**.

Figure 11-64 Paste the Transform algorithm URL into the dialog box URL field

- Navigate to Signing key information (Figure 11-65) → **New...**

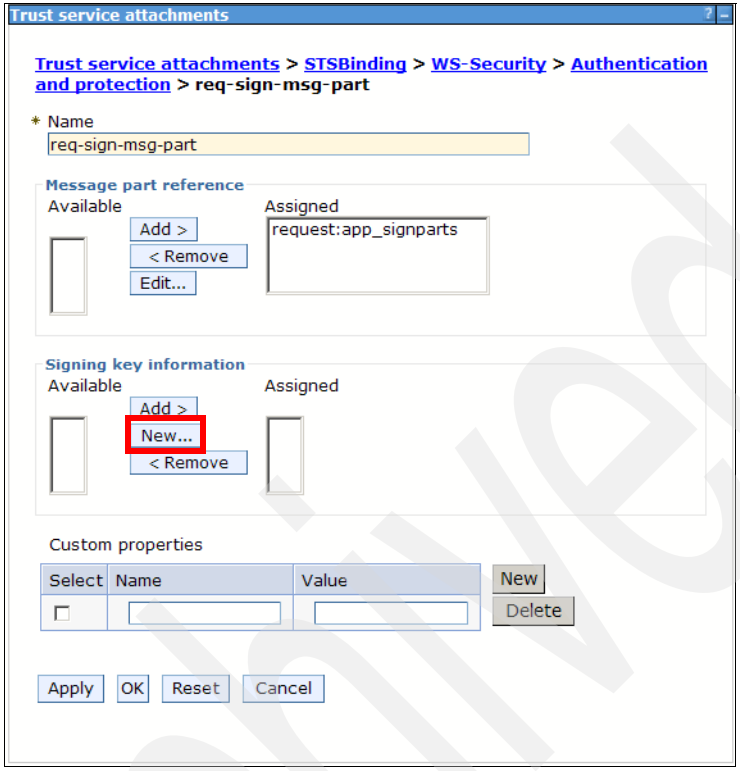


Figure 11-65 New key signing information

- Type `req-sign-keyinfo` in the Name field → Select **AsymmetricBindingInitiatorSignatureToken0** in the Token generator or consumer name field → **OK** → Under Signing key information → Select **req-sign-keyinfo** → **Add>** (Figure 11-66).

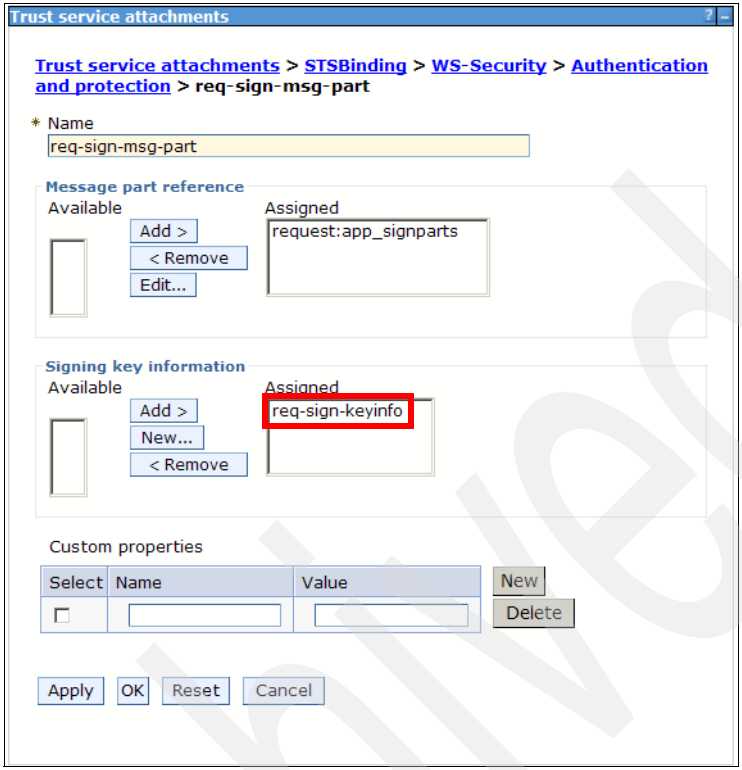


Figure 11-66 Added req-sign-keyinfo

☐ **OK** → **Save.** The signature part is now configured (Figure 11-67).

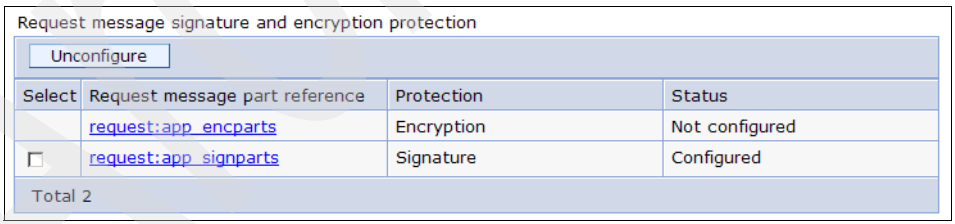


Figure 11-67 Completed configuration of the signature

STSBinding request encryption consumer security binding

In this section we identify the encrypted parts, the encryption algorithm and the keystore used to decrypt the contents of the request for an STS Security Context token.

- In the Administrator Console navigate to **Services** → **Trust service** → **Trust service attachments** → Select the **STSBinding** attached to the **Issue** token → **WS-Security** → **Authentication and protection** → **AsymmetricBindingRecipientEncryptionToken0** (Figure 11-68).

Figure 11-68 Select AsymmetricBindingRecipientEncryptionToken0

- ☐ Verify that the **JAAS login** is set to `wss.consume.x509` → **Apply** → **Callback handler** → In the **Keystore** section → **Custom** as the name → **Custom keystore configuratio**n (Figure 11-69).

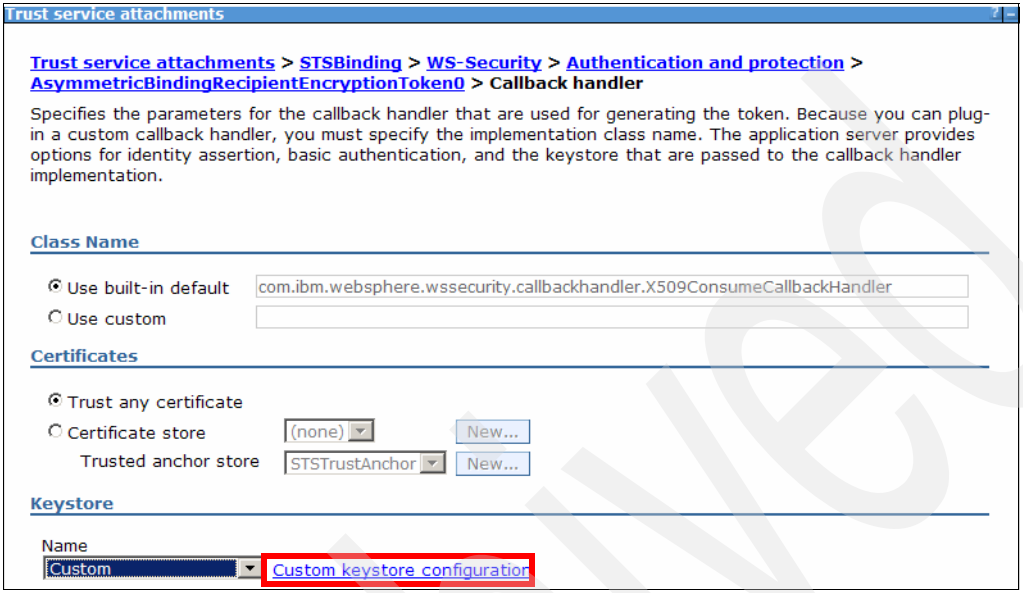


Figure 11-69 Select Custom keystore configuration

- Paste the **Full path** to the receiver.jks keystore → Type the **Password** `sampleapp` → Type in the **Key** information `cn=bob, ou=myou, o=myco` → **Alias** `bob` → **Password** `sampleapp` (Figure 11-70) as provided in Bob's certificate (Table 11-4 on page 468) → **OK** → **OK** → **OK** → **Save**.

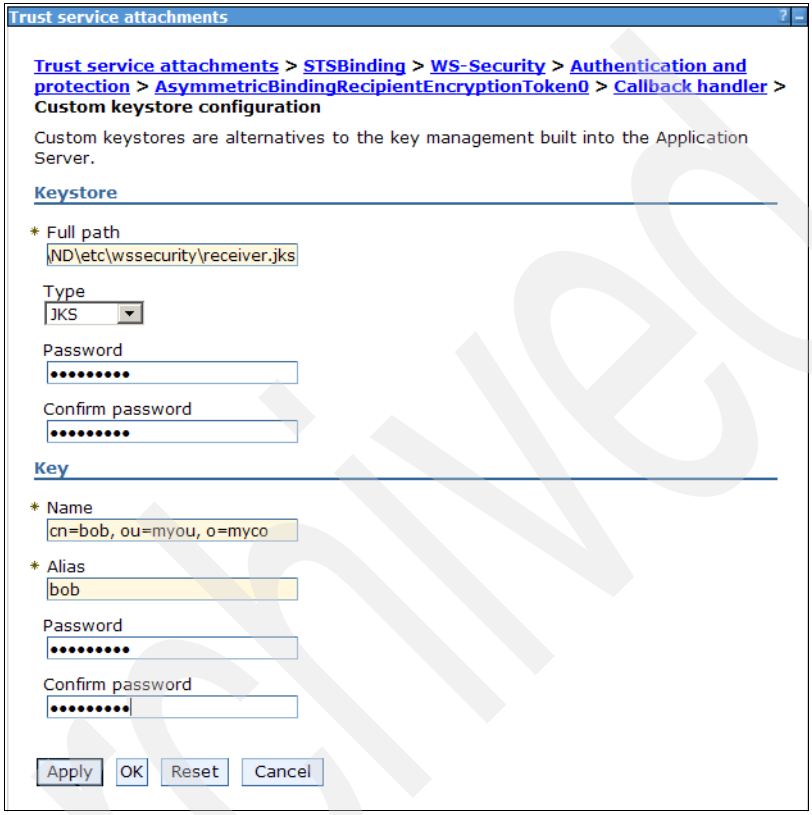


Figure 11-70 Keystore configuration

- Under **Request message signature and encryption protection** → **request:app encparts** → Type `req-enc-msg-part` in **Name** → **Apply**.
- In the **Key information** part → **New** → Type `req-enc-keyinfo` in **Name** → Select AsymmetricBindingRecipientEncryptionToken0 for **Token generator or consumer name** → **OK** (Figure 11-71).

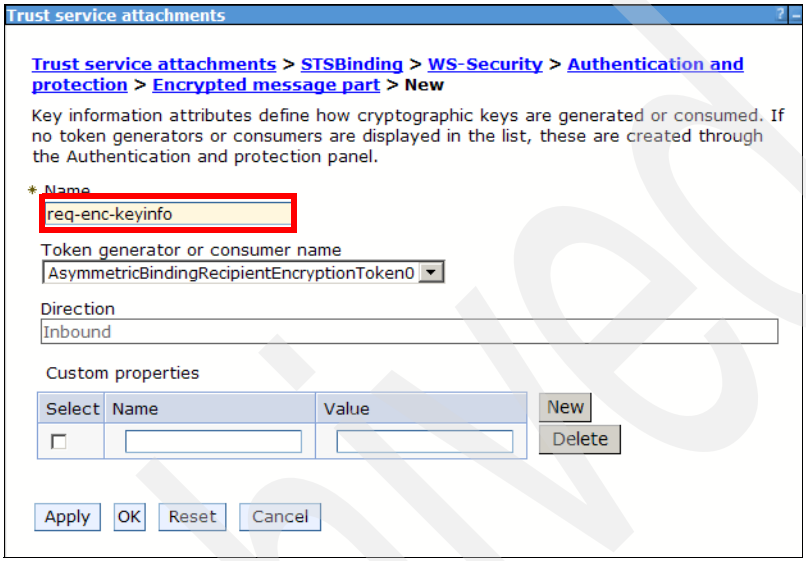


Figure 11-71 New Token generator for inbound encrypted message part

- In the **Key information** part → select `req-enc-keyinfo` → **Add >** (Figure 11-73 on page 483) → **OK** → **Save**.

The configured message signature and encryption protection tab should show both the encoded and signed parts as configured (Figure 11-72).

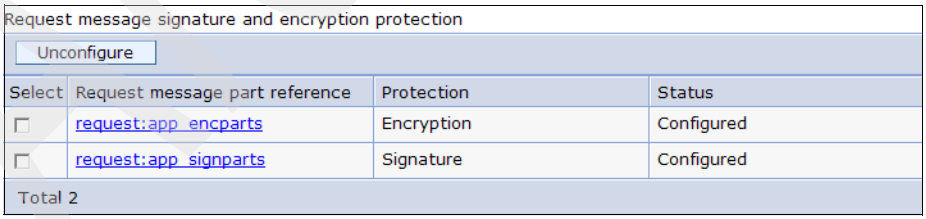


Figure 11-72 Both encoded and signed parts configured

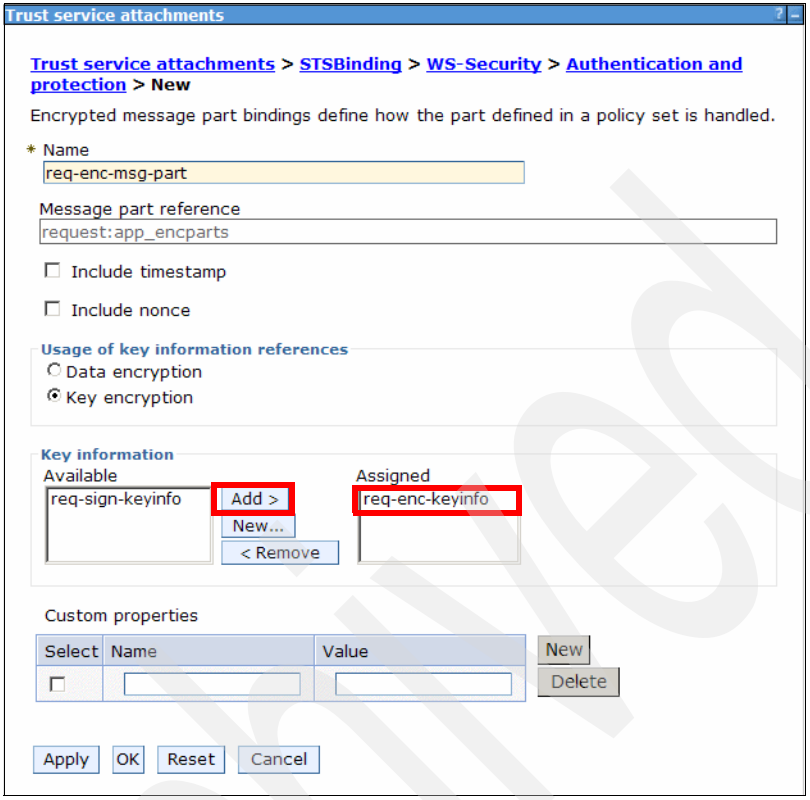


Figure 11-73 Add key information for encoded message part

STSBinding response signature generator security binding

- In the Administrator Console, navigate to **Services** → **Trust service** → **Trust service attachments** → Select the **STSBinding** attached to the **Issue** token → **WS-Security** → **Authentication and protection** → **AsymmetricBindingRecipientSignatureToken0** (Figure 11-74).

Figure 11-74 Select AsymmetricBindingRecipientSignatureToken0

- Verify that the **JAAS login** is set to `wss.consume.x509` → **Apply** → **Callback handler** → In the **Keystore** section → Select **Custom** as the name → **Custom keystore configuration** (Figure 11-69 on page 480).
- Paste the **Full path** to the receiver.jks keystore → Type the **Password** `sampleapp` → Type in the **Key** information `cn=bob, ou=myou, o=myco` → **Alias** `bob` → **Password** `sampleapp` (Similar to Figure 11-70 on page 481) as provided in Bob's certificate (Table 11-4 on page 468) → **OK** → **OK** → **OK** → **Save**.
- Under **Response message signature and encryption protection** → **request:app signparts** (Figure 11-75) → Type `resp-sign-msg-part` in **Name** → **Apply**.

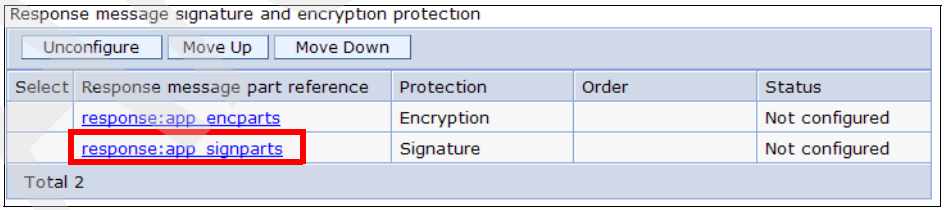


Figure 11-75 Select response:app_signparts

- In the Signing **Key information** part → **New** → (Figure 11-76).

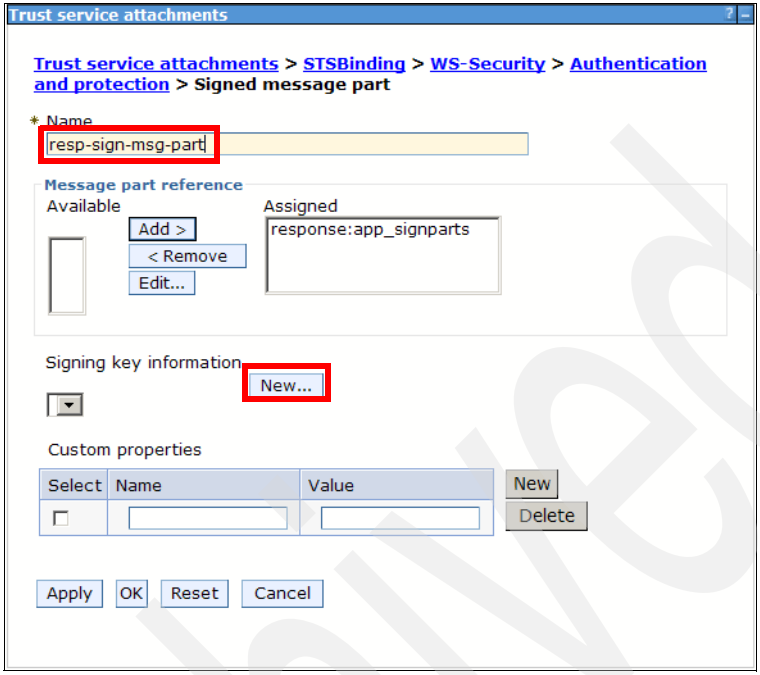


Figure 11-76 New Signing key information

- Type `resp-sign-keyinfo` in **Name** → Select **Security token reference** for the Type → Select **AsymmetricBindingRecipientSignatureToken0** for Token generator or consumer name (Figure 11-77) → **OK** → **Apply**.

Figure 11-77 Provide signing key information

- In the Message part reference tab (see Figure 11-76) → Select **response:app_signparts** → **Edit** → Under Transform algorithms → **New** → Verify the URL `http://www.w3.org/2001/10/xml-exc-c14n#` in a browser (Figure 11-63 on page 475) → Paste into the URL field → **OK** (Figure 11-78) → **OK**.

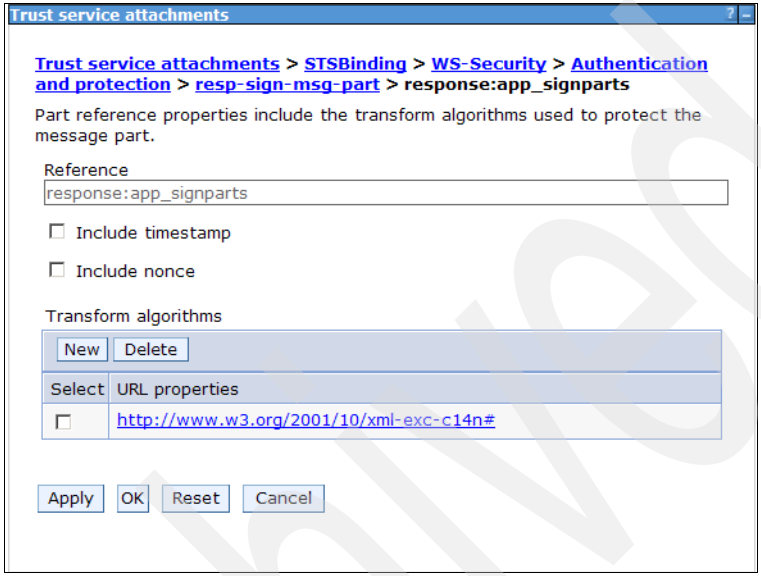


Figure 11-78 Transform algorithm associated with response:app_signparts

Select `resp-sign-keyinfo` under **Signing key information** (Figure 11-79) → **OK** → **Save**.

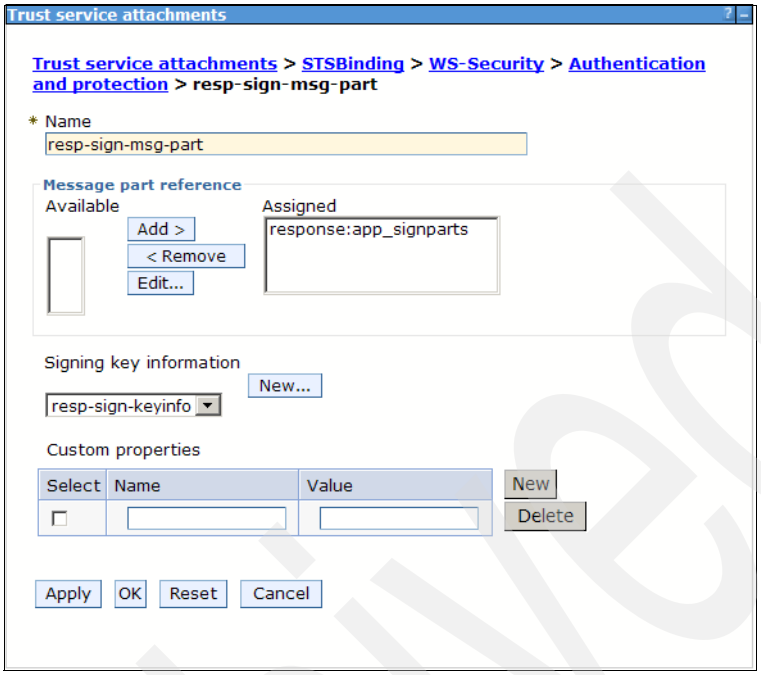


Figure 11-79 Signing key information selected for response signature

STSBinding response encryption generator security binding

- In the Administrator Console navigate to **Services** → **Trust service** → **Trust service attachments** → Select the **STSBinding** attached to the **Issue** token → **WS-Security** → **Authentication and protection** → **AsymmetricBindingInitiatorEncryptionToken0** (Figure 11-80).

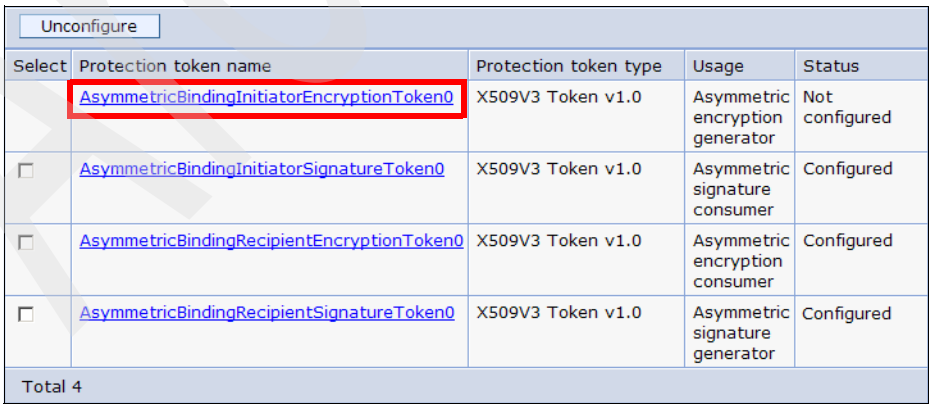


Figure 11-80 Select AsymmetricBindingInitiatorEncryptionToken0

- ☐ Verify that the **JAAS login** is set to `wss.consume.x509` → **Apply** → **Callback handler** → In the **Keystore** section → Select **Custom** as the name → **Custom keystore configuratio**n (Similar to Figure 11-69 on page 480).
- ☐ Paste the **Full path** to the receiver.jks keystore → Type the **Password** `sampleapp` → Type in the **Key** information `cn=alice, ou=myou, o=myco` → **Alias** `alice` → **Password** `sampleapp` (Figure 11-81) as provided in Alice's certificate (Table 11-4 on page 468) → **OK** → **OK** → **OK** → **Save**.

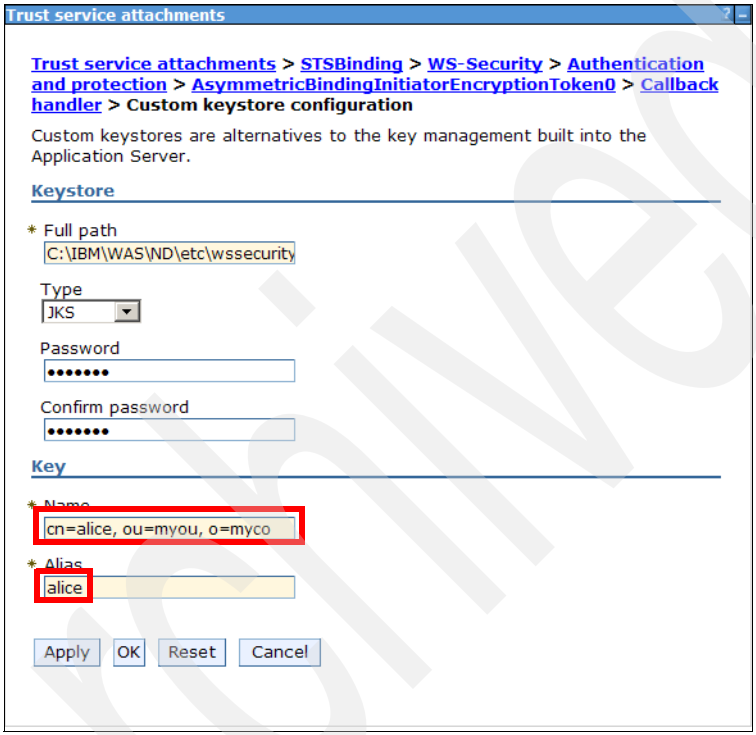


Figure 11-81 Provide Alice's key to encrypt the outgoing response

- ☐ Under **Response message signature and encryption protection** → **request:app signparts** (Figure 11-75 on page 484) → Type `resp-enc-msg-part` in **Name** → **Apply**.
- ☐ Under **Response message signature and encryption protection** → **response:app encparts** (Figure 11-82) → Type `resp-enc-msg-part` in **Name** → **Apply**.

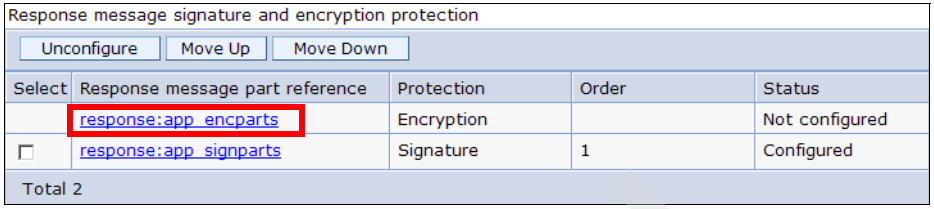


Figure 11-82 Select response:app_signparts

- In the **Key information** part → **New** → (Figure 11-83).

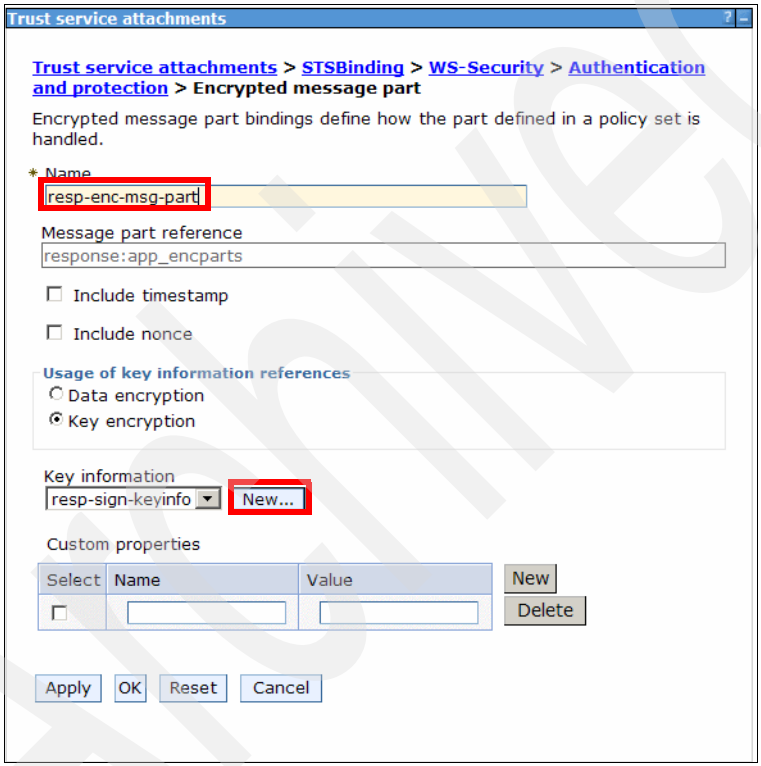


Figure 11-83 New key information for encrypting message part

- Type `resp-enc-keyinfo` in **Name** → Select **Security token reference** for the Type → Select **AsymmetricBindingInitiatorEncryptionToken0** for Token generator or consumer name → **OK** → Select **resp-enc-keyinfo** as the key Information (Figure 11-84) → **Apply** → **OK** → **Save**.

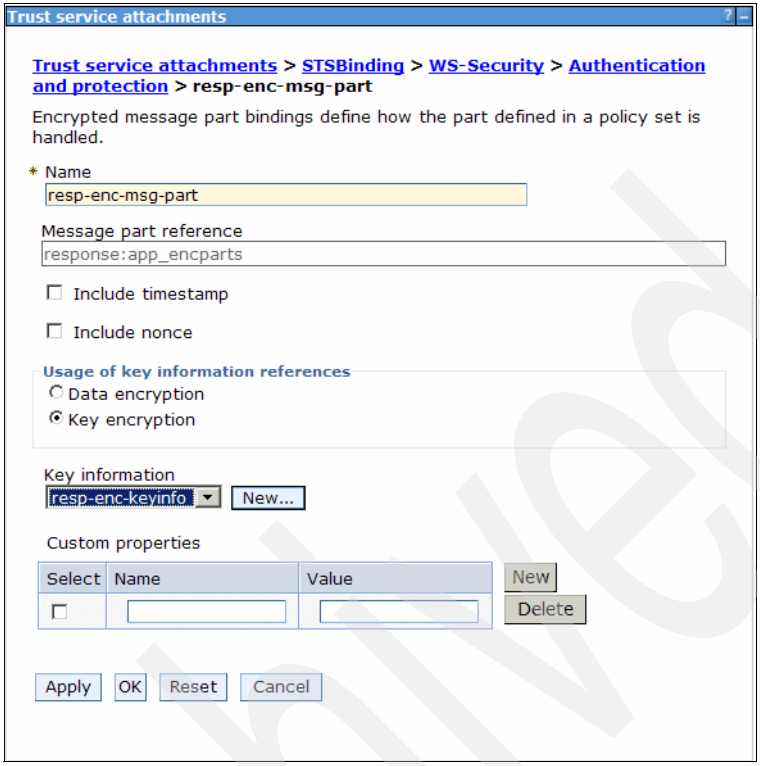


Figure 11-84 Apply resp-enc-keyinfo to resp-enc-msg-part

STSBinding response timestamp expiration security binding

- In the Administrator Console navigate to **Services** → **Trust service** → **Trust service attachments** → Select the **STSBinding** attached to the **Issue** token → **WS-Security** → **Message Expiration** (Figure 11-85).

Figure 11-85 Select Message Expiration

- ☐ Check **Enable message expiration** → Type 5 in the **Message timeout interval** (Figure 11-86) → **OK** → **Save**.

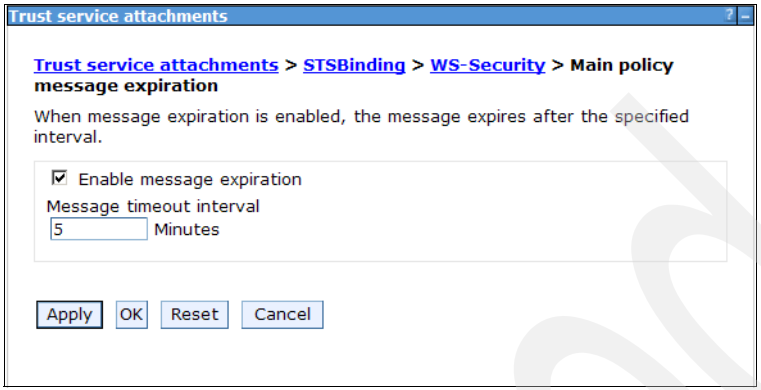


Figure 11-86 Enable message time-out interval

Summary

We have now stepped through configuring the security settings for the STSBinding. There are a couple of places to look review the settings.

The Authentication and protection page shows which custom bindings and tokens required by the Policy set have been configured. (Figure 11-87). Because it does not really show that the configurations are complete, it is best regarded as useful in telling us what parts have definitely not been completed.

Trust service attachments

Trust service attachments > STSBinding > WS-Security > **Authentication and protection**

Configure custom bindings for tokens and message parts that are required by the policy set.

Protection tokens

Unconfigure

Select	Protection token name	Protection token type	Usage	Status
☐	AsymmetricBindingInitiatorEncryptionToken0	X509V3 Token v1.0	Asymmetric encryption generator	Configured
☐	AsymmetricBindingInitiatorSignatureToken0	X509V3 Token v1.0	Asymmetric signature consumer	Configured
☐	AsymmetricBindingRecipientEncryptionToken0	X509V3 Token v1.0	Asymmetric encryption consumer	Configured
☐	AsymmetricBindingRecipientSignatureToken0	X509V3 Token v1.0	Asymmetric signature generator	Configured

Total 4

Authentication tokens

Unconfigure

Select	Security token reference	Authentication token type	Usage	Status
None				

Total 0

Request message signature and encryption protection

Unconfigure

Select	Request message part reference	Protection	Status
☐	request:app_encparts	Encryption	Configured
☐	request:app_signparts	Signature	Configured

Total 2

Response message signature and encryption protection

Unconfigure | Move Up | Move Down

Select	Response message part reference	Protection	Order	Status
☐	response:app_signparts	Signature	1	Configured
☐	response:app_encparts	Encryption	2	Configured

Total 2

Figure 11-87 Configured custom bindings for token and message parts

Another page to look at is the Keys and Certificates page (Figure 11-88). This page lets us review what keys have been configured. We could also use this page to create the configuration for keys before configuring the use of the keys in the various interactions and security functions.

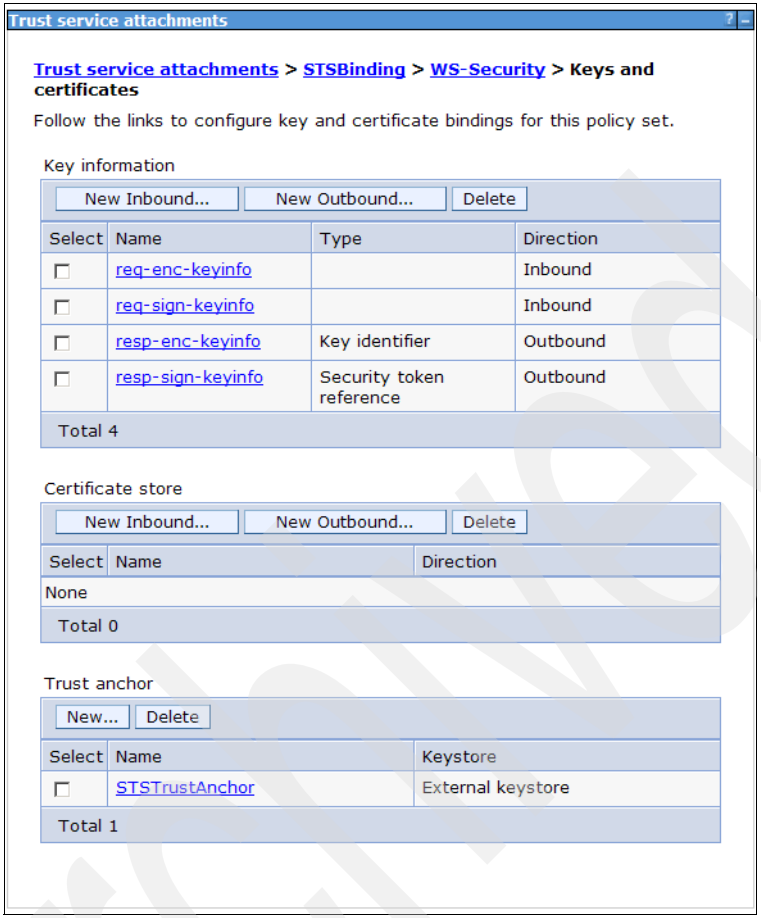


Figure 11-88 Keys and certificates

11.10.2 Configure SCCltBinding for SCCltInterop

To configure the client bindings for the Medical and Credit services, we have two sets of requests and responses to deal with. There are the bootstrap requests sent to the Secure Token Service that we have just configured, and their responses, and also the main set of requests and responses sent to the Credit and Medical check services. Both of these sets of bindings have to be configured (Figure 11-89).

Figure 11-89 Two sets of client bindings to configure

In addition to configuring the WS-Security policy, we also have to configure the WS-ReliableMessaging policy for the credit and medical check clients.

The full list of configurations we have to perform is as follows:

- "SCCltBinding bootstrap request signature security binding" on page 496
- "SCCltBinding bootstrap request encryption security binding" on page 497
- "SCCltBinding bootstrap response signature security binding" on page 498
- "SCCltBinding bootstrap response encryption security binding" on page 499
- "SCCltBinding bootstrap request message expiry binding" on page 500
- "SCCltBinding main request signature security binding" on page 501
- "SCCltBinding main request encryption security binding" on page 502
- "SCCltBinding main response signature security binding" on page 503
- "SCCltBinding main response encryption security binding" on page 504
- "SCCltBinding main request message expiry binding" on page 505
- "SCCltBinding main reliable messaging binding" on page 505

With the exception of the SCCltBinding main reliable messaging binding, we have performed similar configurations before for the SCSBinding, but this time we are configuring a client, so the keystore we use is different. Rather than step through the configurations in detail, we have collated the information for each configuration in a table, which then has to be applied following the equivalent procedure for the SCSBinding.

Tip: Because we have used the same Policy set binding on the MedicalCheck client as the CreditCheck client, *and they are both in the same .ear file*, we only have to define the binding that one time, and then make sure that the binding is attached to the MedicalCheck client. *(Do verify that this is the case; it is well worth the time spent.)*.

SCCltBinding bootstrap request signature security binding

- In the Administrator Console navigate to **Services** → **Service clients** → **CreditCheck** → **SCCltBinding** → **WS-Security** (Figure 11-89 on page 495) → **BootStrap message security policy bindings** → **Authentication and protection** → **AsymmetricBindingInitiatorSignatureToken0** → Complete the dialogs with the data in Table 11-5 → **Save**.

Table 11-5 Attributes and values for AsymmetricBindingInitiatorSignatureToken0

Attribute	Value
JAAS login	wss.generate.x509
Keystore Name	Custom
Keystore path	<%WAS_HOME%>\etc\ws-security\sender.jks
Keystore type	jks
Keystore password	sampleapp
Key	cn=alice,ou=myou,o=myco
Alias	alice
Key password	sampleapp

- In the Administrator Console navigate to **Services** → **Service clients** → **CreditCheck** → **SCCltBinding** → **WS-Security** (Figure 11-89 on page 495) → **BootStrap message security policy bindings** → **Authentication and protection** → **Request message signature and encryption protection** → **request:bootstrap_signparts** → Complete the dialogs with the data in Table 11-6 → **Save**.

Table 11-6 Attributes and values for request:bootstrap_signparts

Attribute	Value
Name	req-sign-msg-part
Signing key information Name	req-sign-keyinfo
Signing key information Type	Security token reference
Signing key information Token generator or consumer name	AsymmetricBindingInitiatorSignatureToken0
Transform algorithm URL	http://www.w3.org/2001/10/xml-exc-c14n#
Assigned message part reference	request:bootstrap-signparts
Signing key information	req-sign-keyinfo

SCCItBinding bootstrap request encryption security binding

- In the Administrator Console navigate to **Services** → **Service clients** → **CreditCheck** → **SCCItBinding** → **WS-Security** (Figure 11-89 on page 495) → **BootStrap message security policy bindings** → **Authentication and protection** → **AsymmetricBindingRecipientEncryptionToken0** → Complete the dialogs with the data in Table 11-7 → **Save**.

Table 11-7 Attributes and values for AsymmetricBindingRecipientEncryptionToken0

Attribute	Value
JAAS login	wss.generate.x509
Keystore Name	Custom
Keystore path	<%WAS_HOME%>\etc\ws-security\sender.jks
Keystore type	jks
Keystore password	sampleapp
Key	cn=bob,ou=myou,o=myco
Alias	bob
Key password	sampleapp

- In the Administrator Console navigate to **Services** → **Service clients** → **CreditCheck** → **SCCItBinding** → **WS-Security** (Figure 11-89 on page 495) → **BootStrap message security policy bindings** → **Authentication and protection** → **Request message signature and encryption protection** → **request:bootstrap_encparts** → Complete the dialogs with the data in Table 11-8 → **Save**.

Table 11-8 Attributes and values for request:bootstrap_encparts

Attribute	Value
Name	req-enc-msg-part
Key information Name	req-enc-keyinfo
Key Information Type	key identifier
Token generator or consumer name	AsymmetricBindingRecipientEncryptionToken0
Message part reference	request:bootstrap-encparts
Key information	req-enc-keyinfo

SCCItBinding bootstrap response signature security binding

- In the Administrator Console navigate to **Services** → **Service clients** → **CreditCheck** → **SCCItBinding** → **WS-Security** (Figure 11-89 on page 495) → **BootStrap message security policy bindings** → **Authentication and protection** → **AsymmetricBindingRecipientSignatureToken0** → Complete the dialogs with the data in Table 11-9 → **Save**.

Table 11-9 Attributes and values for AsymmetricBindingRecipientSignatureToken0

Attribute	Value
JAAS login	wss.generate.x509
Certificates	Trust any certificate
Keystore Name	Custom
Keystore path	<%WAS_HOME%>\etc\ws-security\sender.jks
Keystore type	jks
Keystore password	sampleapp
Key	cn=bob,ou=myou,o=myco
Alias	bob
Key password	sampleapp

- In the Administrator Console navigate to **Services** → **Service clients** → **CreditCheck** → **SCCItBinding** → **WS-Security** (Figure 11-89 on page 495) → **BootStrap message security policy bindings** → **Authentication and protection** → **Response message signature and encryption protection** → **response:bootstrap_signparts** → Complete the dialogs with the data in Table 11-10 → **Save**.

Table 11-10 Attributes and values for response:bootstrap_signparts

Attribute	Value
Name	resp-sign-msg-part
Signing key information Name	resp-sign-keyinfo
Signing key information Token generator or consumer name	AsymmetricBindingRecipientSignatureToken0
Transform algorithm URL	http://www.w3.org/2001/10/xml-exc-c14n#
Assigned message part reference	response:bootstrap-signparts
Assigned signing key information	resp-sign-keyinfo

SCCItBinding bootstrap response encryption security binding

- In the Administrator Console navigate to **Services** → **Service clients** → **CreditCheck** → **SCCItBinding** → **WS-Security** (Figure 11-89 on page 495) → **BootStrap message security policy bindings** → **Authentication and protection** → **AsymmetricBindingInitiatorEncryptionToken0** → Complete the dialogs with the data in Table 11-11 → **Save**.

Table 11-11 Attributes and values for AsymmetricBindingInitiatorEncryptionToken0

Attribute	Value
JAAS login	wss.generate.x509
Keystore Name	Custom
Keystore path	<%WAS_HOME%>\etc\ws-security\sender.jks
Keystore type	jks
Keystore password	sampleapp
Key	cn=alice,ou=myou,o=myco
Alias	alice
Key password	sampleapp

- In the Administrator Console navigate to **Services** → **Service clients** → **CreditCheck** → **SCCItBinding** → **WS-Security** (Figure 11-89 on page 495) → **BootStrap message security policy bindings** → **Authentication and protection** → **Response message signature and encryption protection** → **response:bootstrap_encparts** → Complete the dialogs with the data in Table 11-12 → **Save**.

Table 11-12 Attributes and values for response:bootstrap_encparts

Attribute	Value
Name	resp-enc-msg-part
Key information Name	resp-enc-keyinfo
Token generator or consumer name	AsymmetricBindingInitiatorEncryptionToken0
Message part reference	response:bootstrap-encparts
Assigned key information	resp-enc-keyinfo

SCCItBinding bootstrap request message expiry binding

- In the Administrator Console navigate to **Services** → **Service clients** → **CreditCheck** → **SCCItBinding** → **WS-Security** (Figure 11-89 on page 495) → **BootStrap message security policy bindings** → **Message expiration** → Complete the dialogs with the data in Table 11-13 → **Save**.

Table 11-13 Message expiration

Attribute	Value
Enable message expiration	Checked
Time-out interval	5 minutes

CreditCheck client bootstrap binding summary

The summary configuration panel for the Bootstrap policy authentication and protection is shown in Figure 11-90, and the summary of Bootstrap keys and certificates in Figure 11-91.

Service clients

Service clients > CreditCheck > SCCltBinding > WS-Security > Bootstrap policy authentication and protection

Configure custom bindings for tokens and message parts that are required by the policy set.

Protection tokens

Unconfigure

Select	Protection token name	Protection token type	Usage	Status
☐	AsymmetricBindingInitiatorEncryptionToken0	X509V3 Token v1.0	Asymmetric encryption consumer	Configured
☐	AsymmetricBindingInitiatorSignatureToken0	X509V3 Token v1.0	Asymmetric signature generator	Configured
☐	AsymmetricBindingRecipientEncryptionToken0	X509V3 Token v1.0	Asymmetric encryption generator	Configured
☐	AsymmetricBindingRecipientSignatureToken0	X509V3 Token v1.0	Asymmetric signature consumer	Configured

Total 4

Figure 11-90 CreditCheck client bootstrap policy authentication and protection

Service clients

Service clients > CreditCheck > SCCltBinding > WS-Security > Bootstrap policy keys and certificates

Follow the links to configure key and certificate bindings for this policy set.

Key information

New Inbound... New Outbound... Delete

Select	Name	Type	Direction
☐	req-enc-keyinfo	Key identifier	Outbound
☐	req-sign-keyinfo	Security token reference	Outbound
☐	resp-enc-keyinfo		Inbound
☐	resp-sign-keyinfo		Inbound

Total 4

Figure 11-91 CreditCheck client bootstrap policy keys and certificates

SCCltBinding main request signature security binding

The main policy bindings do not require a keystore: the key is established in the bootstrap interaction. Consequently the configuration is simpler.

- ☐ In the Administrator Console navigate to **Services** → **Service clients** → **CreditCheck** → **SCCltBinding** → **WS-Security** (Figure 11-89 on page 495) → **Main message security policy bindings** → **Authentication and protection** → **SymmetricBindingInitiatorSignatureToken** → Complete the dialogs with the data in Table 11-14 → **Save**.

Table 11-14 Attributes and values for SymmetricBindingInitiatorSignatureToken

Attribute	Value
JAAS login	wss.generate.sct

- ☐ In the Administrator Console navigate to **Services** → **Service clients** → **CreditCheck** → **SCCltBinding** → **WS-Security** (Figure 11-89 on page 495) → **Main message security policy bindings** → **Authentication and protection** → **Request message signature and encryption protection** → **request:app_signparts** → Complete the dialogs with the data in Table 11-15 → **Save**.

Table 11-15 Attributes and values for request:app_signparts

Attribute	Value
Name	sc-req-sign-msg-part
Signing key information Name	sc-req-sign-keyinfo
Signing key information Type	Security token reference
Signing key information Token generator or consumer name	SymmetricBindingInitiatorSignatureToken
Transform algorithm URL	http://www.w3.org/2001/10/xml-exc-c14n#
Assigned message part reference	request:app-signparts
Signing key information	sc-req-sign-keyinfo

SCCltBinding main request encryption security binding

- ☐ In the Administrator Console navigate to **Services** → **Service clients** → **CreditCheck** → **SCCltBinding** → **WS-Security** (Figure 11-89 on page 495) → **Main message security policy bindings** → **Authentication and protection** → **SymmetricBindingRecipientEncryptionToken** → Complete the dialogs with the data in Table 11-16 → **Save**.

Table 11-16 Attributes and values for SymmetricBindingRecipientEncryptionToken

Attribute	Value
JAAS login	wss.generate.sct

- In the Administrator Console navigate to **Services** → **Service clients** → **CreditCheck** → **SCCItBinding** → **WS-Security** (Figure 11-89 on page 495) → **Main message security policy bindings** → **Authentication and protection** → **Request message signature and encryption protection** → **request:app_encparts** → Complete the dialogs with the data in Table 11-17 → **Save**.

Table 11-17 Attributes and values for request:app_encparts

Attribute	Value
Name	sc-req-enc-msg-part
Key information Name	sc-req-enc-keyinfo
Key Information Type	Security token reference
Token generator or consumer name	SymmetricBindingRecipientEncryptionToken
Message part reference	request:app-encparts
Key information	sc-req-enc-keyinfo
Usage of Key information	Data encryption

SCCItBinding main response signature security binding

- In the Administrator Console navigate to **Services** → **Service clients** → **CreditCheck** → **SCCItBinding** → **WS-Security** (Figure 11-89 on page 495) → **Main message security policy bindings** → **Authentication and protection** → **SymmetricBindingRecipientSignatureToken** → Complete the dialogs with the data in Table 11-18 → **Save**.

Table 11-18 Attributes and values for SymmetricBindingRecipientSignatureToken

Attribute	Value
JAAS login	wss.generate.sct

- In the Administrator Console navigate to **Services** → **Service clients** → **CreditCheck** → **SCCltBinding** → **WS-Security** (Figure 11-89 on page 495) → **BootStrap message security policy bindings** → **Authentication and protection** → **Response message signature and encryption protection** → **response:app_signparts** → complete the dialogs with the data in Table 11-19 → **Save**.

Table 11-19 Attributes and values for response:app_signparts

Attribute	Value
Name	sc-resp-sign-msg-part
Signing key information Name	sc-resp-sign-keyinfo
Signing key information Token generator or consumer name	SymmetricBindingRecipientSignatureToken
Transform algorithm URL	http://www.w3.org/2001/10/xml-exc-c14n#
Assigned message part reference	response:app-signparts
Assigned signing key information	sc-resp-sign-keyinfo

SCCltBinding main response encryption security binding

- In the Administrator Console navigate to **Services** → **Service clients** → **CreditCheck** → **SCCltBinding** → **WS-Security** (Figure 11-89 on page 495) → **Main message security policy bindings** → **Authentication and protection** → **SymmetricBindingInitiatorEncryptionToken** → complete the dialogs with the data in Table 11-20 → **Save**.

Table 11-20 Attributes and values for SymmetricBindingInitiatorEncryptionToken

Attribute	Value
JAAS login	wss.generate.sct

- In the Administrator Console navigate to **Services** → **Service clients** → **CreditCheck** → **SCCltBinding** → **WS-Security** (Figure 11-89 on page 495) → **Main message security policy bindings** → **Authentication and protection** → **Response message signature and encryption protection** → **response:app_encparts** → complete the dialogs with the data in Table 11-21 → **Save**.

Table 11-21 Attributes and values for response:app_encparts

Attribute	Value
Name	sc-resp-enc-msg-part
Key information Name	sc-resp-enc-keyinfo
Key information Type	Security token reference
Token generator or consumer name	SymmetricBindingInitiatorEncryptionToken
Message part reference	response:app-encparts
Assigned key information	sc-resp-enc-keyinfo
Usage of key information references	Data encryption

SCCItBinding main request message expiry binding

- In the Administrator Console navigate to **Services** → **Service clients** → **CreditCheck** → **SCCItBinding** → **WS-Security** (Figure 11-89 on page 495) → **Main message security policy bindings** → **Message expiration** → complete the dialogs with the data in Table 11-22 → **Save**.

Table 11-22 Message expiration

Attribute	Value
Enable message expiration	Checked
Time-out interval	5 minutes

SCCItBinding main reliable messaging binding

- In the Administrator Console navigate to **Services** → **Service clients** → **CreditCheck** → **SCCItBinding** → **WS-ReliableMessaging** () → Select **rmBus** for the Bus name → The Messaging engine is automatically selected if there is only one of them, otherwise select the messaging engine (for example, `Node01.server1-rmBus`) → **OK** → **Save**.

CreditCheck client main policy bindings summary

The summary configuration panel for the Bootstrap policy authentication and protection is shown in Figure 11-93, and the summary of Bootstrap keys and certificates in Figure 11-92.

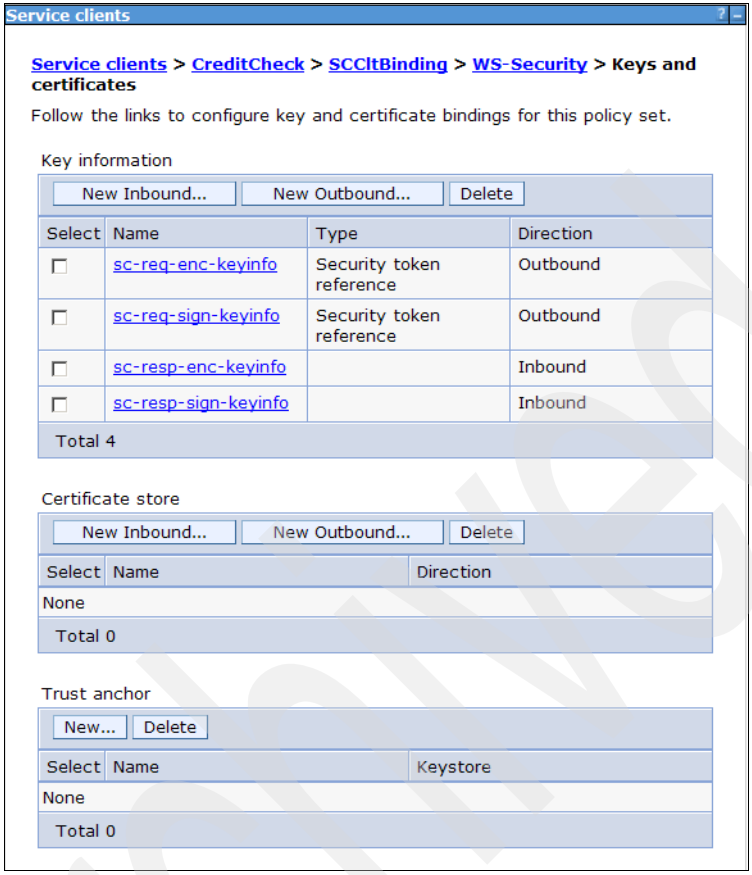


Figure 11-92 CreditCheck client main policy keys and certificates

Notice the two significant differences between the bootstrap security binding configuration and the main policy security binding configuration:

1. The bootstrap policy uses asymmetric encryption algorithms, whereas the main policy uses symmetric algorithms for better performance.
2. The bootstrap policy uses a keystore (sender.jks) to obtain keys and certificates for encryption and decryption, whereas the main policy uses keys encrypted in the message body established by the bootstrap procedure.
   a. Hence sc-req-enc-keyinfo in the main policy binding points to a security token, whereas req-enc-keyinfo in the bootstrap binding points to a key identifier.

b. And also the encryption part of the binding in the main policy security binding uses the key information to encrypt data, whereas in the bootstrap policy security binding the key information is used to encrypt a key (encrypting the key used in the symmetric algorithm in the main policy).

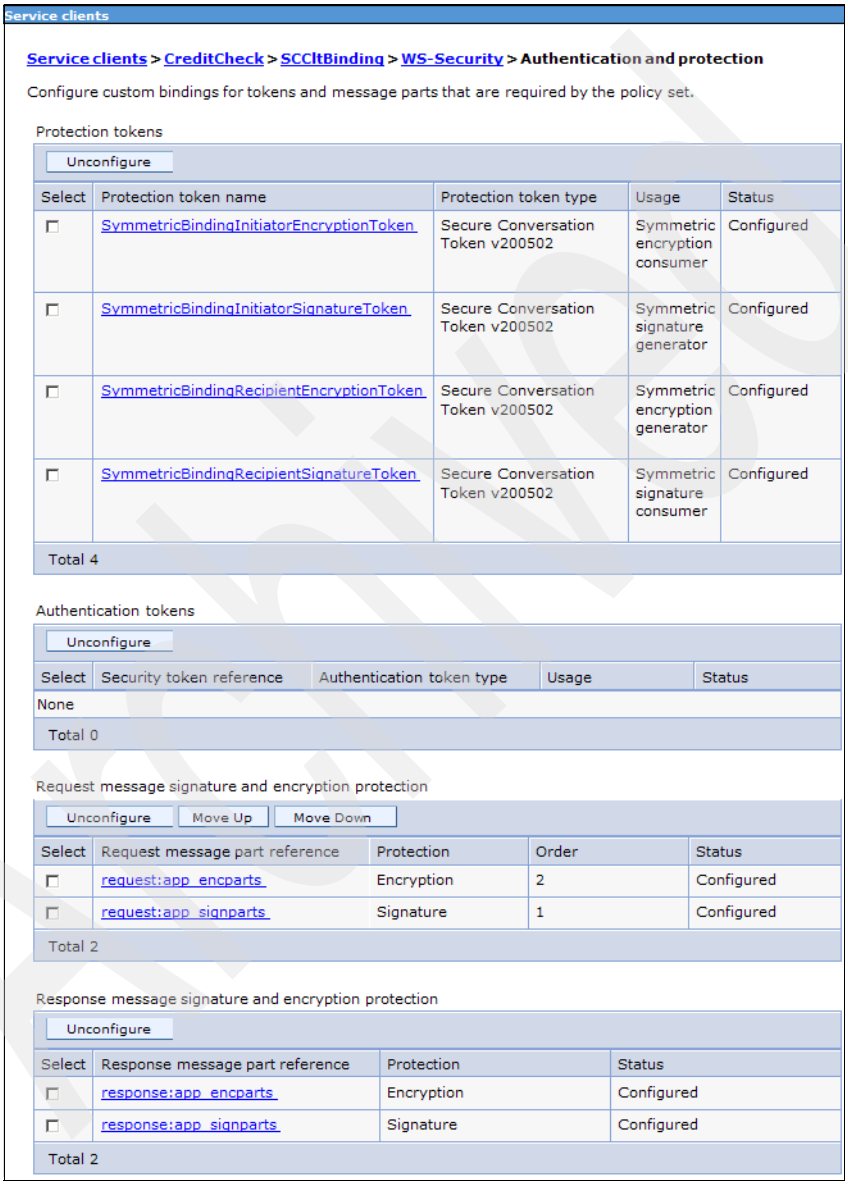


Figure 11-93 CreditCheck client main policy authentication and protection

11.10.3 Configure SCSvcBinding for SCSvcInterop

There are fewer SCSvcBinding bindings for the SCSvcInterop Policy set to configure than bindings for the clients, as there is no bootstrap to configure. Because the configuration procedure is so similar to the client, we provide tables of attributes and values to configure rather than working through the procedures step by step. The bindings to be configured are:

- "SCSvcBinding main request signature security binding" on page 508
- "SCSvcBinding main request encryption security binding" on page 509
- "SCSvcBinding main response signature security binding" on page 510
- "SCSvcBinding main response encryption security binding" on page 511
- "SCSvcBinding main request message expiry binding" on page 511
- "SCSvcBinding main reliable messaging binding" on page 512

But there are two problems:

- Default key lengths used by WebSphere Application Server and Windows Communication Foundation are different. We show how to customize the key lengths in WebSphere Application Server to fix this interoperability problem in 11.10.5, "Configure compatible key sizes" on page 514.
- Because the MedicalCheck provider is in a different .ear file to the CreditCheck provider, and there is no way to export the bindings from one .ear file and import them into another, there is no "supported" way of avoiding redoing all the bindings a second time. Instead, we provide a work around in 11.10.6, "Migrate SCSvcBinding to the MedicalCheck provider" on page 516, to simplify the set up for this example, without endorsing it as a procedure to use on production systems.

SCSvcBinding main request signature security binding

The main policy bindings do not require a keystore: the key is established in the bootstrap interaction. Consequently the configuration is simpler.

☐ In the Administrator Console navigate to **Services** → **Service providers** → **CreditCheck** → **SCSvcBinding** → **WS-Security** (Figure 11-89 on page 495) → **Main message security policy bindings** → **Authentication and protection** → **SymmetricBindingInitiatorSignatureToken** → Complete the dialogs with the data in Table 11-14 → **Save**.

Table 11-23 Attributes and values for SymmetricBindingInitiatorSignatureToken

Attribute	Value
JAAS login	wss.generate.sct

- In the Administrator Console navigate to **Services** → **Service providers** → **CreditCheck** → **SCSvcBinding** → **WS-Security** (Figure 11-89 on page 495) → **Main message security policy bindings** → **Authentication and protection** → **Request message signature and encryption protection** → **request:app_signparts** → Complete the dialogs with the data in Table 11-15 → **Save**.

Table 11-24 Attributes and values for request:app_signparts

Attribute	Value
Name	sc-req-sign-msg-part
Signing key information Name	sc-req-sign-keyinfo
Signing key information Type	Security token reference
Signing key information Token generator or consumer name	SymmetricBindingInitiatorSignatureToken
Transform algorithm URL	http://www.w3.org/2001/10/xml-exc-c14n#
Assigned message part reference	request:app-signparts
Signing key information	sc-req-sign-keyinfo

SCSvcBinding main request encryption security binding

- In the Administrator Console navigate to **Services** → **Service providers** → **CreditCheck** → **SCSvcBinding** → **WS-Security** → **Main message security policy bindings** → **Authentication and protection** → **SymmetricBindingRecipientEncryptionToken** → Complete the dialogs with the data in Table 11-25 → **Save**.

Table 11-25 Attributes and values for SymmetricBindingRecipientEncryptionToken

Attribute	Value
JAAS login	wss.generate.sct

- In the Administrator Console navigate to **Services** → **Service providers** → **CreditCheck** → **SCSvcBinding** → **WS-Security** → **Main message security policy bindings** → **Authentication and protection** → **Request message signature and encryption protection** → **request:app_encparts** → Complete the dialogs with the data in Table 11-26 → **Save**.

Table 11-26 Attributes and values for request:app_encparts

Attribute	Value
Name	sc-req-enc-msg-part
Key information Name	sc-req-enc-keyinfo
Key Information Type	Security token reference
Token generator or consumer name	SymmetricBindingRecipientEncryptionToken
Message part reference	request:app-encparts
Key information	sc-req-enc-keyinfo
Usage of Key information	Data encryption

SCSvcBinding main response signature security binding

- In the Administrator Console navigate to **Services** → **Service providers** → **CreditCheck** → **SCSvcBinding** → **WS-Security** → **Main message security policy bindings** → **Authentication and protection** → **SymmetricBindingRecipientSignatureToken** → Complete the dialogs with the data in Table 11-27 → **Save**.

Table 11-27 Attributes and values for SymmetricBindingRecipientSignatureToken

Attribute	Value
JAAS login	wss.generate.sct

- In the Administrator Console navigate to **Services** → **Service providers** → **CreditCheck** → **SCSvcBinding** → **WS-Security** → **Main message security policy bindings** → **Authentication and protection** → **Response message signature and encryption protection** → **response:app_signparts** → Complete the dialogs with the data in Table 11-28 → **Save**.

Table 11-28 Attributes and values for response:app_signparts

Attribute	Value
Name	sc-resp-sign-msg-part
Signing key information Name	sc-resp-sign-keyinfo
Signing key information Token generator or consumer name	SymmetricBindingRecipientSignatureToken
Transform algorithm URL	http://www.w3.org/2001/10/xml-exc-c14n#

Attribute	Value
Assigned message part reference	response:app-signparts
Assigned signing key information	sc-resp-sign-keyinfo

SCSvcBinding main response encryption security binding

- In the Administrator Console navigate to **Services** → **Service providers** → **CreditCheck** → **SCSvcBinding** → **WS-Security** → **Main message security policy bindings** → **Authentication and protection** → **SymmetricBindingInitiatorEncryptionToken** → Complete the dialogs with the data in Table 11-29 → **Save**.

Table 11-29 Attributes and values for AsymmetricBindingInitiatorEncryptionToken0

Attribute	Value
JAAS login	wss.generate.sct

- In the Administrator Console navigate to **Services** → **Service providers** → **CreditCheck** → **SCSvcBinding** → **WS-Security** → **Main message security policy bindings** → **Authentication and protection** → **Response message signature and encryption protection** → **response:app_encparts** → Complete the dialogs with the data in Table 11-30 → **Save**.

Table 11-30 Attributes and values for response:app_encparts

Attribute	Value
Name	sc-resp-enc-msg-part
Key information Name	sc-resp-enc-keyinfo
Key information Type	Security token reference
Token generator or consumer name	SymmetricBindingInitiatorEncryptionToken
Message part reference	response:app-encparts
Assigned key information	sc-resp-enc-keyinfo
Usage of key information references	Data encryption

SCSvcBinding main request message expiry binding

- In the Administrator Console navigate to **Services** → **Service providers** → **CreditCheck** → **SCSvcBinding** → **WS-Security** → **Main message security policy bindings** → **Message expiration** → Complete the dialogs with the data in Table 11-31 → **Save**.

Table 11-31 Message expiration

Attribute	Value
Enable message expiration	Checked
Time-out interval	5 minutes

SCSvcBinding main reliable messaging binding

- In the Administrator Console navigate to **Services** → **Service providers** → **CreditCheck** → **SCSvcBinding** → **WS-ReliableMessaging** () → Select **rmBus** for the Bus name → The Messaging engine is automatically selected if there is only one of them, otherwise select the messaging engine (for example, `Node01.server1-rmBus`) → **OK** → **Save** .

CreditCheck service SCSvcBinding summary

The CreditCheck service keys and certificates summary is shown in Figure 11-94, and the CreditCheck service authentication and protection bindings summary in Figure 11-95.

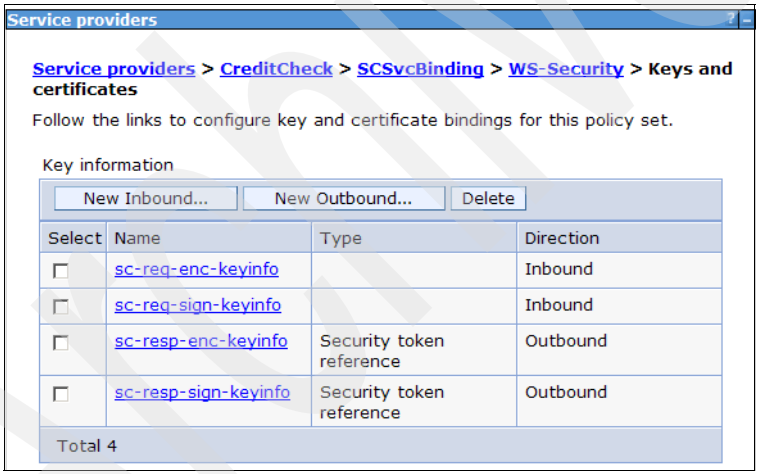


Figure 11-94 CreditCheck service keys and certificate bindings

Service providers

Service providers > CreditCheck > SCSvcBinding > WS-Security > **Authentication and protection**

Configure custom bindings for tokens and message parts that are required by the policy set.

Protection tokens

Unconfigure

Select	Protection token name	Protection token type	Usage	Status
☐	SymmetricBindingInitiatorEncryptionToken	Secure Conversation Token v200502	Symmetric encryption generator	Configured
☐	SymmetricBindingInitiatorSignatureToken	Secure Conversation Token v200502	Symmetric signature consumer	Configured
☐	SymmetricBindingRecipientEncryptionToken	Secure Conversation Token v200502	Symmetric encryption consumer	Configured
☐	SymmetricBindingRecipientSignatureToken	Secure Conversation Token v200502	Symmetric signature generator	Configured

Total 4

Authentication tokens

Unconfigure

Select	Security token reference	Authentication token type	Usage	Status
None				

Total 0

Request message signature and encryption protection

Unconfigure

Select	Request message part reference	Protection	Status
☐	request:app_encparts	Encryption	Configured
☐	request:app_signparts	Signature	Configured

Total 2

Response message signature and encryption protection

Unconfigure Move Up Move Down

Select	Response message part reference	Protection	Order	Status
☐	response:app_signparts	Signature	1	Configured
☐	response:app_encparts	Encryption	2	Configured

Total 2

Figure 11-95 CreditCheck service authentication and protection bindings summary

11.10.4 Configure default binding for WS-ReliableMessaging

We configure the default policy bindings for the ITSO WSRM 10 Persistent policy so that we do not have create a separate binding for each client or service.

- In the left hand navigation pane in the **Administrator Console** select **Service** → **Policy Sets** → **Default Policy Set Bindings**
- Click **WS-ReliableMessaging** → Select **rmBus** → Select server messaging engine from the Messaging engine drop-down (for example, *Node02.server1-rmBus* (Figure 11-96) → **OK** → **Save**.

Figure 11-96 Bind the default WS-ReliableMessaging policy rmBus

11.10.5 Configure compatible key sizes

WebSphere Application Server v6.1 with the Feature pack for Web services 6.1.0.13 and Windows Communication Foundation v3.0 use a different key size for the signing key. WebSphere Application Server uses a key size of 20 bytes, and Windows Communication Foundation, 16 bytes. The key sizes must match.

- In the Administrator Console navigate to **Services** → **Service providers** → **CreditCheck** → **SCSvcBinding** → **WS-Security** → **Main message security policy bindings** → **Authentication and protection** → **SymmetricBindingRecipientSignatureToken** → Under **Custom properties** type `com.ibm.wsspi.wssecurity.DerivedkeyToken.KeyLengthBytes` for the **Name** → `16` for the **Value** → **OK** → **Save** (Figure 11-97).

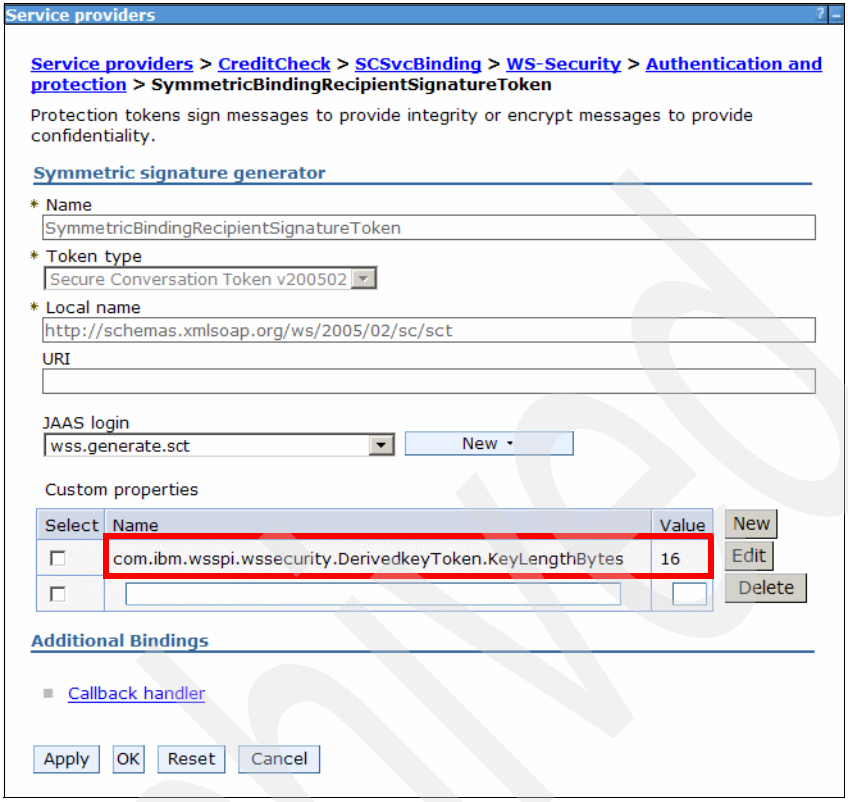


Figure 11-97 Setting the key length to 16 for the CreditCheck service provider

- In the Administrator Console navigate to **Services** → **Service clients** → **CreditCheck** → **SCCltBinding** → **WS-Security** → **Main message security policy bindings** → **Authentication and protection** → **SymmetricBindingInitiatorSignatureToken** → Under **Custom properties** type `com.ibm.wsspi.wssecurity.DerivedkeyToken.KeyLengthBytes` for the **Name** → `16` for the **Value** → **OK** → **Save** (Figure 11-98).

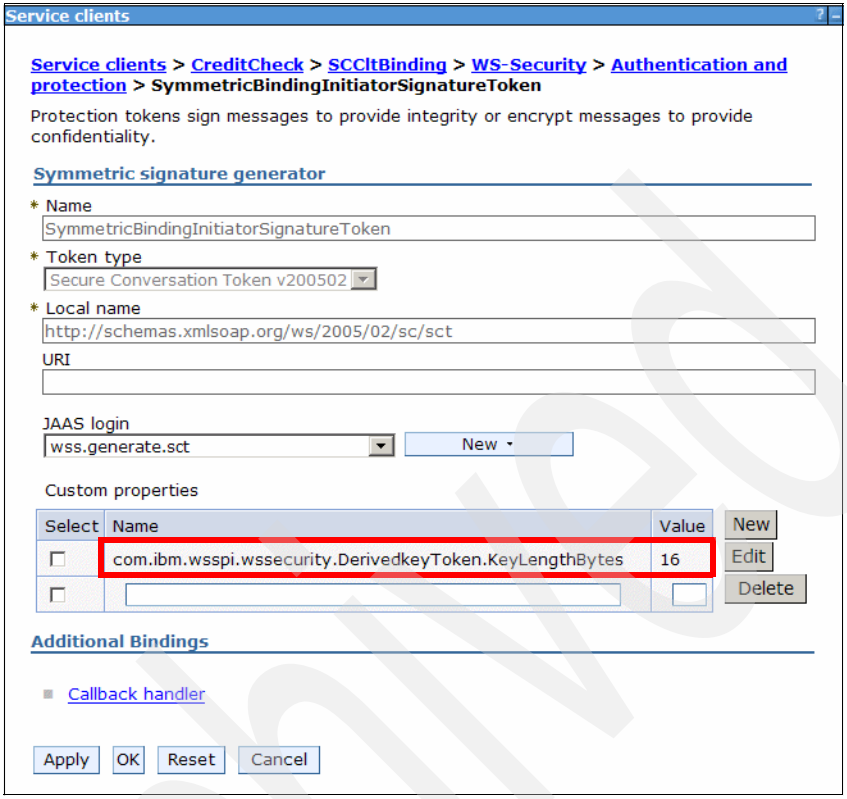


Figure 11-98 Setting key length to 16 in CreditCheck client

11.10.6 Migrate SCSvcBinding to the MedicalCheck provider

Rather than configure the SCSvsBinding bindings for the MedicalCheck provider manually, we copy them from the CreditCheck provider as they are identical. There is no supplied way to do this, so we have to stop WebSphere Application Server, and copy the configuration files in the file system, and then restart WebSphere Application Server.

- ☐ Verify **Services** → **Service providers** → **MedicalCheck** → **SCSvcBinding** → **WS-Security** → **Main message security policy bindings** → **Authentication and protection** is unconfigured.
- ☐ Stop WebSphere Application Server.

- Copy the folder:

  `<WAS_INSTALL_ROOT>\profiles\<PROFILE>\config\cells\<CELLNAME>\applications\CreditCheckEAR.ear\deployments\CreditCheckEAR\META-INF\SCSvcBinding`

  to:

  `<WAS_INSTALL_ROOT>\profiles\<PROFILE>\config\cells\<CELLNAME>\applications\MedicalCheckEAR.ear\deployments\MedicalCheckEAR\META-INF\SCSvcBinding`

  Thereby overwriting the contents of the existing folders.
- Delete the contents of `<WAS_INSTALL_ROOT>\profiles\<PROFILE>\wstemp` and `<WAS_INSTALL_ROOT>\profiles\<PROFILE>\temp`
- Restart WebSphere Application Server
- Verify **Services** → **Service providers** → **MedicalCheck** → **SCSvcBinding** → **WS-Security** → **Main message security policy bindings** → **Authentication and protection** is configured

11.11 Validate WebSphere Application Server sample

Validate the secured deployment of the insurance agent application works correctly on WebSphere Application Server following the steps in 11.6.1, “Deploy application to WebSphere Application Server” on page 429 before distributing it to Windows Communication Foundation.

If you have skipped the configuration of the Insurance agents on WebSphere Application Server, or want to study a running configuration, we have provided a Quickstart configuration procedure using zipped up directories from our profile of WebSphere Application Server.

11.11.1 Quickstart configuration procedure

For the Quickstart procedure, all applications running on WebSphere Application Server are installed manually using the Administrator Console. We do not run the InsuranceAgents client using Rational Application Developer.

> **Note:** This procedure is not recommended for applying QoS’s in a production deployment. The steps listed in this section are provided to apply RSP and WS-RM policy sets and policy bindings to the sample application quickly. In a production environment, policy sets and policy bindings should be created using the Integrated Solutions Console, or the wsadmin scripting tool.

- Install the MedicalCheck, CreditCheck and Underwriter .ear files on WebSphere Application Server following the procedure in 11.8.1, “Reinstall applications on WebSphere Application Server” on page 455.
- Install InsuranceAgents.ear on WebSphere Application Server from the Administrator Console.
- Configure the rmBus using the procedure in 11.9.1, “Create an SI bus to store and manage WS-RM messages” on page 465.
- Log out of the Integrated Solutions Console → Stop the server and backup your WebSphere profile.
- Copy the `ITSO7618\Insurance\mySysKeys` folder contents into the `<%WAS_HOME%>/etc/ws-security` folder..
- Unzip `sts.zip` and `PolicySets.zip` from `ITSO7618\Insurance\WSFP\Policies and Bindings\Quickstart` into `<%WAS_HOME%>/profiles/<PROFILE>/config/cells/<CELL>`
  - The PolicySets folder tree adds its content to the PolicySets folder.
  - The sts folder tree updates content in the sts folder.(Figure 11-99)

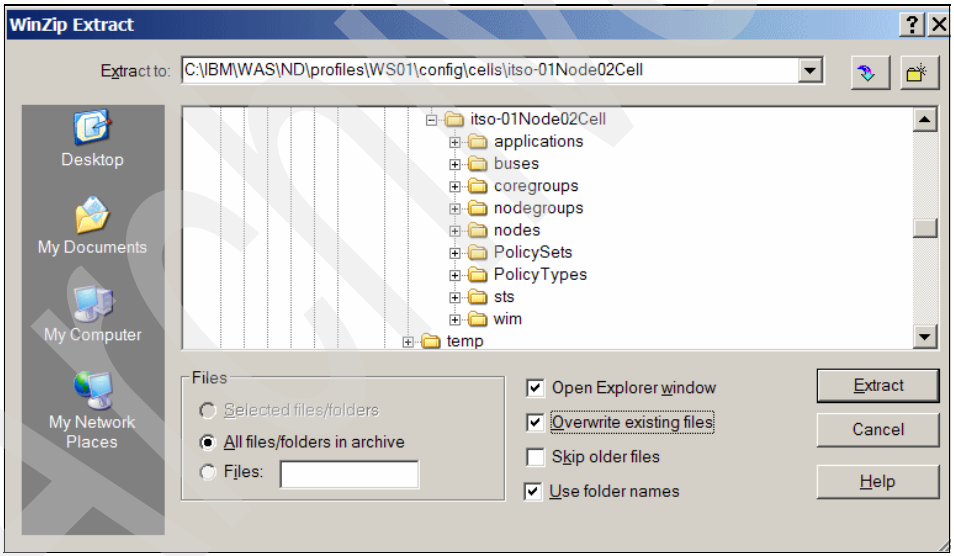


Figure 11-99 Unzipping PolicySets.zip into the correct location

- Unzip the `CreditCheck.zip`, `MedicalCheck.zip`, `InsuranceAgent.zip` and `UnderWriter.zip` files into `<%WAS_HOME%>/profiles/<PROFILE>/config/cells/<CELL>`
  - The CreditCheckEAR.ear folder tree updates content in the CreditCheckEAR.ear folder.

- The MedicalCheckEAR.ear folder tree updates content in the MedicalCheckEAR.ear folder.
- The UnderwriterEAR.ear folder tree updates content in the Underwriter folder.
- The InsuranceAgentEAR.ear folder tree updates content in the InsuranceAgentEAR.ear folder.

Attention: There are long paths and filenames in InsuranceAgent.zip, the symptoms are:

- Requesting a password.
- Copying the unzipped directory produces an error “filename too long” or “path too long”.

Remedies:

- It is tricky to delete a directory with this error. Try moving the directory to the drive root and then deleting (we had to reboot for this to finally work).
- Unzip to a root directory and then move the directory tree.
- Use the **`subst`** command to map a long path to a drive letter.

☐ Ensure that the two binding files include correct path directory where you placed the contents of the mySysKeys folder on your system.

There are four path entries in each file.

```
<%WAS_HOME%>/profiles/<PROFILE>/config/cells/<CELL>/sts/policy/
STSBinding/PolicyTypes/WSSecurity/binding.xml
```

- ☐ Trust services → STSBinding Outbound (Figure 11-100 on page 520)
- ☐ Trust services → STSBinding Inbound (Figure 11-101 on page 521)

```
<%WAS_HOME%>/profiles/<PROFILE>/config/cells/<CELL>/applications/
UnderWriterEAR.ear/deployments/UnderWriterEAR/META-INF/SCCltBinding/
PolicyTypes/WSSecurity/binding.xml
```

- ☐ Underwriter client → SCCltBinding Outbound (Figure 11-102 on page 522)
- ☐ Underwriter client → SCCltBinding Inbound (Figure 11-103 on page 523)

Note: The STSBindings are for the application policy, and the Underwriter client for the bootstrap policy — highlighted in the figures.

We imported the filesystem containing the two binding files into Rational Application Developer to edit the XML using the Rational Application Developer XML editor and then exported the edited files back to the server.

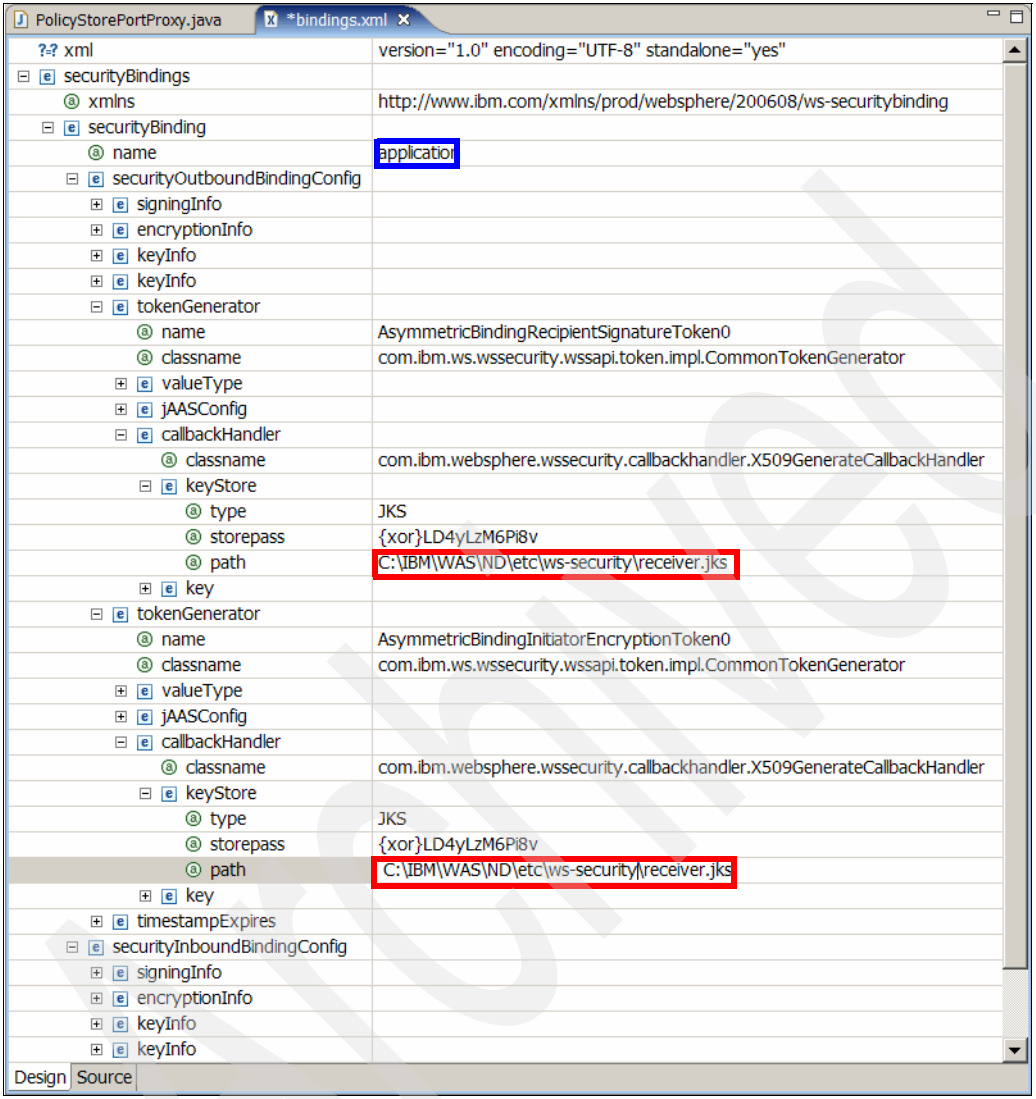


Figure 11-100 Trust services → STSBinding Outbound

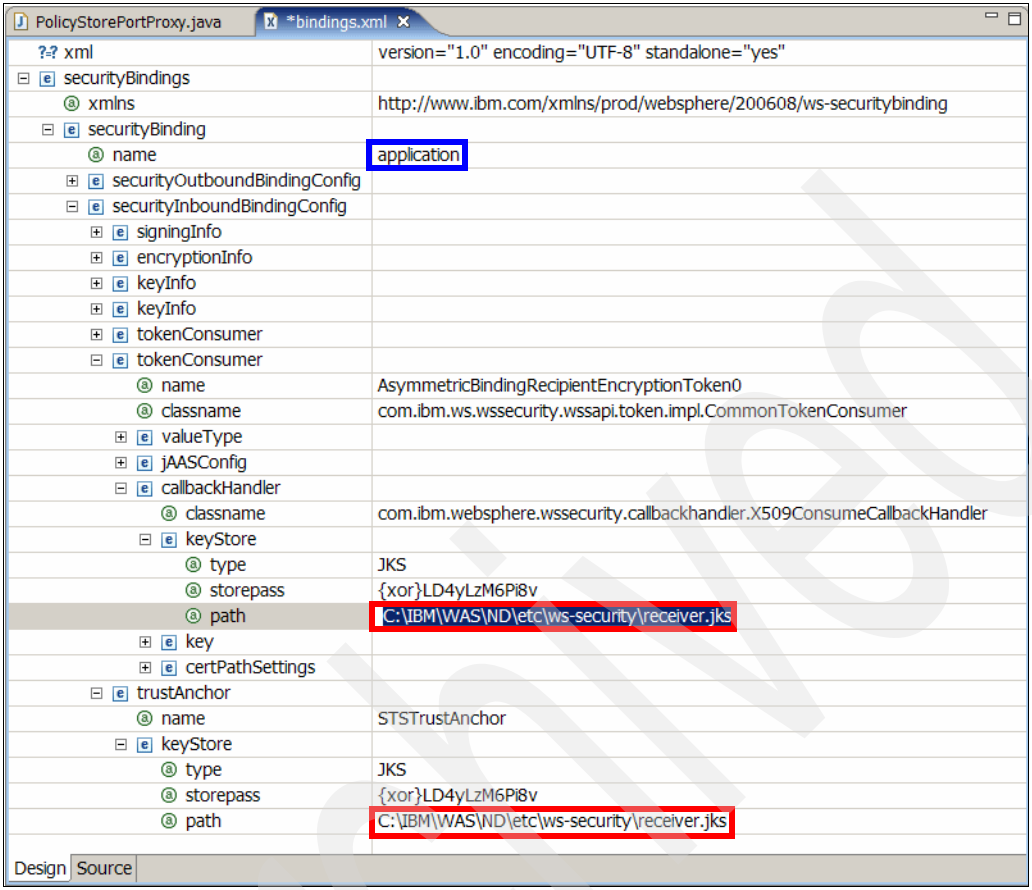


Figure 11-101 Trust services → STSBinding Inbound

*bindings.xml	
?-? xml	version="1.0" encoding="UTF-8" standalone="yes"
securityBindings	
xmlns	http://www.ibm.com/xmlns/prod/websphere/200608/ws-securitybinding
securityBinding	
securityBinding	
name	bootstrap
securityOutboundBindingConfig	
signingInfo	
encryptionInfo	
keyInfo	
keyInfo	
tokenGenerator	
name	AsymmetricBindingInitiatorSignatureToken0
classname	com.ibm.ws.wssecurity.wssapi.token.impl.CommonTokenGenerator
valueType	
jAASConfig	
callbackHandler	
classname	com.ibm.websphere.wssecurity.callbackhandler.X509GenerateCallbackHandler
keyStore	
type	JKS
storepass	{xor}LD4yLzM6Pi8v
path	C:\IBM\WAS\ND\etc\ws-security\sender.jks
key	
tokenGenerator	
name	AsymmetricBindingRecipientEncryptionToken0
classname	com.ibm.ws.wssecurity.wssapi.token.impl.CommonTokenGenerator
valueType	
jAASConfig	
callbackHandler	
classname	com.ibm.websphere.wssecurity.callbackhandler.X509GenerateCallbackHandler
keyStore	
type	JKS
storepass	{xor}LD4yLzM6Pi8v
path	C:\IBM\WAS\ND\etc\ws-security\sender.jks
key	
timestampExpires	
securityInboundBindingConfig	

Design Source

Figure 11-102 Underwriter client → SCCltBinding Outbound

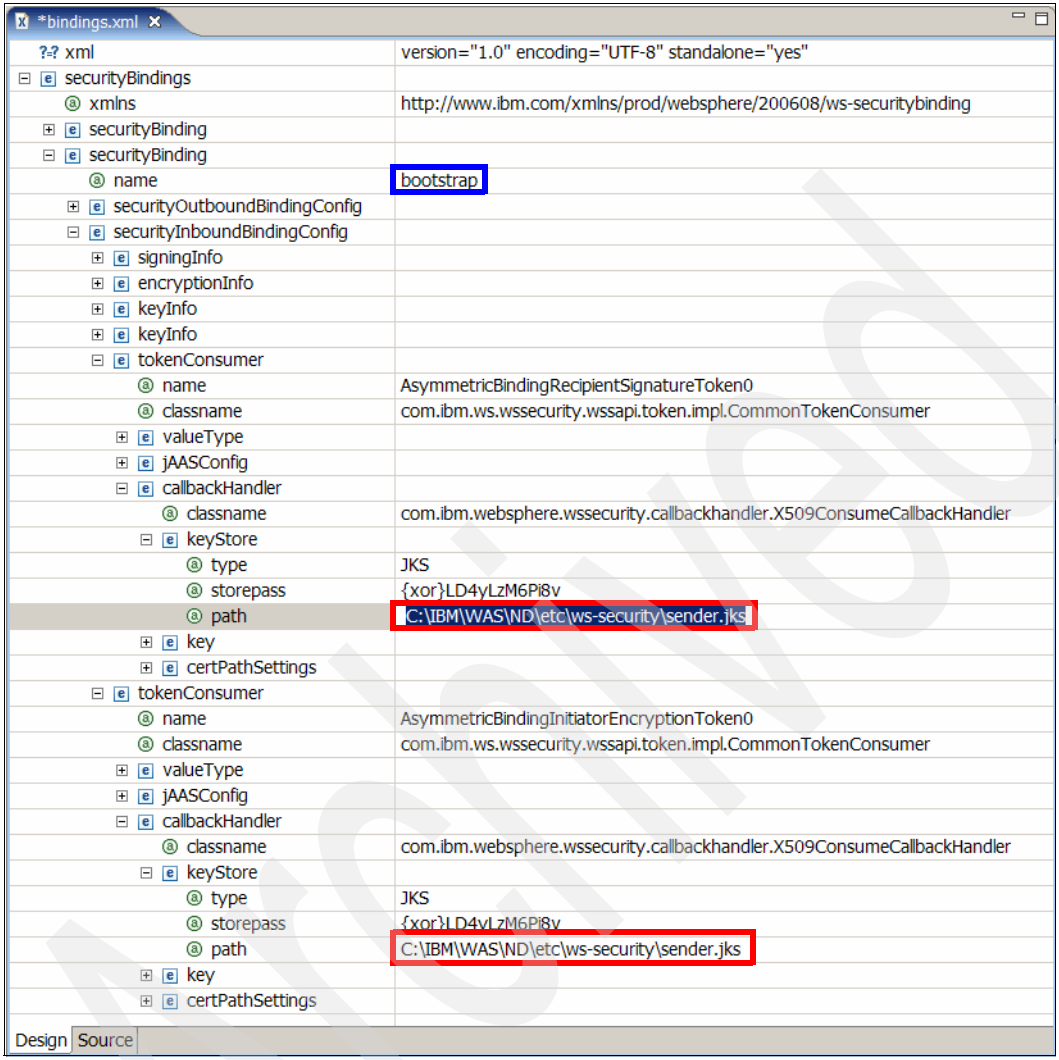


Figure 11-103 Underwriter client → SCCltBinding Inbound

- Start the server.
- Configure the `SCSvcBinding` provider and `SCCltBinding` client bindings for MedicalCheck and CreditCheck to include the newly created WS-ReliableMessaging bus and messaging engine.

> **Note:** There are separate bindings for the services in the two different .ear files, but only one binding for the two clients, as they are in the same .ear file.

The instructions for these are in the following sections:

- "SCCltBinding main reliable messaging binding" on page 505
- "SCSvcBinding main reliable messaging binding" on page 512.

- Configure the WSRM 10 Persistent Binding for the InsuranceAgent Service client, and the PolicyStore Service, and the WSRM 10 Persistent Binding2 for the Underwriter service to include the newly created WS-ReliableMessaging bus and messaging engine.
- If you have not already done so, configure the default WS-ReliableMessaging binding as described in "Configure default binding for WS-ReliableMessaging" on page 514

11.11.2 Validate the WAS RSP configuration

Running the verification test against the secure configuration of the Insurance Underwriter solution is essentially the same as described in 11.6.1, "Deploy application to WebSphere Application Server" on page 429. There are two differences in how the clients are run if you used the QuickStart configuration, which does not use Rational Application Developer to run the clients.

- Start the Insurance Agents' GUI from this URL:

  `http://localhost:9081/InsuranceAgents/sampleUnderWriterSOAPProxy/TestClient.jsp`

- Run the PolicyStoreThinClient.jar from its own directory in a command window using the command:

  ```
  java -classpath <%WAS_HOME%>com.ibm.jaxws.thinclient_6.1.0.jar;.;PolicyStoreThinClient.jar org.example.policystore.PolicyStoreJavaClient
  ```

  Use the PolicyStoreThinClient in the ReliablePolicyStoreThinClient directory.

> **Note:** The **ReliablePolicyStoreThinClient** directory has a jar file exported from Rational Application Developer that includes the WS-ReliableMessaging_10.

11.12 Configure and validate Windows sample

Next we describe the quality of service parameters to apply to the default deployment of the insurance solution to Windows Communication Foundation we made in 11.6.2, “Deploy application to Windows Communication Foundation” on page 434. We configure the secure and reliable services using values that are compatible with the configuration on WebSphere Application Server.

Table 11-32 on page 526 and Table 11-33 on page 527 show the configuration information. There are three different bindings defined in Table 11-32 called:

- reliableBinding
- reliableMtomBinding
- RSPMtomBinding

The bindings are cumulative, in the sense that the attributes on the simpler bindings are also applied to the more complicated bindings, and just for the purposes of the table, the duplicate rows have been omitted.

Table 11-32 Bindings and customizations

Binding	Extension	Attribute	Value
reliable Binding	reliable Session	Ordered	False
reliable Mtom Binding	mtomMessage Encoding	MaxBufferSize	10000000
	httpTransport	MaxBufferPoolSize	1000000
		MaxBufferSize	10000000
		MaxReceivedMessageSize	10000000
RSP Mtom Binding	security	AuthenticationMode	SecureConversation
		DefaultAlgorithmSuite	Basic128Rsa15
		MessageProtectionOrder	SignBeforeEncryptAndEncrypt Signature
		MessageSecurityVersion	WSSecurity11 WSTrustFebruary2005 WSSecureConversationFebruary 2005 WSSecurityPolicy11 BasicSecurityProfile10
		SecurityHeaderLayout	Lax
	security\ secure Conversation Bootstrap	AllowSerializedTokenOnSigningRepl y	True
		AuthenticationMode	MutualCertificate
		MessageProtectionOrder	SignBeforeEncrypt AndEncryptSignature
		RequireSignatureConfirmation	False

In addition to the bindings, there are two endpoint customizations described in Table 11-33 that are applied to the Medical and CreditCheck clients and services to link the services with the security configuration.

Table 11-33 Endpoint behavior customization

Behavior	Customization	Attribute	Value
Server-Cert-Behavior	service Certificate	FindValue	13
		StoreLocation	LocalMachine
		StoreName	My
		X509FindType	FindBySerialNumber
	client Certificate	CertificateValidationMode	PeerTrust
		IncludeWindowsGroup	False
		FindValue	12
		StoreLocation	Current User
		StoreName	My
		X509FindType	FindBySerialNumber
Client-Cert-Behavior	client Certificate	FindValue	12
		StoreLocation	Current User
		StoreName	My
		X509FindType	FindBySerialNumber
	service Certificate\ default Certificate	FindValue	13
		StoreLocation	Current User
		StoreName	TrustedPeople
		X509FindType	FindBySerialNumber
	.\authentication	CertificateValidationMode	PeerTrust

The application makes use of the bindings and behaviors shown in Table 11-34.

Table 11-34 Applications use of bindings and endpoint customization

Application	Client	Service
PolicyStore	reliableBinding	
Underwriter	reliableMtomBinding	
MedicalCheck	Client-Cert-Behavior	Server-Cert-Behavior
CreditCheck	RSPMtomBinding	

Finally, both the MedicalCheck and CreditCheck client endpoint configurations require the client's identity **(Dns=bob)** to be set. This property can be set on the **Identity** tab by clicking on the client's endpoint configuration, as shown in Figure 11-104.

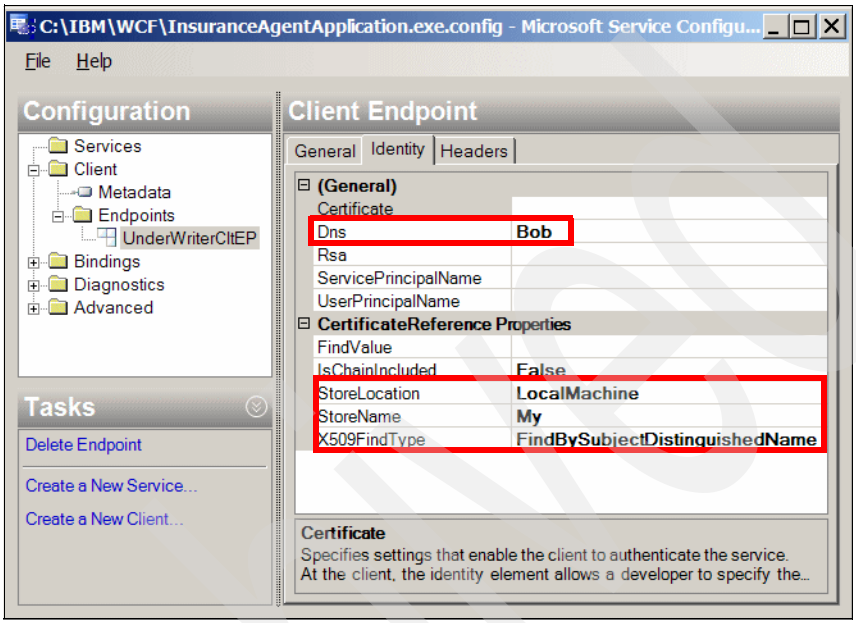


Figure 11-104 Client endpoint identity settings

At this point you should have a good understanding of the configurations bundled with the WCF Insurance underwriter sample application and preconfigured to use WS-ReliableMessaging and Reliable Secure Profile.

Apply secure quality of service configurations

We do not describe the step-by-step procedure to configure Windows Communication Foundation. We suggest you copy the secure configuration .config files from the WCF directory in the additional materials and overwrite the default configuration installed with the sample.

Validate WCF secure configuration

After configuring the secured version of the Insurance Underwriter example on Windows Communication Foundation re-run validation test as a standalone test on Windows Communication Foundation, following the instructions in 11.6.2, "Deploy application to Windows Communication Foundation" on page 434.

11.13 Distribute and test the Insurance sample

The final task is to verify the sample in a distributed environment. We provide one example in which the sample is deployed in a distributed topology, shown in Figure 11-105.

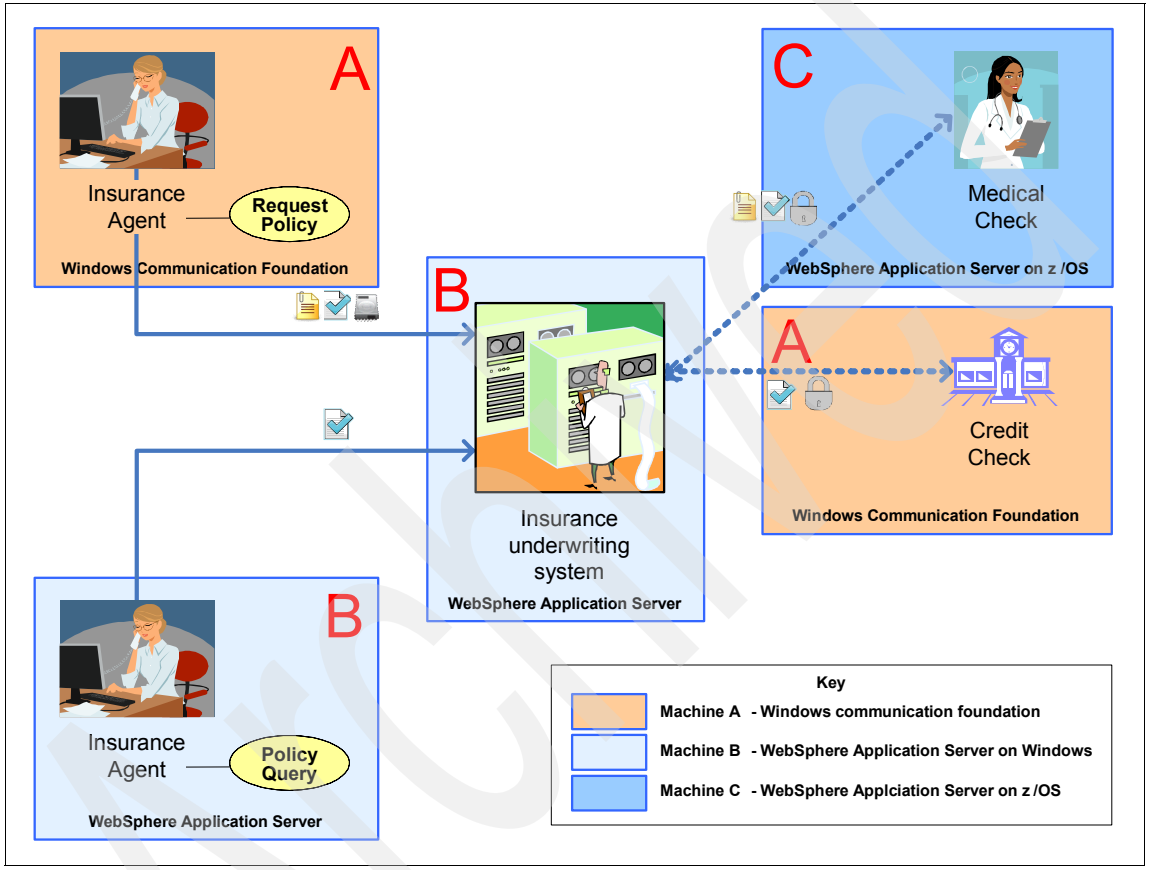


Figure 11-105 Distributed deployment

You can choose a different configuration from the one we demonstrate. As for many of the steps we describe, you have probably already performed them in setting up the single server environments. Use the instructions as guidance to the tasks that have to be performed to complete the configuration in a distributed environment. The main area to be careful about is setting the correct URLs for the services. We strongly recommend that you “ping” the Web service before trying the sample out.

Note: A Web service “ping” is to type the service URL with no parameters in a browser. See Figure 11-106 and Figure 11-107.

Figure 11-106 WebSphere Application Server service ping

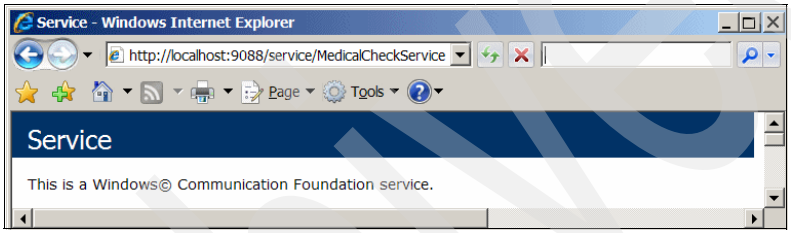


Figure 11-107 Windows Communication Foundation service ping

11.13.1 Install the Insurance Underwriter solution on three machines

Follow these instructions to install the sample application in a distributed environment.

- Install the Windows Communication Foundation sample on machine A by running the **setup.exe** in the WCF directory tree of the additional materials.

  This machine is used to run the Insurance Agent client, **InsuranceAgentApplication.exe,** and the CreditCheck service, **CreditCheckServiceConsole.exe.**

- Install WebSphere Application Server v6.1 and the Feature pack for Web services on machine B, a Windows, Linux®, or AIX® machine. Then install **UnderWriter.ear** from the Administrator Console.

- ☐ Either prepare to run PolicyServiceThinClient from a command window on machine B, either using the .cmd file provided in the additional materials (if you used the Quickstart procedure to configure the server), or run it from Rational Application Developer as done previously in this chapter.
- ☐ Install WebSphere Application Server for z/OS v6.1 and the Feature pack for Web services on Machine C. Then install the MedicalCheck service, **MedicalCheck.ear,** from the Administrator Console.

11.13.2 Machine A: Configure Windows Communication Foundation

We are running the Insurance Agents GUI and the CreditCheck service on Machine A.

- ☐ Follow the instructions in 11.12, “Configure and validate Windows sample” on page 525 to configure and validate the standalone WCF sample
- ☐ Terminate the MedicalCheck and Underwriter services, and the PolicyStore client.
- ☐ Click **Preferences** in the Insurance agents GUI (Figure 11-108) → Specify the URL of the Underwriter service running on WebSphere Application Server → **Apply**.

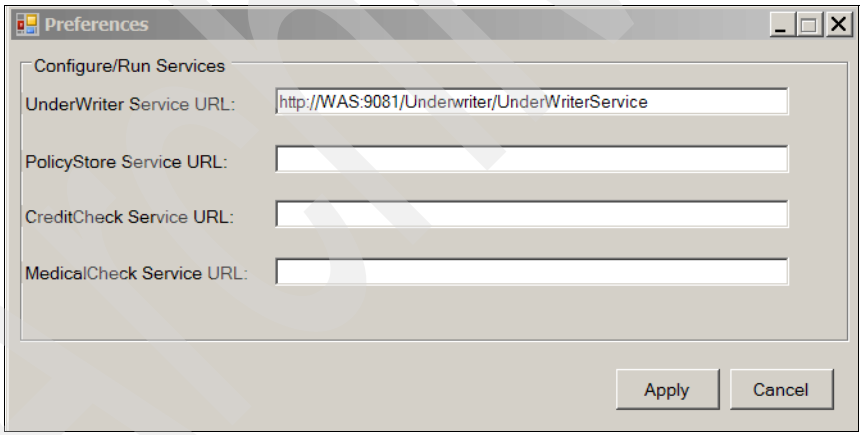


Figure 11-108 Set URL of underwriter service

11.13.3 Machine B: Configure WebSphere Application Server

We are running the Underwriter system, the PolicyStore service and the PolicyStore thin client on Machine B.

- Stop the MedicalCheck and CreditCheck services, and the Insurance Agent application.
- Launch the Underwriter system administrator's configuration page, as shown in Figure 11-109. Configure the URLs to point to the services. Here is the Underwriter URL:

  `http://localhost:9081/UnderWriter/UpdateEndpoint.html`

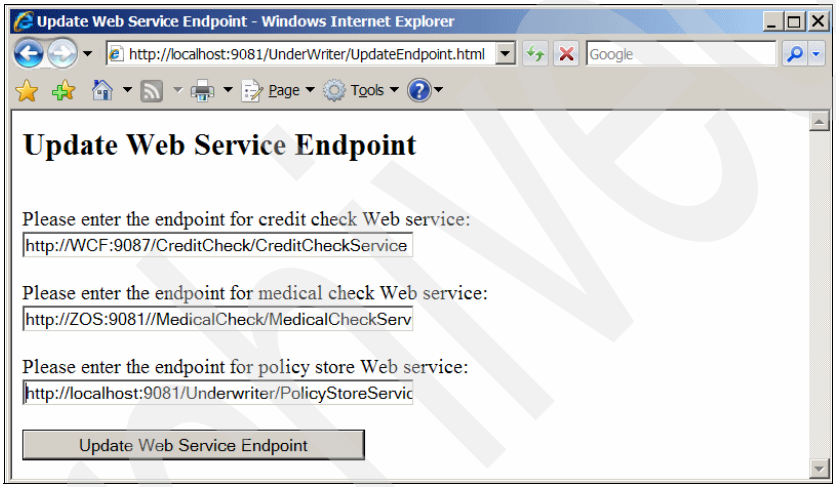


Figure 11-109 Update service endpoints: Administrator's configuration view

11.13.4 Machine C: Configure WebSphere Application Server on z/OS

We are running the MedicalCheck service on WebSphere Application Server for z/OS.

We have to do a complete configuration for the MedicalCheck service. That is a matter of carrying out all of the configuration for some of the policies, bindings, and resources required. You should note that the Security Token Service needs to run on every WebSphere Application Server that is participating in the secure conversation. To overcome this restriction, we need WS-Federation support to enable one server to trust another server with running STS on its behalf.

The steps to configure WebSphere Application Server for z/OS running the MedicalCheck service are as follows:

- Install the MedicalCheck service onto WebSphere Application Server for z/OS
- 11.7.1, “Policy set for persistent WS-ReliableMessaging 1.0” on page 444
- 11.7.2, “Policy set for trust service issuing Secure Conversation token” on page 446
- 11.7.4, “Policy set for a secure conversation service” on page 453
- 11.8.2, “Attach STSInterop Policy set” on page 457
- 11.8.3, “Assign bindings to the STSInterop Policy set” on page 457
- 11.8.4, “Attach SCSvcInterop Policy set” on page 458
- 11.8.5, “Assign bindings for SCSvcInterop Policy set” on page 459
- 11.9.1, “Create an SI bus to store and manage WS-RM messages” on page 465
- 11.9.2, “Configure keystores for WebSphere Application Server” on page 466
- 11.10.1, “Configure STSBinding for STSInterop” on page 469
- 11.10.3, “Configure SCSvcBinding for SCSvcInterop” on page 508
- 11.10.4, “Configure default binding for WS-ReliableMessaging” on page 514
- 11.10.5, “Configure compatible key sizes” on page 514
- Start MedicalCheck.ear → Restart the server

11.13.5 Run the Insurance Underwriter sample

Running the distributed example is little different from running it standalone. We have configured Windows Communication Foundation to use the Underwriter service on WebSphere Application Server, and we configured the Underwriter service to use MedicalCheck on WebSphere Application Server for z/OS and CreditCheck on Windows Communication Foundation.

- Submit an insurance request on Windows Communication Foundation until both the MedicalCheck and CreditCheck services return True and a policy is issued.
- Execute RunPolicySample.bat on WebSphere Application Server to verify the insurance policy is stored in the Underwriter system.

11.13.6 Troubleshooting

We experienced a number of problems getting the solution to work, as there is a good deal of opportunity to make mistakes in the configuration, resulting in unexpected problems — and because some of the configuration has persistent state, the cleanup sometimes requires more than a simple restart of the server.

Here are three suggestions to help you handle problems in your environment:

- Sometimes, with changes from service to client mode or vice-versa, the Feature pack for Web services server hangs due to stale messages in memory. Restart the Feature pack for Web services server if you experience any hangs or internal error messages.
- If you experience problems affecting WS-ReliableMessaging, stop the Feature pack for Web services server and remove the *filestores* directory, under your working server profile. Then restart the server.
- Use tools such as TCPMon SOAP monitor or Wireshark packet sniffer to capture http packets on the wire and then view the SOAP messages.

Part 4

Appendixes

A

Sample code

The listings of sample code used in the book are provided here for your convenience.

The projects listed are:

Echo

The Echo project examples are:

- A-1 EchoServicePortImpl.java
- A-2 EchoClientMainSync.java
- A-8 EchoClientMainAsyncPolling.java
- A-5 EchoClientCallbackHandler.java
- A-7 EchoClientMainAsyncCallback.java
- A-6 EchoClientMainAsyncCallbackWireAsync.java
- A-3 EchoClientMainSyncPortable.java

Echo service

Example: A-1 EchoServicePortImpl.java

```
package com.ibm.was.wssample.sei.echo;
@javax.jws.WebService(
      endpointInterface = "com.ibm.was.wssample.sei.echo.EchoServicePortType",
      targetNamespace = "http://com/ibm/was/wssample/sei/echo/",
      serviceName = "EchoService", portName = "EchoServicePort")
public class EchoServicePortImpl {
   public String echoOperation(String echoInput) throws Fault_Exception {
      if ("Fault".equals(echoInput)) {
         final Fault faultInfo = new Fault();
         faultInfo.setReason("You told me to throw a fault");
         throw new Fault_Exception("echo fault", faultInfo);
      }
      return "JAX WS echo >> " + echoInput;
   }
}
```

Echo clients

The sample client code here differs slightly from the examples in Chapter 7, “JAX-WS programming model” on page 175, but matches the additional materials, in having moved the new classes we wrote into a separate client package.

Example: A-2 EchoClientMainSync.java

```
package com.ibm.itso.clients;
import com.ibm.was.wssample.sei.echo.EchoServicePortProxy;
public class EchoClientMainSync {
   public static void main(String[] args) throws Exception {
//     Get an instance of the proxy class
      final EchoServicePortProxy proxy = new EchoServicePortProxy();
//     Call the Web service via the proxy
      System.out.println("proxy.echoOperation(\"Hello\") returns \""
            + proxy.echoOperation("Hello") + "\"");
   }
}
```

Example A-3 uses the URL and QName constructor to make the client portable, corresponding to . To overcome the compile time error caused by the @WebServiceClient annotation referencing the WSDL location we have copied Echo.wsdl into the portable client project. *We have also modified the generated code to give it a relative reference and make the project portable* (see Example A-4).

Example: A-3 EchoClientMainSyncPortable.java

```
package com.ibm.itso.clients;
import java.net.URL;
import javax.xml.namespace.QName;
import com.ibm.was.wssample.sei.echo.EchoServicePortProxy;
public class EchoClientMainSyncPortable {
   static final String endpointURL = "http://localhost:9081/EchoService/EchoService";
   public static void main(String[] args) throws Exception {
      // Get an instance of the proxy class
      final EchoServicePortProxy proxy = new EchoServicePortProxy(
            new URL(endpointURL + "?WSDL"),
            new QName("http://com/ibm/was/wssample/sei/echo/", "EchoService"));
      // Call the Web service via the proxy
      System.out.println("proxy.echoOperation(\"Hello\") returns \""
            + proxy.echoOperation("Hello") + "\"");
   }
}
```

Example: A-4 EchoService.java - relative wsdl reference

```
@WebServiceClient(name = "EchoService",
      targetNamespace = "http://com/ibm/was/wssample/sei/echo/",
      wsdlLocation = "file:./META-INF/Echo.wsdl")
```

Example: A-5 EchoClientCallbackHandler.java

```
package com.ibm.itso.clients;
import java.util.concurrent.ExecutionException;
import javax.xml.ws.AsyncHandler;
import javax.xml.ws.Response;
import com.ibm.was.wssample.sei.echo.EchoOperationResponse;
import com.ibm.was.wssample.sei.echo.FaultException;
public class EchoClientCallbackHandler implements
      AsyncHandler<EchoOperationResponse> {
   public void handleResponse(Response<EchoOperationResponse> resp) {
      try {
         final EchoOperationResponse eor = resp.get();
         System.out.println("echoOperation async invocation complete.");
         System.out.println("echoOperation async response is: "
               + eor.getEchoResponse());
      } catch (final ExecutionException ee) {
         final FaultException fe = (FaultException) ee.getCause();
         System.out.println("echoOperation failed:  reason = "
               + fe.getFaultInfo().getReason());
      } catch (final InterruptedException ie) {
         System.out.println("EchoOperation received Interrupted Exception");
      }
   }
}
```

Example: A-6 EchoClientMainAsyncCallbackWireAsync.java

```
package com.ibm.itso.clients;
import javax.xml.ws.BindingProvider;
import com.ibm.was.wssample.sei.echo.EchoServicePortProxy;
import java.util.concurrent.Future;
public class EchoClientMainAsyncCallbackWireAsync {
   public static void main(String[] args) throws Exception {
      final EchoServicePortProxy proxy = new EchoServicePortProxy();
      // Set wire async
      final BindingProvider bp = (BindingProvider) proxy._getDescriptor()
            .getProxy();
      bp.getRequestContext().put(
            "com.ibm.websphere.webservices.use.async.mep", Boolean.TRUE);
      // Set up the callback handler.
      final EchoClientCallbackHandler callbackHandler =
         new EchoClientCallbackHandler();
      // Make the Web service call.
      final Future<?> response = proxy.echoOperationAsync("Hello",
            callbackHandler);
      System.out.println("Wait 5 seconds.");
      // Give the callback handler a chance to be called.
      Thread.sleep(5000);
      System.out.println("echoOperation async is now ending.");
   }
}
```

Example: A-7 EchoClientMainAsyncCallback.java

```
package com.ibm.itso.clients;
import java.util.concurrent.Future;
import com.ibm.was.wssample.sei.echo.EchoServicePortProxy;
public class EchoClientMainAsyncCallback {
   public static void main(String[] args) throws Exception {
      final EchoServicePortProxy proxy = new EchoServicePortProxy();
      // Set up the callback handler.
      final EchoClientCallbackHandler callbackHandler =
         new EchoClientCallbackHandler();
      // Make the Web service call.
      final Future<?> response = proxy.echoOperationAsync("Hello",
            callbackHandler);
      System.out.println("Wait 5 seconds.");
      // Give the callback handler a chance to be called.
      Thread.sleep(5000);
      System.out.println("echoOperation async is now ending.");
   }
}
```

Example: A-8 EchoClientMainAsyncPolling.java

```
package com.ibm.itso.clients;
import java.util.concurrent.ExecutionException;
import javax.xml.ws.Response;
import com.ibm.was.wssample.sei.echo.EchoOperationResponse;
import com.ibm.was.wssample.sei.echo.EchoServicePortProxy;
import com.ibm.was.wssample.sei.echo.FaultException;
public class EchoClientMainAsyncPolling {
   private static final int SLEEPER = 2; // Polling interval in seconds
   private static final int TIMEOUT = 240; // Error timeout in seconds
   public static void main(String[] args) throws Exception {
      final EchoServicePortProxy proxy = new EchoServicePortProxy();
      final Response<EchoOperationResponse> resp = proxy
            .echoOperationAsync("Hello");
      // Poll for the response.
      Thread.sleep(1000 * SLEEPER);
      int waiting = TIMEOUT;
      while (!resp.isDone()) {
         // Check for timeout
         if (waiting <= 0) {
            System.out.println("echoOperation async timed out.");
            System.exit(0);
         }
         System.out.println("echoOperation async still not complete.");
         Thread.sleep(1000 * SLEEPER);
         waiting -= SLEEPER;
      }
      // Get the response and print it.
      try {
         final EchoOperationResponse eor = resp.get();
         System.out.println("echoOperation async invocation complete.");
         if (eor != null) {
            System.out.println("echoOperation async response is: "
                  + eor.getEchoResponse());
         }
      } catch (final ExecutionException ee) {
         final FaultException fe = (FaultException) ee.getCause();
         System.out.println("echoOperation failed:  reason = "
               + fe.getFaultInfo().getReason());
      }
   }
}
```

MTOM

The MTOM project examples are:

- A-9 AttachmentPortImpl.java
- A-10 MTOMClientMain.java

MTOM service

Example: A-9 `AttachmentPortImpl.java`

```
package com.ibm.was.wssample.sei.mtom;
import javax.xml.transform.Source;
@javax.jws.WebService(
      endpointInterface = "com.ibm.was.wssample.sei.mtom.Attachment",
      targetNamespace = "http://com/ibm/was/wssample/sei/mtom/",
      serviceName = "AttachmentService",
      portName = "AttachmentPort",
      wsdlLocation = "WEB-INF/wsdl/MTOM.wsdl")
@javax.xml.ws.BindingType(
      value = javax.xml.ws.soap.SOAPBinding.SOAP11HTTP_MTOM_BINDING)
public class AttachmentPortImpl {
   public Source sendXML(Source xml) {
      return xml;
   }
   public byte[] sendBinary(byte[] binary) {
      return binary;
   }
}
```

MTOM client

Example: A-10 MTOMClientMain.java

```
package com.ibm.itso.clients;
import java.io.InputStreamReader;
import java.io.StringReader;
import javax.xml.transform.Source;
import javax.xml.transform.stream.StreamSource;
import javax.xml.ws.BindingProvider;
import javax.xml.ws.soap.SOAPBinding;
import com.ibm.was.wssample.sei.mtom.AttachmentPortProxy;
public class MTOMClientMain {
   public static void main(String[] args) throws Exception{
//    Get an instance of the proxy class
      final AttachmentPortProxy proxy = new AttachmentPortProxy();
//    Enable MTOM
      BindingProvider bp = (BindingProvider)
            (proxy._getDescriptor().getProxy());
      ((SOAPBinding) bp.getBinding()).setMTOMEnabled(true);
//    Make the Web service call to sendBinary.
      System.out.println("sendBinary(\"Hello\") returned \""
            + new String(proxy.sendBinary("Hello".getBytes())) + "\"");
//    Make the Web service call to sendXML.
      Source ss = new StreamSource(new StringReader("<Hello/>"));
      ss = proxy.sendXML(ss);
      final InputStreamReader reader = (InputStreamReader) ((StreamSource) ss)
            .getReader();
      final char[] chars = new char[100];
      final int length = reader.read(chars);
      System.out.println("sendXML(\"<Hello/>\") returned \""
            + new String(chars, 0, length) + "\"");
   }
}
```

Calculator

The Calculator examples are:

- A-11 CalculatorSOAPImpl.java
- A-12 CalculatorClientMain.java
- A-13 CalculatorClientMainWSS.java

Calculator service

Example: A-11 CalculatorSOAPImpl.java

```
package com.ibm.itso;
@javax.jws.WebService(
      endpointInterface = "com.ibm.itso.Calculator",
      targetNamespace = "http://itso.ibm.com",
      serviceName = "Calculator",
      portName = "CalculatorSOAP",
      wsdlLocation = "WEB-INF/wsdl/Calculator.wsdl")
public class CalculatorSOAPImpl {
   public double add(double i, double j) {
      System.out.println("Calculator service "+i+" + "+j+" = "+(i+j));
      return i+j;
   }
   public double subtract(double i, double j) {
      System.out.println("Calculator service "+i+" - "+j+" = "+(i-j));
      return i-j;
   }
}
```

Calculator clients

There are two versions of the Calculator client, the first simpler version using Policy sets and the second, more complicated version uses the WSS API rather than using Policy sets. The explanation of the second example is found in Chapter 9, “Secure conversation” in the section “Configure Web service security using the WSS APIs example” on page 316.

Example: A-12 CalculatorClientMain.java

```
package com.ibm.itso.clients;
import java.net.URL;
import javax.xml.namespace.QName;
import com.ibm.itso.CalculatorSOAPProxy;
public class CalculatorClientMain {
   private static final String endpointURL = "http://localhost:9081/Calculator/Calculator";
   public static void main(String[] args) throws Exception {
      CalculatorSOAPProxy proxy = new CalculatorSOAPProxy(
            new URL(endpointURL + "?wsdl"),
            new QName("http://itso.ibm.com", "Calculator"));
      System.out.println("Calculator client 1 + 1 = " + proxy.add(1.0, 1.0));
      System.out.println("Calculator client 1 - 1 = " + proxy.subtract(1.0, 1.0));
   }
}
```

Example: A-13 CalculatorClientMainWSS.java

```
package com.ibm.itso.clients;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.InputStream;
import java.net.URL;
import java.security.InvalidAlgorithmParameterException;
import java.security.NoSuchAlgorithmException;
import java.security.NoSuchProviderException;
import java.security.cert.CertStore;
import java.security.cert.CertificateException;
import java.security.cert.CertificateFactory;
import java.security.cert.CollectionCertStoreParameters;
import java.security.cert.X509Certificate;
import java.util.HashSet;
import java.util.Map;
import java.util.Set;
import javax.xml.namespace.QName;
import javax.xml.ws.BindingProvider;
import com.ibm.itso.Calculator;
import com.ibm.itso.CalculatorSOAPProxy;
import com.ibm.websphere.wssecurity.callbackhandler.X509ConsumeCallbackHandler;
import com.ibm.websphere.wssecurity.callbackhandler.X509GenerateCallbackHandler;
import com.ibm.websphere.wssecurity.wssapi.WSSConsumingContext;
import com.ibm.websphere.wssecurity.wssapi.WSSException;
import com.ibm.websphere.wssecurity.wssapi.WSSFactory;
import com.ibm.websphere.wssecurity.wssapi.WSSGenerationContext;
```

```
import com.ibm.websphere.wssecurity.wssapi.decryption.WSSDecryption;
import com.ibm.websphere.wssecurity.wssapi.encryption.WSSEncryption;
import com.ibm.websphere.wssecurity.wssapi.signature.WSSSignature;
import com.ibm.websphere.wssecurity.wssapi.token.SecurityToken;
import com.ibm.websphere.wssecurity.wssapi.token.X509Token;
import com.ibm.websphere.wssecurity.wssapi.verification.WSSVerification;

public class CalculatorClientMainWSS {
   private static final String KEY_PATH = "C:/IBM/WAS/ND/etc/ws-security/samples/";
   private static final String endpointURL = "http://localhost:9081/Calculator/Calculator";
   public static void main(String[] args) throws Exception {
      CalculatorSOAPProxy proxy = new CalculatorSOAPProxy(
            new URL(endpointURL + "?wsdl"),
            new QName("http://itso.ibm.com", "Calculator"));
      Calculator port = proxy._getDescriptor().getProxy();
      BindingProvider bp = (BindingProvider) port;
      Map<String, Object> requestContext = bp.getRequestContext();
      Map<String, Object> responseContext = bp.getRequestContext();
      /** **************set the signature info*************************** */
      // Generate the WSSFactory instance (step: a)
      WSSFactory factory = WSSFactory.getInstance();
      // Generate the WSSGenerationContext instance (step: b)
      WSSGenerationContext gencont = factory.newWSSGenerationContext();
      X509GenerateCallbackHandler callbackHandler = new X509GenerateCallbackHandler(
            "", KEY_PATH + "dsig-sender.ks", "jks", "client".toCharArray(),
            "soaprequester", "client".toCharArray(),
            "CN=SOAPRequester, OU=TRL, O=IBM, ST=Kanagawa, C=JP", null);
      // Generate the security token to be used for the signature (step:c)
      SecurityToken token =
            factory.newSecurityToken(X509Token.class, callbackHandler);
      // Generate the WSSSignature instance (step: d)
      WSSSignature sig = factory.newWSSSignature(token);
      // Add the WSSSignature to WSSGenerationContext (step: e)
      gencont.add(sig);
      /** *********set encryption information****************************** */
      // Generate the callback handler
      X509GenerateCallbackHandler callbackHandler2 = new X509GenerateCallbackHandler(
            "", KEY_PATH + "enc-sender.jceks", "jceks", "storepass".toCharArray(),
            "bob", null, "CN=Bob, O=IBM, C=US", null);
      // Generate the security token used for encryption (step: c)
      SecurityToken token2 =
         factory.newSecurityToken(X509Token.class, callbackHandler2);
      // Generate WSSEncryption instance (step: d)
      WSSEncryption enc = factory.newWSSEncryption(token2);
      // DEFAULT: WSSEncryption.KW_RSA_OAEP
```

```
enc.setKeyEncryptionMethod(WSSEncryption.KW_RSA15);
gencont.add(enc);
gencont.process(requestContext);
/** ********set decryption information**************************** */
WSSConsumingContext concont = factory.newWSSConsumingContext();
X509ConsumeCallbackHandler callbackHandler3 = new X509ConsumeCallbackHandler(
      "", KEY_PATH + "enc-sender.jceks", "jceks", "storepass".toCharArray(),
      "alice", "keypass".toCharArray(), "CN=Alice, O=IBM, C=US");
// Generate the WSSDecryption instance (step: d)
WSSDecryption dec =
   factory.newWSSDecryption(X509Token.class, callbackHandler3);
dec.addAllowedKeyEncryptionMethod(WSSEncryption.KW_RSA15);
concont.add(dec);
/** *********set signature verification info******************** */
String certpath = KEY_PATH + "intca2.cer";
// The location of the X509 certificate file
X509Certificate x509cert = null;
try {
   InputStream is = new FileInputStream(certpath);
   CertificateFactory cf = CertificateFactory.getInstance("X.509");
   x509cert = (X509Certificate) cf.generateCertificate(is);
} catch (FileNotFoundException e1) {
   throw new WSSException(e1);
} catch (CertificateException e2) {
   throw new WSSException(e2);
}
Set<Object> eeCerts = new HashSet<Object>();
eeCerts.add(x509cert);
// Create the certificate store
java.util.List<CertStore> certList = new java.util.ArrayList<CertStore>();
CollectionCertStoreParameters certparam =
   new CollectionCertStoreParameters(eeCerts);
CertStore cert = null;
try {
   cert = CertStore
         .getInstance("Collection", certparam, "IBMCertPath");
} catch (NoSuchProviderException e1) {
   throw new WSSException(e1);
} catch (InvalidAlgorithmParameterException e2) {
   throw new WSSException(e2);
} catch (NoSuchAlgorithmException e3) {
   throw new WSSException(e3);
}
if (certList != null) {
   certList.add(cert);
```

```
        }
        // Generate the callback handler
        X509ConsumeCallbackHandler callbackHandler4 = new X509ConsumeCallbackHandler(
              KEY_PATH + "dsig-receiver.ks", "jks", "server".toCharArray(),
              certList, java.security.Security.getProvider("IBMCertPath"));
        // Generate the WSSVerification instance (step: d)
        WSSVerification ver =
           factory.newWSSVerification(X509Token.class,callbackHandler4);
        // Add the WSSVerification to the WSSConsumingContext (step: j)
        concont.add(ver);
        concont.process(responseContext);
        System.out.println("Calculator client 1 + 1 = " + proxy.add(1.0, 1.0));
        System.out.println("Calculator client 1 - 1 = "+ proxy.subtract(1.0, 1.0));
    }
}
```

Insurance sample

The insurance sample has two external services:

- “CreditCheck” on page 551
- “MedicalCheck” on page 552

And two internal services:

- “UnderwriterSOAPImpl” on page 553
- “PolicyStore service” on page 556

Two callback handlers for the asynchronous clients to the CreditCheck and MedicalCheck services which are embedded in the Underwriter service:

- “CreditCheckCallbackHandler” on page 555
- “MedicalCheckCallbackHandler” on page 556

There are two clients:

- “PolicyStoreThinClient.java” on page 557
- InsuranceAgent client (JSP, not listed).

And two utility functions and a Web page to manage changing endPoints for testing:

- “ManageEndpoint” on page 558
- “UpdateEndpoint Web page” on page 559
- “SetEndpointServlet” on page 559

CreditCheck

Example: A-14 CreditCheckSOAPImpl.java

```
package org.example.creditcheck;
import java.util.Random;
import org.example.cc_customerinfo.CCCustomerInfoType;
@javax.jws.WebService (endpointInterface="org.example.creditcheck.CreditCheckPort",
      targetNamespace="http://www.example.org/CreditCheck/",
      serviceName="CreditCheck",
      portName="CreditCheckSOAP",
      wsdlLocation="WEB-INF/wsdl/CreditCheck.wsdl")
public class CreditCheckSOAPImpl{
/**
 * This method evaluates the credit rating of customer.
 * It uses the Java Random function to return the result.
 * @param customerInfo
 */
    public boolean evaluateCreditRating(CCCustomerInfoType customerInfo) {
        Random creditCheckResult = new Random();
        boolean result= false;
        int count =0;
// If the result is false, it will loop additional two times.
        while(result==false & count<3){
            result = creditCheckResult.nextBoolean();
            if(result==true) break;
            count++;
        }
    System.out.println("Credit check result is " + result);
    return result;
    }
}
```

MedicalCheck

Example: A-15 MedicalCheckSOAPImpl.java

```
package org.example.medicalcheck;
import java.util.Random;
import org.example.mc_customerinfo.MCCustomerInfoType;
import org.example.mc_customerinfo.MCMedicalRecordType;
@javax.jws.WebService(endpointInterface = "org.example.medicalcheck.MedicalCheckPort",
      targetNamespace = "http://www.example.org/MedicalCheck/",
      serviceName = "MedicalCheck",
      portName = "MedicalCheckSOAP",
      wsdlLocation = "WEB-INF/wsdl/MedicalCheck.wsdl")
@javax.xml.ws.BindingType(
      value = javax.xml.ws.soap.SOAPBinding.SOAP11HTTP_MTOM_BINDING)
public class MedicalCheckSOAPImpl {
   public boolean evaluateRecord(MCCustomerInfoType customerInfo,
         MCMedicalRecordType medicalRecord) {
      Random medicalCheckResult = new Random();
      boolean result = false;
      int count = 0;
// If the result is false, it will additional two times. If the result
      while (result == false & count < 3) {
         result = medicalCheckResult.nextBoolean();
         if (result == true) break;
         count++;
      }
      System.out.println("Medical check result is " + result);
      return result;
   }
}
```

UnderwriterSOAPImpl

Example: A-16 UnderwriterSOAPImpl.java

```
package org.example.underwriter;
import java.util.Random;
import javax.xml.ws.Response;
import org.example.Model.PolicyFactory;
import org.example.cc_customerinfo.CCCustomerInfoType;
import org.example.creditcheck.CreditCheckSOAPProxy;
import org.example.creditcheck.EvaluateCreditRatingResponse;
import org.example.management.ManageEndpoint;
import org.example.mc_customerinfo.MCCustomerInfoType;
import org.example.mc_customerinfo.MCMedicalRecordType;
import org.example.medicalcheck.EvaluateRecordResponse;
import org.example.medicalcheck.MedicalCheckSOAPProxy;
import org.example.uw_customerinfo.UWCustomerInfoType;
import org.example.uw_customerinfo.UWMedicalRecordType;

@javax.jws.WebService(endpointInterface = "org.example.underwriter.UnderWriterPort",
      targetNamespace = "http://www.example.org/UnderWriter/",
      serviceName = "UnderWriter",
      portName = "UnderWriterSOAP",
      wsdlLocation = "WEB-INF/wsdl/UnderWriter.wsdl")
@javax.xml.ws.BindingType(
      value = javax.xml.ws.soap.SOAPBinding.SOAP11HTTP_MTOM_BINDING)
public class UnderWriterSOAPImpl  {
   private static final int SLEEPER = 2000; // Polling interval 2 seconds
   private static final int WAITING = 15000;  // Polling limit 15 seconds
   public void createPolicy(UWCustomerInfoType customerInfo,
         UWMedicalRecordType medicalRecord) {
      try {
         CreditCheckSOAPProxy ccProxy = new CreditCheckSOAPProxy();
         MedicalCheckSOAPProxy mcProxy = new MedicalCheckSOAPProxy();
         ManageEndpoint me = ManageEndpoint.getInstance();
// set the endpoint based on the endpoint retrieved from the ManageEndpoint.java
         ccProxy._getDescriptor().setEndpoint(me.getCcEndpointUrl());
         mcProxy._getDescriptor().setEndpoint(me.getMcEndpointUrl());
         System.out.println("The endpoint for credit check Web service is "
               + ccProxy._getDescriptor().getEndpoint());
         System.out.println("The endpoint for medical check Web service is "
               + mcProxy._getDescriptor().getEndpoint());
// set message values from the input received from the Insurance Agent
         CCCustomerInfoType ccCustomerInfo = setCustomerInfo(customerInfo);
         MCCustomerInfoType mcCustomerInfo = setMedicalInfo(customerInfo);
```

```
        MCMedicalRecordType mCMedicalRecord = setMedicalRecord(medicalRecord);
// Invoke services to check credit and medical results
        Response<EvaluateCreditRatingResponse> creditCheckResp
              = ccProxy.evaluateCreditRatingAsync(ccCustomerInfo);
        Response<EvaluateRecordResponse> medCheckResp
              = mcProxy.evaluateRecordAsync(mcCustomerInfo, mCMedicalRecord);
// Waiting for the response from medical check and credit check.
        int waiting = WAITING;
        while (!creditCheckResp.isDone() & !medCheckResp.isDone()) {
           if (waiting <= 0) {
              System.out.println(">> CLIENT: ERROR - Timeout waiting for reply.");
              System.exit(0);
           }
           System.out.println(">> CLIENT: Check invocation still not complete");
           Thread.sleep(SLEEPER);
           waiting -= SLEEPER;
        }
// Get the results
        Boolean creditCheckResult = creditCheckResp.get().isResult();
        Boolean medicalCheckResult = medCheckResp.get().isResult();
        if (creditCheckResult && medicalCheckResult) {
           Random random = new Random();
           long policyNumber = Math.abs(random.nextLong());
           PolicyFactory policyFactory = new PolicyFactory();
           policyFactory.addPolicy(customerInfo.getSsn(), policyNumber);
           System.out.println("Generated policy number is " + policyNumber);
        } else {
           System.out.println(
                 "Policy was denied due to customer background check failure.");
        }
     } catch (Exception e) {
        System.out.println("Policy was not generated due to processing failure.");
        System.out.println(">> CLIENT: ERROR.");
        e.printStackTrace();
     }
   }
   private MCMedicalRecordType setMedicalRecord(UWMedicalRecordType medicalRecord) {
      MCMedicalRecordType mCMedicalRecord = new MCMedicalRecordType();
      mCMedicalRecord.setAttachment(medicalRecord.getAttachment());
      return mCMedicalRecord;
   }
   private MCCustomerInfoType setMedicalInfo(UWCustomerInfoType customerInfo) {
      MCCustomerInfoType mcCustomerInfo =
         new MCCustomerInfoType();
      mcCustomerInfo.setGender(customerInfo.getAge());
```

```
      mcCustomerInfo.setName(customerInfo.getName());
      return mcCustomerInfo;
   }
   private CCCustomerInfoType setCustomerInfo(UWCustomerInfoType customerInfo) {
      CCCustomerInfoType ccCustomerInfo =
         new CCCustomerInfoType();
      ccCustomerInfo.setName(customerInfo.getName());
      ccCustomerInfo.setSsn(customerInfo.getSsn());
      return ccCustomerInfo;
   }
}
```

CreditCheckCallbackHandler

Example: A-17 CreditCheckCallbackHandler.java

```
package org.example.creditcheck;
import java.util.concurrent.ExecutionException;
import javax.xml.ws.AsyncHandler;
import javax.xml.ws.Response;
public class CreditCheckCallbackHandler implements
      AsyncHandler<EvaluateCreditRatingResponse> {
   private EvaluateCreditRatingResponse output;
   public void handleResponse(Response<EvaluateCreditRatingResponse> response) {
      try {
         output = response.get();
      } catch (ExecutionException e) {
         System.out.println(">> CLIENT: Connection Exception");
      } catch (InterruptedException e) {
         System.out.println(">> CLIENT: Interrupted Exception");
      }
   }
   public EvaluateCreditRatingResponse getResponse() {
      return output;
   }
}
```

MedicalCheckCallbackHandler

Example: A-18 MedicalCheckCallbackHandler.java

```
package org.example.medicalcheck;
import java.util.concurrent.ExecutionException;
import javax.xml.ws.AsyncHandler;
import javax.xml.ws.Response;
public class MedicalCheckCallbackHandler implements
      AsyncHandler<EvaluateRecordResponse> {
   private EvaluateRecordResponse output;
   public void handleResponse(Response<EvaluateRecordResponse> response) {
      try {
         output = response.get();
      } catch (ExecutionException e) {
         System.out.println(">> CLIENT: Connection Exception");
      } catch (InterruptedException e) {
         System.out.println(">> CLIENT: Interrupted Exception");
      }
   }
   public EvaluateRecordResponse getResponse() {
      return output;
   }
}
```

PolicyStore service

Example: A-19 PolicyStorePortImpl.java

```
package org.example.policystore;
import org.example.Model.PolicyFactory;
@javax.jws.WebService(endpointInterface = "org.example.policystore.PolicyStorePort",
      targetNamespace = "http://www.example.org/PolicyStore/",
      serviceName = "PolicyStoreService",
      portName = "PolicyStorePort",
      wsdlLocation = "WEB-INF/wsdl/PolicyStore.wsdl")
public class PolicyStorePortImpl {
   public String policyRequest(String ssn) {
      PolicyFactory policyFactory = new PolicyFactory();
      long policyNumber = policyFactory.getPolicies(ssn);
      return policyNumber.toString();
   }
}
```

PolicyStoreThinClient.java

Example: A-20 PolicyStoreThinClient

```
package org.example.policystore;
import java.net.URL;
import javax.xml.namespace.QName;
public class PolicyStoreJavaClient {
   public static void main(String[] args) throws Exception {
      String ssn = "100";
      String policyURL = "http://localhost:9081/UnderWriter/PolicyStoreService";
      String policyQName = "http://www.example.org/PolicyStore/";
      String policyService = "PolicyStoreService";
      for (int i = 0; i < args.length; i++) {
         switch(i) {
         case 0: ssn = args[0]; break;
         case 1: policyURL = args[1]; break;
         case 2: policyQName = args[2]; break;
         case 3: policyService = args[3]; break;
         default:
         }
      }
      System.out.println("Social security number is " + ssn);
      System.out.println("Policy store URL " + policyURL);
      System.out.println("Policy store QName " + policyQName);
      System.out.println("Policy store Service " + policyService);
      try {
         PolicyStorePortProxy proxy = new PolicyStorePortProxy(
               new URL(policyURL + "?wsdl"),
               new QName(policyQName, policyService));
// Restore the policyURL parm, overriding the endpoint in the WSDL
         proxy._getDescriptor().setEndpoint(policyURL);
         System.out.println("The policy number is " + proxy.policyRequest(ssn));
      } catch (Exception e) {
         System.out.println("The policy number does not exist.");
      }
   }
}
```

ManageEndpoint

Example: A-21 ManageEndpoint.java

```
package org.example.management;
import org.example.policystore.PolicyStorePortProxy;
public class ManageEndpoint {
   private static ManageEndpoint instance = new ManageEndpoint();
   // These are the default endpoint for Credit check and Medical check Web service
   private String ccEndpointUrl = "http://localhost:9081/CreditCheck/CreditCheck";
   private String mcEndpointUrl = "http://localhost:9081/MedicalCheck/MedicalCheck";
   private String psEndpointUrl = "http://localhost:9081/UnderWriter/PolicyStoreService";
   public static ManageEndpoint getInstance() { return instance; }
   public String getCcEndpointUrl() { return ccEndpointUrl; }
   public void setCcEndpointUrl(String ccEndpointUrl) {
      this.ccEndpointUrl = ccEndpointUrl;
      }
   public String getMcEndpointUrl() { return mcEndpointUrl; }
   public void setMcEndpointUrl(String mcEndpointUrl) {
      this.mcEndpointUrl = mcEndpointUrl;
   }
   public String getPsEndpointUrl() { return psEndpointUrl; }
   public void setPsEndpointUrl(String psEndpointUrl) {
      this.psEndpointUrl = psEndpointUrl;
      PolicyStorePortProxy psProxy = new PolicyStorePortProxy();
      psProxy._getDescriptor().setEndpoint(
            this.psEndpointUrl);
   }
}
```

UpdateEndpoint Web page

Example: A-22 UpdateEndpoint.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Update Web Service Endpoint</title>
</head>
<body>
<H2>Update Web Service Endpoint</H2>
<FORM action="SetEndpoint" method="post"><BR>
Please enter the endpoint for credit check Web
service:   <INPUT type="text" name="ccid" size="50"
   maxlength="100"> <br>
<BR>
Please enter the endpoint for medical check Web service: <INPUT
   type="text" name="mcid" size="50" maxlength="100"> <BR>
<br>
Please enter the endpoint for policy store Web service: <input
   type="text" name="psid" size="50" maxlength="100"><br>
<BR>
<INPUT type="submit" name="submitbutton"
   value="Update Web Service Endpoint"></FORM>
</body>
</html>
```

SetEndpointServlet

Example: A-23 SetEndpointServlet.java

```
package org.example.servlet;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.example.management.ManageEndpoint;
public class SetEndpointServlet extends javax.servlet.http.HttpServlet implements
javax.servlet.Servlet {
   private static final long serialVersionUID = -6949582528640708 50L;
   public SetEndpointServlet() {super(); }
   protected void doGet(HttpServletRequest request, HttpServletResponse response)
```

```
    throws ServletException, IOException {
       doPost(request, response);
    }
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
       ManageEndpoint me= ManageEndpoint.getInstance();
       me.setCcEndpointUrl(request.getParameter("ccid"));
       //Get the parameter from the input form of the HTML page
       System.out.println("New endpoint for Credit check is  " +
             request.getParameter("ccid"));
       me.setMcEndpointUrl(request.getParameter("mcid"));
       System.out.println("New endpoint for Medical check is  "+
             request.getParameter("mcid"));
       me.setPsEndpointUrl(request.getParameter("psid"));
       System.out.println("New endpoint for Policy Store is  "+
             request.getParameter("psid"));
       response.setContentType("text/html");
       PrintWriter out = response.getWriter();
       out.println(+
             "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.01 Transitional//EN\">");
       out.println("<HTML><HEAD><TITLE>Endpoint has been updated.</TITLE></HEAD>");
       out.println("<BODY>");
       out.println(=
             "<h2>The endpoint for Credit check Web service has been updated to " +
             me.getCcEndpointUrl());
       out.println(
             "<br><br>The endpoint for Medical check Web service has been updated to "
             + me.getMcEndpointUrl());
       out.println(
             "<br><br>The endpoint for Policy Store Web service has been updated to "
             + me.getPsEndpointUrl());
       out.println("</h2><br><br><h2>");
       out.println(
          "</h2><FORM><INPUT TYPE=\"button\" VALUE=\"“ +
             “Back to Update Web service endpoint page\" " +
             "onClick=\"history.go(-1);return true;\"> </FORM> </BODY></HTML>");
    }
}
```

B

Additional material

This book refers to additional material that can be downloaded from the Internet as described below.

Locating the Web material

The Web material associated with this book is available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser at:

`ftp://www.redbooks.ibm.com/redbooks/SG247618`

Alternatively, you can go to the IBM Redbooks Web site at:

`ibm.com/redbooks`

Select the **Additional materials** and open the directory that corresponds with the IBM Redbooks form number, SG247618.

Using the Web material

The additional Web material that accompanies this book is organized with a directory for each chapter for which there are samples. In each directory you will find all the files to recreate the examples in the book.

File names suffixed with PI should be loaded using the Project Interchange importer in Rational Application Developer.

System requirements for downloading the Web material

We created the examples in a VMWare Windows 2003 image. The total VMware image is about 8GB without Visual Studio 2008, and 16GB with Studio.

We ran the samples on a Thinkpad T61 T7500 @2.2GHz Intel® Core Duo with 3GB RAM.running Windows XP SP2. This was more than ample to run the examples.

How to use the Web material

Create a subdirectory (folder) on your workstation, and unzip the contents of the Web material zip file into this folder.

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this book.

IBM Redbooks

For information about ordering these publications, see “How to get Redbooks” on page 565. Note that some of the documents referenced here may be available in softcopy only.

- *WebSphere Process Server for z/OS: Configuring a Network Deployment Environment*

  http://www.redbooks.ibm.com/redpapers/pdfs/redp4388.pdf

- *“Patterns: Implementing an SOA using an Enterprise Service Bus*

  http://www.redbooks.ibm.com/abstracts/sg246346.html

- *WebSphere Application Server Network Deployment V6: High Availability Solutions”*

  http://www.redbooks.ibm.com/abstracts/sg246688.html

- *WebSphere Application Server V6 Scalability and Performance Handbook*

  http://www.redbooks.ibm.com/abstracts/sg246392.html

- *Web Services Handbook for WebSphere Application Server v6.1*

  http://www.redbooks.ibm.com/abstracts/sg247257.html

Other publications

These publications are also relevant as further information sources:

- *Web services Platform Architecture,* Sanjiva Weerawarana, et al. 2005 Prentice Hall, ISBN 0-13-148874-0

Online resources

These Web sites are also relevant as further information sources:

- Three articles on *"Achieving Web services interoperability between the WebSphere Web Services Feature Pack and Windows Communication Foundation"*

  http://www.ibm.com/developerworks/websphere/library/techarticles/0710_levay/0710_levay.html

- Three articles on "*JAX-WS client APIs in the Web Services Feature Pack for WebSphere Application Server V6.1"*

  http://www.ibm.com/developerworks/websphere/library/techarticles/0707_thaker/0707_thaker.html

- *Manage WebSphere resources with the WSDM support in the WebSphere Application Server V6.1 Web Services Feature Pack*

  http://www.ibm.com/developerworks/websphere/library/techarticles/0802_chang/0802_chang.html

- *Troubleshooting JAX-WS applications with the WebSphere Application Server V6.1 Feature Pack for Web Services*

  http://www.ibm.com/developerworks/websphere/library/techarticles/0803_adams/0803_adams.html

- *Web services interoperability with the WebSphere Web Services Feature Pack and Apache Axis2, Part 1: Test basic SOAP and WS-Addressing interoperability*

  http://www.ibm.com/developerworks/websphere/library/techarticles/0803_das/0803_das.html

- WebSphere Application Server v6.1 Infocenter

  http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/index.jsp

- WebSphere Application Server for z/OS program directory

  ftp://ftp.software.ibm.com/software/webserver/appserv/library/v61/BOS5G111.PDF

- *Don Bagwell, "Feature pack for Web services".*

  http://www-03.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/WP101084

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, at this Web site:

ibm.com/redbooks

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services

Index

G

H

I

J

Q

R

S

T

U

V

W

X

Z

Redbooks

Web Services Feature Pack for WebSphere Application Server V6.1

(1.0” spine)
0.875”<->1.498”
460 <-> 788 pages

Web Services Feature Pack for WebSphere Application Server V6.1

Asynchronous

Reliable

Secure

The WebSphere 6.1 Feature pack for Web services offers a new and improved implementation of Java Web services and improved usability. Some of the changes simplify existing solutions; others enable new solutions requiring more secure and reliable services, asynchronous interaction patterns, and more complex data models.

The Feature pack for Web services offers a preliminary implementation of the WS-I Reliable and Secure WS-I profile, which is expected to proceed to ratification through 2008.

This IBM Redbooks publication includes an extensive interoperability example using Reliable and Secure Web services with the Windows Communications Foundation. There are also lots of examples showing you how to use the feature pack, use the new Web services Policy sets, and code Java Web services using the JAX-WS specification.

For architects and designers, there are chapters on business scenarios that will benefit from the feature pack, and examples of patterns that are readily implemented using the feature pack.

INTERNATIONAL TECHNICAL SUPPORT ORGANIZATION

BUILDING TECHNICAL INFORMATION BASED ON PRACTICAL EXPERIENCE

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

For more information:
ibm.com/redbooks

SG24-7618-00 ISBN 0738431346