

WebSphere Version 6 Web Services Handbook Development and Deployment

**WebSphere Application Server V6.0
Rational Application Developer V6.0**

**Latest Web services
technologies and standards**

**Best practices and
advanced techniques**



**Ueli Wahli
Thomas Kjaer
Brett Robertson
Fumiko Satoh
Franz-Josef Schneider
Witold Szczepanik
Chris Whyley**



International Technical Support Organization

**WebSphere Version 6 Web Services Handbook
Development and Deployment**

July 2005

Archived

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

First Edition (July 2005)

This edition applies to IBM WebSphere Application Server Version 6.0 and IBM Rational Application Developer for WebSphere Software Version 6.0.

Most of the samples have been validated with early code of IBM Rational Application Developer Version 6.0.1, and the necessary changes are indicated in the text.

© Copyright International Business Machines Corporation 2005. All rights reserved.

Note to U.S. Government Users Restricted Rights -- Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Notices	xix
Trademarks	xx
Preface	xxi
Changes to the previous redbook SG24-6891	xxii
The team that wrote this redbook	xxiii
Become a published author	xxvi
Comments welcome	xxvi
Part 1. Web services concepts	1
Chapter 1. Web services introduction	3
Introduction	4
Service-oriented architecture	4
Characteristics	6
Requirements	6
Web services	7
Properties of Web services	9
A short history of Web services	10
Summary	12
More information	12
Chapter 2. Web services standards	13
Categorization of Web services standards	14
Core standards	16
Description and discovery	17
Messaging	20
Management	23
Business processes	25
Transactions	26
Security	27
User experience	29
J2EE 1.4 and Java JSRs	30
Web services organizations and groups	34
Internet Engineering Task Force	34
Java Community Process	34
OASIS	35
World Wide Web Consortium	35
Web Services Interoperability Organization	35

UDDI	36
Companies working on Web services	36
IBM	36
Microsoft	36
Vertical industry standards organizations	37
Summary	38
More information	38
Chapter 3. Introduction to SOAP	39
Overview	40
The three pillars of SOAP	41
Overall message format: Envelope with header and body	41
Encoding rules	42
RPC representation	42
SOAP elements	44
Namespaces	44
URN	45
SOAP envelope	45
Headers	46
Body	48
Error handling	48
Advanced topics	49
Data model	49
Mappings	51
Communication styles	52
Encodings	54
Messaging modes	55
Implementations	56
SOAP implementation general architecture	56
Apache SOAP 2.3 implementation	57
Apache Axis	62
WebSphere Web services engine	64
Microsoft SOAP Toolkit	65
Other toolkits and server implementations	65
Outlook	65
Summary	66
More information	67
Chapter 4. Introduction to WSDL	69
Overview	70
WSDL document	70
WSDL document anatomy	71
WSDL definition	77

Types	77
Messages	78
Port types	79
Bindings	81
Service definition	82
Port definition	83
WSDL bindings	83
SOAP binding	84
HTTP binding	85
MIME binding	86
WSDL API	86
Outlook	88
Summary	89
More information	89
Chapter 5. JAX-RPC (JSR 101)	91
Terminology: JAX-RPC and JSR 101	92
JAX-RPC basics	92
JAX-RPC client	94
JAX-RPC client programming styles	95
Static stub	96
Dynamic proxy	97
Dynamic invocation interface (DII)	98
Which style to use	98
Managed and unmanaged JAX-RPC clients	99
JAX-RPC specification details	99
Data type mapping: XML to Java, Java to XML	100
Summary	101
More information	101
Chapter 6. Web Services for J2EE	103
Web services for J2EE overview	104
Client programming model	105
Overview	105
Client concepts	106
Packaging	108
Roles	110
Server programming model	111
Overview	111
Server concepts	112
Packaging	116
Roles	117
Transactions	118

Handlers	118
Security	119
WSEE implementations in WebSphere	120
SOAP over HTTP	120
SOAP over JMS	121
Summary	122
More information	122
Chapter 7. Introduction to UDDI	123
UDDI overview	124
Static versus dynamic Web services	124
UDDI registry structure	125
Interactions with UDDI	127
New features in UDDI Version 3	129
Keys assigned by publisher	129
Human-friendly URI-based keys	130
Complex registry topologies	130
Advanced security features	130
Policies	130
Data model updates	131
Extended inquiry API	131
Subscription API	131
Registry management	131
UDDI support in WebSphere Application Server	132
Advanced features of UDDI	132
Modeling features for complex business entities	132
External taxonomies	132
Powerful inquiry	133
Internationalization features	133
Peer-based replication	134
UDDI business registries on the Web	134
Web front ends for registries	136
Finding information	137
Publishing information	139
Java APIs for dynamic UDDI interactions	141
UDDI4J overview	142
Writing UDDI clients	143
Private UDDI registries	148
Motivation for the use of private UDDI registries	148
Possible scenarios for private UDDI registries	149
Benefits of private UDDI registries	150
Additional considerations for private UDDI registries	150
WebSphere private UDDI registry	151

Summary	154
More information	154
Chapter 8. Web Services Inspection Language	155
Overview	156
WS-Inspection document	157
WS-Inspection document anatomy	158
WS-Inspection and UDDI relationship	160
WS-Inspection definition	161
Services	162
Links	163
WS-Inspection bindings	164
WSDL binding	164
UDDI binding	166
WS-Inspection document publishing	167
WS-Inspection examples	168
WS-Inspection API	170
Outlook	172
Summary	172
More information	172
Chapter 9. Web services security	173
Security overview	174
Web services security exposures	175
WS-Security	177
Evolution of the WS-Security specification	179
WS-Security support in WebSphere Application Server	180
WS-Security road map	181
When to use WS-Security	182
Example of WS-Security	182
Authentication	184
Integrity	185
Confidentiality	189
Transport-level security	192
SOAP/HTTP transport-level security	192
When to use transport-level security	193
Summary	194
More information	194
Chapter 10. Web services interoperability	195
Definition	196
Web Services Interoperability Organization	196
WS-I Basic Profile V1.1 and Simple SOAP Binding Profile V1.0	197
WS-I Attachments Profile V1.0	197

WS-I tools	198
WS-I conformance claims	199
WebSphere interoperability	200
Interoperability with .NET	201
RPC/literal WSDL	201
WS-I conformance claims	202
SwA not supported	202
WSDL import statements	202
Mandatory header handling	203
UTF-16 WSDL	203
User exception handling	203
Object inheritance	204
Null and empty array handling	204
Null primitives and dates	205
WS-Security support	205
Representation of arrays in WSDL	205
Summary	207
More information	207
Chapter 11. Web services architectures	209
Service-oriented architecture	210
Enterprise service bus	211
Web services versus service-oriented architectures	212
Web services protocol stack	213
Message exchange patterns	215
One-way	215
Asynchronous two-way	216
Request-response	217
Workflow-oriented	218
Publish-subscribe	219
Composite	220
SOAP processing model	221
Web service gateways	222
Summary	224
More information	224
Chapter 12. Best practices	225
Generic best practices	226
Be WS-I compliant	226
Use simple data types	226
Avoid nillable primitives	226
Avoid fine-grained Web services	226
Avoid Web services for intra-application communication	227

Use short attribute, property, and tag names	227
Avoid deep nesting of XML structures	227
Apply common sense (also known as being defensive)	228
WebSphere Application Server best practices	228
Use the WebSphere Web services engine	228
Use caching of Web services as provided by the platform	228
Summary	229
More information	229
Part 2. Implementing and using Web services	231
Chapter 13. IBM products for Web services	233
WebSphere Application Server Version 6	234
What is new in WebSphere Application Server Version 6	234
Rational software development products	236
Rational Web Developer for WebSphere Software Version 6	236
Rational Application Developer for WebSphere Software Version 6	237
Rational Software Architect	237
WebSphere Studio Application Developer Integration Edition V5.1	237
WebSphere Studio Enterprise Developer Version 5.1.2	237
Web services support in Rational Application Developer	238
Web services tooling	238
Web services runtime environments	240
Web services configuration settings	241
Web services preferences	242
Summary	245
More information	245
Chapter 14. Sample application: Weather forecast	247
Weather forecast application components	248
Service modules	248
Business module	249
Data module	249
Back-end module	250
Information flow	250
Weather forecast application implementation	252
Weather database	253
Extract of the source code	253
Data transfer object: Weather	254
Business object: WeatherForecast	254
Data access: WeatherDAO	256
Predictor: WeatherPredictor	258
JavaBean Web service: WeatherJavaBean	259
EJB Web service: WeatherEJB	259

Summary	260
Chapter 15. Development overview	261
Overview	262
Build phase	263
Deploy phase	263
Run phase	263
Manage phase	263
Building a new Web service	264
Bottom-up	264
Top-down	265
Composing a Web service	266
Types of Web services implementation	267
Building a new Web service client	267
Static client	268
Dynamic client with known service type	269
Dynamic client with unknown service type	271
Types of client implementations	271
Summary	272
More information	272
Chapter 16. Develop Web services with Application Developer V6.0 ..	273
Overview	274
Selected scenarios	274
Preparation	275
Creating a Web service from a JavaBean	275
Web Service wizard	275
Generated files	283
Testing the Web service	286
Creating Web service clients	288
Stand-alone Java client	289
Web JavaServer Faces client	291
Creating a Web service from a session bean	297
Running the Web Service wizard	297
Generated code	298
Running clients against the EJB Web service	298
Creating a Web service top-down from WSDL	299
Creating the skeleton JavaBean	299
Implementing the JavaBean skeleton Web service	301
Creating a composed Web service	301
Web services and JMS binding	303
Creating an EJB router project	303
Running the Web Service wizard	303

Creating a J2EE application client for SOAP over JMS	306
Running the managed client	306
Running the managed client outside of Application Developer.	307
Creating a Web service from a URL.	308
Using Web service handlers.	309
Creating a server-side Web service handler	309
Testing the Web service handler.	311
Creating a client-side Web service handler.	312
Using handlers in a stand-alone client	313
Handlers and handler chains	315
Closing comments about using handlers	315
Using attachments	316
How do I use attachments today?	317
Implementing attachments	317
Implementing clients	322
Generating WSDL from Java	325
Web Services Atomic Transaction	326
Deployment descriptors for atomic transactions	328
Implementing a simple atomic transaction	328
Activating atomic transaction support	332
Testing atomic transactions	332
SOAP message for atomic transaction	333
Summary	335
 Chapter 17. Test and monitor Web services	 337
Testing Web services	338
Testing modes.	338
Testing approaches.	339
Web Services Explorer.	339
Starting the Web Services Explorer	340
Working with the Web Services Explorer	341
Web services sample test JSPs	345
Generating client-side code	345
Testing with test client JSPs	347
Universal Test Client	349
Starting the Universal Test Client	349
Testing a Web service with the Universal Test Client	349
Testing an EJB with the Universal Test Client	351
Web services component test	352
Creating a Web service component test	352
Monitoring with Tivoli Performance Viewer	363
Enabling and starting Tivoli Performance Viewer	363
Monitoring Web service requests	365

TCP/IP Monitor	368
Defining a TCP/IP Monitor configuration	368
TCP/IP Monitor view	369
Monitoring one server	372
WebSphere Application Server TCP/IP Monitor	372
Starting tcpmon	372
Summary	375
More information	375
 Chapter 18. Deploy and run Web services in WebSphere Application Server V6.0	 377
Overview	378
WebSphere Application Server general concepts	378
Administration basics	378
WebSphere Application Server topology building blocks	378
Upgrading from Version 5	380
Administrative console	380
Enterprise application deployment	382
Configuring the server with a data source	382
Installing an enterprise application	384
Regenerating the HTTP server plug-in	391
Starting and stopping enterprise applications	391
Web services deployment in WebSphere environment	392
Web services enabling with the SoapEarEnabler tool	392
Running the applications against the real server	392
Running a stand-alone client against the real server	393
Summary	394
More information	394
 Chapter 19. Command-line tools, Ant, and multiprotocol binding	 395
Command-line tools	396
Using the command-line tools	397
Bean2WebService	397
Deploying a Web service to a server	402
Running the generated client	403
EJB2WebService	404
WSDL2WebService	405
WSDL2Client	408
UDDIPublish/UDDIUnpublish	411
WebSphere Web services Ant tasks	412
Creating Web services Ant tasks	413
Running the Ant script	418
Conclusions about Ant	418

Multiprotocol binding	418
Command-line example for multiprotocol binding	419
WSDL file with EJB binding	420
Generating a client	422
Summary	424
Part 3. Advanced Web services techniques	425
Chapter 20. Web services interoperability tools and examples	427
Interoperability tools in Application Developer	428
Setting the compliance level	428
WSDL validator	429
WS-I message validation	429
Interoperability examples	431
Prerequisites	431
Apache Axis V1.0 example	431
Client development	431
Comparing the Axis client to the WebSphere client	433
Conclusion	433
Microsoft .NET example	434
Environment setup	434
Client development	435
Client test	441
Verifying WS-I compliance	442
Summary	443
More information	443
Chapter 21. Securing Web services	445
Overview	446
Typical scenario for WS-Security	447
Authentication	448
Integrity and confidentiality	450
Kerberos	454
Establishing a security context	455
Features of WS-Security in Application Server V6.0	458
Supported specifications	458
Unsupported specifications	460
Extensions in WebSphere Application Server V6.0	460
Architecture and deployment model	467
High-level architecture	467
Configuration structure	469
Development of WS-Security	474
How to define WS-Security configuration	475
Generating sample key stores	480

Authentication	484
Integrity	498
Confidentiality	516
Adding a security timestamp	528
Testing on WebSphere Application Server	531
Enabling security on the server	531
Enabling the TCP/IP Monitor	534
Testing the application with WS-Security	534
Debugging and tracing	540
Typical errors	541
Generating WS-Security sample configurations	543
Running the Web Service wizard with security	543
Scenario to generate and modify security definitions	544
Modifying generated definitions to use own key stores	545
Adding authentication and timestamp	547
Configuring WS-Security on a real Application Server	547
Modifying binding configurations	547
Adding a custom JAAS configuration	551
Configuring the distributed nonce cache in a cluster environment	552
Configuring certificate caching	554
Compatibility and migration with Version 5.x	555
WS-Security runtimes in WebSphere Application Server V6.0	556
Migrating from Version 5.x	556
How to migrate an application with Version 5.x WS-Security	557
Summary	558
More information	558
Chapter 22. Web services and the service integration bus	559
Overview	560
Motivation for using the bus	562
Installation	563
Installing the SDO repository	563
Installing the resource adapter	564
Installing the core application	565
Installing the endpoint listener applications	565
Using the bus	566
Performing administrative tasks	566
Buses	569
Endpoint listeners	571
Inbound services	573
Outbound services	575
UDDI references	577
Publishing WSDL for services	578

Web services gateway	580
Gateway instances	581
Gateway services	582
Proxy services	584
Message manipulation	588
JAX-RPC handlers	588
Mediations	590
Security	595
Web services security (WS-Security) in the bus	595
HTTP endpoint listener authentication	598
Operation-level authorization	600
Using HTTPS with the bus	601
Proxy server authentication	601
Gateway migration from Application Server V5.x	602
Using the Version 6.0 gateway migration utility	603
Gateway coexistence with Version 5.x	605
Advanced bus configuration	607
Accessing bus services directly using a JAX-RPC client	607
Using the bus with the weather forecast application	610
Preparation	610
Configuration steps	611
Summary	617
Troubleshooting	617
More information	617
Chapter 23. Implementing a private UDDI registry	619
Installing a private UDDI registry	620
Installing using the integrated test environment	620
Installing using WebSphere Application Server Network Deployment	621
Using the UDDI registry	622
Using the UDDI GUI	622
Using the UDDI Explorer with the private UDDI registry	628
Using command-line tools	630
Summary	631
More information	631
Chapter 24. Web services caching	633
Web services caching	634
Caching Web services	635
Cache configuration	635
Creating the cache configuration file	636
Cache monitor	638
Web services server cache	638

Web services client cache	641
Simple load test	643
Test result analysis	645
Summary	646
More information	646
Part 4. Appendixes	647
Appendix A. Installation and setup	649
Installation planning	650
WebSphere Application Server Version 6.0 requirements	650
Installing WebSphere Application Server V6.0.	651
Installing IBM HTTP Server.	653
Installing the Web server plug-ins	653
Installing Rational Application Developer V6.0.	654
Installing fixes	655
Installing the Agent Controller	656
Setting up Rational Application Developer	657
Starting Application Developer	657
Starting the WebSphere Application Server V6.0 test environment	659
Configuring Application Developer	659
WebSphere Application Server profiles	660
Multiple profiles for Application Developer	660
Creating a profile for the Application Developer test environment	660
Using the new server in Application Developer.	661
Using a remote server.	662
Installing the sample code	663
Appendix B. WS-Security configuration details: Mapping V5/V6, predefined properties, sample forms	665
Configuration mapping: Version 5.x to Version 6.0	666
Mapping for authentication	666
Mapping for integrity	683
Mapping for confidentiality	686
Mapping for timestamp	688
List of predefined properties.	689
Property list of deployment descriptor extension	689
Forms for WS-Security configuration	692
Deployment descriptor configuration.	692
Binding configuration.	695
Appendix C. Additional material	701
Locating the Web material	701
Using the Web material	702

System requirements for downloading the Web material	702
How to use the Web material	702
List of enterprise applications	703
Installing the base weather forecast application.	705
Importing the base applications	705
Setting up the WEATHER database	708
Selecting DB2 or Cloudscape	709
Deploying the enterprise applications to the server	710
Testing the enterprise applications	711
Setting up messaging for the JMS Web service.	713
Starting the administrative console	713
Creating a service integration bus.	714
Creating the queue connection factories.	716
Creating the queue	719
Creating an activation specification.	720
Important JNDI names	720
Saving the changes.	721
Configuring JMS using a JACL script	721
Restarting the server.	722
Importing solutions	723
Abbreviations and acronyms	725
Related publications	729
IBM Redbooks	729
Other publications	730
Online resources	730
How to get IBM Redbooks	731
Help from IBM	731
Index	733

Notices

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

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

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

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

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

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

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

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

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

COPYRIGHT LICENSE:

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

Trademarks

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

ClearCase®
Cloudscape™
CICS®
DB2®
developerWorks®

@server®
IBM®
ibm.com®
Rational®
Redbooks™

Redbooks (logo) ™
Tivoli®
WebSphere®
zSeries®

The following terms are trademarks of other companies:

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

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

Intel, Intel logo, Intel Inside, Intel Inside logo, Intel Centrino, Intel Centrino logo, Celeron, Intel Xeon, Intel SpeedStep, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

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

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® Redbook describes the concepts of Web services from various perspectives. It presents the major building blocks on which Web services rely. Here, well-defined standards and new concepts are presented and discussed.

While these concepts are described as vendor independent, this book also presents the IBM view and illustrates with suitable demonstration applications how Web services can be implemented using the IBM product portfolio, especially IBM WebSphere® Application Server Version 6.0 and IBM Rational® Application Developer for WebSphere Software Version 6.0.

This redbook is a rewrite of the redbook *WebSphere Version 5.1 Application Developer 5.1.1 Web Services Handbook*, SG24-6891-01. The new book covers the latest specifications in regard to Web services and Web services security. The new book uses the same weather forecast application as the base for all examples, but updated to Version 6 of the products.

This book is structured into three parts:

- ▶ Part 1 presents the underlying concepts, architectures, and specifications for the use of Web services.
- ▶ Part 2 shows how Web services can be implemented and deployed using the latest IBM products. Here, we introduce the weather forecast application, which we use in many ways to demonstrate these concepts and features.
- ▶ Part 3 shows some advanced techniques, such as Web services security, interoperability, and the service integration bus.

Changes to the previous redbook SG24-6891

New in this version of the publication are:

- ▶ The description of updated standards and specifications, as well as a complete list of applicable standards
- ▶ Two chapters about interoperability, including a Microsoft® .NET example
- ▶ A chapter about architectures
- ▶ A chapter about best practices
- ▶ A chapter about testing, including using the component test facility
- ▶ A chapter about command-line tools and Apache Ant (replacing the WebSphere Software Development Kit for Web Services information)
- ▶ A chapter about the service integration bus
- ▶ A chapter about Web services caching
- ▶ An appendix with details about the Web services security implementation

Removed in this version of the publication are:

- ▶ The chapter about workflows and business processes, because a Version 6 implementation is not available yet
- ▶ The chapter about the Web services gateway, because its function is now part of the service integration bus
- ▶ The chapter about Rational Application Developer Integration Edition, because a Version 6 product is not available yet
- ▶ The chapter about the WebSphere Software Development Kit (SDK) for Web Services, because its tools are now part of Rational Application Developer and are covered in the chapter about command-line tools
- ▶ The chapter about the Web services scenario, because we implemented many different versions of the weather forecast application

All the samples have been tested on WebSphere Application Server V6.0 and Rational Application Developer V6.0. The weather forecast application has been rewritten using multiple components.

The team that wrote this redbook

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



Ueli Wahli is a Consultant IT Specialist at the IBM International Technical Support Organization in San Jose, California. Before joining the ITSO 20 years ago, Ueli worked in technical support at IBM Switzerland. He writes extensively and teaches IBM classes worldwide about application development, object technology, WebSphere Application Server, and WebSphere Studio and Rational application development products. In his ITSO career, Ueli has produced more than 30 IBM Redbooks™. Ueli holds a degree in Mathematics from the Swiss Federal Institute of Technology.

Thomas Kjaer is an IT Specialist at IBM Global Services in Aarhus, Denmark. He has eight years of experience in application development, system integration, and Java™ consulting. He holds a B.A. in Information Science from Aarhus University, Denmark and has co-authored the IBM Redbook *IBM WebSphere Commerce Suite SPE Customization*, SG24-5958. His areas of expertise include application design and architecture on the WebSphere product family, including WebSphere Application Server, Commerce Suite, and Application Developer.

Brett Robertson is an IT Architect working for IBM Global Services, based in Sydney, Australia. He has 10 years of experience in the telecommunications and banking industries and has been working exclusively on e-business projects since 1997 in application development. His areas of expertise include architecture, J2EE and Java technologies, Web infrastructure, and security. Brett holds a B.E. in Computer Engineering from the University of New South Wales, Sydney.

Fumiko Satoh is a researcher at the Tokyo Research Laboratory, Japan. She has been with IBM Research for four years, focusing on Web services security. Her experience includes studying metadata for video digest, studying Web services for mobile devices, developing Web services applications with Web services security for customers, and designing and developing the Web services security runtime for WebSphere Application Server. Her current research interests include securing Web services and Web services security-related specifications. She received her Master's degree in Physics from the Tokyo Institute of Technology. Fumiko worked remotely in Japan to write two chapters.

Franz-Josef Schneider is an IT Specialist for the IBM Software Services for WebSphere consultancy, based in Mainz, Germany. He has five years of experience in supporting and consulting projects on J2EE and WebSphere Application Server. Josef has a University of Applied Sciences degree in electrical engineering. His areas of expertise cover various application development and infrastructure aspects: workshops, design and implementation solutions, performance monitoring and tuning, applying best practices, and problem analysis techniques.

Witold Szczeponek is a Senior IT Architect working for IBM Global Services, Germany. He has been with IBM for nine years, working as a programmer, application designer, and solution architect in areas of network management, telecommunications network management, and Web applications. His experience includes object-oriented analysis, design, and implementation with a focus on J2EE applications. Witold holds Master's degrees from the University of Oklahoma, U.S., and the Technische Universität Braunschweig, Germany. He also co-authored the first IBM Redbook about IBM Patterns for e-business and has reviewed and contributed to several publications about Web services and J2EE architectures.

Chris Whyley is a Technical Leader in Web services testing at the IBM Hursley software laboratory in the U.K. He has been with IBM for six years and has worked with a variety of companies in the banking, telecommunications, and media industries, implementing e-business solutions. Chris's skills lie in IP network management, J2EE application programming, and Web development, where he holds a patent award for an aspect of e-commerce design. Chris has a degree in Modern European History from the University of Warwick and is an ISEB certified software tester.

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

- ▶ Chris Brealey, from the IBM Software Group in Toronto, Canada, for his general help on the Application Developer tooling for Web services
- ▶ Peter Swithinbank from the Raleigh ITSO Center for his assistance with .NET interoperability
- ▶ Dave Russell from the IBM Software Group in Toronto, Canada, for his WS-I expertise and reviews of the interoperability chapters
- ▶ Zina Mostafia from the IBM Software Group in Toronto, Canada, for providing a fix for a problem that hindered us to work with the multiprotocol bindings
- ▶ Ken Ueno, Masayoshi Teraguchi, Yasumasa Kajinaga, Takeshi Imamura, Satoshi Makino, and Yuichi Nakamura from the Tokyo Research Lab, and Henry Chung from WebSphere Security Development, for their help on Web services security
- ▶ Jean-Philippe Delpiroux and Christophe Telep from IBM France for their help with the component test example
- ▶ Barry Gunner, Mark Lewis, Anthony Elder, Ian Lerner, Malcolm Ayres, and Paul Harris from IBM Hursley, U.K., for their help with the service integration bus
- ▶ Stan Cox from the IBM Software Group in Raleigh for his help with WebSphere Web services caching
- ▶ Rich Scheuerle from the IBM Software Group in Austin for his expertise and reviews in regard to the WebSphere Web services engine

Become a published author

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

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

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

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

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

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

ibm.com/redbooks

- ▶ Send your comments in an e-mail to:

redbook@us.ibm.com

- ▶ Mail your comments to:

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



Part 1

Web services concepts

In this part, we introduce the underlying concepts of Web services:

- ▶ Simple Object Access Protocol (SOAP)
- ▶ Web Services Description Language (WSDL)
- ▶ JAX-RPC (JSR 101)
- ▶ Web Services for J2EE (JSR 109)
- ▶ Universal Description, Discovery, and Integration (UDDI)
- ▶ Web Services Inspection Language (WSIL)
- ▶ Web services security
- ▶ Web services interoperability

Then, we discuss architectural patterns and provide guidelines:

- ▶ Web services architectures
- ▶ Best practices

Web services introduction

This chapter introduces Web services, a technology that enables you to invoke applications using Internet protocols and standards. The technology is called “Web services” because it integrates services (applications) using Web technologies (the Internet and its standards).

Even though this document focuses on Web services, which we cover in detail in the subsequent chapters, we first introduce a much broader concept, called the *service-oriented architecture* (SOA), that promises to better integrate today’s highly heterogeneous environments using an approach that links services together to build complex, yet manageable solutions. We then show how Web services implement a service-oriented architecture.¹

¹ It should be noted that Web services are not the only technology that can be used to implement service-oriented architectures.

Introduction

There is a strong trend for companies to integrate existing systems to implement IT support for business processes that cover the entire business cycle. Today, interactions already exist using a variety of schemes that range from very rigid point-to-point electronic data interchange (EDI) interactions to open Web auctions. Many companies have already made some of their IT systems available to all of their divisions and departments, or even their customers or partners on the Web. However, techniques for collaboration vary from one case to another and are thus proprietary solutions; systems often collaborate without any vision or architecture.

Thus, there is an increasing demand for technologies that support the connecting or sharing of resources and data in a very flexible and standardized manner. Because technologies and implementations vary across companies and even within divisions or departments, unified business processes could not be smoothly supported by technology. Integration has been developed only between units that are already aware of each other and that use the same static applications.

Furthermore, there is a need to further structure large applications into building blocks in order to use well-defined components within different business processes. A shift towards a *service-oriented* approach will not only standardize interaction, but also allows for more flexibility in the process. The complete value chain within a company is divided into small modular functional units, or services. A service-oriented architecture thus has to focus on how services are described and organized to support their dynamic, automated discovery and use.

Companies and their sub-units should be able to easily provide services. Other business units can use these services in order to implement their business processes. This integration can be ideally performed during the runtime of the system, not just at the design time.

Service-oriented architecture

This section is a short introduction to a service-oriented architecture, its key concepts, and requirements. You will find a more thorough description of service-oriented architectures in Chapter 11, “Web services architectures” on page 209. It should be observed that the presented architecture makes no statements about the infrastructure or protocols it uses. Therefore, the service-oriented architecture can be implemented not only using Web technologies.

A service-oriented architecture consists of three basic components:

- ▶ Service provider
- ▶ Service broker
- ▶ Service requestor

However, each component can also act as one of the two other components. For instance, if a service provider needs some more information that it can only acquire from some other service, it acts as a service requestor while still serving the original request. Figure 1-1 shows the operations each component can perform.

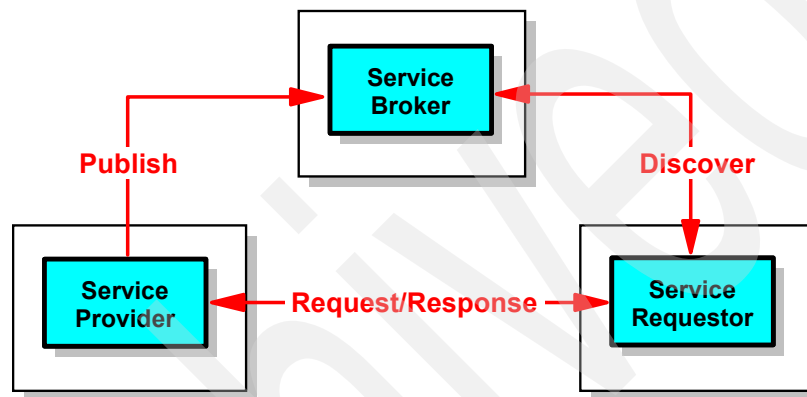


Figure 1-1 Web services components and operations

- ▶ The *service provider* creates a Web service and possibly publishes its interface and access information to the service broker.

Each provider must decide which services to expose, how to make trade-offs between security and easy availability, and how to price the services (or, if they are free, how to exploit them for other value). The provider also has to decide what category the service should be listed in for a given broker service and what sort of trading partner agreements are required to use the service.

- ▶ The *service broker* (also known as *service registry*) is responsible for making the Web service interface and implementation access information available to any potential service requestor.

The implementers of a broker have to decide about the scope of the broker. Public brokers are available all over the Internet, while private brokers are only accessible to a limited audience, for example, users of a company-wide intranet. Furthermore, the width and breadth of the offered information has to be decided. Some brokers will specialize in breadth of listings. Others will offer high levels of trust in the listed services. Some will cover a broad landscape of services, and others will focus within a given industry. Brokers

will also arise that simply catalog other brokers. Depending on the business model, a broker might attempt to maximize the look-up requests, number of listings, or accuracy of the listings.

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

One important issue for users of services is the degree to which services are statically chosen by designers compared to those dynamically chosen at runtime. Even if most initial usage is largely static, any dynamic choice opens up the issues of how to choose the best service provider and how to assess quality of service. Another issue is how the user of services can assess the risk of exposure to failures of service suppliers.

Characteristics

The presented service-oriented architecture employs a loose coupling between the participants. Such a loose coupling provides greater flexibility:

- ▶ In this architecture, a client is not coupled to a server, but to a service. Thus, the integration of the server to use takes place outside of the scope of the client application programs.
- ▶ Old and new functional blocks are encapsulated into components that work as services.
- ▶ Functional components and their interfaces are separated. Therefore, new interfaces can be plugged in more easily.
- ▶ Within complex applications, the control of business processes can be isolated. A business rule engine can be incorporated to control the workflow of a defined business process. Depending on the state of the workflow, the engine calls the respective services.
- ▶ Services can be incorporated dynamically during runtime.
- ▶ Bindings are specified using configuration files and can thus easily be adapted to new needs.

Requirements

For an efficient use of a service-oriented architecture, a number of requirements have to be fulfilled:

- ▶ **Interoperability between different systems and programming languages**

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

► **Clear and unambiguous description language**

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

► **Retrieval of the service**

To allow a convenient integration at design time or even runtime of the system, we require a mechanism that provides search facilities to retrieve suitable available services. Such services should be classified into computer-accessible, hierarchical categories, or taxonomies, based on what the services in each category do and how they can be invoked.

► **Security**

Protection of the services, including the information passed to and received from the service against unauthorized and malicious access, must be supported by the platform to win the confidence of the requestor (chain)—at the end the business customers. The type and extent of security depends on the type and placement of the participants—service requestors and service providers—and the services themselves. Service usage monitoring and security incident action plans have to be in place to detect unauthorized access (attempts) and trigger counter measures. Security is required to empower and retain authenticated and authorized requestors/customers while fencing off everything and everyone.

Web services

Web services are a relatively new technology that implements a service-oriented architecture. During the development of this technology, a major focus was put on making functional building blocks accessible over standard Internet protocols that are independent from platforms and programming languages.

If we had to describe Web services using just one sentence, we would use the following:

Web services are self-contained, modular applications that can be described, published, located, and invoked over a network.

Web services perform encapsulated business functions, ranging from simple request-reply to full business process interactions. These services can be new applications or just wrapped around existing legacy systems to make them network-enabled. Services can rely on other services to achieve their goals.

Figure 1-2 shows the relationship between the core elements of Web services in a service-oriented architecture (SOA).

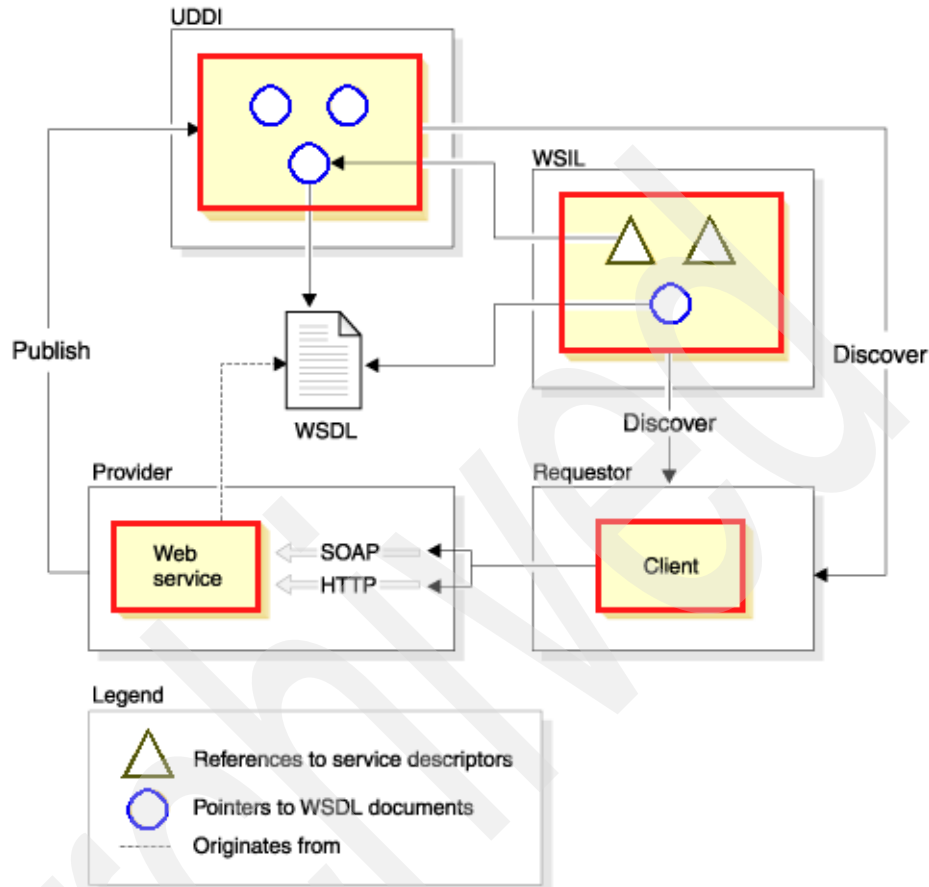


Figure 1-2 Main building blocks in an SOA approach based on Web services

The following are the core technologies used for Web services. These technologies are covered in detail in the subsequent chapters.

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

SOAP (Simple Object Access Protocol) is a network, transport, and programming language and platform-neutral protocol that allows a client to call a remote service. The message format is XML.

WSDL (Web Services Description Language) is an XML-based interface and implementation description language. The service provider uses a WSDL document in order to specify the operations a Web service

provides and the parameters and data types of these operations. A WSDL document also contains the service access information.

- WSIL** (Web Services Inspection Language) is an XML-based specification about how to locate Web services without the necessity of using UDDI. However, WSIL can be also used together with UDDI, that is, it is orthogonal to UDDI and does not replace it.
- UDDI** (Universal Description, Discovery, and Integration) is both a client-side API and a SOAP-based server implementation that can be used to store and retrieve information on service providers and Web services.

Properties of Web services

All Web services share the following properties:

► **Web services are self-contained.**

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

► **Web services are self-describing.**

The definition of the message format travels with the message; no external metadata repositories or code generation tools are required.

► **Web services can be published, located, and invoked across the Web.**

This technology uses established lightweight Internet standards such as HTTP. It leverages the existing infrastructure. Some additional standards that are required to do so include SOAP, WSDL, and UDDI.

► **Web services are modular.**

Simple Web services can be aggregated to more complex ones, either using workflow techniques or by calling lower-layer Web services from a Web service implementation. Web services can be chained together to perform higher-level business functions. This shortens development time and enables best-of-breed implementations.

► **Web services are language-independent and interoperable.**

The client and server can be implemented in different environments. Existing code does not have to be changed in order to be Web service enabled. Basically, any language can be used to implement Web service clients and servers.²

² In this publication, however, we assume that Java is the implementation language for both the client and the server side of the Web service.

- ▶ **Web services are inherently open and standard-based.**

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

- ▶ **Web services are loosely coupled.**

Traditionally, application design has depended on tight interconnections at both ends. Web services require a simpler level of coordination that allows a more flexible reconfiguration for an integration of the services in question.

- ▶ **Web services are dynamic.**

Dynamic e-business can become reality using Web services, because with UDDI and WSDL, the Web service description and discovery can be automated. In addition, Web services can be implemented and deployed without disturbing clients that use them.

- ▶ **Web services provide programmatic access.**

The approach provides no graphical user interface; it operates at the code level. Service consumers have to know the interfaces to Web services but do not have to know the implementation details of services.

- ▶ **Web services provide the ability to wrap existing applications.**

Already existing stand-alone applications can easily be integrated into the service-oriented architecture by implementing a Web service as an interface.

- ▶ **Web services build on proven, mature technology.**

There are a lot of commonalities, as well as a few fundamental differences, with other distributed computing frameworks. For example, the transport protocol is text based and not binary.³

A short history of Web services

The Internet began its success story in the early nineties, even though it was used in the academic world before for many years. The main driver for the Internet's success was the World Wide Web, whose main innovation was the easy access to information, from any place, using standard Internet protocols and a simple data access protocol that enabled the implementation browsers on a variety of platforms. Together with the spread of the WWW, the Internet and its related technologies became the *de facto* standard to connect computers all around the world.

³ While using a text-based protocol adds to the Web services' capability of being platform independent, it also can be seen as a drawback, because the textual representation adds too much overhead on the protocol.

With the spread of the Internet, it became clear that the infrastructure that was introduced by the Internet could be used not just to retrieve information that was to be presented using a browser (called *human-to-application*, H2A, scenarios). Rather, there was also an increased demand for *application-to-application* (A2A) communication using the existing technologies. And, it was hoped that the existing protocols could be used for this purpose.

However, it soon became clear that this was not the case. HTTP had been designed with the retrieval of information in mind, following a very simple access path that basically relies on documents being linked together by means of hypertexts. The protocol does not provide for complex operations that arise from A2A scenarios. And some of the protocols that were defined at this time could not be used either because they did not fit into the Web world or they were too restrictive.

In late 1999, Microsoft published an XML-based protocol, called SOAP, that could be used for A2A scenarios. As it was one among many protocols suggested, it may be due to the fact that IBM started supporting SOAP in early 2000 that eventually led to a public acceptance of SOAP by the industry.

At this point in time, SOAP was just a protocol to perform complex A2A scenarios. However, it quickly gained popularity and it was clear that there was a need for better describing and finding the services that were implemented using SOAP. The term *Web services* was coined several months later, when IBM, Microsoft, and Ariba jointly published the Web Services Description Language (WSDL). Eventually, UDDI was also introduced, thus completing the set of standards and protocols that make up the basis of Web services.

In the following years, many propositions were made about how to improve the technology such that it can be used not just for simple service invocation but can also be leveraged in more demanding environments. Among the most important ones, the Web services security (WSS) suite of standards is of particular interest, because it allows for a quality of service that is required by many enterprises and organizations. As of this writing, more than 40 specifications and standards were published.

Summary

In this chapter, we introduced Web services and service-oriented architectures. Web services are self-contained, modular applications that can be described, published, located, and invoked over a network. We described the basic standards that are used in building service-oriented solutions and their relation to service-oriented architectures. We also described the history of Web services.

More information

General introductions to Web services can be found at:

<http://www.ibm.com/developerworks/webservices>

The following Web site provides a collection of IBM resources about Web services. For example, you can find an introduction to the SOA in a white paper titled *Web Services Conceptual Architecture* (WSCA 1.0):

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

More information is provided in the article *Energize e-business with Web services from the IBM WebSphere software platform*, available at:

<http://www.ibm.com/developerworks/library/ibm-lunar.html>

You can find more information about the history of SOAP at:

<http://www.ibm.com/developerworks/webservices/library/ws-ref1.html>

Also, the following site provides some more information about how the Web service standards evolved:

<http://www.mywebservices.org/index.php/article/articleview/1202/1/24/?PHPSESSID=7ac73a8d162a0ebb652bc5f0c7218886>

Web services standards

Today, there is an abundance of Web services standards¹ available, and it is not always easy to recognize how these standards are grouped and how they relate to each other. Unfortunately, there exists no such thing as a single and simple Web services protocol stack that would allow for an easy categorization of Web services standards. This is not really surprising, because the underlying concepts of message exchanging can be used for more than just message transport and service invocation and many of the standards cover more than one aspect of the Web service technology.

In this chapter, we categorize the Web services standards and provide a short description of each of them.

Web services standards are evolving at a rapid pace. It might be useful to consult an online reference of Web services standards, such as is hosted on the IBM developerWorks® Web site, available at:

<http://www.ibm.com/developerworks/views/webservices/standards.jsp>

¹ The term *standard* is often misleading. Most standards are just *specifications* that have been submitted to the public in order to gather feedback and to improve them without the burden of slowly moving standard committees or organizations. Nonetheless, they can be considered as standards because many vendors implement their products complying to these standards.

Categorization of Web services standards

To better clarify the abundance of Web services standards, we decided to group the different standards into categories that loosely depend on each other (Table 2-1). Although this representation resembles a protocol stack, it should not be understood as such.

Table 2-1 Web services standards categorized

Category	Standard	Jump
Core	▶ SOAP ▶ WSDL ▶ UDDI ▶ XML	Click Click Click Click
Description and discovery	▶ WS-Inspection (also called WSIL) ▶ WS-Discovery ▶ WS-MetadataExchange ▶ WS-Policy	Click Click Click Click
Messaging	▶ ASAP ▶ SOAP Messages with Attachments (SwA) ▶ SOAP MTOM ▶ WS-Addressing ▶ WS-Notification (consisting of) - WS-BaseNotification - WS-BrokeredNotification - WS-Topics ▶ WS-Eventing ▶ WS-Enumeration ▶ WS-MessageDelivery ▶ WS-Reliability ▶ WS Reliable Messaging ▶ WS-Resources (consisting of) - WS-ResourceProperties - WS-ResourceLifetime - WS-BaseFaults - WS-ServiceGroup ▶ WS-Transfer	Click Click Click Click Click Click Click Click Click Click Click Click
Management	▶ WSDM ▶ WS-Manageability ▶ SPML ▶ WS-Provisioning	Click Click Click Click
Business processes	▶ BPEL4WS ▶ WS-CDL ▶ WS-CAF	Click Click Click

Category	Standard	Jump
Transactions	▶ WS-Coordination ▶ WS-Transaction ▶ WS-AtomicTransaction ▶ WS-BusinessActivity	Click Click Click Click
Security	▶ XML-Encryption ▶ XML-Signature ▶ WS-Security ▶ WS-SecureConversation ▶ WS-SecurityPolicy ▶ WS-Trust ▶ WS-Federation ▶ SAML	Click Click Click Click Click Click Click Click
User experience	▶ WSRP	Click
J2EE and Java JSR	▶ JSR 101: JAX-RPC ▶ JSR 109: Implementing Enterprise Web Services ▶ JSR 31: JAXB ▶ JSR 67: JAXM ▶ JSR 93: JAXR ▶ JSR 110: WSDL4J ▶ JSR 172: J2ME Web Services ▶ JSR 173: StAX ▶ JSR 181: Web Services Metadata for Java ▶ JSR 208: JBI ▶ JSR 222: JAXB 2.0 ▶ JSR 224: JAX-RPC 2.0 ▶ JSR 921: Implementing Enterprise Web Services 1.1	Click Click Click Click Click Click Click Click Click Click Click Click Click

This collection is by no means exhaustive; although it is current at the time of writing, the Web services standards space is rapidly evolving and new specifications are being introduced every few months.

You can find an online version of all the Web services related standards at:

<http://www.ibm.com/developerworks/views/webservices/standards.jsp>

Note: When assembling the content this chapter, we read many standards and specifications found on the Internet. However, when we later tried to re-read some of the content, we found out that some of the URLs changed. Even though we verified all URLs at the end of writing, bear with us if we point you to some non-existent URL.

Core standards

As complex as Web services might appear when you look at them for the first time, there are only three standards that describe the core of Web services. These standards describe the protocol for Web services (SOAP), the language to describe Web services (WSDL), and the means to publish and discover Web services (UDDI).

Foundational standards on which core Web services standards build on include the XML family of specifications (XML and XML Schema) and the IETF HTTP(S) standards. At least some understanding of XML and its related standards is useful to better understand the Web services core standards.

None of the SOAP, WSDL or XML Schema standards specify the use of a particular version of the XML specification, WS-I Basic Profile Version 1.1 mandates the use of Version 1.0 of the XML W3C Recommendation to maximize interoperability.

SOAP: Simple Object Access Protocol

This standard provides the definition of the structured and typed XML-based information that is exchanged between parties in a distributed environment. SOAP messages are self-contained and describe the structure and type of the information they carry within the message. This allows for very flexible solutions.

Each SOAP message consists of an envelope that contains an arbitrary number of headers and one body that carries the payload. SOAP messages might contain exceptions to report failures or unexpected conditions.

Even though SOAP implements a stateless, one-way paradigm, it can be used to implement more complex paradigms such as request/response and solicit/response.

As of this writing, SOAP Version 1.2 is in use. For more information, refer to:

<http://www.w3.org/2000/xp/Group/>

WSDL: Web Services Description Language

This standard describes Web services as abstract service endpoints that operate on messages. Both the operations and the messages are defined in an abstract manner, while the actual protocol used to carry the message and the endpoint's address are concrete.

WSDL² is not bound to any particular protocol or network service. It can be extended to support many different message formats and network protocols. However, because Web services are mainly implemented using SOAP and HTTP, the corresponding bindings are part of this standard.

As of this writing, WSDL 1.1 is in use and WSDL 2.0 is a working draft. For more information, refer to:

<http://www.w3.org/TR/wsd1>

UDDI: Universal Description, Discovery, and Integration

The Universal Description, Discovery, and Integration standard defines means to publish and to discover Web services. Of the three Web services standards, this is the least important one, because one can also implement and deploy Web services without UDDI. However, in certain situations, UDDI is a must.

As of this writing, UDDI Version 3.0 has been finalized, but UDDI Version 2.0 is still more commonly used. For more information, refer to:

<http://www.uddi.org/>
<http://www.oasis-open.org/specs/index.php#wssv1.0>

XML

XML is the foundation of Web services and is not described in this document, but you can find more information about XML at:

<http://www.w3.org/XML/>

Note: The remainder of this chapter covers additional standards that do not make up the core standards defining Web services. You might want to skip these standards at the moment. You can always return to this chapter and obtain more information later. If you choose to skip the standards, you might want to continue with “Web services organizations and groups” on page 34.

Description and discovery

The standards and specifications in this category are related to describing and locating Web services either over the Internet or through means of local resources.

² The acronym WSDL is sometimes also referred to as *Web Services Definition Language*. It is an interesting side note that even the W3C does not use the term consistently.

WS-Inspection: Web Services Inspection Language (WSIL)

WS-Inspection describes how to locate Web service descriptions on some server and how this information needs to be structured. As such, WSIL can be viewed as a *lightweight* UDDI.

The WS-Inspection specification was developed jointly by Microsoft and IBM in November 2001. More information can be found at:

<http://www.ibm.com/developerworks/webservices/library/ws-wsilspec.html>

WS-Discovery

The Web Services Dynamic Discovery defines a multicast discovery protocol to locate Web services. By default, probes are sent to a multicast group, and target services that match return a response directly to the requester. To scale to a large number of endpoints, the protocol defines the multicast suppression behavior if a discovery proxy is available in the network. To minimize the need for polling, target services that want to be discovered send an announcement when they join and leave the network.

WS-Discovery was developed by Microsoft, BEA, Canon, and Intel®. For more details, refer to:

<http://msdn.microsoft.com/ws/2004/02/discovery>

WS-MetadataExchange

Web services use metadata to describe what other endpoints have to know to interact with them. Specifically, WS-Policy describes the capabilities, requirements, and general characteristics of Web services; WSDL describes abstract message operations, concrete network protocols, and endpoint addresses used by Web services; XML Schema describes the structure and contents of XML-based messages received and sent by Web services. To bootstrap communication with a Web service, the WS-MetadataExchange specification defines three request-response message pairs to retrieve these three types of metadata.

The public draft specification was produced by Microsoft, IBM, Computer Associates, SAP, BEA, and webMethods. More information can be found at:

<http://msdn.microsoft.com/library/en-us/dnglobspec/html/ws-metadataexchange.pdf>
<http://www.ibm.com/developerworks/library/specification/ws-mex/>

WS-Policy

WS-Policy provides a general purpose model and syntax to describe and communicate the policies of a Web service. WS-Policy defines a policy to be a collection of one or more policy assertions. Some assertions specify traditional requirements and capabilities that will ultimately manifest on the wire (for

example, authentication scheme, transport protocol selection). Some assertions specify requirements and capabilities that have no wire manifestation yet are critical to proper service selection and usage (for example, privacy policy, QoS characteristics). WS-Policy provides a single policy grammar to allow both kinds of assertions to be reasoned about in a consistent manner.

WS-Policy is a standard developed by IBM, BEA, Microsoft, SAP, Sonic, and VeriSign. Additional information can be found at:

<http://www.ibm.com/developerworks/library/specification/ws-polfram/>

WS-PolicyAssertions

This specification defines a common set of policy assertions for Web services. The assertions defined by this specification cover text encoding, natural language support, versioning of specifications, and predicates.

WS-PolicyAssertions is based on the WS-Policy framework that was described earlier. It was developed by BEA, IBM, Microsoft, and SAP. More information can be found at:

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

WS-PolicyAttachment

The WS-Policy specification defines an abstract model and an XML-based grammar for policies. It defines two general-purpose mechanisms for associating policies with the subjects to which they apply.

This specification is based on the WS-Policy framework that was described earlier. It was developed by BEA, IBM, Microsoft, SAP, Sonic, and VeriSign. More information can be found at:

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

DNS Endpoint Discovery (DNS-EPD)

This document introduces mechanisms for DNS-based discovery of Web service endpoints that represent common or well-known services. DNS-EPD introduces a process for resolving the location of common services that is similar in nature to using the telephone white pages directory (as opposed to business-oriented services that are more likely to need a taxonomical, yellow pages-like directory approach). With DNS-EPD, if a client wants to locate a specific instance of a Web service, the client would go to DNS and resolve the current location of that service by name.

DNS-EPD is a specification that has been developed by IBM and has been submitted to IETF as an Internet Draft. More information can be found at:

<http://www.ietf.org/internet-drafts/draft-snell-dnsepd-01.txt>

Messaging

Adding attachments to SOAP messages are a major area of activity within the Web services community. After much contention between standards bodies and key players, the W3C MTOM standard appears to be the emerging standard that will be adopted by the industry.

Other areas of change in the messaging and encoding stack include the WS-Addressing and WS-Notification specifications.

ASAP: Asynchronous Services Access Protocol

The ASAP standard aims to create a very simple extension of SOAP to allow for asynchronous or long-running Web services.

ASAP is a specification being developed by the OASIS organization. The technical committee has not yet developed a draft of the standard. More information can be found at:

http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=asap

SOAP Messages with Attachments (SwA)

This specification defines a binding for a SOAP 1.1 message to be carried within a MIME multipart/related message in such a way that the processing rules for the SOAP 1.1 message are preserved. The MIME multipart mechanism for encapsulation of compound documents can be used to bundle entities related to the SOAP 1.1 message such as attachments. Rules for the usage of URI references to refer to entities bundled within the MIME package are specified.

The SOAP with Attachments API for Java (SAAJ) implements the specification. This specification is a submission to the W3C. For more information, refer to:

<http://www.w3.org/TR/SOAP-attachments>

SOAP Message Transmission Optimization Mechanism

The SOAP Message Transmission Optimization Mechanism (MTOM) enables SOAP bindings to optimize the transmission or wire format, or both, of a SOAP message by selectively encoding portions of the message, while still presenting an XML Infoset to the SOAP application. Optimization is available only for binary content.

MTOM is a standard developed by W3C. It was originally a specification developed by AT&T, BEA, Microsoft, Canon, SAP, and Tibco called Proposed Infoset Addendum to SOAP Messages with Attachments (PASwA). The MTOM specification has reached the candidate recommendation stage.

More information can be found at:

<http://www.w3c.org/TR/2004/CR-soap12-mtom-20040826/>
<http://www.w3.org/TR/soap12-mtom/>

WS-Addressing

WS-Addressing defines how message headers direct messages to a service or agent, provides an XML format for exchanging endpoint references, and defines mechanisms to direct replies or faults to a specific location.

The WS-Addressing standard has been developed by BEA, IBM, Microsoft, SAP, and Sun Microsystems. It has been submitted to W3C and is expected to be ratified as a W3C Recommendation in the October 2005 time frame.

More information can be found at:

<http://www.w3c.org/2002/ws/addr/>
<http://www.ibm.com/developerworks/library/specification/ws-addr/>
<http://www.w3.org/Submission/ws-addressing/>

WS-Notification

WS-Notification defines a standardized way in which Web services interact using the notification (or publish-subscribe) pattern.

In the notification pattern, a Web service, or other entity, disseminates information to a set of other Web services, without having to have prior knowledge of these other Web services.

WS-Notification is a standard developed by IBM, Akamai Technologies, Globus Alliance, Hewlett-Packard, SAP, Sonic, and Tibco. It has been submitted to OASIS for ratification and a Working Draft document has been published. More information can be found at:

http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsn
<http://www.ibm.com/developerworks/library/specification/ws-notification>

WS-Eventing

The WS-Eventing specification defines a baseline set of operations that allow Web services to provide asynchronous notifications to interested parties. WS-Eventing defines the simplest level of Web services interfaces for notification producers and notification consumers, including standard message exchanges to be implemented by service providers that want to act in these roles, along with operational requirements expected of them.

The WS-Eventing specification is being developed by IBM, BEA, Computer Associates, Microsoft, Sun Microsystems, and Tibco. It provides similar functionality to that of WS-BaseNotification specification (WS-BaseNotification is

one of the WS-Notification specifications). IBM has joined the author group to align the two specifications and reduce potential for overlap and incompatibilities. More information can be found at:

<http://www.ibm.com/developerworks/webservices/library/specification/ws-eventing/>

WS-Enumeration

The WS-Enumeration specification defines a framework to access information using a cursor rather than retrieving all information with one Web service invocation. The means to implement the cursor is an enumeration context that can be used by clients to request the information.

The WS-Enumeration specification is being developed by BEA, Computer Associates, Microsoft, and Sonic. More information can be found at:

<http://xml.coverpages.org/WS-enumeration200409.pdf>

WS-MessageDelivery

This specification defines an abstract set of message delivery properties that enable message delivery for Web services that use message exchange patterns associated with WSDL documents. It allows for a definition of complex message exchange patterns and shows how it can be used to implement a *callback pattern*.

The WS-MessageDelivery specification is being developed by Arjuna, Cyclone Commerce, Enigmatec, IONA, Nokia, Oracle, SeeBeyond Technology, and Sun Microsystems. More information can be found at:

<http://www.w3.org/Submission/ws-messagedelivery/>

WS-Reliability

As of this writing, there is still contention between the standard submitted to OASIS and the IBM/Microsoft-backed specification. You can find more information at:

<http://www.oasis-open.org/committees/wsrn/charter.php>

WS-ReliableMessaging

This standard allows applications to send and receive messages simply, reliably, and efficiently even with application, platform, or network failure.

WS-ReliableMessaging is a standard developed by IBM, BEA, Microsoft, and Tibco. A competing standard of the same name is being developed by OASIS based on a submission by Fujitsu, Hitachi, Oracle, NEC, Sonic, and Sun Microsystems. For more information, refer to:

<http://www.ibm.com/developerworks/webservices/library/ws-rm/>
<http://www.oasis-open.org/committees/wsm/charter.php>
http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsm

WS-Resources

The Web services resource framework defines a family of specifications for accessing stateful resources using Web services. It includes the WS-ResourceProperties, WS-ResourceLifetime, WS-BaseFaults, and WS-ServiceGroup specifications. The motivation for these new specifications is that although Web service implementations typically do not maintain state information during their interactions, their interfaces must frequently allow for the manipulation of state, that is, data values that persist across and evolve as a result of Web service interactions.

WS-Resources is a standard developed by IBM, Globus Alliance, and Hewlett-Packard. The standard has been submitted to OASIS (WSRF) for ratification and Working Drafts have been published. More information can be found at:

<http://www.ibm.com/developerworks/library/ws-resource/index.html>
http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrf

WS-Transfer

The WS-Transfer specification defines a protocol to access XML-based entities using Web services technologies. It also defines means to create and delete entities through factories.

This specification was developed by BEA, Computer Associates, Microsoft, Sonic, and Systinet. More information can be found at:

<http://xml.coverpages.org/WS-transfer200409.pdf>

Management

This section lists the standards of the management category.

WSDM: Web Services Distributed Management

Web Services Distributed Management (WSDM) defines Web services management, including using Web services architecture and technology to manage distributed resources. The scope includes developing the model of a Web service as a manageable resource.

WSDM is a standard being developed by OASIS based on the WS-Management submission from IBM, Computer Associates, and Talking Blocks. More information can be found at:

<http://www.oasis-open.org/committees/wsdm/charter.php>
http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsdm

WS-Manageability

WS-Manageability defines the manageability model for a Web service and how to access that model using Web services. The specification defines Web services manageability as a set of capabilities for discovering the existence, availability, health, performance, and usage, as well as the control and configuration of a Web service within the Web services architecture.

This specification was developed by IBM, Computer Associates, and Talking Blocks. More information can be found at:

<http://www.ibm.com/developerworks/webservices/library/ws-manage/>

SPML: Service Provisioning Markup Language

The Service Provisioning Markup Language (SPML) is an OASIS-approved standard intended to define and standardize an XML-based framework for exchanging user, resource, and service provisioning information. More information can be found at:

<http://www.oasis-open.org/committees/download.php/4137/os-pstc-spml-core-1.0.pdf>

Information published by the OASIS Provisioning Services Technical Committee can be found at:

<http://www.oasis-open.org/committees/provision/charter.php>

WS-Provisioning

WS-Provisioning is a specification developed by IBM to facilitate interoperability between provisioning systems in a consistent manner. The specification has been submitted to the OASIS Provisioning Services Technical Committee to be considered for inclusion in the SPML V2.0 standard. The specification can be found at:

<http://www.ibm.com/developerworks/library/ws-provis/>

Information published by the OASIS Provisioning Services Technical Committee can be found at:

<http://www.oasis-open.org/committees/provision/charter.php>

Business processes

This section lists the standards of the business process category.

BPEL: Business Process Execution Language

This standard describes a notion to specify business process behavior using Web services. It is sometimes called *BPEL4WS*, Business Process Execution Language for Web Services. BPEL enables users to describe business process activities as Web services and define how they can be connected to accomplish specific tasks.

The WS-BPEL standard is currently at Version 1.1. It was developed by IBM, Microsoft, Siebel Systems, BEA, and SAP, and has been submitted to the OASIS organization. A technical committee has been formed, but a draft specification has not yet been published. More information can be found at:

<http://www.ibm.com/developerworks/library/ws-bpel/>

<http://www.ibm.com/developerworks/webservices/library/ws-bpel/>

http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel

WS-CDL

The Web Services Choreography Description Language (WS-CDL) is an XML-based language that describes peer-to-peer collaborations of parties by defining, from a global viewpoint, their common and complementary observable behavior, where ordered message exchanges result in accomplishing a common business goal.

WS-CDL is a complementary standard to the IBM, Microsoft, and BEA WS-BPEL and other choreography description languages. WS-CDL is developed by W3C and a Working Draft document was published on October 12, 2004. Additional information can be found at:

<http://www.w3c.org/2002/ws/chor/>

WS-CAF

The Web Services Composite Application Framework (WS-CAF) defines an open framework for supporting coordinated and transactional compositions of multiple Web services applications. WS-CAF is distinct from BPEL in that it takes an autonomous choreography approach compared to BPEL's directed orchestration. BPEL technology is designed for scenarios where there is a central point or organization in control of the business process.

At this point, OASIS has not produced a draft specification. For more information, refer to:

http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=ws-caf

Transactions

Figure 2-1 shows the relationships between transactions standards.

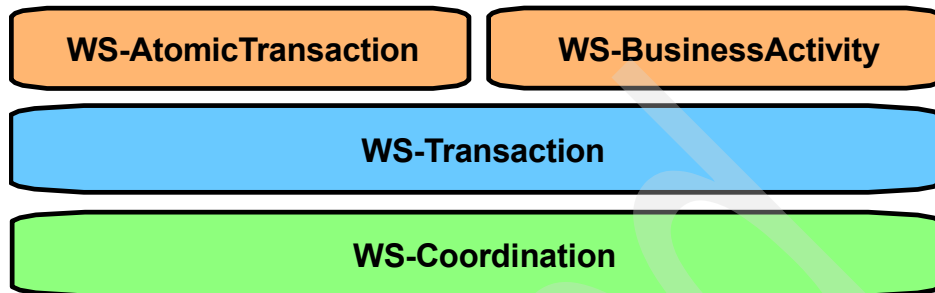


Figure 2-1 Relationship between WS-Transaction standards

WS-Coordination (WS-COOR)

The WS-Coordination specification describes an extensible framework for providing protocols that coordinate the actions of distributed applications. Such coordination protocols are used to support a number of applications, including those that have to reach consistent agreement on the outcome of distributed activities.

WS-Coordination is a standard developed by IBM, Microsoft, and BEA. A similar standard, WS-CAF, is being developed by OASIS based on WS-Context, WS-Coordination Framework, and WS-Transaction Management specifications published by Arjuna, Fujitsu, Iona, Oracle, and Sun Microsystems. Additional information can be found at:

<http://www.ibm.com/developerworks/library/ws-coor/>

http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=ws-caf

WS-Transaction

The WS-Transaction specification describes coordination types that are used with the extensible coordination framework described in the WS-Coordination specification. It defines two coordination types: WS-AtomicTransaction and WS-BusinessActivity. Developers can use either or both of these coordination types when building applications that require consistent agreement on the outcome of distributed activities.

WS-Transaction is a standard developed by IBM, BEA, and Microsoft. More information can be found at:

<http://www.ibm.com/developerworks/library/specification/ws-tx/>

WS-AtomicTransaction (WS-AT)

The WS-AtomicTransaction specification describes the atomic transaction coordination type that is based on the WS-Coordination framework. It defines an atomic transaction context and its usage. This specification supersedes the WS-Transaction specification. More information can be found at:

<http://www.ibm.com/developerworks/library/specification/ws-tx/>

WS-BusinessActivity (WS-BA)

The WS-BA specification describes the business activity coordination type that is based on the WS-Coordination framework. Contrary to the WS-AT specification, this specification deals with (possibly) long-running transactions, user interaction, and exception handling of business cases. It supersedes the WS-Transaction specification. For more information, refer to:

<http://www.ibm.com/developerworks/library/specification/ws-tx/>

Security

The standards of the security category deal with the security enablement of Web services.

XML-Encryption

This standard specifies a process for encrypting data and representing the result in XML. The data can be arbitrary data (including an XML document), an XML element, or an XML element content. The result of encrypting data is an XML encryption element that contains or references the cipher data.

XML-Encryption has been published as a W3C recommendation in December 2002. More information can be found at:

<http://www.w3.org/Encryption/2001/>

XML-Signature

This standard specifies a process for encrypting data and representing the result in XML. The data can be arbitrary data (including an XML document), an XML element, or an XML element content. The result of encrypting data is an XML encryption element that contains or references the cipher data.

XML-Signature has been published as a joint W3C and IETF recommendation in December 2002. More information can be found at:

<http://www.w3.org/Signature/>

WS-Security

The WS-Security specification describes extensions to SOAP that allow for *quality of protection* of SOAP messages. This includes, but is not limited to, message authentication, message integrity, and message confidentiality. The specified mechanisms can be used to accommodate a wide variety of security models and encryption technologies. It also provides a general-purpose mechanism for associating security tokens with message content.

The WS-Security 1.0 standard was published in March 2004. The standard also includes the UsernameToken profile and X.509 Certificate Token profile. Additional token profiles for REL and SAML are currently published as Committee Drafts. For additional information, refer to:

<http://www.oasis-open.org/specs/index.php#wssv1.0>

http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss

WS-SecureConversation

The Web Services Secure Conversation Language is built on top of the WS-Security and WS-Policy/WS-Trust models to provide secure communication between services. WS-Security focuses on the message authentication model but not a security context, and thus is subject several forms of security attacks. This specification defines mechanisms for establishing and sharing security contexts, and deriving keys from security contexts, to enable a secure conversation.

This is a draft specification developed by BEA, Computer Associates, IBM, Layer 7, Microsoft, Netegrity, Oblix, OpenNetwork, Ping Identity, Reactivity, RSA Security, VeriSign, and Westbridge. Additional information can be found at:

<http://www.ibm.com/developerworks/library/specification/ws-secon/>

WS-SecurityPolicy

The Web Services Security Policy Language defines a model and syntax to describe and communicate security policy assertions within the larger policy framework. It covers assertions for security tokens, data integrity, confidentiality, visibility, security headers, and the age of a message.

This is a draft specification developed by IBM, Microsoft, RSA, and VeriSign. More information can be found at:

<http://www.ibm.com/developerworks/library/ws-secpol/>

<http://www.ibm.com/developerworks/webservices/library/ws-secpol/>

WS-Trust

The Web Services Trust Language (WS-Trust) uses the secure messaging mechanisms of WS-Security to define additional primitives and extensions for security token exchange to enable the issuance and dissemination of credentials within different trust domains.

WS-Trust is a standard developed by IBM, BEA, Computer Associates, Layer 7, Microsoft, Netegrity, Oblix, OpenNetwork, Ping Identity, Reactivity, RSA Security, VeriSign, and Westbridge Technology. More information can be found at:

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

WS-Federation

This specification defines mechanisms that are used to enable identity, account, attribute, authentication, and authorization federation across different trust realms.

WS-Federation is a standard developed by IBM, BEA, Microsoft, RSA, and VeriSign. It is a competing standard to those developed by the OASIS SAML standard and the Liberty Alliance project. More information can be found at:

<http://www.ibm.com/developerworks/webservices/library/ws-fed/>

SAML: Security Assertion Markup Language

The Security Assertion Markup Language (SAML) is a suite of specifications that define interoperability between different security domains. This is a natural requirement for Web services single sign-on, or distributed transactions. More information can be found at:

<http://www.oasis-open.org/specs/index.php#samlv1.1>

User experience

There is only one standard in this category, because it is a relatively new trend and might also lead to the creation of other standards and specifications.

WSRP: Web Services for Remote Portlets

Web Services for Remote Portlets (WSRP) is a Web services standard that allows for the *plug-and-play* of portals, other intermediary Web applications that aggregate content, and applications from disparate sources. WSRP Version 1.0 is an approved OASIS standard. More information can be found at:

http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrp

J2EE 1.4 and Java JSRs

The Java 2 Platform, Enterprise Edition Version 1.4 features complete Web services support through the new JAX-RPC 1.1 API, which supports service endpoints based on servlets and enterprise beans. JAX-RPC 1.1 provides interoperability with Web services based on the WSDL and SOAP protocols. The J2EE 1.4 platform also supports the Web Services for J2EE specification (JSR 921), which defines deployment requirements for Web services and uses the JAX-RPC programming model. In addition to numerous Web services APIs, the J2EE 1.4 platform also features support for the WS-I Basic Profile 1.0. This means that in addition to platform independence and complete Web services support, J2EE 1.4 offers platform Web services interoperability.

The J2EE 1.4 platform also introduces the J2EE Management 1.0 API, which defines the information model for J2EE management, including the standard Management EJB (MEJB). The J2EE Management 1.0 API uses the Java Management Extensions API (JMX). The J2EE 1.4 platform also introduces the J2EE Deployment 1.1 API, which provides a standard API for deployment of J2EE applications.

The J2EE platform now makes it easier to develop Web front ends with enhancements to the Java servlet and JavaServer Pages (JSP) technologies. Servlets now support request listeners and enhanced filters. JSP technology has simplified the page and extension development models with the introduction of a simple expression language, tag files, and a simpler tag extension API, among other features. This makes it easier than ever for developers to build JSP-enabled pages, especially those who are familiar with scripting languages.

Other enhancements to the J2EE platform include the J2EE Connector Architecture, which provides incoming resource adapter and Java Messaging Service (JMS) plugability. New features in Enterprise JavaBeans (EJB) technology include Web service endpoints, a timer service, and enhancements to EJB QL and message-driven beans. The J2EE 1.4 platform also includes enhancements to deployment descriptors. They are now defined using XML Schema, which can also be used by developers to validate their XML structures.

For more information, refer to:

<http://java.sun.com/j2ee/faq.html#new>

This section highlights current and proposed Java standards and APIs that are relevant to Web services.

JSR 101: Java APIs for XML-based RPC (JAX-RPC)

The Java API for XML-based RPC (JAX-RPC) enables Java technology developers to build Web applications and Web services incorporating XML-based RPC functionality according to the SOAP 1.1 specification. By using JAX-RPC, developers can rapidly achieve Web services interoperability based on widely adopted standards and protocols.

The latest specification, V1.1, is part of the Java Web Services Developer Pack. It entered final release stage October 28, 2003. See also “JSR 224: Java API for XML-based RPC (JAX-RPC) 2.0” on page 33. More information can be found at:

<http://java.sun.com/xml/downloads/jaxrpc.html>

JSR 109: Implementing Enterprise Web Services

This specification defines the Web services for J2EE architecture. This is a service architecture that leverages the J2EE component architecture to provide a client and server programming model that is portable and interoperable across application servers, provides a scalable secure environment, and yet is familiar to J2EE developers.

JSR 109 entered final release stage November 15, 2002. For more information, refer to:

<http://jcp.org/aboutJava/communityprocess/final/jsr109/index.html>

JSR 921 Implementing Enterprise Web Services 1.1 is the follow-on specification. Refer to “JSR 921: Implementing Enterprise Web Services 1.1” on page 34.

JSR 31: Java Architecture for XML Data Binding (JAXB)

The Java Architecture for XML Data Binding (JAXB) gives Java developers an efficient and standard way of mapping between XML and Java code. Java developers using JAXB are more productive because they can write less code themselves and do not have to be experts in XML. JAXB makes it easier for developers to extend their applications with XML and Web services technologies.

JSR 31 is part of the Java Web Services Developer Pack. JSR 31 entered final release stage March 4, 2003. See also “JSR 222: Java Architecture for XML Binding (JAXB) 2.0” on page 33. More information can be found at:

<http://java.sun.com/xml/downloads/jaxb.html>

JSR 67: Java APIs for XML Messaging 1.0 (JAXM)

The Java API for XML Messaging (JAXM) enables applications to send and receive document-oriented XML messages. JAXM implements Simple Object Access Protocol (SOAP) 1.1 with attachments messaging so that you can focus on building, sending, receiving, and decomposing messages for your applications instead of programming low-level XML communications routines.

JSR 67 is an optional component of J2SE or J2EE. JSR 67 entered final release stage December 20, 2001 and was final October 21, 2003. Additional information can be found at:

<http://java.sun.com/xml/jaxm/index.jsp>

JSR 93: Java API for XML Registries 1.0 (JAXR)

The Java API for XML Registries (JAXR) API provides a uniform and standard Java API for accessing different kinds of XML registries. An XML registry is an enabling infrastructure for building, deploying, and discovering Web services. This version of the JAXR specification includes detailed bindings between the JAXR information model and both the ebXML Registry and the UDDI Registry 2.0 specifications.

JSR 93 is part of the Java Web Services Developer Pack. JSR 93 entered final release stage June 11, 2002. More information can be found at:

<http://java.sun.com/xml/downloads/jaxr.html>

JSR 110: Java APIs for WSDL (WSDL4J)

The Web Services Description Language for Java Toolkit (WSDL4J) enables the creation, representation, and manipulation of WSDL documents describing services.

JSR 110 is available as a download, but not included in any official J2SE or J2EE product. JSR 110 entered final release stage March 25, 2003. For more information, refer to:

<http://sourceforge.net/projects/wsd14j>

JSR 172: J2ME Web Services

JSR 172 defines an optional package that provides standard access from J2ME to Web services. JSR 172 entered final release stage March 3, 2004. More information can be found at:

<http://www.jcp.org/en/jsr/detail?id=172>

JSR 173: Streaming API for XML

The Streaming API for XML (StAX) is a Java-based API for pull-parsing XML. This API has been developed primarily to improve the performance of SOAP stack implementations.

JSR 173 entered final release stage March 25, 2004. For more information, refer to:

<http://www.jcp.org/en/jsr/detail?id=173>

JSR 181: Web Services Metadata for the Java Platform

This JSR defines an annotated Java format that uses Java Language Metadata (JSR 175) to enable easy definition of Java Web services in a J2EE container.

JSR 181 entered the public review stage September 20, 2004 and is awaiting public review ballot. For more information, refer to:

<http://www.jcp.org/en/jsr/detail?id=181>

JSR 208: Java Business Integration (JBI)

This JSR extends J2EE with business integration SPIs. These SPIs enable the creation of a Java business integration environment for specifications such as WSCI, BPEL4WS, and the W3C Choreography Working Group.

JSR 208 entered early draft review stage October 8, 2004. It should be noted that a number of industry heavy-weights, including IBM and BEA, do not support this JSR specification and are concentrating on support for the broader industry-wide BPEL specification. More information can be found at:

<http://www.jcp.org/en/jsr/detail?id=208>

JSR 222: Java Architecture for XML Binding (JAXB) 2.0

JAXB 2.0 is the next version of JAXB, The Java Architecture for XML Binding. This JSR proposes additional functionality while retaining ease of development as a key goal.

JSR 222 entered early draft review stage June 23, 2004. Additional information is available at:

<http://www.jcp.org/en/jsr/detail?id=222>

JSR 224: Java API for XML-based RPC (JAX-RPC) 2.0

The JAX-RPC 2.0 specification extends the existing JAX-RPC 1.0 specification with new features.

JSR 224 entered early draft review stage June 23, 2004. More information is available at:

<http://www.jcp.org/en/jsr/detail?id=224>

JSR 921: Implementing Enterprise Web Services 1.1

JSR 921 is the follow-on specification to JSR 109 and includes minor updates. See “JSR 109: Implementing Enterprise Web Services” on page 31. More information is available at:

<http://www.jcp.org/aboutJava/communityprocess/maintenance/jsr921/>

Web services organizations and groups

Web services are industry-wide open standards that do not belong to single company or organization. Instead, there are several organizations that define Web service standards and oversee their evolution.

In the remainder of this section, we briefly introduce the most important participants in the world of Web services. The short quotations at the beginning of the descriptions is quoted from the organizations’ Web pages.

Internet Engineering Task Force

“The Internet Engineering Task Force (IETF) is a large open international community of network designers, operators, vendors, and researchers concerned with the evolution of the Internet architecture and the smooth operation of the Internet.”

The IETF is organized into several groups who are publishing their working documents as *Internet Drafts* which can become *Requests for Comment* after some time. For more information, refer to:

<http://www.ietf.org>

Java Community Process

“Since its introduction in 1998 as the open, participative process to develop and revise the Java technology specifications, reference implementations, and test suites, the Java Community Process (JCP) program has fostered the evolution of the Java platform in cooperation with the international Java developer community.”

The JCP is publishing Java Specification Requests (JSR) that either propose a new specification or a substantial upgrade of an existing specification. There are several, Web service-related JSRs issued by the JCP. More information can be found at:

<http://www.jcp.org>

OASIS

“The Organization for the Advancement of Structured Information (OASIS) is a not-for-profit, international consortium that drives the development, convergence, and adoption of e-business standards.”

The OASIS Web site can be found at:

<http://www.oasis-open.org>

World Wide Web Consortium

“The World Wide Web Consortium (W3C) develops interoperable technologies (specifications, guidelines, software, and tools) to lead the Web to its full potential. W3C is a forum for information, commerce, communication, and collective understanding.”

Among the many *drafts* and *recommendations* the W3C has published, the most relevant are with respect to XML, SOAP, and WSDL. For more information, refer to:

<http://www.w3c.org>

Web Services Interoperability Organization

“The Web Services Interoperability Organization (WS-I) is an open industry effort chartered to promote Web Services interoperability across platforms, applications, and programming languages. The organization brings together a diverse community of Web services leaders to respond to customer needs by providing guidance, recommended practices, and supporting resources for developing interoperable Web services.”

WS-I defines *profiles* that consist of specific Web service-related standards together with recommendations and guidelines regarding implementation and interoperability issues. More information about WS-I and the WS-I profiles can be found at:

<http://www.ws-i.org>

UDDI

Originally a separate consortium, the UDDI.org has since been subsumed by the OASIS organization (see “OASIS” on page 35). The UDDI home page is available at:

<http://www.uddi.org>

Companies working on Web services

In addition, there are many corporations that also support Web services and are defining new specifications in areas that are not covered by the available standards yet. Among these companies, IBM and Microsoft have to be pointed out because both are major supporters of Web services. In order to understand these companies' commitment, we quote some statements they made about the importance of Web services.

IBM

International Business Machines Corp., <http://www.ibm.com>

Samuel J. Palmisano³: *On demand integration is also why we have placed a huge bet on standards, from the Internet protocols and Linux® to grid computing and Web services. Without open technical interfaces and agreed-upon standards, even integration within a single enterprise would remain a gargantuan task.*

The IBM Web site dedicated to Web services is available at:

<http://www.ibm.com/developerworks/webservices>

This site contains sections for users new to SOA and Web services, downloads and products, open source projects, demonstrations, technical library, training, forums, special offers, and events.

Microsoft

Microsoft Corp., <http://www.microsoft.com>

The promise of interoperability across systems, applications, and programming languages isn't just possible, it's becoming pervasive with the tremendous potential of XML-based systems to unify the computing landscape. [...]

³ Samuel J. Palmisano, Chairman, President, and Chief Executive Officer, quoted from his 2003 Annual Report. The full document can be found at:

http://ftp.software.ibm.com/annualreport/2003/2003_ibm_ar_letter.pdf

Microsoft intends to work with key industry partners and standards bodies on these [specifications for security, routing, reliable messaging, and transactions] and other specifications important to Web services.⁴

The Microsoft Web site dedicated to Web services is available at:

<http://msdn.microsoft.com/webservices/>

Vertical industry standards organizations

Besides the cross-industry standards that already exist and those that are currently being defined, other standards exist that are only relevant to specific industries.⁵

You can find a nice overview of some of the vertical organization standards at:

http://developers.sun.com/techtopics/webservices/standards.html#xml_vert_orgs

Another source of information about vertical industry standards is available at:

<http://www.webmethods.com/meta/default/folder/0000006963>

Because an attempt to list all standards is futile at best, we can only give pointers to some of the information that can be found on the Internet. You are encouraged to search the Internet for more information, in particular if you are looking for standards for a specific industry only.

⁴ Quoted from Microsoft's Commitment to Standards and the Standards Process, available at:
<http://www.microsoft.com/net/business/standards.asp>

⁵ Because these standards are related to specific industries and not across all industries, they are called vertical.

Summary

In this chapter, we presented the standards and specifications that define the *Web services universe*. In order to better find your way around the many existing standards and specifications, we grouped the standards in categories. We briefly described each standard and also included information about the JSRs that are related to Web services and J2EE.

To give a better overview, we also described the organizations and companies that define Web service standards and specifications and thus drive the Web services to new domains.

More information

An online list of current and emerging Web services standards can be found at the IBM developerWorks Web site by navigating to *developerWorks* → *SOA and Web services* → *Technical library* → *Standards*, available at:

<http://www.ibm.com/developerworks/views/webservices/standards.jsp>

Detailed information about individual Java Specification Requests and their current status are available at:

<http://www.jcp.org/>

An article on the, sometimes difficult, standardization of Web services is *Java integration spec shunned by IBM and BEA*, available at:

http://searchwebservices.techtarget.com/originalContent/0,289142,sid26_gci1020556,00.html?track=NL-201&ad=495795

Introduction to SOAP

In this chapter, we introduce SOAP, the specification covering the exchange of XML-based messages between the three main actors in the service-oriented architecture (SOA).

We cover the SOAP 1.1 specification in detail and present a couple of examples of SOAP messages. We cover the details of the SOAP architecture. We explore SOAP implementations such as Apache SOAP 2.3 and its successor, Apache Extensible Interaction System (Axis).

We also briefly touch upon other SOAP implementations, such as the Microsoft SOAP Toolkit. Finally, we outline recent developments and future trends in this field in the form of SOAP 1.2.

Overview

Simple Object Access Protocol (SOAP) is a specification for the exchange of structured information in a decentralized, distributed environment. As such, it represents the main way of communication between the three main actors in SOA: The service provider, service requestor, and service broker. The main goal of its design is to be simple and extensible. Originally proposed by Microsoft, it is now submitted to W3C as the basis of the XML Protocol Working Group by several companies, including IBM. At the time of writing of this book, the current standard is 1.2. You can get more details at:

<http://www.w3.org/TR/SOAP>

Because SOAP 1.2 is not included in WS-I Basic Profile 1.0 (WS-I BP 1.0) and J2EE 1.4 Web services adheres to this specification, only SOAP 1.1 is supported by J2EE 1.4 Web services.

SOAP is an XML-based protocol that consists of three parts: An *envelope* that defines a framework for describing message content and process instructions, a set of *encoding rules* for expressing instances of application-defined data types, and a *convention* for representing remote procedure calls and responses.

SOAP is, in principle, transport protocol-independent and can, therefore, potentially be used in combination with a variety of protocols such as HTTP, JMS, SMTP, or FTP. Right now, the most common way of exchanging SOAP messages is through HTTP, which is also the only protocol supported by WS-I Basic Profile 1.0.

The way SOAP applications communicate when exchanging messages is often referred to as the message exchange pattern (MEP). The communication can be either one-way messaging, where the SOAP message only goes in one direction, or two-way messaging, where the receiver is expected to send back a reply. Message exchange patterns are explored in detail in Chapter 11, “Web services architectures” on page 209.

In this book, we describe how to use SOAP in combination with HTTP, HTTP extension framework, and JMS. SOAP is also operating system independent and not tied to any programming language or component technology.

Because SOAP deals with objects serialized to plain text and not with stringified remote object references (interoperable object references, IORs, as defined in CORBA), distributed garbage collection has no meaning and is not covered. For the same reason, SOAP clients do not hold any stateful references to remote objects, and without this, activation in a Java RMI way is also impossible.

Due to these characteristics, it does not matter what technology is used to implement the client, as long as the client can issue XML messages. Similarly, the service can be implemented in any language, as long as it can process XML messages. Also, both the server and client sides can reside on any suitable platform.

In this chapter we discuss the W3C SOAP 1.1 specification and the SOAP 2.3 implementation from Apache. We also discuss Apache Axis, the SOAP 2.3 successor, and the IBM WebSphere Web services engine. There are other Java implementations as well as non-Java implementations, such as Microsoft SOAP Toolkit, which we only briefly touch upon.

The three pillars of SOAP

This section discusses the key aspects of XML-based message exchange.

Overall message format: Envelope with header and body

A SOAP message is an *envelope* containing zero or more *headers* and exactly one *body*:

- ▶ The envelope is the top element of the XML document, providing a container for control information, the addressee of a message, and the message itself.
- ▶ Headers contain control information, such as quality of service attributes.
- ▶ The body contains the message identification and its parameters.

Both the headers and the body are child elements of the envelope.

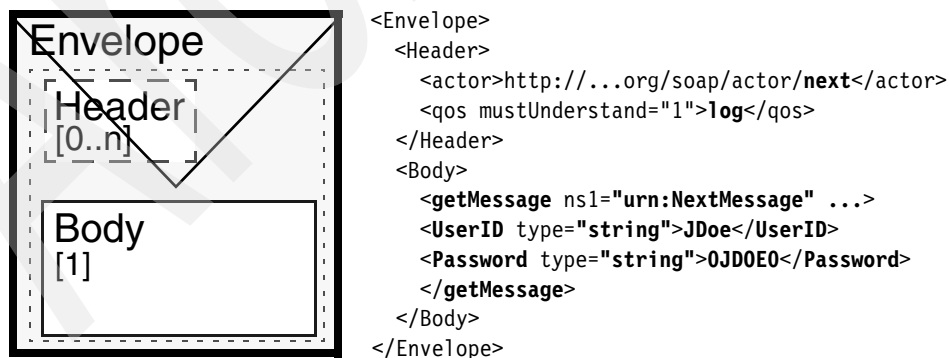


Figure 3-1 Example of a conceptualized SOAP message

Figure 3-1 shows a conceptualized SOAP request message based on a scenario of a personal text message recorder, similar to a recording phone answering machine, where the user can listen to the recorded messages:

- ▶ The header tells *who* and *how* to deal with the message. The actor next (or omitted actor) is the default actor and declares the receiver as the server who has to do what the body says. Furthermore, the server must understand the application-defined quality of service log (and implement and execute it, as the name implies: Log the service request).
- ▶ The body tells *what* has to be done: Get and return the next message for Jon Doe, in detail, invoke the `getMessage` method on the service object instance `NextMessage` passing the two string arguments `UserID` and `Password` with the values `JDoe` and `OJD0E0`.

More information about the envelope, header, and body is covered in “SOAP elements” on page 44.

Encoding rules

Encoding rules (of course included in a real SOAP message) define a serialization mechanism that can be used to exchange instances of application-defined data types.

SOAP defines a programming language-independent data type schema based on the XML Schema Descriptor (XSD), plus encoding rules for all data types defined according to this model.

RPC representation

The remote procedure call (RPC) representation is a convention suited to represent remote procedure calls and the related response messages. As arguments in remote method invocation, we normally use relatively simple data structures, although, with conventions such as *XML Literal*, it is possible to transfer more complex data. This convention, however, is only covered by SOAP implementations and standards beyond SOAP, such as the JSR-101 Java APIs for XML-based RPC—or briefly, JAX-RPC—and is not a part of the SOAP standard. For more information about JAX-RPC, see Chapter 5, “JAX-RPC (JSR 101)” on page 91.

The use of this RPC representation in a plain SOAP context is optional. If the convention is not used, the communication style is purely *message-oriented*. When working with the message-oriented style, also called *document-oriented* communication, almost any XML document can be exchanged. Refer to “Communication styles” on page 52 for more information.

Figure 3-2 shows an example of a SOAP request message embedded in an HTTP request:

- ▶ The standard HTTP header contains the URL of the SOAP server, which in this case is `/www.messages.com/webapp/servlet/rpcrouter`. Relative to this URL, the Web service is identified by `urn:NextMessage`.
- ▶ After the header comes a SOAP envelope that contains the message to be transmitted. Here, the method invocation is the SOAP RPC representation of a call to the method `getMessage(UserID, Password)` of a Web service called `urn:Nextmessage` residing on the SOAP server.
- ▶ `http://schemas.xmlsoap.org/soap/encoding/` specifies the encoding that is used to convert the parameter values from the programming language on both the client side and the server side to XML and vice versa.

```
POST /webapp/servlet/rpcrouter HTTP/1.1
Host: www.messages.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: ""

<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <ns1:getMessage xmlns:ns1="urn:NextMessage"
      soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
      <UserID xsi:type="xsd:string">JDoe</UserID>
      <Password xsi:type="xsd:string">0JD0E0</Password>
    </ns1:getMessage>
  </soapenv:Body>
</soapenv:Envelope>
```

Figure 3-2 A SOAP message embedded in an HTTP request

Figure 3-3 shows a (possible) response to the above request:

- ▶ First comes the standard HTTP header.
- ▶ After the header comes a SOAP envelope that contains the message to be transmitted. In this message, the service returned the requested message.

```

HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<soapenv:Envelope
  xmlns:soapenv="..." xmlns:xsd="..." xmlns:xsi="...">
  <soapenv:Body>
    <ns1:getResponseMessage xmlns:ns1="urn:NextMessage"
      soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
      <return xsi:type="xsd:string">Call mom!</return>
    </ns1:getResponseMessage>
  </soapenv:Body>
</soapenv:Envelope>

```

Figure 3-3 A SOAP message embedded in an HTTP response

SOAP messages are fundamentally one-way transmissions from a sender to a receiver, but, as previously illustrated, SOAP messages are often combined to implement patterns such as request/response.

SOAP elements

This section discusses the major elements of a SOAP message.

Namespaces

The use of namespaces plays an important role in SOAP message, because a message can include several different XML elements that must be identified by a unique namespace to avoid name collision.

Especially the WS-I Basic Profile 1.0 requires that all application-specific elements in the body must be namespace qualified to avoid name collision.

Table 3-1 shows the namespaces of SOAP and WS-I Basic Profile 1.0 used in this book.

Table 3-1 SOAP namespaces

Prefix	Namespace URI	Explanation
SOAP-ENV	http://schemas.xmlsoap.org/soap/envelope/	SOAP 1.1 envelope namespace
SOAP-ENC	http://schemas.xmlsoap.org/soap/encoding/	SOAP 1.1 encoding namespace

Prefix	Namespace URI	Explanation
	http://www.w3.org/2001/XMLSchema-instance	Schema instance namespace
	http://www.w3.org/2001/XMLSchema	XML Schema namespace
	http://schemas.xmlsoap.org/wsdl	WSDL namespace for WSDL framework
	http://schemas.xmlsoap.org/wsdl/soap	WSDL namespace for WSDL SOAP binding
	http://ws-i.org/schemas/conformanceClaim/	WS-I Basic Profile

URN

A *unified resource name* (URN) uniquely identifies the service to clients. It must be unique among all services deployed in a single SOAP server, which is identified by a certain network address. A URN is encoded as a *universal resource identifier* (URI). We commonly use the format `urn:UniqueServiceID`. `urn:NextMessage` is the URN of our message exchange Web service.

All other addressing information is transport dependent. When using HTTP as the transport, for example, the URL of the HTTP request points to the SOAP server instance on the destination host. For the message exchange service, the URL could be `http://www.messages.com/webapp/servlet/rpcrouter`.

This namespace URI identifying the method name in SOAP is very similar to the interface ID scoping a method name in distributed computing environments, such as DCOM or CORBA, or the name and the associated remote interface of an Enterprise JavaBean (EJB).

SOAP envelope

The envelope is the top element of the XML document representing the message with the following structure:

```
<SOAP-ENV:Envelope .... >
  <SOAP-ENV:Header name="nmtoken">
    <SOAP-ENV:HeaderEntry.... />
  </SOAP-ENV:Header>
  <SOAP-ENV:Body name="nmtoken">
    [message payload]
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

In general, a SOAP message is a (possibly empty) set of headers plus one body. The SOAP envelope also defines the namespace for structuring messages. The entire SOAP message (headers and body) is wrapped in this envelope.

Note that the message body uses a service-specific namespace, `urn:NextMessage`, in our examples (Figure 3-2 on page 43 and Figure 3-3 on page 44). This namespace is different from `SOAP-ENV`, the namespace used by the envelope, which is defined by the SOAP specification. Therefore, the application can use its own domain-specific vocabulary when creating message bodies.

Headers

Headers are a generic and flexible mechanism for extending a SOAP message in a decentralized and modular way without prior agreement between the parties involved. They allow control information to pass to the receiving SOAP server and also provide extensibility for message structures.

Headers are optional elements in the envelope. If present, the element *must* be the first immediate child element of a SOAP envelope element. All immediate child elements of the header element are called *header entries*.

There is a predefined header attribute called `SOAP-ENV:mustUnderstand`. The value of the `mustUnderstand` attribute is either 1 or 0. The absence of the SOAP `mustUnderstand` attribute is semantically equivalent to the value 0.

If it is present in a header element and set to 1, the service provider must implement the semantics defined by the element:

```
<thens:qos xmlns:thens="someURI" SOAP-ENV:mustUnderstand="1">
  3
</thens:qos>
```

In the example, the header element specifies that a service invocation must fail if the service provider does not support the quality of service (qos) 3 (whatever qos=3 stands for in the actual invocation and servicing context).

A SOAP intermediary is an application that is capable of both receiving and forwarding SOAP messages on their way to the final destination. In realistic situations, not all parts of a SOAP message may be intended for the ultimate destination of the SOAP message, but, instead, may be intended for one or more of the intermediaries on the message path. Therefore, a second predefined header attribute, `SOAP-ENV:actor`, is used to identify the recipient of the header information. The value of the SOAP actor attribute is the URI of the mediator, which is also the final destination of the particular header element (the mediator

does not forward the header). For detailed information about intermediaries, refer to “SOAP processing model” on page 221.

If the actor is omitted or set to the predefined default value, the header is for the actual recipient and the actual recipient is also the final destination of the message (body). The predefined value is:

`http://schemas.xmlsoap.org/soap/actor/next`

If a node on the message path does not recognize a `mustUnderstand` header and the node plays the role specified by the actor attribute, the node must generate a SOAP `MustUnderstand` fault. Whether the fault is sent back to the sender depends on the MEP in use. For request/response, the WS-I BP 1.0 requires the fault to be sent back to the sender. Also, according to WS-I BP 1.0, the receiver node must discontinue normal processing of the SOAP message after generating the fault message.

Headers can also carry authentication data, digital signatures, encryption information, and transactional settings.

Headers can also carry client-specific or project-specific controls and extensions to the protocol; the definition of headers is not just up to standard bodies.

Note: The header must not include service instructions (that would be used by the service implementation).

WS-I conformance header

With the WS-Interoperability Basic Profile, it is possible to specify an optional header to indicate which profile, or in the future profiles, the SOAP message complies with. For WS-I Basic Profile 1.0 conformance, the following header element can be added to the SOAP message:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/" >
  <soap:Header>
    <!-- other headers -->
    <wsi:Claim conformsTo="http://ws-i.org/profiles/basic/1.0"
      xmlns:wsi="http://ws-i.org/schemas/conformanceClaim/" />
    <!-- other headers -->
  </soap:Header>
  <soap:Body>
    <!-- body content -->
  </soap:Body>
</soap:Envelope>
```

The `ws:Claim` element is only informative and because of that is always optional. In addition, the WS-I BP specifications forbids the sender to make use of the `mustUnderstand` attribute on the claim block, making it impossible to enforce a receiver to process the claim block.

Body

The SOAP body element provides a mechanism for exchanging information intended for the ultimate recipient of the message. The body element is encoded as an immediate child element of the SOAP envelope element. If a header element is present, then the body element *must* immediately follow the header element. Otherwise it *must* be the first immediate child element of the envelope element.

All immediate child elements of the body element are called *body entries*, and each body entry is encoded as an independent element within the SOAP body element. In the most simple case, the body of a basic SOAP message consists of:

- ▶ A message name.
- ▶ A reference to a service instance. In Apache SOAP, a service instance is identified by its URN. This reference is encoded as the namespace attribute.
- ▶ One or more parameters carrying values and optional type references.

Typical uses of the body element include invoking RPC calls with appropriate parameters, returning results, and error reporting. *Fault elements* are used in communicating error situations. The messages can contain almost any XML construct except document type definitions (DTDs) and processing instructions.

Error handling

SOAP itself predefines one body element, which is the *fault element* used for reporting errors. If present, the fault element must appear as a body entry and must not appear more than once within a body element. The fields of the fault element are defined as follows:

- ▶ `faultcode` is a code that indicates the type of the fault. SOAP defines a small set of SOAP fault codes covering basic SOAP faults:
 - `soapenv:Client`, indicating incorrectly formatted messages
 - `soapenv:Server`, for delivery problems
 - `soapenv:VersionMismatch`, which can report any invalid namespaces for envelope element

- `soapenv:MustUnderstand`, for errors regarding the processing of header content
- ▶ `faultstring` is a human-readable description of the fault. It must be present in a fault element.
- ▶ `faultactor` is an optional field that indicates the URI of the source of the fault. It is similar to the SOAP actor attribute, but instead of indicating the destination of the header entry, it indicates the source of the fault. The value of the `faultactor` attribute is a URI identifying the source that caused the error. Applications that do not act as the ultimate destination of the SOAP message must include the `faultactor` element in a SOAP fault element.
- ▶ `detail` is an application-specific field that contains detailed information about the fault. It must not be used to carry information about errors belonging to header entries. Therefore, the absence of the `detail` element in the fault element indicates that the fault is not related to the processing of the body element (the actual message).

For example, a `soapenv:Server` fault message is returned if the service implementation throws a `SOAPException`. The exception text is transmitted in the `faultstring` field.

Although SOAP 1.1 permits the use of custom-defined `faultcodes`, WS-I Basic Profile only permits the use of the four codes defined in SOAP 1.1.

Advanced topics

In the following sections, we discuss more advanced SOAP concepts, such as the different communication styles, the SOAP data model, the available encodings, and the corresponding type mappings. Although these concepts are rarely a concern in simple SOAP architectures, you will very quickly find them useful after you try to implement a nontrivial Web service.

Data model

One of the promises of SOAP is interoperability between different programming languages. That is the purpose of the SOAP data model, which provides a language-independent abstraction for common programming language types. It consists of:

Simple XSD types Basic data types found in most programming languages such as `int`, `String`, and `Date`.

Compound types	There are two kinds of compound types, <i>structs</i> and <i>arrays</i> :
Structs	Named aggregated types. Each element has a unique name, its <i>accessor</i> . An accessor is an XML tag. Structs are conceptually similar to records in languages, such as Pascal, or methodless classes with public data members in object-based programming languages.
Arrays	Elements in an array are identified by position, not by name. This is the same concept found in languages such as C and Java. SOAP also supports partially transmitted and sparse arrays. Array values can be structs or other compound values. Also, nested arrays (which means arrays of arrays) are allowed.

All elements and identifiers comprising the SOAP data model are defined in the namespace SOAP-ENC. It is worth noting that the SOAP standard only defines the rules of how data types can be constructed; a project-specific XML Schema has to define the actual data types.

A SOAP request message, such as `getMessage` in Figure 3-2 on page 43, is modeled as a struct containing an accessor for each in and in/out parameter. Figure 3-4 shows this.

```
<ns1:getMessage xmlns:ns1="urn:NextMessage"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <UserID xsi:type="xsd:string">JDoe</UserID>
  <Password xsi:type="xsd:string">0JD0E0</Password>
</ns1:getMessage>
```

Figure 3-4 A SOAP request message

In the example in Figure 3-4, the accessors are `UserID` and `Password`. The accessor names correspond to the names of the parameters, and the message types to the programming language data types (`xsd:string` and `java.lang.String`). The parameters must appear in the same order as in the method signature. The prefixes `xsd` and `xsi` reference the namespaces <http://www.w3.org/2001/XMLSchema> and <http://www.w3.org/2001/XMLSchema-instance>, respectively.

In the example shown in Figure 3-5, the argument is an array whose values are structs.

```

<ns1:getSubscribers xmlns:ns1="urn:SubscriberList"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENC:Array SOAP-ENC:arrayType="xxx:Subscribers[2]">
    <Subscribers>
      <UserID xsi:type="xsd:string">JDoe</UserID>
      <Password xsi:type="xsd:string">0JD0E0</Password>
    </Subscribers>
    <Subscribers>
      <UserID xsi:type="xsd:string">MDoe</UserID>
      <Password xsi:type="xsd:string">0JMD0E0</Password>
    </Subscribers>
  </SOAP-ENC:Array>
</ns1:getSubscribers>

```

Figure 3-5 A SOAP request message with an array of structs

The SOAP data model makes self-describing messages possible. No external schema is needed to understand an XML element such as:

```
<UserID xsi:type="xsd:string">JDoe</UserID>
```

SOAP provides a preferred encoding for all data types defined according to this model. We cover this subject in the next section.

Note: The use of a data model and associated encoding is optional.

Mappings

A *mapping* defines an association between a qualified XML element name, a Java class name, and one of the encodings as previously introduced. Therefore, mappings are not implementation language-independent.

A mapping specifies how, under the given encoding, an incoming XML element with a fully qualified name is to be converted to a Java class, and vice versa. We refer to the two mapping directions as *XML to Java* and *Java to XML*, respectively.

Any SOAP runtime environment holds a table of such mapping entries, the `SOAPMappingRegistry`. Table 3-2 shows standard Java-related mappings.

Table 3-2 SOAP Java-related mappings

Java	SOAP	serializer/deserializer
String	xsd:string	<built-in>/StringDeserializer
boolean	xsd:boolean	<built-in>/BooleanDeserializer
Boolean	xsd:boolean	<built-in>/BooleanObjectDeserializer

Java	SOAP	serializer/deserializer
double	xsd:double	<built-in>/DoubleDeserializer
Double	xsd:double	<built-in>/DoubleObjectDeserializer
long	xsd:long	<built-in>/LongDeserializer
Long	xsd:long	<built-in>/LongObjectDeserializer
float	xsd:float	<built-in>/FloatDeserializer
Float	xsd:float	<built-in>/FloatObjectDeserializer
int	xsd:int	<built-in>/IntDeserializer
Integer	xsd:int	<built-in>/IntObjectDeserializer
short	xsd:short	<built-in>/ShortDeserializer
Short	xsd:short	<built-in>/ShortObjectDeserializer
byte	xsd:byte	<built-in>/ByteDeserializer
Byte	xsd:byte	<built-in>/ByteObjectDeserializer
BigDecimal	xsd:decimal	<built-in>/DecimalDeserializer
GregorianCalendar	xsd:timeInstant	CalendarSerializer
Date	xsd:date	DateSerializer
QName	xsd:QName	QNameSerializer

If a data type is supposed to be used under a certain encoding, exactly one mapping must exist for it in this registry. Most standard Java types and JavaBeans are supported by default. Non-standard (custom) data types require the introduction of *custom mappings* on the client side and on the server side.

Communication styles

SOAP supports two different communication styles:

Document Also known as *message-oriented* style: This style provides a lower layer of abstraction, and requires more programming work. The *in* parameter is any XML document; the *response* can be anything (or nothing). This is a very flexible communication style that provides the best interoperability.

RPC The remote procedure call is a synchronous invocation of operation returning a result, conceptually similar to other RPCs.

This style is exploited by many Web service tools and featured in many tutorials.

Messages, parameters, and invocation APIs look different for RPC and document styles. The decision about which style to use is made at design time; a different client API is used.

SOAP enables applications to encapsulate and exchange RPC calls using the extensibility and flexibility of XML. To make a method call, the following information is needed:

- ▶ The URI of the target object
- ▶ A method name
- ▶ An optional method signature
- ▶ The parameters of the method
- ▶ Optional header data

Using SOAP for RPC does not imply any specific SOAP protocol binding; using SOAP for RPC is not limited to the HTTP protocol binding. When using HTTP as the protocol binding, however, an RPC call maps naturally to an HTTP request and an RPC response maps to an HTTP response.

Figure 3-6 shows the interactions between the client and server and the server-side processing of a service invocation in RPC communication.

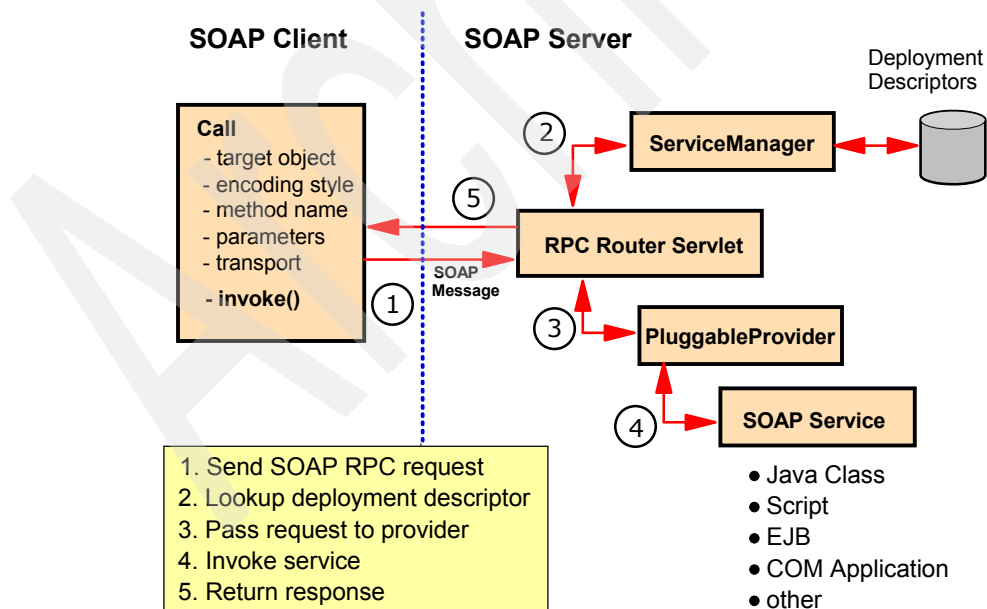


Figure 3-6 SOAP client and server interaction

Web service invocation using RPC involves the following steps (Figure 3-6):

1. A SOAP client generates a SOAP RPC request document and sends it to a RPC router.
2. The router contacts the service manager to obtain a deployment descriptor.
3. Based on routing information from the deployment descriptor, the router forwards the request to a service provider.
4. The service provider invokes the requested service and returns a result to the router.
5. The router sends the service result message to the client.

It is worth noting that this is a very simple scenario. More sophisticated scenarios would use additional steps, such as the ones related to security policy.

Encodings

In distributed computing environments, *encodings* define how data values defined in the application can be translated to and from a protocol format. We refer to these translation steps as *serialization* and *deserialization*, or, synonymously, *marshalling* and *unmarshalling* (even Apache SOAP uses both pairs of terms).

When implementing a Web service, we have to choose one of the tools and programming or scripting languages that support the Web services model, for example, Java. However, the protocol format for Web services is XML, which is independent of the programming language. Thus, SOAP encodings tell the SOAP runtime environment how to translate from data structures constructed in a specific programming language into SOAP XML and vice versa.

The following encodings are defined:

SOAP encoding The *SOAP encoding* enables marshalling/unmarshalling of values of data types from the SOAP data model. This encoding is defined in the SOAP 1.1 standard.

Literal The *literal* encoding is a simple XML message that does not carry encoding information. Usually, an XML Schema describes the format and data types of the XML message.

Literal XML The *literal XML* encoding enables direct conversion of existing XML DOM tree elements into SOAP message content and vice versa. This encoding style is not defined by the SOAP standard, but is in the Apache SOAP 2.3 implementation. This encoding is not used in newer SOAP engines.

XMI *XML metadata interchange (XMI)* is defined by the Apache SOAP implementation. We do not use this encoding in this document.

The encoding to be used by the SOAP runtime can be specified at deploy time or at runtime. If it is specified at deploy time, it appears in the WSDL specification of the Web service. Tools can then analyze this specification and configure the SOAP runtime to use the desired encoding.

At runtime, the SOAP client API allows the specification of the encoding for the entire message and for each of its parameters. On the server side, the encoding style is specified in the *deployment descriptor* of the Web service, an XML document that provides information to the SOAP runtime about the services that should be made available to clients. It contains information such as the URN for the service and method and class details (in case the service is implemented in Java) or the name of a script. The settings on the client side and the server side have to match.

Messaging modes

The two styles (RPC, document) and two most common encodings (encoded, literal) can be freely intermixed to what is called a SOAP messaging mode. Although SOAP supports four modes, only three of the four modes are generally used, and further, only two are preferred by the WS-I Basic Profile.

- ▶ **Document/literal**—Provides the best interoperability between Java and non-Java implementations, and is also recommended for Java-to-Java applications.
- ▶ **RPC/literal**—Possible choice between Java implementations. Although RPC/literal is WS-I compliant, it is not frequently used in practice. There are a number of usability issues associated with RPC/literal, including, but not limited to, the mapping of Java arrays to RPC/literal WSDL.
- ▶ **RPC/encoded**—Early Java implementations (WebSphere Application Server Versions 4 and 5.0) supported this combination, but it does not provide interoperability with non-Java implementations.
- ▶ **Document/encoded**—Not used in practice.

Because the WS-I Basic Profile prefers the use of literal, only document/literal and RPC/literal should be used for WS-I conformance. The SOAP encoding is not recommended by the WS-I BP mainly because of complexity and interoperability problems.

Implementations

So far, we have discussed only the SOAP specification. To build a Web services-based solution, the standard should be implemented in one of the programming languages. In the following sections, we discuss some of the most popular SOAP implementations:

- ▶ Apache SOAP 2.3
- ▶ Apache Axis
- ▶ WebSphere Web services engine

Important: We discuss the three implementations in the historical sequence. However, we recommend that you only use the WebSphere Web services engine for any applications running on WebSphere Application Server.

SOAP implementation general architecture

Figure 3-7 contains a high-level component model showing the conceptual architecture of both the service provider (SOAP server) and the service requestor (SOAP client).

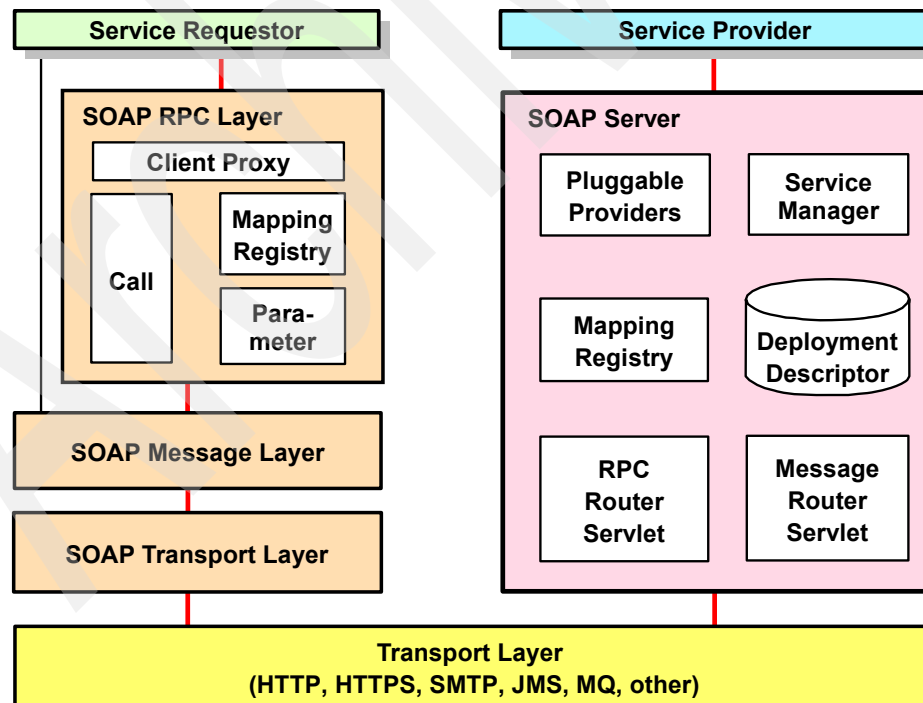


Figure 3-7 High-level SOAP component model

The client can invoke SOAP messages through the RPC layer (RPC style) or directly against the message layer (document style). Various transport protocols, such as HTTP or SMTP, connect the requestor and the provider.

On the provider side, RPC and message router servlets receive the incoming requests. Providers route them to the Java service implementation. The server is configured through deployment descriptor files.

Both on the requestor side and on the provider side, there are mapping registries providing access to serializer/deserializer classes implementing an encoding scheme.

Apache SOAP 2.3 implementation

The basis of the Apache SOAP implementation is IBM SOAP for Java (SOAP4J), the Java-based SOAP implementation that was submitted to the open source community.

SOAP server

Apache SOAP 2.3, included with other implementations in WebSphere Application Server Version 5 and WebSphere Studio Application Developer Version 5, provides an implementation of a SOAP server for deploying and invoking Web services.

Figure 3-8 gives an overview of the Apache SOAP server components.

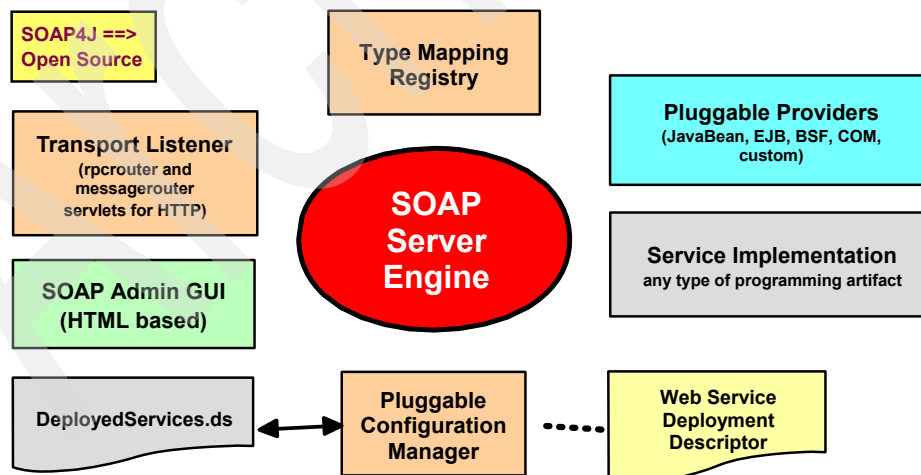


Figure 3-8 Components of Apache SOAP server implementation

For now, the important elements in this architecture are the `rpcrouter` and `messagingrouter` servlets, the deployment descriptor (explained later), and the type mapping registry. These components implement the SOAP concepts introduced so far.

The pluggable providers link a Web service and its implementation. The service implementation is your Java code executing the invoked Web service. We do not go into detail about the configuration manager and the administrative GUI here. Refer to the Apache SOAP user documentation for more information.

Server deployment

Deployment stands for configuring a Web service implementation so that it becomes available within a hosting SOAP server. The following steps have to be performed when a Web service is deployed to the SOAP server:

- ▶ Create a code artifact that is supported by one of the Apache SOAP providers.
- ▶ Ensure that parameters to the method/function are serializable by SOAP and exist within the SOAP type mapping registry. Otherwise, develop and register custom mappings.
- ▶ Create an entry in the Apache SOAP deployment descriptor for the service.

Figure 3-9 shows a Web service implementation Java class implementing the first step for the Exchange Web service.

```
public class NextMessage{
    public String getMessage(String UserID, String Password) {
        System.out.println("getMessage(" + UserID + ", " + Password + ")");
        return "Call mom!"; // fixed value for now
    }
}
```

Figure 3-9 Java class implementing the Web service

Figure 3-10 shows the corresponding deployment descriptor, which is read by the SOAP server at startup time.

```

<root>
  <isd:service xmlns:isd="http://xml.apache.org/xml-soap/deployment"
    id="urn:NextMessage" checkMustUnderstands="1">
    <isd:provider type="java" scope="Request" methods="getMessage">
      <isd:java class="NextMessage" static="false"/>
    </isd:provider>
  </isd:service>
</root>

```

Figure 3-10 Web service deployment descriptor (SOAP 2.3)

This deployment descriptor defines the URN, `urn:NextMessage`, and the name of the Java implementation class, `NextMessage`. There is one accessible method, `getMessage`.

The Web service scope is `Request`. This scope attribute can be set to `Application`, `Request`, or `Session`:

- ▶ If the scope is `Application`, a singleton instance of the service is created at server startup time (like a servlet).
- ▶ A service with the `Request` scope is instantiated whenever a message for it is received.
- ▶ If the scope is `Session`, the lifetime of the service instance is bound to the duration of the underlying transport session (for example, the `HttpSession` in the case that HTTP is the transport protocol).

The actual deployment step can either be performed using the administrative GUI that comes with Apache SOAP or programmatically.

SOAP client API

There are two key abstractions in the SOAP client API, which is defined in the `org.apache.soap` package and its sub-packages:

<code>Call</code>	Contains the URN, the SOAP address of the router servlet on the SOAP servers, and the name of the method to be called. A call object contains <code>Parameter</code> instances as data members.
<code>Parameter</code>	Contains the parameter value, type, and encoding style.

As a SOAP developer, you might have to use the following classes as well:

QName	Qualified name: Combination of an XML namespace and a local name. QName instances are used to identify XML data types and other elements in an XML document.
SOAPMappingRegistry	Maps types and encoding styles to the available serializer and deserializer classes.
SOAPHttpTransport	Provides access to the HTTP transport layer. For example, this class can be used to modify proxy and authentication settings.

Let us take a look at an example of a SOAP 2.3 client (Figure 3-11).

```
public class MySoapClient {
    public static void main(String[] args){
        Call call = new Call();
        call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
        call.setTargetObjectURI ("urn:NextMessage");
        call.setMethodName ("getMessage");
        Vector params = new Vector();
        Parameter userIDParam = new Parameter(
            "UserID", String.class, "JDoe", Constants.NS_URI_SOAP_ENC);
        params.addElement(userIDParam);
        Parameter passwordParam = new Parameter(
            "Password", String.class, "OJDOEO", Constants.NS_URI_SOAP_ENC);
        params.addElement(passwordParam);

        call.setParams(params);
        Response resp = null;
        URL url = new URL ("http://www.messages.com/soap/servlet/rpcrouter");
        resp = call.invoke (url, "urn:NextMessage"); // url, soapActionURI
        // soapActionURI is URN for Apache, "" for most other servers
        if (resp.generatedFault()) {
            Fault fault=resp.getFault();
            System.out.println(" Fault code: " + fault.getFaultCode());
            System.out.println(" Fault string: " + fault.getFaultString());
        } else {
            Parameter result=resp.getReturnValue();
            Object o = result.getValue();
            System.out.println("Result: " + o);
        }
    }
}
```

Figure 3-11 SOAP 2.3 client

The message that travels if this code is executed is the same message we inspected in Figure 3-2 on page 43.

You have to perform the following steps when developing a SOAP client:

- ▶ Obtain the interface description of the SOAP service so that you know what the signatures of the methods that you want to invoke are. Either contact the service provider directly, or use UDDI to do so (note that this step is not shown in the example).
- ▶ Make sure that there are serializers registered for all parameters that you will be sending and deserializers for all information that you will be receiving (this holds true for the example). Otherwise, develop and register the custom mapping.
- ▶ Create and initialize the `org.apache.soap.rpc.Call` object.
 - Set the target URI in the `Call` object using the `setTargetObjectURI` method.
 - Set the method name that you want to invoke into the `Call` object using the `setMethodName` method.
 - Create any `Parameter` objects necessary for the RPC call, and add them to the `Call` object using the `setParams` method.
- ▶ Execute the `Call` object's `invoke` method and capture the `Response` object that is returned from `invoke`.
- ▶ Check the `Response` object to see if a fault was generated using the `generatedFault` method.
- ▶ If a fault was returned, retrieve it using the `getFault` method. Otherwise, extract any result or returned parameters using the `getReturnValue` and `getParams` methods, respectively.

The SOAP client API is a string-oriented, weakly typed interface. This is due to the fact that it is a fixed API that is unaware of the signatures of the messages that are exchanged over it.

Usually, programmers do not have to work with this rather cumbersome API directly because there are tools wrapping it. For example, code generated from WSDL-aware tools can provide a more type-oriented, easier-to-code interface.

Apache SOAP implementation also represents the basis of another open-source project, Axis, which we cover in the next section.

Apache Axis

The Apache Extensible Interaction System (Axis) is basically a SOAP engine. It represents a framework for constructing SOAP processors such as clients, servers, or gateways. Axis is an Apache open-source project and is written in Java. Axis Version 1.1 is available, and Version 1.2 is in alpha.

Besides being a SOAP engine, Axis also includes the following features:

- ▶ A server that plugs into servlet engines such as WebSphere Application Server or Apache Tomcat
- ▶ Extensive support for the Web Services Description Language (WSDL)
- ▶ Tools that generate Java classes from WSDL and vice versa (WSDL2Java and Java2WSDL)
- ▶ A tool for monitoring TCP/IP packets

The TCP/IP Monitor is a very efficient tool for Web services debugging and troubleshooting.

Axis represents the third generation of Apache SOAP, which began at IBM as SOAP4J, and is often referred to as Apache SOAP 3.x. However, it does not share the code of the Apache SOAP 2.x project, but has been redesigned from scratch. It is based on the idea of configurable chains of message handlers, which would implement small bits of functionality in a flexible manner.

Axis uses the event-based simple API for XML (SAX) instead of DOM parsing to perform significantly better than earlier versions of Apache SOAP. The extendable Axis architecture enables the developer to insert extensions into the engine.

The core of the Axis SOAP engine is completely transport-independent. Therefore, it enables SOAP message exchange using different communication channels such as SMTP, FTP, or message-oriented middleware.

Axis supports the JAX-RPC API with JavaBeans as Web service implementations. Axis does not support enterprise Web services, for example, session EJBs as Web service implementations.

Axis server architecture

Figure 3-12 shows the basic architecture of Axis in the server.

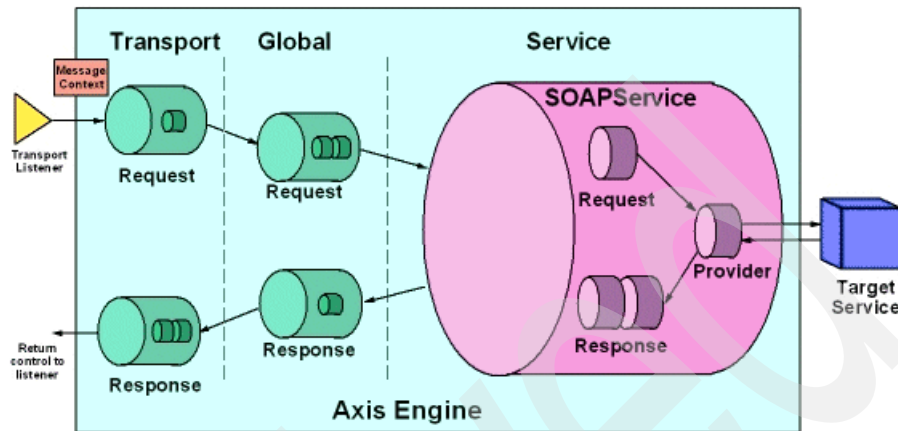


Figure 3-12 Axis architecture

The large cylinders represent chains, and the small cylinders represent handlers within a chain. The main flow elements are:

- ▶ The *transport listener* puts the protocol-specific data into a Message object and the message data into a MessageContext object.
- ▶ The Axis engine looks up the transport, which might contain a *request chain*, a *response chain*, or both. Each chain is a sequence of *handlers* that are invoked in turn.
- ▶ After the transport chain, the *global* chain of handlers is invoked.
- ▶ One of the handlers sets the `serviceHandler` field in the MessageContext, and the Axis engine invokes that *service* (an instance of SOAPService).
- ▶ The service invokes its request chain (if present) and finally the *provider*, which is the handler responsible for invoking the back-end logic.
- ▶ The response is processed by the respective response handlers in the service, global, and transport chains.

Axis client architecture

The client architecture is basically a mirror image of the server architecture:

- ▶ The client application sends a message.
- ▶ The Axis engine invokes the *service* chain, then the *global* chain, and then the *transport* chain, where the final handler, the *sender*, sends the message to the target.

Axis subsystems

Axis consists of several subsystems working together with the aim of separating responsibilities cleanly and making Axis modular. Subsystems that are properly layered enable parts of a system to be used without having to use all the parts.

Figure 3-13 shows the layering of subsystems. The lower layers are independent of the higher layers. The stacked boxes represent mutually independent, although not necessary exclusive, alternatives. For example, the HTTP, SMTP, and JMS transports are independent of each other but can be used together.

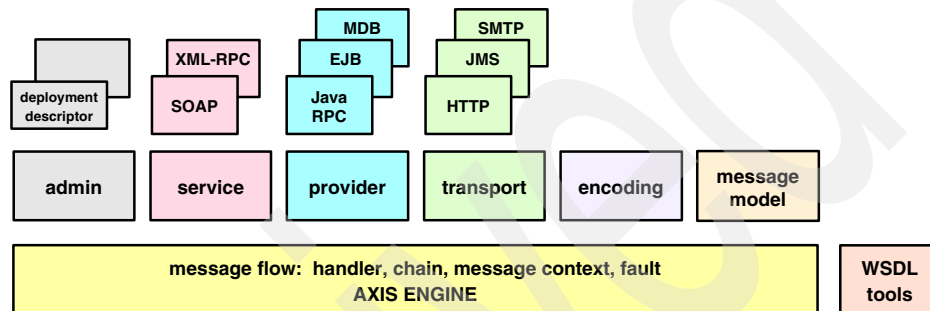


Figure 3-13 Axis subsystems

Implementations

Axis 1.0 is implemented in WebSphere Application Server Versions 5.0.5, 5.1, and 6.0 as alternative SOAP engine. The preferred engine is, however, the WebSphere Web services engine.

WebSphere Web services engine

With Versions 5.0.2, 5.1, and 6.0 of WebSphere Application Server comes a SOAP engine written by IBM. This engine, commonly referred to as *WebSphere Web services engine*, is based on Axis principles but extended for performance and for enterprise Web services (support for session EJB as Web services and for SOAP over JMS).

We implement our sample application using WebSphere Application Server Version 6.0 (Application Server for short) and Rational Application Developer Version 6.0 (Application Developer for short), all of which include the WebSphere Web services engine.

Tip: We recommend that you always use the WebSphere Web services engine for all applications running on WebSphere Application Server.

Microsoft SOAP Toolkit

Microsoft has its own SOAP implementation in the Microsoft SOAP Toolkit. At the time of writing of this book, the current version was Version 3.0. SOAP is also part of .NET, the Microsoft strategic platform for Web services.

Other toolkits and server implementations

Currently, Web service engines and development tools often appear in the market, for all kinds of platforms written in all kinds of languages for all kinds of devices to get simple connectivity. In addition, many SOAP and SOAP-RPC user communities, in which you can join and participate, are on the Web. Use, for example, this link as a starting point for your own research:

<http://www.google.com/search?q=soap+web+service+server>

Outlook

At the time of writing this book, the SOAP 1.2 specification was in its final phase. It introduces several changes to SOAP 1.1. It is beyond the scope of this book to go into the details of the differences between SOAP 1.1 and SOAP 1.2 specifications. Recent information about this subject is available at:

<http://www.w3.org/TR/soap12-part0/#L4697>

In October 2002, Apache Axis 1.0 was officially released; in June 2003, Axis 1.1 became available. Among other functions, Axis implements most of the SOAP 1.2 specification.

Summary

SOAP represents the information exchange mechanism between the three main actors in the Web service model: A Web service provider, a Web service requestor, and a Web service broker. It is based on XML documents and is designed to be simple and extensible. It is also independent of actual Web services implementation and therefore enables interoperability between Web services implementations on different platforms. SOAP is defined by the W3C SOAP specification. Its current version is Version 1.1, with Version 1.2 in preparation.

Currently, there are several SOAP implementations available:

- ▶ The Apache SOAP 2.3 implementation is an open-source Java-based implementation based on the IBM SOAP4J implementation and is a part of several commercial packages, including WebSphere Application Server Version 5 and WebSphere Studio Application Developer Version 5.
- ▶ The Apache Axis implementation is a follow-up project of the Apache SOAP V2.X project and is often referred to as the Apache SOAP 3.0 implementation.
- ▶ IBM WebSphere Application Server Versions 5.0.2, 5.1, and 6.0 have their own WebSphere Web services engine.
- ▶ Other companies (for example, Microsoft) have other SOAP implementations based on different programming and scripting languages.

As a communication protocol, SOAP enables the publication and invocation of Web services and represents the basic *plumbing* of a Web services infrastructure. However, this is not enough for the successful implementation of Web services:

- ▶ A client has to obtain the information about the interface of a service, the server URL, the URN, the methods, the types, and the type mappings. This is provided by WSDL.
- ▶ A client has to learn about the existence of a service and its characteristics, which is the function of UDDI.
- ▶ A Web Services Inspection Language (WSIL) document provides location information to invoke Web services.
- ▶ We introduce these three technologies in the chapters that follow.

More information

The SOAP specification can be downloaded from:

<http://www.w3.org/TR/SOAP>
<http://www.w3.org/TR/soap12-part0/>
<http://www.w3.org/TR/soap12-part1/>

For information about Apache SOAP, refer to the user and API documentation and the FAQ list, available at:

<http://www.apache.org/soap>

For information about Apache Axis, refer to the user and API documentation and the FAQ list, available at:

<http://xml.apache.org/axis/index.html>

The IBM developerWorks Web site provides many articles about SOAP (enter SOAP into the search engine):

<http://www.ibm.com/developerworks>

For example, here are two articles about SOAP type mapping:

<http://www.ibm.com/developerworks/library/ws-soapmap1/>
<http://www.ibm.com/developerworks/library/ws-soapmap2/>

Introduction to WSDL

This chapter provides an introductory view to Web Services Description Language (WSDL) 1.1. WSDL specifies the characteristics of a Web service using an XML format, describing what a Web service can do, where it resides, and how it is invoked. WSDL is extensible to allow descriptions of different bindings, regardless of what message formats or network protocols are used to communicate.

WSDL 1.1 provides a notation serving these purposes. The WSDL specification is a joint effort by Ariba, IBM, and Microsoft. It is not yet an official standard; at the time of the writing of this book, its status was “submission acknowledged” by the W3C.

The Web Services Description Language Specification 1.2 is a working draft at this time. Therefore, we do not make any specific reference to WSDL 1.2, but we include some information in “Outlook” on page 88. However, some of the current implementations already implement selected features of the 1.2 draft specification.

Overview

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

- ▶ Name of the Web service and addressing information
- ▶ Protocol and encoding style to be used when accessing the public operations of the Web service
- ▶ Type information: Operations, parameters, and data types comprising the interface of the Web service, plus a name for this interface

A WSDL specification uses XML syntax; therefore, there is an XML Schema for it. A valid WSDL document consists of one or more files. If there is more than one file, the use of the import element is required. This import element creates the needed references to locate the different files of the WSDL document. We recommend this split to maintain a clearer service definition based on the level of abstraction of the parts.

WSDL document

The WSDL document contains the following main elements:

Types	A container for data type definitions using some type system, such as XML Schema.
Message	An abstract, typed definition of the data being communicated. A message can have one or more typed parts.
Port type	An abstract set of one or more operations supported by one or more ports.
Operation	An abstract description of an action supported by the service that defines the input and output message and optional fault message.
Binding	A concrete protocol and data format specification for a particular port type. The binding information contains the protocol name, the invocation style, a service ID, and the encoding for each operation.
Service	A collection of related ports.
Port	A single endpoint, which is defined as an aggregation of a binding and a network address.

Note that WSDL does not introduce a new type definition language. WSDL recognizes the need for rich type systems for describing message formats and supports the XML Schema Definition (XSD) specification.

WSDL 1.1 introduces specific binding extensions for various protocols and message formats. There is a WSDL SOAP binding, which is capable of describing SOAP over HTTP, a direct HTTP interface (any plain servlet, for example), and a MIME binding. The language is extensible and allows the definition of other binding mechanisms, such as Java and SOAP over Java Messaging Service (JMS).

It is worth noting that WSDL does not define any mapping-to-programming languages such as Java; rather, the bindings deal with transport protocols. This is a major difference from interface description languages, such as CORBA interface definition language (IDL), which has language bindings.

WSDL document anatomy

Figure 4-1 shows the elements comprising a WSDL document and the various relationships between them.

The diagram should be read in the following way:

- ▶ One WSDL document contains zero or more services. A service contains zero or more port definitions (service endpoints), and a port definition contains a specific protocol extension.
- ▶ The same WSDL document contains zero or more bindings. A binding is referenced by zero or more ports. The binding contains one protocol extension, where the style and transport are defined, and zero or more operations bindings. Each of these operation bindings is composed of one protocol extension, where the action and style are defined, and one to three messages bindings, where the encoding is defined.
- ▶ The same WSDL document contains zero or more port types. A port type is referenced by zero or more bindings. This port type contains zero or more operations, which are referenced by zero or more operations bindings.
- ▶ The same WSDL document contains zero or more messages. An operation usually points to an input and an output message, and optionally to some faults. A message is composed of zero or more parts.
- ▶ The same WSDL document contains zero or more types. A type can be referenced by zero or more parts.
- ▶ The same WSDL document points to zero or more XML Schemas. An XML Schema contains zero or more XSD types that define the different data types.

The containment relationships shown in the diagram directly map to the XML Schema for WSDL.

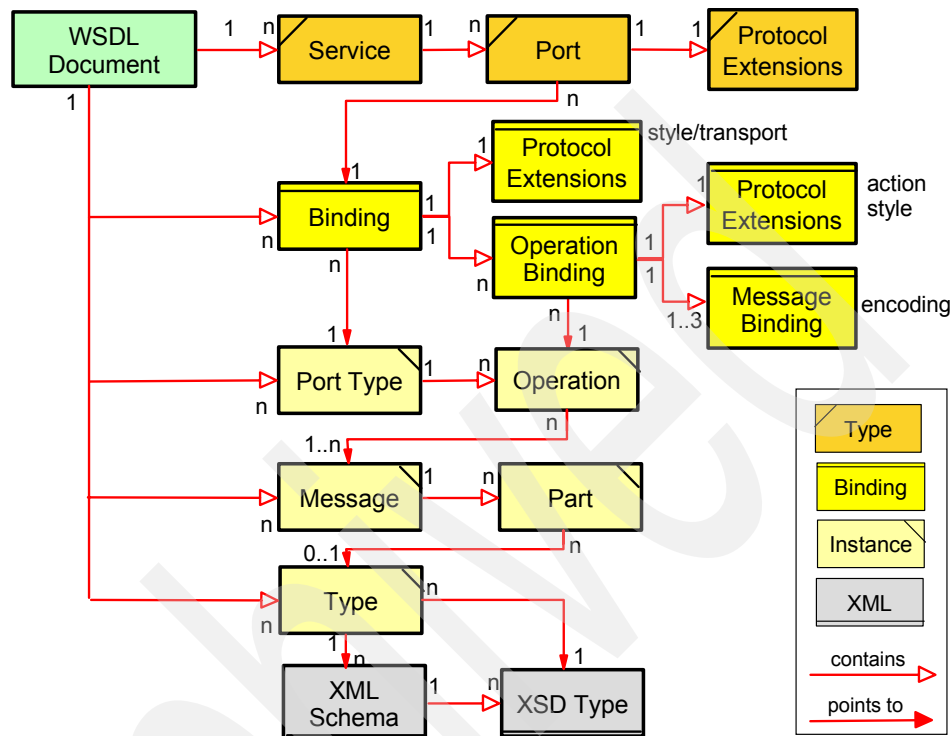


Figure 4-1 WSDL elements and relationships

In practice, a WSDL document can be split into multiple documents using the import element (see “Physical files” on page 75).

Example

Let us now give an example of a simple, complete, and valid WSDL file. As we will see, even a simple WSDL document contains quite a few elements with various relationships to each other. Figure 4-2 and Figure 4-3 contain the WSDL file example. This example is analyzed in detail later in this chapter.

The example is provided as one unique file. We will see later that it is possible to fragment this WSDL document in more than one part. As an exercise, you can try identifying the different elements in Figure 4-1 while examining the example.


```

<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://address.jaxrpc.samples"
  xmlns:apachsoap="http://xml.apache.org/xml-soap"
  xmlns:impl="http://address.jaxrpc.samples"
  xmlns:intf="http://address.jaxrpc.samples"
  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <wsdl:types>
    <schema targetNamespace="http://address.jaxrpc.samples"
      xmlns="http://www.w3.org/2001/XMLSchema">
      <import namespace="http://schemas.xmlsoap.org/soap/encoding/" />
      <complexType name="AddressBean">
        <sequence>
          <element name="street" nillable="true" type="xsd:string"/>
          <element name="zipcode" type="xsd:int"/>
        </sequence>
      </complexType>
      <element name="AddressBean" nillable="true" type="impl:AddressBean"/>
    </schema>
    <import namespace="http://www.w3.org/2001/XMLSchema"/>
  </wsdl:types>

  <wsdl:message name="updateAddressRequest">
    <wsdl:part name="in0" type="intf:AddressBean"/>
    <wsdl:part name="in1" type="xsd:int"/>
  </wsdl:message>
  <wsdl:message name="updateAddressResponse">
    <wsdl:part name="return" type="xsd:string"/>
  </wsdl:message>
  <wsdl:message name="updateAddressFaultInfo">
    <wsdl:part name="fault" type="xsd:string"/>
  </wsdl:message>

```

Figure 4-2 WSDL simple example: Part 1

```

<wsdl:portType name="AddressService">
  <wsdl:operation name="updateAddress" parameterOrder="in0 in1">
    <wsdl:input message="intf:updateAddressRequest"
      name="updateAddressRequest"/>
    <wsdl:output message="intf:updateAddressResponse"
      name="updateAddressResponse"/>
    <wsdl:fault message="intf:updateAddressFaultInfo"
      name="updateAddressFaultInfo"/>
  </wsdl:operation>
</wsdl:portType>

<wsdl:binding name="AddressSoapBinding" type="intf:AddressService">
  <wsdlsoap:binding style="rpc"
    transport="http://schemas.xmlsoap.org/soap/http"/>
  <wsdl:operation name="updateAddress">
    <wsdlsoap:operation soapAction=""/>
    <wsdl:input name="updateAddressRequest">
      <wsdlsoap:body
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="http://address.jaxrpc.samples" use="encoded"/>
    </wsdl:input>
    <wsdl:output name="updateAddressResponse">
      <wsdlsoap:body
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="http://address.jaxrpc.samples" use="encoded"/>
    </wsdl:output>
    <wsdl:fault name="updateAddressFaultInfo">
      <wsdlsoap:body
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="http://address.jaxrpc.samples" use="literal"/>
    </wsdl:fault>
  </wsdl:operation>
</wsdl:binding>

<wsdl:service name="AddressServiceService">
  <wsdl:port binding="intf:AddressSoapBinding" name="Address">
    <wsdlsoap:address
      location="http://localhost:8080/axis/services/Address"/>
  </wsdl:port>
</wsdl:service>
</wsdl:definitions>

```

Figure 4-3 WSDL simple example: Part 2

Physical files

A WSDL document can be created in one or more physical files. If they are more than one, we have to connect these files using an *import* element. This separation of files can be convenient to hold the abstraction of concepts and to allow better maintenance.

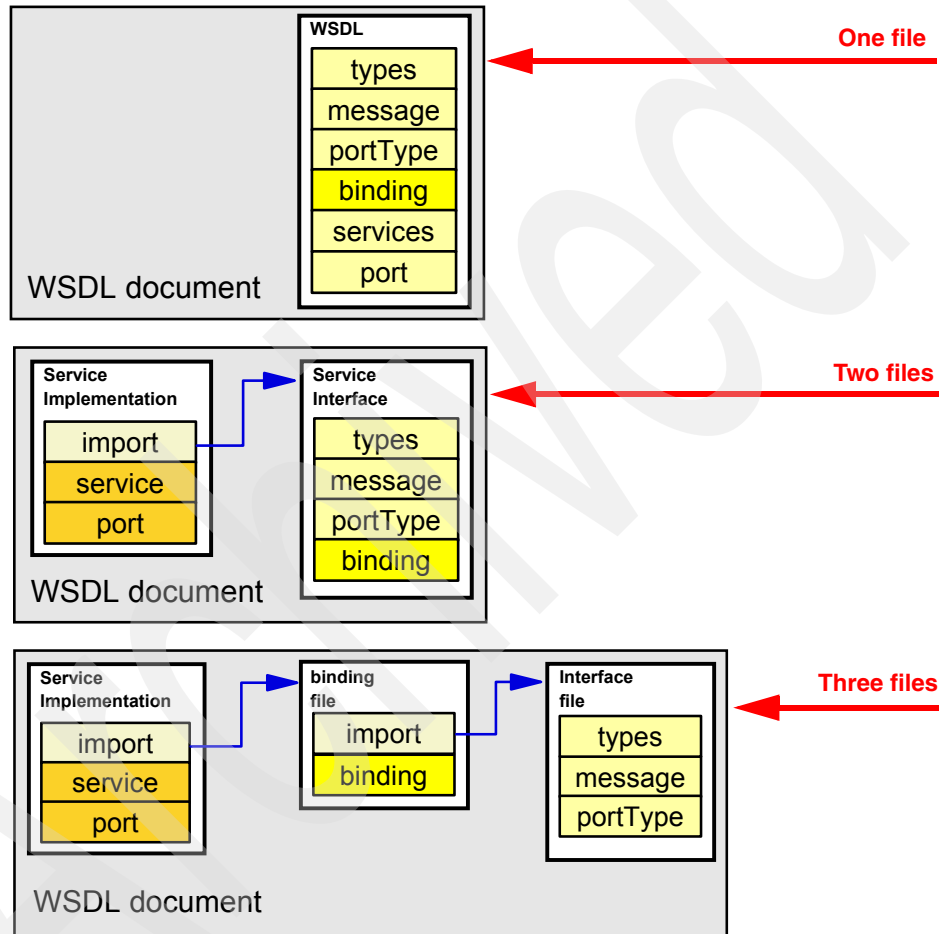


Figure 4-4 WSDL document structure as one, two, or three files

Figure 4-4 shows the same Web service described in one, two, or three files:

- ▶ One file, typically used in Axis and in Application Developer Versions 5.1/6.0
- ▶ Two files, typically used in Application Developer Version 4
- ▶ Three files, typically used in Application Developer Version 5.0

All three examples stand for the same Web service. Therefore, it is important not to be confused by the number of files used to define the WSDL document. There is only one WSDL specification that defines the elements of a WSDL document; how many files are used to store the document is up to the implementer.

Namespaces

WSDL uses the XML namespaces listed in Table 4-1.

Table 4-1 WSDL namespaces

Prefix	Namespace URI	Explanation
wsdl	http://schemas.xmlsoap.org/wsdl/	Namespace for WSDL framework.
soap	http://schemas.xmlsoap.org/wsdl/soap/	SOAP binding.
http	http://schemas.xmlsoap.org/wsdl/http/	HTTP binding.
mime	http://schemas.xmlsoap.org/wsdl/mime/	MIME binding.
soapenc	http://schemas.xmlsoap.org/soap/encoding/	Encoding namespace as defined by SOAP 1.1.
soapenv	http://schemas.xmlsoap.org/soap/envelope/	Envelope namespace as defined by SOAP 1.1.
xsi	http://www.w3.org/2000/10/XMLSchema-instance	Instance namespace as defined by XSD.
xsd	http://www.w3.org/2000/10/XMLSchema	Schema namespace as defined by XSD.
tns	(URL to WSDL file)	The <i>this namespace</i> (tns) prefix is used as a convention to refer to the current document. Do not confuse it with the XSD <i>target namespace</i> , which is a different concept.

The first four namespaces are defined by the WSDL specification itself; the next four definitions reference namespaces that are defined in the SOAP and XSD standards. The last one is local to each specification. Note that in our example, we do not use real namespaces; the URIs contain localhost.

WSDL definition

The WSDL definition contains types, messages, operations, port types, bindings, ports, and services.

Also, WSDL provides an optional element called `wsdl:document` as a container for human-readable documentation.

Types

The `types` element encloses data type definitions used by the exchanged messages. WSDL uses XML Schema Definitions (XSDs) as its canonical and built-in type system:

```
<definitions .... >
  <types>
    <xsd:schema .... />(0 or more)
  </types>
</definitions>
```

The XSD type system can be used to define the types in a message regardless of whether or not the resulting wire format is XML.

There is an extensibility element (placeholder for additional XML elements, that is) that can be used to provide an XML container element to define additional type information in case the XSD type system does not provide sufficient modeling capabilities.

In our example, the type definition, shown in Figure 4-5, is where we specify that there is a complex type called `AddressBean`, which is composed of two elements, a `street` and a `zipcode`. We also specify that the type of the `street` is a string and the type of the `zip code` is a number (`int`).

```

<wsdl:types>
  <schema targetNamespace="http://address.jaxrpc.samples"
    xmlns="http://www.w3.org/2001/XMLSchema">
    <import namespace="http://schemas.xmlsoap.org/soap/encoding/" />
    <complexType name="AddressBean">
      <sequence>
        <element name="street" nillable="true" type="xsd:string"/>
        <element name="zipcode" type="xsd:int"/>
      </sequence>
    </complexType>
    <element name="AddressBean" nillable="true" type="impl:AddressBean"/>
  </schema>
  <import namespace="http://www.w3.org/2001/XMLSchema" />
</wsdl:types>

```

Figure 4-5 Types definition in our WSDL document example

Messages

Messages consist of one or more logical parts. A message represents one interaction between a service requestor and service provider. If an operation is bidirectional (an RPC call returning a result, for example), at least two message definitions are used in order to specify the transmission on the way to and from the service provider:

```

<definitions .... >
  <message name="nmtoken"> (0 or more)
    <part name="nmtoken" element="qname" (0 or 1) type="qname" (0 or 1)/>
    (0 or more)
  </message>
</definitions>

```

The abstract message definitions are used by the operation element. Multiple operations can refer to the same message definition.

Operations and messages are modeled separately in order to support flexibility and simplify reuse of existing specifications. For example, two operations with the same parameters can share one abstract message definition.

In our example, the messages definition, shown in Figure 4-6, is where we specify the different parts that compose each message. The request message `updateAddressRequest` is composed of an `AddressBean` part and an `int` part. The response message `updateAddressResponse` is composed of a `string` part. The fault message `updateAddressFaultInfo` is composed of a `string` part.

```

<wsdl:message name="updateAddressRequest">
  <wsdl:part name="in0" type="intf:AddressBean"/>
  <wsdl:part name="in1" type="xsd:int"/>
</wsdl:message>
<wsdl:message name="updateAddressResponse">
  <wsdl:part name="return" type="xsd:string"/>
</wsdl:message>
<wsdl:message name="updateAddressFaultInfo">
  <wsdl:part name="fault" type="xsd:string"/>
</wsdl:message>

```

Figure 4-6 Message definition in our WSDL document example

Port types

A port type is a named set of abstract operations and the abstract messages involved:

```

<wsdl:definitions .... >
  <wsdl:portType name="nmtoken">
    <wsdl:operation name="nmtoken" .... /> (0 or more)
  </wsdl:portType>
</wsdl:definitions>

```

Operations

WSDL defines four types of operations that a port can support:

One-way	The port receives a message. There is an <i>input message</i> only.
Request-response	The port receives a message and sends a correlated message. There is an input message followed by an <i>output message</i> .
Solicit-response	The port sends a message and receives a correlated message. There is an output message followed by an input message.
Notification	The port sends a message. There is an output message only. This type of operation could be used in a publish/subscribe scenario.

Each of these operation types can be supported with variations of the following syntax. Presence and order of the input, output, and fault messages determine the type of message:

```
<wsdl:definitions .... >
  <wsdl:portType .... > (0 or more)
    <wsdl:operation name="nmtoken" parameterOrder="nmtokens">
      <wsdl:input name="nmtoken" (0 or 1) message="qname"/> (0 or 1)
      <wsdl:output name="nmtoken" (0 or 1) message="qname"/> (0 or 1)
      <wsdl:fault name="nmtoken" message="qname"/> (0 or more)
    </wsdl:operation>
  </wsdl:portType >
</wsdl:definitions>
```

Note that a request-response operation is an abstract notion. A particular binding must be consulted to determine how the messages are actually sent:

- ▶ Within a single transport-level operation, such as an HTTP request/response message pair, which is the preferred option
- ▶ As two independent transport-level operations, which can be required if the transport protocol only supports one-way communication

In our example, the port type and operation definition, shown in Figure 4-7, are where we specify the port type, called `AddressService`, and a set of operations. In this case, there is only one operation, called `updateAddress`.

We also specify the interface that the Web service provides to its possible clients, with the input message `updateAddressRequest`, the output message `updateAddressResponse`, and the `updateAddressFaultInfo` that are used in the transaction.

```
<wsdl:portType name="AddressService">
  <wsdl:operation name="updateAddress" parameterOrder="in0 in1">
    <wsdl:input message="intf:updateAddressRequest"
      name="updateAddressRequest"/>
    <wsdl:output message="intf:updateAddressResponse"
      name="updateAddressResponse"/>
    <wsdl:fault message="intf:updateAddressFaultInfo"
      name="updateAddressFaultInfo"/>
  </wsdl:operation>
</wsdl:portType>
```

Figure 4-7 Port type and operation definition in our WSDL document example

Bindings

A binding contains:

- ▶ Protocol-specific general binding data, such as the underlying transport protocol and the communication style for SOAP.
- ▶ Protocol extensions for operations and their messages include the URN and encoding information for SOAP, for example.

Each binding references one port type; one port type can be used in multiple bindings. All operations defined within the port type must be bound in the binding. The pseudo XSD for the binding looks like this:

```
<wsdl:definitions .... >
  <wsdl:binding name="nmtoken" type="qname"> (0 or more)
    <!-- extensibility element (1) --> (0 or more)
    <wsdl:operation name="nmtoken"> (0 or more)
      <!-- extensibility element (2) --> (0 or more)
      <wsdl:input name="nmtoken" (0 or 1) > (0 or 1)
        <!-- extensibility element (3) -->
      </wsdl:input>
      <wsdl:output name="nmtoken" (0 or 1) > (0 or 1)
        <!-- extensibility element (4) --> (0 or more)
      </wsdl:output>
      <wsdl:fault name="nmtoken"> (0 or more)
        <!-- extensibility element (5) --> (0 or more)
      </wsdl:fault>
    </wsdl:operation>
  </wsdl:binding>
</wsdl:definitions>
```

As we have already seen, a port references a binding. The port and binding are modeled as separate entities in order to support flexibility and location transparency. Two ports that merely differ in their network address can share the same protocol binding.

The extensibility elements `<!-- extensibility element (x) -->` use XML namespaces in order to incorporate protocol-specific information into the language- and protocol-independent WSDL specification. We introduce the extensibility elements supporting SOAP, HTTP, and MIME in “WSDL bindings” on page 83.

In our example, the binding definition, shown in Figure 4-8, is where we specify our binding name, `AddressSoapBinding`. The connection must be SOAP HTTP, and the style must be `rpc`. We provide a reference to our operation, `updateAddress`; define the input message `updateAddressRequest` and the output message `updateAddressResponse`, both to be SOAP encoded; and the `fault`

message, which is `literal`. Because the fault information is always one string only, it is suitable to use `literal` for the encoding.

```
<wsdl:binding name="AddressSoapBinding" type="intf:AddressService">
  <wsdlsoap:binding style="rpc"
    transport="http://schemas.xmlsoap.org/soap/http"/>
  <wsdl:operation name="updateAddress">
    <wsdlsoap:operation soapAction=""/>
    <wsdl:input name="updateAddressRequest">
      <wsdlsoap:body
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="http://address.jaxrpc.samples" use="encoded"/>
    </wsdl:input>
    <wsdl:output name="updateAddressResponse">
      <wsdlsoap:body
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="http://address.jaxrpc.samples" use="encoded"/>
    </wsdl:output>
    <wsdl:fault name="updateAddressFaultInfo">
      <wsdlsoap:body
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="http://address.jaxrpc.samples" use="literal"/>
    </wsdl:fault>
  </wsdl:operation>
</wsdl:binding>
```

Figure 4-8 Binding definition in our WSDL document example

Service definition

A service definition merely bundles a set of ports together under a name, as the following pseudo XSD definition of the service element shows. This pseudo XSD notation is introduced by the WSDL specification:

```
<wsdl:definitions .... >
  <wsdl:service name="nmtoken"> (0 or more)
    <wsdl:port .... /> (0 or more)
  </wsdl:service>
</wsdl:definitions>
```

Multiple service definitions can appear in a single WSDL document.

Port definition

A port definition describes an individual endpoint by specifying a single address for a binding:

```
<wsdl:definitions .... >
  <wsdl:service .... > (0 or more)
    <wsdl:port name="nmtoken" binding="qname"> (0 or more)
      <!-- extensibility element (1) -->
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

The binding attribute is of type QName, which is a qualified name (equivalent to the one used in SOAP). It refers to a binding. A port contains exactly one network address; all other protocol-specific information is contained in the binding.

Any port in the implementation part must reference exactly one binding in the interface part.

<!-- extensibility element (1) --> is a placeholder for additional XML elements that can hold protocol-specific information. This mechanism is required, because WSDL is designed to support multiple runtime protocols. For SOAP, the URL of the RPC router servlet is specified as the SOAP address here.

In our example, the service and port definition, shown in Figure 4-9, is where we specify our service, called `AddressServiceService`, that contains a collection of our ports. In this case, there is only one that uses the `AddressSoapBinding` and is called `Address`. In this port, we specify our connection point as, for example, `http://localhost:9080/Router/services/Address`.

```
<wsdl:service name="AddressServiceService">
  <wsdl:port binding="intf:AddressSoapBinding" name="Address">
    <wsdl:soap:address
      location="http://localhost:9080/Router/services/Address"/>
  </wsdl:port>
</wsdl:service>
```

Figure 4-9 Service and port definition in our WSDL document example

WSDL bindings

We now investigate the WSDL extensibility elements supporting SOAP, HTTP, and MIME transport bindings. Other bindings, such as EJB, JMS, and plain Java, are available as well.

SOAP binding

WSDL includes a binding for SOAP 1.1 endpoints, which supports the specification of the following protocol-specific information:

- ▶ An indication that a binding is bound to the SOAP 1.1 protocol
- ▶ A way of specifying an address for a SOAP endpoint
- ▶ The URI for the SOAPAction HTTP header for the HTTP binding of SOAP
- ▶ A list of definitions for headers that are transmitted as part of the SOAP envelope

Table 4-2 lists the corresponding extension elements.

Table 4-2 SOAP extensibility elements in WSDL

Extension and attributes		Explanation
<soap:binding ...>		Binding level; specifies defaults for all operations.
	transport="uri" (0 or 1)	Binding level; transport is the runtime transport protocol used by SOAP (HTTP, SMTP, and so on).
	style="rpc document" (0 or 1)	The style is one of the two SOAP communication styles, rpc or document.
<soap:operation ... >		Extends operation definition.
	soapAction="uri" (0 or 1)	URN.
	style="rpc document" (0 or 1)	See binding level.
<soap:body ... >		Extends operation definition; specifies how message parts appear inside the SOAP body.
	parts="nmtokens"	Optional; allows externalizing message parts.
	use="encoded literal"	encoded: messages reference abstract WSDL type elements; encodingStyle extension used. literal: messages reference concrete XSD (no WSDL type); usage of encodingStyle is optional.
	encodingStyle="uri-list"(0 or 1)	List of supported message encoding styles.
	namespace="uri" (0 or 1)	URN of the service.
<soap:fault ... >		Extends operation definition; contents of fault details element.

Extension and attributes		Explanation
	name="nmtoken"	Relates soap:faul t to wsd l :faul t for operation.
	use, encodingStyle, namespace	See soap:body.
<soap:address ... >		Extends port definition.
	location="uri"	Network address of RPC router.
<soap:header ... >		Operation level; shaped after <soap:body ...>.
<soap:headerfaul t ... >		Operation level; shaped after <soap:body ...>.

For an example of extensibility elements, refer to Figure 4-8 on page 82.

Note that the WSDL specification deals with encoding only. The mappings to be used for a specific type under a certain encoding are beyond the scope of this book. They are part of the SOAP client and server runtime configuration (client API and deployment descriptor, respectively).

HTTP binding

WSDL includes a binding for HTTP 1.1 GET and POST verbs to describe the interaction between a Web browser and a Web application. This allows applications other than Web browsers to interact with the application (its controller servlets, for example).

The following protocol-specific information is required to describe a Web service that can be accessed through HTTP:

- ▶ An indication that a binding uses HTTP GET or POST
- ▶ An address for the port
- ▶ A relative address for each operation (relative to the base address defined by the port)

Table 4-3 lists the defined extension elements.

Table 4-3 HTTP extension elements in WSDL

Extension	Explanation
<http:address location="uri" />	Extends the port definition and contains the base URL.
<http:binding verb="nmtoken" />	The HTTP operation to be performed (nmtoken=GET or POST).

Extension	Explanation
<code><http:operation location="uri"/></code>	Extends the operation binding and specifies the relative URL.
<code><http:urlEncoded/></code>	Message parts are encoded into the HTTP request URI using the standard URI-encoding rules.
<code><http:urlReplacement/></code>	Message parts are encoded into the HTTP request URI using a replacement algorithm.

MIME extension elements might have to be used as well (see the next section).

MIME binding

The response message of a Web service might be formatted according to the MIME format `multipart/related`, returning mixed content, such as images and text documents. WSDL provides support for describing such messages.

Table 4-4 lists the extensions that are available to describe a Web service that can be accessed using MIME.

Table 4-4 MIME extension elements in WSDL

Extension name	Explanation
<code><mime:content part="nmtoken"? type="string"?/></code>	Name and MIME type of WSDL message part
<code><mime:multipartRelated></code>	Describes each part of a multipart/related message
<code><soap:body></code>	Same as in SOAP binding
<code><mime:mimeXml part="nmtoken"?/></code>	For XML elements that do not travel inside a SOAP envelope

WSDL API

There is a WSDL Java API called WSDL4J, exposing the WSDL object model. Its capabilities include the parsing of the contents of a WSDL document and programmatic creation of new WSDL documents. Note that it is always possible to use XML parsers or XSL transformations. Currently, WSDL4J is an open source project available on the IBM developerWorks Web site:

<http://www-124.ibm.com/developerworks/projects/wsd4j/>
<http://www.ibm.com/developerworks/opensource>

WSDL4J will be a reference implementation for JSR 110 (Java APIs for WSDL). Primarily, it is a set of Java interfaces that can be implemented by anyone. The Java package name is `javax.wsdl`.

Figure 4-10 is an example in which we provide a function to obtain all the port addresses available for a specific SOAP binding in a specified WSDL document. These addresses are returned in a vector element of strings with the URL locations.

```
private Vector getWSDLPort(String fileName, String serviceName) {
    final String serviceNameSpace = "http://wsj.itso";
    int endPointCounter = 0;
    Service service;
    Port port = null;
    Vector serviceEndpoint = new Vector();
    try {
        // Read WSDL document and get definitions element
        WSDLFactory wsdlFactory = WSDLFactory.newInstance();
        WSDLReader wsdlReader = wsdlFactory.newWSDLReader();
        Definition definition = wsdlReader.readWSDL(null, fileName);
        // Get the ports for the WeatherForecast_SEIService
        service = definition.getService(new
            QName(serviceNameSpace, serviceName));
        Map ports = service.getPorts();
        Collection values = ports.values();
        for (Iterator iterator = values.iterator(); iterator.hasNext();) {
            port = (Port) iterator.next();
            List list = port.getExtensibilityElements();
            for (Iterator iter = list.iterator(); iter.hasNext();) {
                // The SOAP binding is an extensibility element of Port
                ExtensibilityElement element =
                    (ExtensibilityElement) iter.next();
                if (element instanceof SOAPAddress) {
                    SOAPAddress soapAddress = (SOAPAddress) element;
                    serviceEndpoint.add(soapAddress.getLocationURI());
                }
            }
        }
    } catch (WSDLException e) { e.printStackTrace(); }
    return serviceEndpoint;
}
```

Figure 4-10 WSDL API example

Outlook

WSDL 1.1 is currently in the process of being standardized by the W3C. However, new working draft specifications Web Services Description Language (WSDL) Version 1.2 and Web Services Description Language (WSDL) Version 1.2: Bindings have been released. These specifications provide a new conceptual framework approach to improve the operability and the components definition.

Version 1.2 enhancements to Version 1.1 are:

- ▶ WSDL 1.2 includes language clarifications that make it easier for developers to understand and use WSDL.
- ▶ WSDL 1.2 provides support for W3C recommendations, including XML Schemas and XML Information Set.

XML Information Set (Infoset) is an abstract data set that provides a consistent set of definitions for use in other specifications that have to refer to the information in a well-formed XML document. See:

<http://www.w3.org/TR/xml-infoset/>

- ▶ WSDL 1.2 adopts a conceptual framework approach to define the description components, which makes them simpler and more flexible.
- ▶ WSDL 1.2 provides a better definition for the HTTP 1.1 binding and will soon provide a binding for SOAP 1.2, which allows description of services using the most current version of SOAP.

Tool support for WSDL is already available, as covered in Part 2, “Implementing and using Web services” on page 231.

An example of another innovation in the WSDL area is the Web Services Invocation Framework (WSIF), providing a WSDL-centric service requestor API. WSIF is transport agnostic; thus, a single WSIF client can support multiple runtime protocols simultaneously, such as SOAP, a direct HTTP connection, and a local Java call.

Summary

This introduction has shown the power of WSDL. WSDL provides an abstract part that is language and protocol independent, as well as bindings for the runtime protocols used in the service-oriented architecture (SOA).

This chapter has also shown that even a simple Web service definition has to cover many interrelated aspects, yielding a rather complex specification file. Writing WSDL documents from scratch is an error-prone task; therefore, there is a strong need for tool support. We cover these tools in Part 2, “Implementing and using Web services” on page 231.

More information

As of today, few WSDL tutorials or other supporting information are available. As WSDL becomes more widespread, this will change.

The WSDL 1.1 specification is available at:

<http://www.w3.org/TR/wsd1>

For further information about WSDL Version 1.2, visit:

<http://www.w3.org/TR/wsd12>

<http://www.w3.org/TR/wsd12-bindings>

JAX-RPC (JSR 101)

This chapter explains the Java API for XML-based RPC (JAX-RPC) style programming model. The JAX-RPC programming model is defined by the Web services standard JSR 101.

JAX-RPC provides the programming model for SOAP-based applications by abstracting the runtime details and providing mapping services between Java and WSDL.

JSR 101 formalizes the procedure for invoking Web services in an RPC-like manner and the specification is now part of the J2EE 1.4 specification.

In this chapter, we cover the details of JAX-RPC 1.1 and the changes made from JAX-RPC 1.0. At the time of writing, JAX-RPC 2.0 in form of JSR 224 is in early draft.

JAX-RPC defines the specification of distributed computing. Some of the earlier specifications of distributed computing are the Java RMI-IIOP, OMG CORBA, and Microsoft COM.

Terminology: JAX-RPC and JSR 101

Let us start with discussing the terminology. In Web services discussions, the terms JAX-RPC and JSR 101 are used interchangeably. This chapter also uses both the terms. However, at their core, they do not mean the same thing. There is a distinction between these terms. JAX-RPC is the programming style, and JSR 101 is the specification document. JSR 101 lays down the requirements of JAX-RPC 1.0. Over time, as JAX-RPC picks up more momentum, it will encompass wider usage scenarios and might at that point implement newer JSRs in addition to or instead of JSR 101.

What is new in JAX-RPC 1.1:

- ▶ Extensive support for XML Schema in WSDL
 - Nearly all predefined simple types
 - Simple types derived by restriction
 - `xsd:List`
 - Attributes
 - Complex types with simple content
 - Wild cards
 - Name collision
- ▶ WS-I Basic Profile 1.0 support
 - RPC/literal in WSDL
 - One-way operations

What is coming in JAX-RPC 2.0:

- ▶ Features regarding ease of development
- ▶ WSDL metadata (JSR 175)
- ▶ JAXB 2.0 integration
- ▶ SOAP 1.2 and WSDL 1.2 support
- ▶ Support for synchrony and new transport protocols

JAX-RPC basics

Java API for XML-based RPC (JAX-RPC), as the name suggests, is a Java API that facilitates distributed computing in a Web services environment. JAX-RPC-based Java applications can easily communicate with non-Java-based technologies in the RPC style fashion.

JAX-RPC compliance mandates client-side requirements and server-side requirements. Figure 5-1 and Figure 5-2 show JSR 101 clients and the JSR 101 server, respectively. JSR 101 clients can interoperate with any JAX-RPC compliant server.

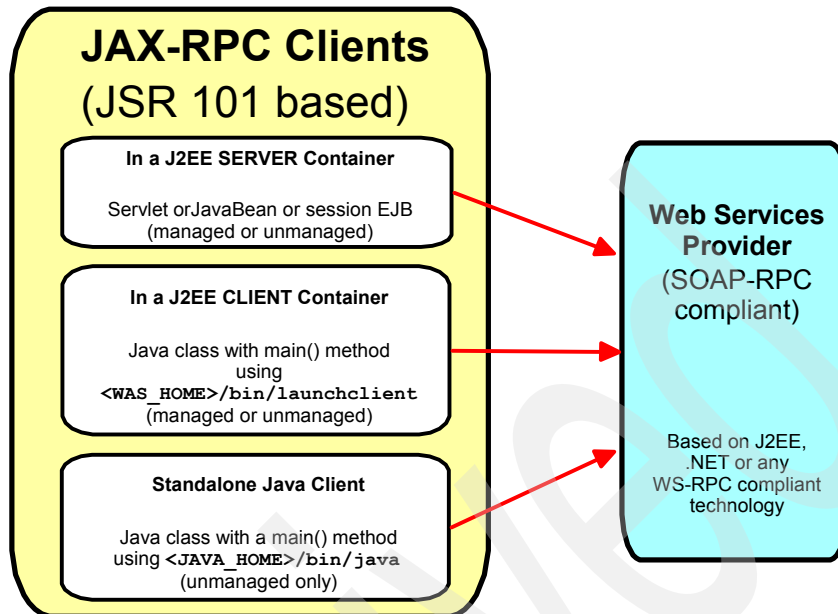


Figure 5-1 JSR 101 client interoperates with any SOAP-RPC compliant server

A JAX-RPC server can interoperate with any SOAP-RPC client, Java based or otherwise.

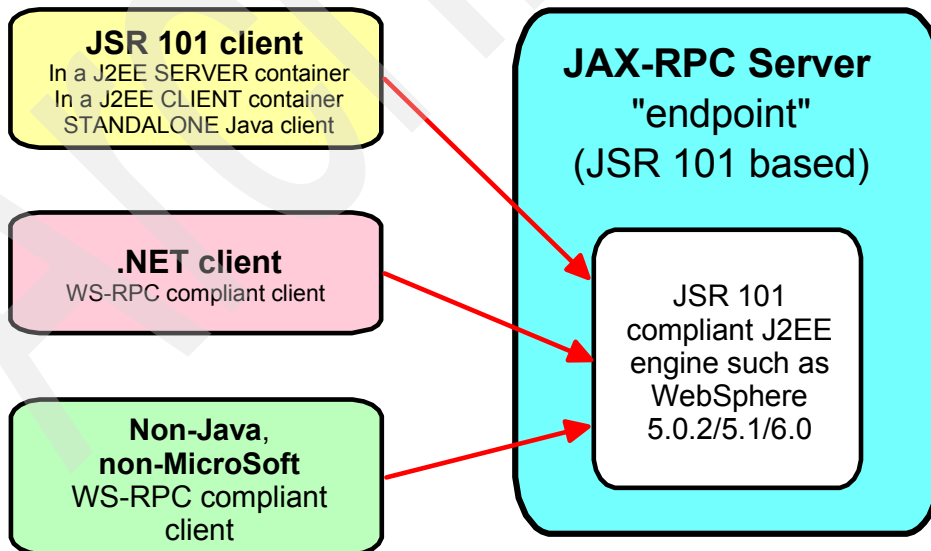


Figure 5-2 JSR 101 server interoperates with any SOAP-RPC compliant client

A JAX-RPC server application's entry point is also known as an *endpoint*. A Web service endpoint is described using a Web Services Description Language (WSDL) document. JAX-RPC is about Web services interoperability across heterogeneous platforms and languages. This makes JAX-RPC a key technology for Web services-based integration.

JAX-RPC client

A JAX-RPC client is capable of invoking a Web service irrespective of whether the service has been defined on the J2EE platform or on a non-Java platform. JAX-RPC clients do not necessarily have to be Java clients. For Java clients to be JAX-RPC compliant, they have to comply with JSR 101. Figure 5-1 explains the various forms of JAX-RPC clients.

JAX-RPC clients can run inside a J2EE container or as a stand-alone Java client. If running inside a J2EE container, they can be coded as *managed* or *unmanaged* clients. If running as a stand-alone Java client, they can only be unmanaged. We discuss unmanaged and managed clients in “Managed and unmanaged JAX-RPC clients” on page 99.

Example 5-1 shows the most simple example of a JAX-RPC client.

Example 5-1 Simple JAX-RPC client

```
package itso.test;

import java.io.*;
import java.util.*;
import itso.test.*;

public class WeatherForecastClient {
    public static void main(String [] args) {
        try{
            WeatherForecastServiceLocator ws1 = new WeatherForecastServiceLocator();
            WeatherForecastService ws = (WeatherForecastService) ws1.getWeather();
            String temperature = ws.getTemperature();
            System.out.println(temperature);
            System.out.println("WeatherForecastClient completed");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

In this example, we highlighted the three essential operations of a JAX-RPC client:

- ▶ Instantiate the *locator* class. The locator class has information about the Web service. The locator class is populated based on the content of the Web service's WSDL file.
- ▶ Instantiate the *service* using the locator class. The service interface is a local representation of the Web service. The service interface is implemented by the *stub* class.
- ▶ Invoke a method on the service interface.

JAX-RPC client programming styles

If the Web service is not expected to ever change, the mechanism of Example 5-1 works very well. This mechanism is known as a *static stub*-based invocation of a Web service. But if the Web service were to change, the client would have to be changed accordingly. We, therefore, provide capability for clients to be *dynamic*. There are three types of Web services clients (Figure 5-3):

- ▶ Static stub
- ▶ Dynamic proxy
- ▶ Dynamic invocation interface (DII)

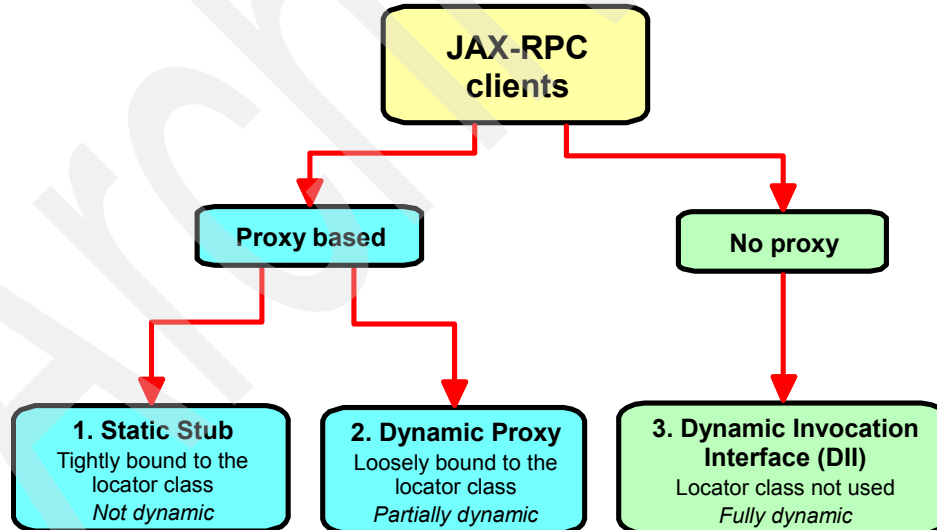


Figure 5-3 Static and dynamic JAX-RPC clients

Static stub

Let us fully understand the static stub client of Example 5-1 on page 94. A static stub-based JAX-RPC client uses proxy classes to communicate with the Web service (Figure 5-4). These proxy classes are generated from the WSDL of the Web service. In WebSphere Application Server, the proxy classes can be generated by the tool `<WAS_HOME>/bin/WSDL2JAVA`.

After the proxy classes have been generated, they are copied to the client machine. The client can then invoke the Web service based only on these proxy classes.

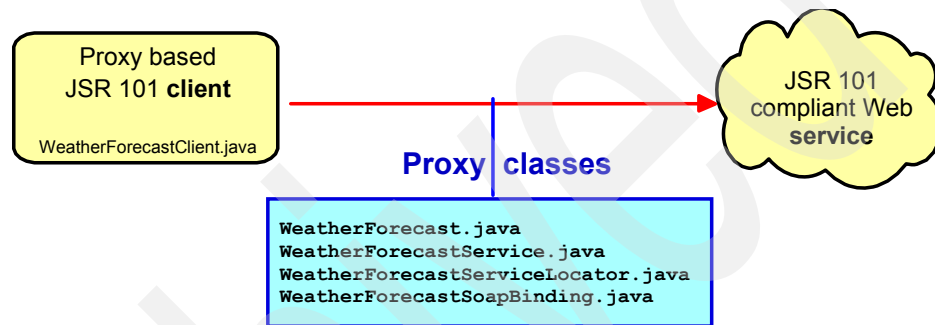


Figure 5-4 Proxy-based JAX-RPC client

The four proxy classes are:

- ▶ Service endpoint interface (SEI): `WeatherForecast`—defines the method signatures of the Web service.
- ▶ Service interface: `WeatherForecastService`—defines the service methods of the locator class (for example, retrieving the SEI).
- ▶ Service locator class: `WeatherForecastServiceLocator`—implements the service interface (provides access to the SEI).
- ▶ Binding stub: `WeatherForecastSoapBinding`—implements the SEI (makes the actual calls to the Web service).

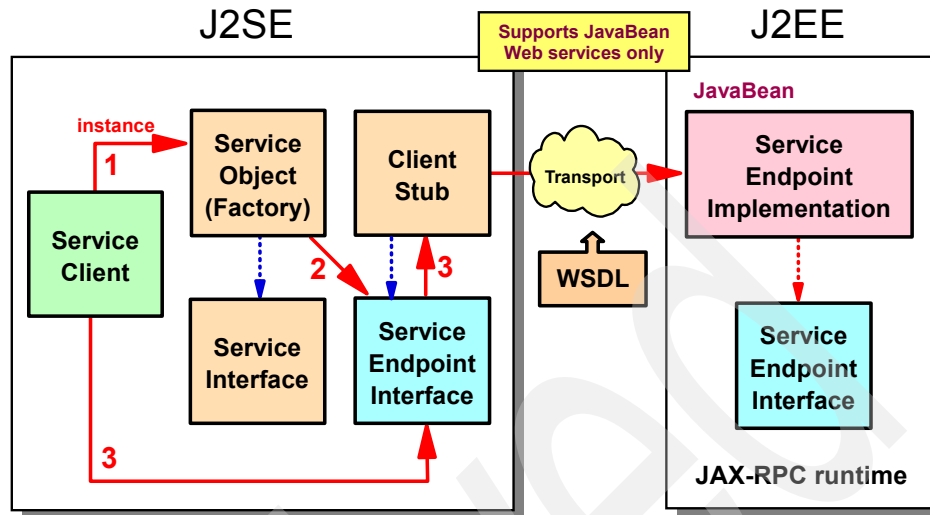


Figure 5-5 JAX-RPC static client calling sequence

At runtime, the client instantiates the service locator class, calls it to retrieve the SEI (actually the binding stub), and then calls the SEI to invoke the Web service. Figure 5-5 shows the calling sequence in a Java implementation:

1. The client instantiates the service locator.
2. The client calls the service locator to retrieve the SEI (an instance of the client stub that implements the SEI is returned).
3. The client invokes a Web service through the SEI.

The client can be a J2SE client that invokes a Web service in a J2EE-compliant application server. The Web service implementation is a JavaBean. To support an EJB Web service, refer to Chapter 6, "Web Services for J2EE" on page 103.

Dynamic proxy

Let us see an example of a dynamic, proxy-based JAX-RPC client (Example 5-2).

In dynamic proxy clients, the default destination of the Web service can be changed in the client by specifying a different destination in the client application.

Example 5-2 Dynamic proxy client, only partially dynamic

```
import javax.xml.namespace.QName;
import java.io.*;
import java.util.*;
```

```

public class WeatherForecastDynamicProxyClient {

    public static void main(String [] args){
        try{
            WeatherForecastServiceLocator ws1 = new WeatherForecastServiceLocator();
            QName qn = new QName("http://www.somewhere.com", "WeatherForecast");
            WeatherForecast ws = (WeatherForecast)
                ws1.getPort(qn,WeatherForecast.class);
            String temperature = ws.getTemperature();
            System.out.println(temperature);
            System.out.println("DynamicProxyJavaClient completed");
        } catch (Exception e){
            e.printStackTrace();
        }
    }
}

```

At runtime the service locator is instantiated. The SEI is retrieved using a destination (QName).

Dynamic invocation interface (DII)

DII is used when the WSDL of the Web service can change considerably over time. DII-based clients do not use proxy classes, but instead read the entire WSDL file during runtime:

- ▶ Instantiate a DII *service* class.
- ▶ Instantiate a Call object (Call is a class provided by JAX-RPC).
- ▶ Populate the Call object.
- ▶ Invoke the Web service operation on the Call object.

Which style to use

Table 5-1 lists the usage scenarios of the three styles.

Table 5-1 Client styles

Static stub	Dynamic proxy	DII
Web service not expected to change	Some changes to the Web service expected, such as the location of the service	Considerable changes to the Web service expected, such as: - Location of the service - Request/response format - Data types
Most common scenario	Less common	Less common, see note that follows

Important: The support for the dynamic invocation interface in WebSphere Application Server is limited. Complicated Web services that use complex types are not supported and may cause DII to fail.

Managed and unmanaged JAX-RPC clients

So far, we have discussed unmanaged JAX-RPC clients. Managed clients allow the J2EE container to instantiate the proxy classes. Example 5-3 shows a code snippet of a managed, dynamic, proxy-based JAX-RPC client. A static stub-based client can also be coded as a managed client.

Example 5-3 Managed, dynamic, proxy-based client

```
InitialContext ic = new InitialContext();
WeatherForecastServiceLocator wsl =
    (WeatherForecastServiceLocator)
    ic.lookup("java:comp/env/service/WeatherForecastService");
QName qn = new QName("http://www.somewhere.com", "WeatherForecast");
WeatherForecast ws = (WeatherForecast)
wsl.getPort(qn, WeatherForecast.class);

String temperature = ws.getTemperature();
System.out.println(temperature);
System.out.println("DynamicProxyJavaClient completed");
```

The main difference in a managed client and unmanaged client (Example 5-2 on page 97 versus Example 5-3) is the way the WeatherForecastServiceLocator class is instantiated:

- ▶ For a managed client, the locator class is instantiated by the container, by calling the lookup method on the default InitialContext. Note that this requires a deployment descriptor and is really using enterprise Web services (see Chapter 6, “Web Services for J2EE” on page 103).
- ▶ For an unmanaged client, the locator class is instantiated by calling a constructor of the locator class.

JAX-RPC specification details

JSR 101 provides details about JAX-RPC. We provide a brief overview of these details.

Data type mapping: XML to Java, Java to XML

JAX-RPC data flows as XML. For this purpose, the JAX-RPC client has to convert Java data types into XML, and the JAX-RPC server has to convert XML data into Java data types, and vice versa on the result flow (Figure 5-6).

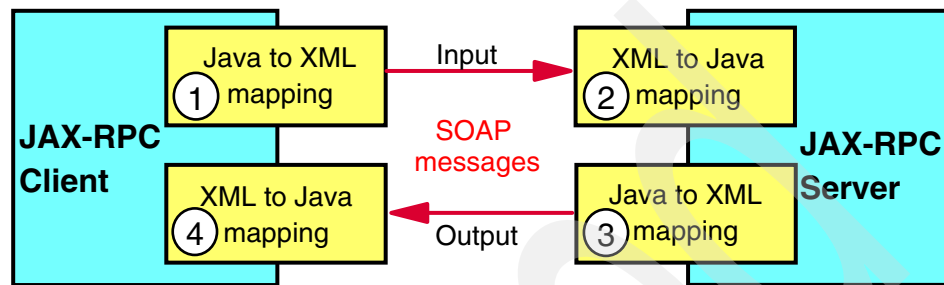


Figure 5-6 Mapping and encoding stages for a Web service

Support is provided to convert simple data types, such as `xsd:string` and `xsd:integer` to `java.lang.String` and `java.math.BigInteger`. The complete list is available at <http://schemas.xmlsoap.org/soap/encoding/>.

In addition to simple data types, JAX-RPC also specifies the mapping of data structures such as arrays. These arrays can contain elements of simple data types or of user-defined data types (JavaBeans). Refer to the specification for details about the mappings.

There are four steps in the process, indicated by the numbers in Figure 5-6:

1. Client input mapping (Java to XML)—This takes the parameter from the Java client and maps it using the input mapping style.
2. Server input mapping (XML to Java)—The inbound parameters are deserialized from the XML style in the message to Java types, which are then used to invoke the method in the JavaBean.
3. Server output mapping (Java to XML)—After the JavaBean has completed its method execution, the return value is inserted into the SOAP reply using the output mapping style.
4. Client output mapping (XML to Java)—The final stage is performed by SOAP for the client proxy, which maps the returned XML elements into Java types.

The mapping between Java and XML is defined in a mapping file. The mapping file, itself an XML file, defines how the Java objects (parameters and results) involved in the Web service call are serialized and deserialized. Serialization is performed using helper classes that can be generated from the WSDL file. See “JAX-RPC mapping deployment descriptor” on page 109 for more information.

Summary

In this chapter, we described JAX-RPC, as defined by the Web services standard JSR 101.

We examined JAX-RPC servers and clients, and the different styles of clients, static and dynamic.

More information

The complete JSR 101 specification is at:

<http://www.jcp.org/en/jsr/detail?id=101>

These are some informative articles available on the IBM developerWorks Web site:

<http://www.ibm.com/developerworks/webservices/library/ws-jsrart/>

<http://www.ibm.com/developerworks/webservices/library/ws-jaxrpc1>

<http://www.ibm.com/developerworks/xml/library/x-tipjaxrpc/>

Web Services for J2EE

In this chapter, we introduce the Web Services for J2EE specification (WSEE). WSEE defines the required architecture for Web services for the Java 2 Platform Enterprise Edition (J2EE) environment.

WSEE is defined in a Java Community Process (JCP) specification, JSR 921, which is the updated specification for JSR 109 (Implementing Enterprise Web Services), and is also known as JSR 109 1.1.

WSEE standardizes the packaging, deployment, and programming model for Web services in a J2EE environment. WSEE-compliant services are portable and interoperable across different application server platforms.

WebSphere Application Server Version 6 supports the most current WSEE Version 1.1. The compliance to WSEE is a defined requirement in the J2EE 1.4 specification.

In this chapter, we discuss the following topics:

- ▶ Web services for J2EE overview
- ▶ Client programming model
- ▶ Server programming model
- ▶ Handlers
- ▶ Security

Web services for J2EE overview

Prior to the appearance of the Web Services for J2EE specification (WSEE), there was no standard definition of how to deploy a Web service in a J2EE environment. Thus, the process to do so was mainly dependant on the destination runtime. WSEE standardizes the process and makes it portable to every J2EE-compliant server platform.

WSEE leverages J2EE technologies defining the needed mechanism to standardize a deployment model for Web services. This standardization wants to achieve the interoperability across different compliant J2EE platforms, transforming the migration among them into a routine process ensuring that vendors interoperate.

WSEE defines the concepts, interfaces, file formats, and responsibilities to support the model for defining and deploying Web services. WSEE-compliant Web service providers can ensure that their services can be deployed later in servers that comply with J2EE and WSEE specifications. WSEE enables developers, assemblers, and deployers to configure Web services through XML-based deployment descriptors.

With the IBM development and deployment tooling, the complexity to define and modify the WSEE deployment descriptors is taken from the developer.

In much the same way that servlets tied together a set of concepts such as cookies and HTTP Session, and EJBs tied together techniques such as RMI, JTA/JTS, and JDBC with a programming and runtime model, WSEE defines the same for implementing and using Web services in J2EE.

WSEE adds additional artifacts to those defined by JAX-RPC (see Chapter 5, “JAX-RPC (JSR 101)” on page 91) and brings JAX-RPC to the J2EE container. Although WSEE does not restrict any implementation, it only defines two:

- ▶ Stateless session EJB in an EJB container
- ▶ Java class running in a Web container

For these two implementation models, and for both the client and server sides, the specification details are:

- ▶ The programming model for J2EE components with Web services
 - How J2EE server components can be defined to be accessible as Web services
 - How J2EE client components and applications can use JAX-RPC to access Web services
- ▶ The assembly of the components with Web services

- ▶ The deploying of these components as Web services components
 - How J2EE server components can be described as Web services
 - How J2EE client components can be described for accessing Web services using JAX-RPC

Client programming model

In this section, we describe the WSEE client programming model. We provide an overview of the characteristics of the client, a description of the different clients, an explanation of the client deployment descriptors, and the roles that take part in the Web service development and assembly.

Overview

The client programming model provides the guidelines for accessing Web services in a J2EE environment. This client must be compliant with the client programming model defined by JAX-RPC (see Chapter 5, “JAX-RPC (JSR 101)” on page 91). Web service clients not running in a J2EE environment cannot use WSEE and will run with pure JAX-RPC.

Programmers can be sure that clients compliant to this specification can access any Web service running in a Web Services for J2EE container and can call any SOAP 1.1 Web service. How the service implementation is realized is transparent to the client.

The client can be a Web service, a J2EE application, or an arbitrary Java application. WSEE is used by the client to access and invoke the Web service methods. The client invokes Web service methods without distinguishing whether those are performed locally or in a remote runtime environment. Further, the client has no control over the life cycle of the Web service.

There is no state persisted over multiple Web service calls. The client must assume that all Web service calls are stateless.

The port provider and the container configuration define the view of the client that includes:

- ▶ The methods to access the port of a Web service— Service interface
- ▶ The methods to call a Web service—Service endpoint interface

Figure 6-1 shows the general J2EE Web service client components.

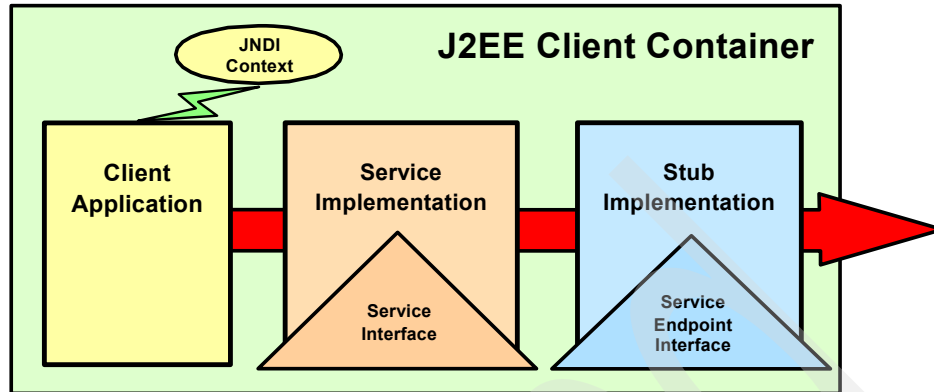


Figure 6-1 General client architectural model

Client concepts

The J2EE client container provides the WSEE runtime that is used by a client to access and invoke Web service methods. The client uses a JNDI lookup to find a service object. A service object implements the service interface as defined by JAX-RPC. The client gets a stub or a proxy by using the factory function of the service object. A stub represents the Web service instance on the client side. Regardless, the Web service client should use the Web service interface and not the stub. Web service stubs are generated during deployment and are specific to the client runtime.

The container is responsible for the JNDI name to service implementation mapping. We recommend that you organize all logical service reference names under the JNDI subcontext *service*. Web service JNDI names are defined in the WSEE client deployment descriptor. We talk about this in more detail in the rest of this chapter.

WSEE specifies three mechanisms for invoking a Web service:

Static stub

Static, because the stub is created before runtime. This requires complete WSDL knowledge.

Dynamic proxy

Dynamic, because the proxy class is created during runtime. Only a partial WSDL definition is required (port type and bindings).

Dynamic invocation interface

Does not require WSDL knowledge. The signature of the remote procedure or the name of the service are unknown until runtime.

These styles are defined in the JAX-RPC specification. See Chapter 5, “JAX-RPC (JSR 101)” on page 91 for detailed information about these styles. We now show the WSEE specifics for these styles.

Static stub

The static stub client is statically bound to a service endpoint interface (SEI), a WSDL port, and a port component. The stub is used by the client to invoke the Web service. A static client is also tied to a specific protocol and transport type.

The static stub is the easiest of the three styles in respect to development. But even a small change in the WSDL document provokes the client useless; the stub must be regenerated. The use of this client is only recommended for Web service where the probability for modifications at the Web service definition is very low. Regardless, the Web service WSDL file is not needed at runtime.

Figure 6-2 shows the static client calling sequence in a J2EE environment.

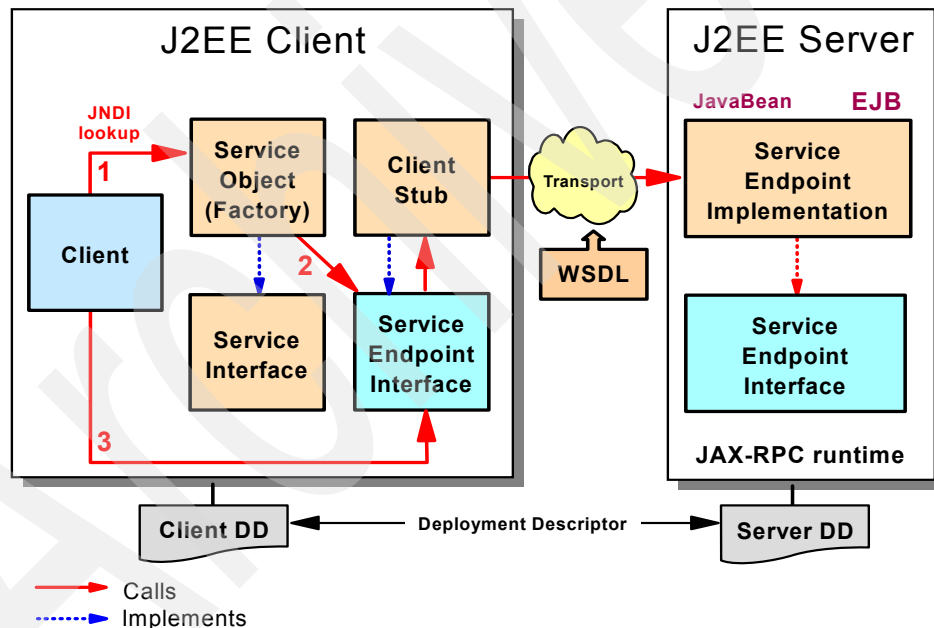


Figure 6-2 Static client calling sequence

In the static client calling sequence:

1. The client makes an JNDI lookup to get an instance of the service object, which implements a service interface.
2. The client uses a factory method of the service object to retrieve the client stub. The client stub implements the SEI.

3. The client invokes the Web service through the SEI.

The configurations for the Web service client and server side are represented by the client and server deployment descriptor shown at the bottom in Figure 6-2.

Dynamic proxy

A dynamic proxy is not tied to an specific stub and, therefore, does not have the restrictions of a static stub client. The dynamic proxy client stub is generated at runtime when a Web service method is called. The Web service WSDL file must be available to the JAX-RPC runtime, because the WSDL file is parsed to create the specific stub.

After the dynamic proxy has been generated by the runtime, the client uses the same mechanism to invoke the Web service as with the static stub implementation.

Using the dynamic proxy, the binding and Web service location can change without the need to manually regenerate the stub. The Web service method signature must not change.

Dynamic invocation interface (DII)

The dynamic invocation interface uses a `javax.xml.rpc.Call` instance for dynamic invocation. Unlike the two previous clients, the DII has to previously configure the `Call` object, providing the following information:

- ▶ Operation name
- ▶ Parameters names, types, and modes
- ▶ Port type and address of a target service endpoint
- ▶ Protocol and transport properties

Note: Refer to the description in “Dynamic invocation interface (DII)” on page 98 for more information and WebSphere restrictions.

Packaging

WSEE defines the artifacts for the Web services client packaging. The packaging structure is specific to the used Web services client type. A WSEE client deployment package must contain the following:

- ▶ WSDL file (see Chapter 4, “Introduction to WSDL” on page 69)
- ▶ Service endpoint interface class (see Chapter 5, “JAX-RPC (JSR 101)” on page 91)
- ▶ Service implementation bean class, and dependent classes
- ▶ Web services client deployment descriptor
- ▶ JAX-RPC mapping file

The first three components have been introduced in previous chapters. Here, the Web services client deployment descriptor and the JAX-RPC mapping file are explained in more detail.

Web service for J2EE client deployment descriptor

The WSEE specification does not define the client deployment descriptor name. Rational Application Developer Version 6 (also Rational Web Developer) adds the WSEE deployment information for J2EE 1.4 Web service clients to existing client project deployment descriptors. Those deployment descriptors are used as defined in the J2EE specification:

- ▶ Web service EJB client—META-INF/ejb-jar.xml
- ▶ Web service Web client—WEB-INF/web.xml

For J2EE 1.3 client projects, a separate deployment descriptor is generated:

- ▶ Web service EJB client—META-INF/webservicesclient.xml
- ▶ Web service Web client—WEB-INF/webservicesclient.xml

WSEE client deployment descriptors contain service reference entries. The J2EE client container will use these definitions to deploy the Web services client. A client deployment descriptor contains following information:

- ▶ Description—Web service description.
- ▶ Service reference—The logical name of the reference used by JNDI to look up the service reference.
- ▶ Service type—The fully qualified name of the service interface; returned by the JNDI lookup.
- ▶ WSDL definition—The location of the service WSDL.
- ▶ JAX-RPC mapping—The location of the JAX-RPC mapping file.
- ▶ Service port—The qualified name of the referenced WSDL service.
- ▶ Ports—The port component provides the local representation of our Web service, the service endpoint interface.
- ▶ Scope—If the client is an EJB, the scope provides the association between the service and the EJB client using the component-scope-refs.
- ▶ Handlers—Defined Web service JAX-RPC handlers.

JAX-RPC mapping deployment descriptor

The name for the JAX-RPC mapping deployment descriptor is not specified by WSEE. Application Developer uses these names for the mapping file:

- ▶ Web service EJB client: META-INF/<WSDL-Filename>_mapping.xml
- ▶ Web service Web client: WEB-INF/<WSDL-Filename>_mapping.xml

The mapping deployment descriptor contains the mapping information between the WSDL definition and the Java interfaces. The main elements that compose a JAX-RPC mapping deployment descriptor are:

- ▶ The package mapping—Contains the relation between the XML namespace and Java packages.
- ▶ The type mapping—Contains the type mapping between Java types and XML types. It is the most remarkable component.

There are four fundamental elements for type mapping:

- Class type—Contains the fully qualified name of the Java bean that stands for the type.
 - Root qname—Contains the qualified name of the bean as it appears in the WSDL definition.
 - Scope—Contains the scope of the type. Possible values are `simpleType`, `complexType`, and `element`.
 - Variables—Contains the mappings between the data members of the Java bean and the XML elements.
- ▶ The service interface mapping—Contains the mapping for the WSDL service definition. The three fundamental elements are:
 - Service interface—Contains the fully qualified name of the service interface Java class.
 - WSDL service name—Contains the qualified name of the service as it appears in the WSDL definition.
 - Port mapping—Contains the mapping for the WSDL ports.
 - ▶ The service endpoint interface mapping—Contains the fully qualified class name of the service endpoint interface. It also contains the mappings for WSDL messages, port types, and bindings.

Roles

There are no new roles defined by the WSEE specification. All used roles can be mapped to existing J2EE platform roles. The roles used in WSEE are:

- ▶ Developer—Responsible for client definition, implementation, and creating the J2EE modules
- ▶ Assembler—Responsible for assembling the modules into an application
- ▶ Deployer—Responsible for publishing the deployed clients and resolving client references

Table 6-1 provides a more detailed description of the role responsibilities.

Table 6-1 Roles and responsibilities for Web service client development

	Developer	Assembler	Deployer
Client implementation	Implement the client. Can implement handlers.		Generate the stubs from the JAX-RPC mapping DD information.
Web services client DD	Define service references. Can define handler references.	Can configure handlers. Assembly modules in an EAR file.	Resolve service and port references.
JAX-RPC mapping DD	Provide the mapping.		
Application server			Deploy the application.

Server programming model

In this section, we describe the WSEE server programming model. We provide an overview of the characteristic of the server, a description of the different server implementations, an explanation of the deployment descriptors used on the server side, and a description of the roles that take part in the application development cycle.

Overview

The server programming model provides the server guidelines for standardizing the deployment of Web services in a J2EE server environment. Depending on the runtime, two implementation models are described:

- Web container—A Java class according to a JAX-RPC servlet-based service
- EJB container—A stateless session EJB

Either of these provides the information to define a port component. A port component defines the server view of a Web service providing a portable Web services programming model. A port component makes available a service entry point defined in a WSDL port address to grant access to the operations stated in the WSDL definition.

Each port model has a service endpoint interface that defines the operations exposed by a Web service and a service implementation bean that implements the business logic for all the operations defined in the service endpoint interface.

The implementation and the packaging of that service depend on the container in which the port is deployed.

The life cycle can vary based on the implementation methodology used and is totally controlled by the associated container. In general, the life cycle of a Web service follows the same steps as its container. A port is created, initialized, executed, and destroyed following the container criteria and without port operations interference.

Figure 6-3 shows an architectural model of a general server.

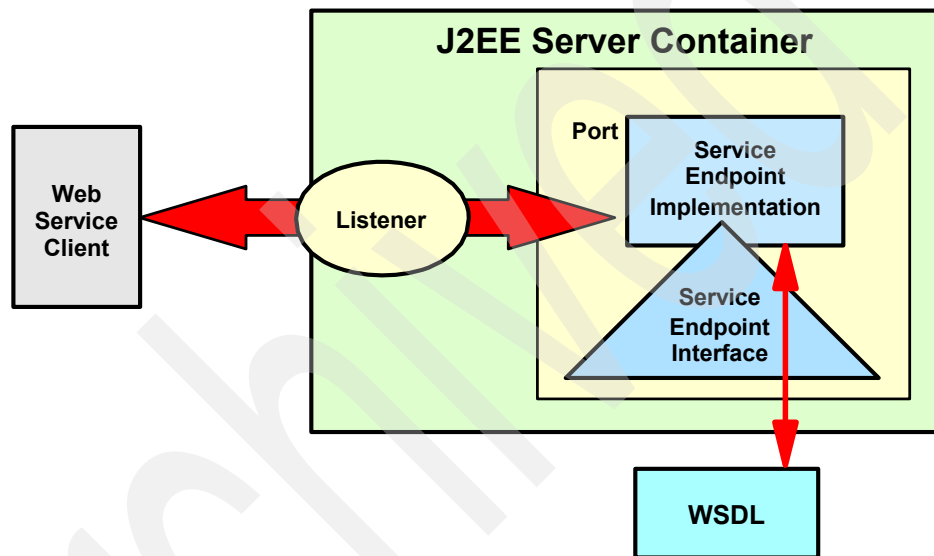


Figure 6-3 General server architectural model

Server concepts

The fundamental part of a Web service is the port component. A port component defines the programming model artifacts that make the Web service a portable server application. That programming model includes:

- ▶ WSDL definition—Provides the description of a Web service.
- ▶ Service endpoint interface (SEI)—Defines the operations available for a Web service. Must follow JAX-RPC mapping rules.
- ▶ Service implementation bean—Implements the SEI methods to provide the business logic of a Web service.
- ▶ Security role references—Provides instance-level security check across the different modules.

Service implementation bean

WSEE defines two possible implementation methods for a Web service:

- ▶ A stateless session EJB
- ▶ JAX-RPC servlet-based service that invokes a JavaBean

Note: An EJB Web service implementation runs in an EJB container. A JavaBean implemented as a JAX-RPC Web service always runs in a Web container.

EJB container programming model

A stateless session EJB can be used to implement a Web service. Therefore, a stateless session EJB has to follow some requirements:

- ▶ The stateless session EJB must have a default public constructor, a default EJB create method, and one or more EJB remote methods.
- ▶ The service endpoint interface must be a subset of the methods of the remote interface of the session EJB. The remote interface does not have to implement the endpoint interface. The methods must be public, but neither final nor static.
- ▶ The stateless session EJB must be a stateless object.
- ▶ The class must be public, but neither final nor abstract.
- ▶ The class must not have a `finalize` method.

Because the stateless session EJB runs in an EJB container, the life cycle of the implementation bean is the same as is stated in the Enterprise JavaBeans specification. Figure 6-4 shows this life cycle.

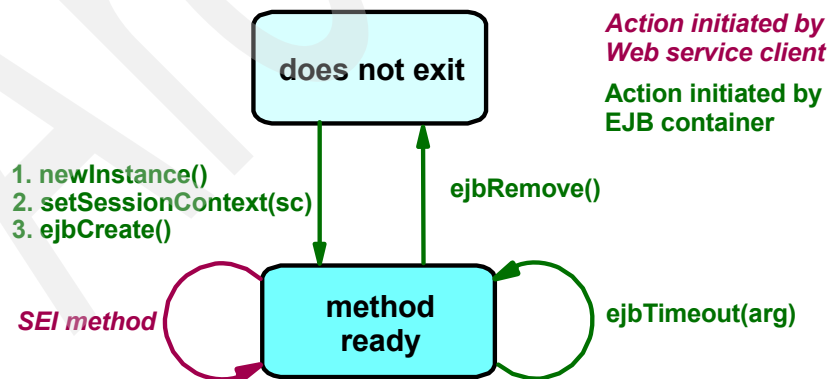


Figure 6-4 Life cycle of a service implementation bean in an EJB container

The EJB container uses the `ejbCreate` and the `ejbRemote` methods to notify a service implementation bean instance of a change in its state. After the container has called the `ejbCreate` method, the service implementation bean is ready for dispatching a request to clients using any of the SEI methods defined for the Web service. The dispatching of these SEI methods is initiated by the client.

The correct J2EE deployment module for an EJB service endpoint is an EJB JAR file. This JAR file contains the stateless session EJB, the required classes, the WSDL definition, and the Web services deployment descriptors. Finally, the modules containing the port components are packaged in an EAR file following the J2EE specifications.

WSEE specifies the exact location of the Web service deployment artifacts. Figure 6-5 shows the enterprise archive structure.

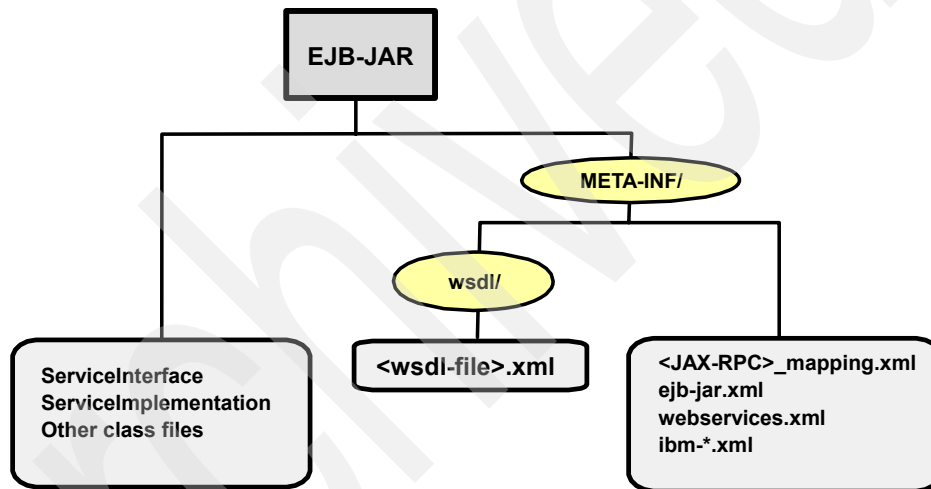


Figure 6-5 WSEE EJB archive package structure

Web container programming model

A Java class can be used to implement a Web service. The Java class must follow the JAX-RPC specifications to create a service implementation bean that runs within a Web container (see Chapter 5, “JAX-RPC (JSR 101)” on page 91). The requirements for Java class are:

- ▶ The Java class must have a default public constructor.
- ▶ The Java class must implement the method signatures of the service endpoint interface. Only those methods are exposed to the client.
- ▶ The Java class must be stateless.
- ▶ The Java class must be public, but neither final nor abstract.

- The Java class must not have a `finalize` method.

The Java class life cycle is controlled by the Web container (Figure 6-6).

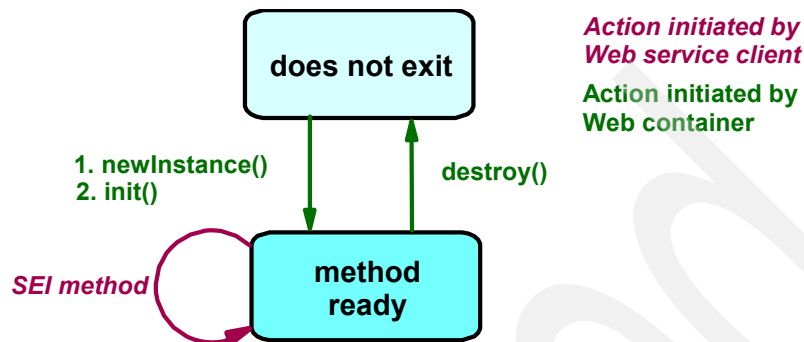


Figure 6-6 Life cycle of a service implementation bean in a Web container

The Web container uses the `init` and the `destroy` methods to notify a service implementation bean instance of a change in its state. After the container has called the `init` method, the service implementation bean is ready for dispatching requests to clients using any of the SEI methods defined for the Web service. The dispatching of these SEI methods is initiated by the client.

The correct J2EE deployment module for a Java class JAX-RPC service endpoint is a WAR file archive. This WAR file contains all the required classes, the WSDL definition, and the Web services deployment descriptors (Figure 6-7).

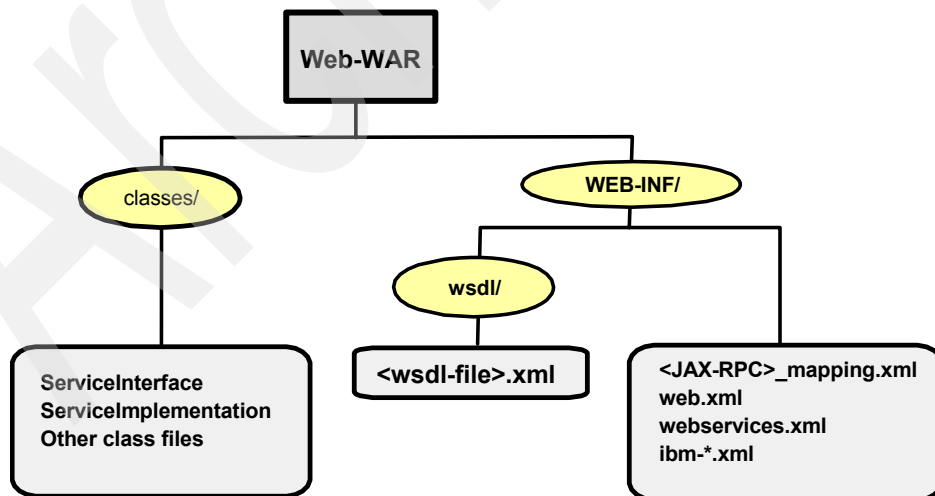


Figure 6-7 WSEE Web archive package structure

Finally, the modules containing the port components are packaged into an EAR file following the J2EE specifications.

Server container responsibilities

A server container provides a JAX-RPC runtime environment for invoking Web services ports. The container is responsible for:

- ▶ Listening to Web services SOAP HTTP requests
- ▶ Parsing the inbound message
- ▶ Mapping the messages to the implementation class and method
- ▶ Creating Java objects from the SOAP envelope
- ▶ Invoking the service implementation bean handlers and instance methods
- ▶ Capturing the response
- ▶ Mapping the Java response objects into a SOAP message
- ▶ Creating the message envelope
- ▶ Sending the message to the client

Packaging

WSEE defines the artifacts for the Web services server packaging. The packaging structure is specific to the Web services server type. The contents of the server package is specified in WSEE as follows:

- ▶ Service endpoint interface classes
- ▶ Generated service interface, and dependent classes (if used)
- ▶ WSDL file
- ▶ Web services client deployment descriptor
- ▶ JAX-RPC mapping file

The first three components have been introduced in previous chapters. Here, the Web services client deployment descriptor and the JAX-RPC mapping file are explained in more detail.

The deployment descriptors used for the server programming model are the Web service deployment descriptor, `webservices.xml`, and the JAX-RPC mapping deployment descriptor.

Web service deployment descriptor

A WSEE server deployment descriptor defines the Web services to deploy in a J2EE container and defines the mapping between WSDL ports and port components. The deployment information is presented in the module-specific deployment descriptor, `ejb-jar.xml` or `web.xml`, and in the Web service deployment descriptor.

A Web service deployment descriptor contains following information:

Service name	The unique name of the service. This name is the same name of the <code>wsdl:service</code> element.
WSDL file	The location of the WSDL description of the service.
JAX-RPC mapping	The location of the WSDL-Java mapping file.
Port	The port component defines the server view, providing the access point and implementation details.
name	The unique name of the WSDL port element for this service. This name is the same name of the <code>wsdl:port</code> element.
qname	The qualified name of the referenced WSDL port.
SEI	The fully qualified class name of the service endpoint interface.
bean class	The implementation name of the Web service. This name must match the name of the <code>ejb-name</code> element stated in the <code>ejb-jar.xml</code> file. For the Web container programming model, the <code>servlet-link</code> element is used in the <code>webservices.xml</code> and the name must match the name of the <code>servlet-name</code> stated in the <code>web.xml</code> file.
Handlers	The handlers associated with the Web service reference.

JAX-RPC mapping deployment descriptor

This deployment descriptor contains the mapping information between the WSDL definition and the Java interfaces. The server and client mapping descriptors are identical.

Refer to “JAX-RPC mapping deployment descriptor” on page 109 for a detailed explanation.

Roles

There are no new J2EE roles defined by the WSEE specification. All needed roles can be mapped to existing J2EE roles. The three roles used in WSEE are:

- ▶ **Developer**—Responsible for service definition, implementation, and packaging within J2EE modules
- ▶ **Assembler**—Responsible for assembling the modules into an application
- ▶ **Deployer**—Responsible for publishing the deployed services and resolving server references

Table 6-2 shows more detailed descriptions of the role responsibilities.

Table 6-2 Roles and responsibilities for Web service server development

	Developer	Assembler	Deployer
Service implementation	Implement the server. Can implement handlers.		
WSDL	Provide WSDL document.		Resolve endpoint addresses.
Web services DD	Provide port components definitions. Can define handler references.	Resolve module dependencies. Can configure handlers. Assemble modules in an EAR file.	
JAX-RPC mapping DD	Provide the mapping.		
Application server			Deploy the application.

Transactions

WSEE defines that a Web service does not execute under a global transaction. An existing client transaction will be suspended before the Web services port is accessed. See “WS-Transaction” on page 26.

Handlers

Handlers provide a mechanism for intercepting the SOAP message. The use of handlers is possible on the client and server side. Handlers are transport independent. A handler can examine and potentially modify the content of a SOAP message. Handlers can be used for logging, SOAP header processing, or caching. A limited form of encryption is also possible.

A handler can operate on an out-going and in-coming SOAP request. A handler can preprocess a message before it is routed to the business logic of a Web service or a client. A handler can also process a message before the message is sent to its receiver.

A handler is always associated with a particular port component, which is defined in the WSEE deployment descriptor. Multiple handlers are processed in a strict order, called a *HandlerChain*. The handler execution order is defined in the WSEE deployment descriptor.

To be able to do application-specific SOAP header processing, the client and server must agree on the specific SOAP header semantics.

Starting with WSEE 1.1, handler support for EJB Web services is defined.

A handler's life cycle is controlled by the container and, thus, runs under the execution context of the application. Handlers have access to the resources and environment variables. Handlers cannot divert a message and cannot modify the WSDL operation and the part types associated with that operation.

Figure 6-8 shows the handler life cycle. The container uses the `init` and the `destroy` methods to notify a handler instance of a change in its state. After the container has called the `init` method, the handler is ready for dispatching a request using the `handleRequest`, `handleResponse`, and `handleFault` methods.

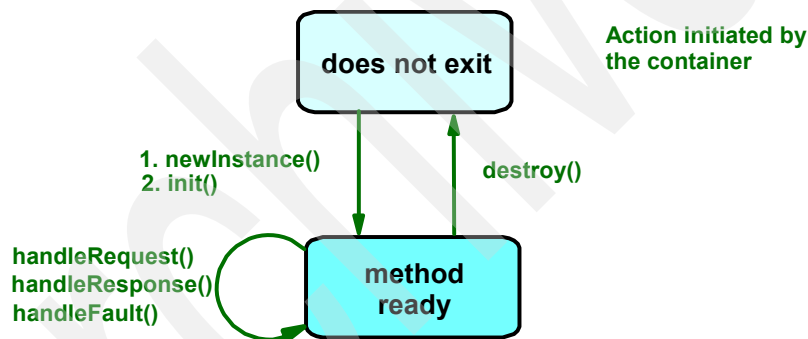


Figure 6-8 Life cycle of a handler

Security

The security specification defined by WSEE is not based on the WS-Security specification. The WSEE specification defines following security sections:

- ▶ Authentication—Defined by J2EE
 - Basic authentication, using user ID and password
 - Symmetric HTTPS, using digital certificates for requestor and server
- ▶ Authorization—Defined by J2EE
 - Securing the HTTP POST access of the JAX-RPC service endpoint

- Integrity and confidentiality—Relies on the HTTPS protocol
 - Secure communication by using SSL

Web services security, defined by the WS-Security standard, is described in Chapter 9, “Web services security” on page 173.

WSEE implementations in WebSphere

WebSphere Application Server 6.0 fully supports the Web Services for J2EE 1.1 specification. Both SOAP over HTTP and SOAP over JMS are supported.

SOAP over HTTP

Figure 6-9 shows the implementation of SOAP over HTTP:

- The client request to the Java proxy is handled by the SOAP client and is routed to the server over HTTP.
- In the server, the WebSphere SOAP engine calls a JavaBean Web service as a servlet, or uses a servlet in a Web router module to invoke an EJB Web service.

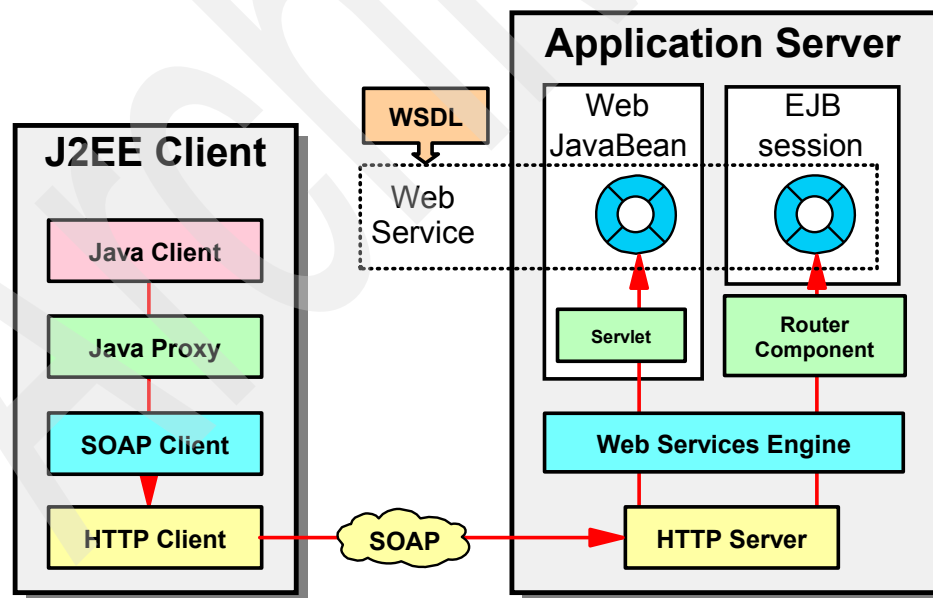


Figure 6-9 WSEE with SOAP over HTTP

SOAP over JMS

Figure 6-10 shows the implementation of SOAP over JMS:

- ▶ The client request to the Java proxy is handled by the SOAP client and is placed into a JMS queue through a JMS sender.
- ▶ In the server, a message-driven EJB (MDB) listens to the JMS queue and routes the message to the WebSphere SOAP engine.
- ▶ The WebSphere Web services engine invokes an EJB Web service.
- ▶ Optionally, the server replies to the client using a dynamic queue.

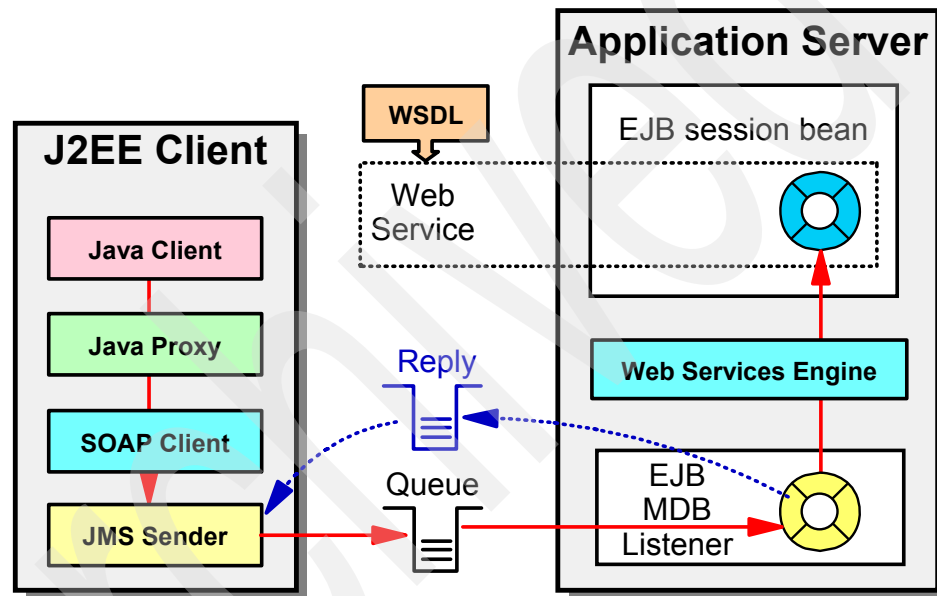


Figure 6-10 WSEE with SOAP over JMS

Summary

In this chapter, we presented the Web services for J2EE specification. This specification standardizes the deployment of Web services and Web services clients in a Java 2 Platform, Enterprise Edition (J2EE).

We reviewed the concepts, interfaces, and file formats introduced by the specification. We introduced the requirements to create a Web service client, analyzing its different phases, deployment characteristics, life cycle, and roles.

We introduced the deployment of Web service in a similar way and discussed the two runtime possibilities developed by the specification:

- ▶ EJB container
- ▶ Web container

Finally, we introduced the handler concept and showed the marginal Web services security specifications defined in WSEE.

Because WSEE is part of J2EE 1.4, it will be adopted and implemented by application server providers. We believe that the process of migration and deployment will become as good as we have today with other J2EE components.

More information

The best source for more information is the Web Services for J2EE specification, JSR 921, available at:

<http://www.jcp.org/en/jsr/detail?id=921>

The first version of Web services for J2EE specification, JSR 109, can be downloaded here:

<http://www.jcp.org/en/jsr/detail?id=109>

There is also a foundation article to help you understand the WSEE specification, *Build interoperable Web services with JSR-109*, available at:

<http://www.ibm.com/developerworks/webservices/library/ws-jsrart/>

Introduction to UDDI

This chapter provides an introduction to Universal Description, Discovery, and Integration (UDDI), as well as some advanced topics. We describe UDDI Version 2.0.4 and Version 3.0.

Version 2 of UDDI is implemented by the major UDDI registries currently in production. Both UDDI V2 and UDDI V3 are supported by the Web services tooling in IBM Rational Application Developer Version 6.0, and a private UDDI V3¹ registry implementation is shipped with Rational Application Developer V6.0 and IBM WebSphere Application Server Version 6.0.

In this chapter, we discuss the following topics:

- ▶ UDDI overview
- ▶ New features in UDDI Version 3
- ▶ UDDI repositories on the Web
- ▶ Web front ends for registries
- ▶ Java API for dynamic UDDI interactions
- ▶ Private UDDI registries

Note: The example use the UDDI4J API, which is deprecated. There was not time to redo the samples using the IBM UDDI Version 3 Client for Java.

¹ Backward-compatible with UDDI V1 and V2 SOAP Inquiry and Publish APIs.

UDDI overview

UDDI stands for Universal Description, Discovery, and Integration, and is the name for a specification that defines a way to store and retrieve information about a business and its technical interfaces, in our case, Web services. One implementation of this specification is the UDDI Business Registry, or UBR. This is a group of Web-based UDDI nodes, which together form a UDDI registry. These nodes are run on separate sites by several companies and can be used by anyone who wants to make information available about a business or entity, as well as anyone who wants to find that information.

A UDDI registry makes it possible to discover what technical programming interfaces are provided for interacting with a business for such purposes as electronic commerce or information retrieval.

UDDI addresses a number of business problems. First, it helps broaden and simplify business-to-business (B2B) interaction. For the manufacturer who needs to create many relationships with different customers, each with its own set of standards and protocols, UDDI provides a highly flexible description of services using virtually any interface. The specifications allow the efficient and simple discovery of a business and the services it offers by publishing them in the registry.

UDDI is based on existing standards, such as XML and SOAP. It is a technical discovery layer. It defines:

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

In fact, UDDI is a search engine for application clients rather than human beings; however, many implementations provide a browser interface for human users.

The central source of information about UDDI is the following Web site:

<http://www.uddi.org>

This site is operated by OASIS, which is a not-for-profit, global consortium that drives the development, convergence, and adoption of e-business standards.

Static versus dynamic Web services

You often read about *static* and *dynamic* Web services. Static Web services mean that the service provider and the service requestor know about each other at design time. There is a WSDL document that was found in a UDDI registry, or, more often, directly sent to the client application developer by the provider for

further use in the development tool. During runtime, it is very clear (mostly hard coded) what the URL (access point) of the partner is.

Dynamic Web services describe the fact that at design and development time the client does not know the explicit server and business entity where it will invoke a service. The client only knows an interface to call and finds one or more concrete providers for that kind of service through exploring UDDI registries at runtime.

UDDI registry structure

There are several types of information that have to be stored in such a registry, including information about the company, in UDDI called a business entity, that offers services and the description of each service including interface specifications and pointers to servers where those services can be invoked.

The data to be stored can be divided into six types of information that build up the data model of the registry:

- ▶ **Business entity**—The list of business entities is similar to the white and yellow pages. A business entity describes a company or organization. Those entities provide information such as business name, contacts, descriptions, identifiers, and categorization.
- ▶ **Business service**—This is non-technical information about a service that is provided. A business service is a descriptive container used to group a set of Web services related to a business process or group of services. In addition, categorization is available on this level. A business service maps to a WSDL service.
- ▶ **Binding template**—This contains service access information. These are the technical Web service descriptions relevant for application developers who want to find and invoke a Web service. In many cases, a binding template points to an implementation address (for example, a URL) and maps to a WSDL port. This entity is sometimes also called an *access locator*.
- ▶ **tModel**—A *tModel* (technical model) is a technical fingerprint holding metadata about type specifications and categorization information. Its attributes are key, name, optional description, and URL. The simplest tModel contains some text characterizing a service.

One type of tModel is sometimes referred to as a *service type*. In this case, the tModel points to a service description document (for example, through a URL). The type specification itself, which can be a WSDL document or any other formal specification, is not part of the UDDI model.

- **Taxonomy**—A taxonomy is a scheme for categorization. There is a set of standard taxonomies, such as the North American Industry Classification System (NAICS) and the Universal Standard Products and Services Classification (UNSPSC). In specific circumstances, you can publish your own taxonomies, but this is typically not possible using the Web client or publishing APIs because it needs special authorization and actions by the UDDI registry operator. The IBM private registry and the Web Services Explorer support user-defined custom taxonomies in addition to the standard NAICS, UNSPSC, and ISO-3166 taxonomies.

UDDI Version 3 uses the term *value set* for taxonomy.

- **Publisher assertions**—These are also called *business relationships*. There are different kinds of relationships: Parent-child, peer-peer, and identity. This makes it possible to model complex businesses, such as subsidiaries, external business partners, or internal divisions.

Figure 7-1 displays this data model with the entities previously introduced. It also shows their relationships and the link to the WSDL documents.

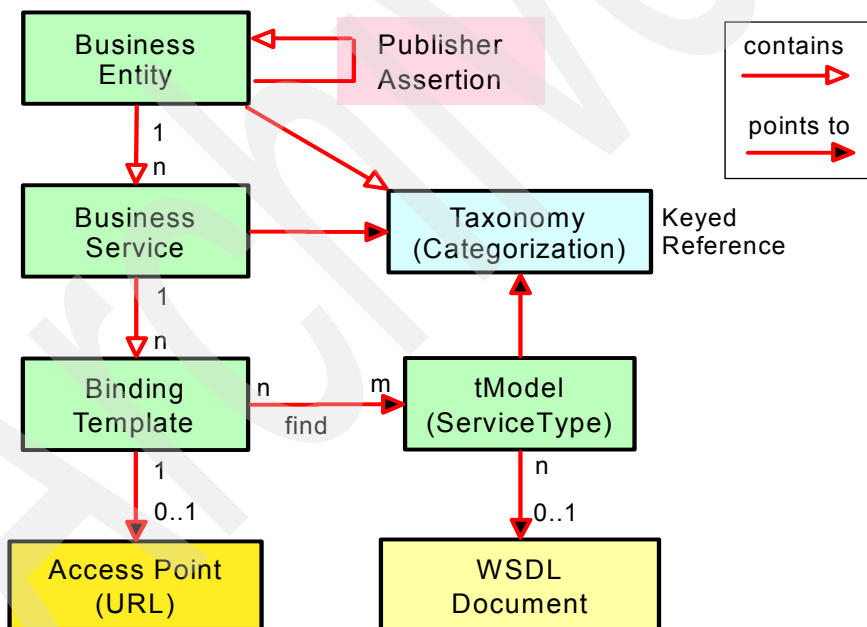


Figure 7-1 UDDI entities and their relationships

The business entity tops the containment hierarchy, containing one or more business service entities. Those services, in turn, contain binding template entities. The tModel instances (service types) are not contained by the business entity, but referenced by the binding template entities.

A binding template points to an access point URL where the service can be invoked. This is a common use of the binding template, but not required. The binding template could point to a phone number as a contact point.

A service type entity (tModel) has a reference to a Web service type specification, which typically is a WSDL document. Note that the UDDI registry merely contains a URL pointing to the Web site of the service provider where the document can be found, not the WSDL document itself.

Businesses, services, and tModels can contain zero to many references to taxonomy entries to categorize their content.

WSDL references are implemented as URLs; therefore, any other specification format can easily be supported as well. UDDI is not bound to WSDL. In general, you do not work with WSDL files at all when working with UDDI registries.

There is an m:m relationship between the binding template and the tModel. Keep in mind that a tModel is just a technical fingerprint containing quite abstract metadata. Even if a service points to several tModels, it does not necessarily have to implement multiple interfaces on a technical level.

The UDDI data model is designed to be flexible. Therefore, there can be more than one technical description for one service (a formal specification and a supporting tutorial, for example), and vice versa, one tModel can be used to describe several similar business services.

The possibility for the requestor to dynamically bind to a provided service through a given tModel is a real benefit of the Web service architecture.

Interactions with UDDI

Using UDDI registries in general contains three different tasks, because on one side there are service providers who want to publish some information, and on the other side there are service requestors who want to find some services and finally use them. So, the tasks using a UDDI registry are:

- ▶ Publishing information
- ▶ Finding information
- ▶ Using the obtained information

Figure 7-2 illustrates these typical steps of interacting with a UDDI registry.

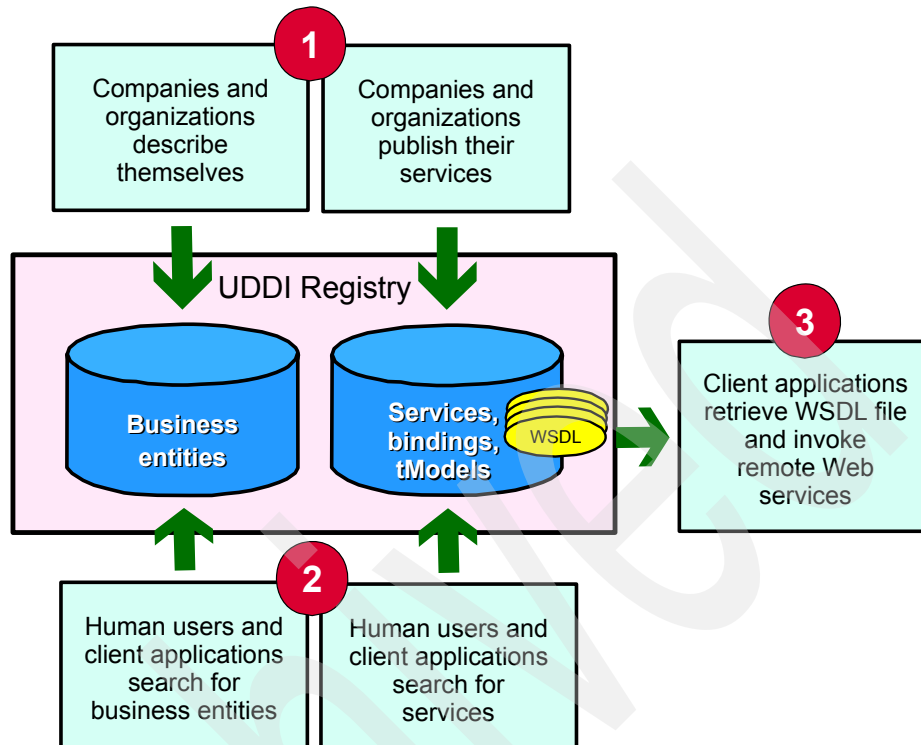


Figure 7-2 Interactions with UDDI registries

Publishing information

There are the six types of information that can be published to UDDI registries:

- ▶ Business entity information
- ▶ Business service information
- ▶ Binding templates
- ▶ tModels
- ▶ Taxonomy information
- ▶ Publisher assertions (business relationships)

Finding information

After business entities have published their entity descriptions and services, clients (service requestors) might try to find this information.

Clients have a number of possible ways to explore a registry. Humans do that by using HTML clients of the registry, and applications use UDDI APIs to do that automatically, as described in “Java APIs for dynamic UDDI interactions” on page 141.

In any case, the most likely strategy will be one of the following:

- ▶ Find concrete business entities and explore their services.
- ▶ Find services of a certain type regardless of which entity provides them.

Using the information

After finding the services, the final step is to use the service that was obtained by the exploration step. This is typically done by accessing the service provider using the interface (messages and parameters) found in the downloaded WSDL file and the access point information (URL). We describe this step in detail in the second part of this book.

New features in UDDI Version 3

At the time this book was written, UDDI Version 3.0.1 has been published as an OASIS standard.

The enhancements specified by this new version include multi-registry topologies, increased security features, improved WSDL support, a new subscription API, and core information model advances.

Note that some of the features in the UDDI Version 3 specification are optional, and an implementor can choose not to implement them.

Keys assigned by publisher

Every item in a Version 2 UDDI registry, such as business entities, services, and tModels is assigned a key (UUID), which uniquely identifies that entity within the registry. In Version 2, copying a UDDI entry from one registry to another and preserving the unique key is not allowed.

Version 3 of UDDI enables you to copy entries using the same key, because the key generation feature is extended. The behavior of copying entities is known as *entity promotion*. The solution to the problem is that the generation of entity keys is no longer exclusively performed by registry nodes. Depending on policies, publishers can request permission to use a partition of the registry's root keyspace (using a `tModel:keyGenerator` request). If this request is successful, publishers can publish entities by supplying an entity key within the partition they own. If the publisher supplies a key—if it is allowed by policy—the supplied entity key is used instead of the node generating a new key.

Human-friendly URI-based keys

Besides the new ability to promote keys across registries, a new format for UDDI keys is introduced in Version 3. Prior versions mandated that keys had to be a formatted universally unique identifier (UUID). Version 3 removes this restriction and recommends the usage of a key scheme based on DNS names. This enables publishers to establish a key partition of the registry's root key space and then generate keys based on that partition. For example, a valid Version 3 key might look like this:

```
uddi:mycompany.com:4711  
uddi:ibm.com:itso:0815
```

These human-friendly keys enable organizations to manage their own key space using internally established conventions and structures.

Complex registry topologies

In order to support these more complex topologies of UDDI registries, there are new possibilities for setting a UDDI registration in certain relationships to another. UDDI Version 3 introduces the notions of *root* and *affiliate* registries as part of its guidance on inter-registry associations. The existence of a root registry enables its affiliates to share data with the root registry and among themselves with the assurance that keys remain unique.

Advanced security features

Another advancement is the support for digital signatures. Entities can be digitally signed to provide better data integrity and authenticity. When exploring the registry, you can filter for signed entries to be sure you get the originally published information that came from a trusted source. These digital signatures are also kept when promoting entities to different registries.

Policies

There are many different scenarios in which a UDDI registry can be used:

- ▶ Public registry on the Internet
- ▶ Partly public registry in the extranet
- ▶ Private registry inside the intranet
- ▶ Private registry for development (for example, inside Application Developer)
- ▶ Stable private registry for test areas
- ▶ Scalable registry for production environments

Each of these registries requires slightly different behavior because of its intended use, so there are a set of *policies* that can be applied to the registry that changes the actual behavior regarding:

- ▶ Authorization models
- ▶ Data custody and confidentiality
- ▶ Key generation
- ▶ Value set validation
- ▶ Subscription
- ▶ User publication limits
- ▶ Audit policy

Data model updates

Version 3 also introduces a set of improvements in the data model, which defines how the entities are stored in the registry. Some of the most important extensions are:

- ▶ Categorization of binding templates.
- ▶ More complex categorization features such as derivation of taxonomies.
- ▶ XML Schemas define more precisely the possible values stored in UDDI.
- ▶ Advanced support for internationalization.

Extended inquiry API

The programmatic exploration was extended to support more complex queries by nesting queries into one round-trip call. Wildcards can be used more flexibly, and special result set features allow processing of very large query results. In addition, more find qualifiers are supported by the API.

Subscription API

The new subscription API includes robust support for notification of changes in the registry. Users can establish a subscription based on a specific query or set of entities in which the user is interested. This makes it possible to be notified of new businesses or services that are registered and to monitor existing businesses or services.

Registry management

To be more independent of the external taxonomy provider's server availability and to improve performance, the UDDI registry contains some caching mechanisms for external taxonomies. Also, replication between UDDI registries is improved.

UDDI support in WebSphere Application Server

A UDDI registry that supports Version 3 of the standard runs on WebSphere Application Server Version 6.0 and is shipped with both WebSphere Application Server and WebSphere Application Server Network Deployment.

Advanced features of UDDI

In this section, we describe some of the advanced functions of UDDI.

Modeling features for complex business entities

This feature enables companies that might be spread over several countries to present themselves as many business entities that are related to each other. There are different kinds of relationships: parent-child, peer-peer, and identity. This makes it possible to model relationships such as subsidiaries, external business partners, or internal divisions.

Also, the inquiry API offers functions, such as `find_relatedBusinesses`, to allow searches for related businesses of one company.

The feature called *service projection* enables one business entity to reference a service that another (related) business entity offers. Therefore, it is possible to define a service that is offered by, for example, more than one department of a large company. The creator of a projected service is not allowed to alter anything in that service, but to the service requestor it looks as though the service is owned by that business entity.

External taxonomies

Originally, there were only three taxonomies that could be used to categorize services: NAICS (North American Industry Classification System), UNSPSC (Universal Standard Products and Services Classification), and geographic taxonomies. These categorization standards were checked internally in the UDDI registry to avoid the setting of invalid codes.

In Version 2 of the standard, there is the possibility for companies to specify their own external taxonomy, which is not checked internally by the UDDI server. The provider of the taxonomy must also offer a standardized Web service to validate values against.

The use of external taxonomies is a controlled process, so the UDDI Business Registry operator has to approve it.

Powerful inquiry

The inquiry API enables the dynamic exploration of the UDDI registry, especially the search for services. This API was extended to support more powerful features dealing with more complex query requirements.

Combining categories

The `combineCategoryBags` qualifier enables you to group all of the taxonomy data associated with a business and all of its contained services (including any service projections) into a single collection that the search is performed against. This is useful because it reduces the steps in finding a business of interest by looking in both the business and its constituent services at the same time. So, you could, for example, in one step look up services that are categorized both as Construction, as per NAICS, and contain a geographic locator such as AT, stating that it is an Austrian company.

Advanced search using categorization

Similarly, the `serviceSubset` filter enables you to search for businesses using taxonomy criteria, which are tested only against the categorizations associated with a business' constituent services. Categorizations associated with the business itself are not included in the search.

Qualifier for searching

Finally, the `orLikeKeys` qualifier is particularly useful because it enables pseudo-complex queries. It enables you to combine search criteria with *OR* and not only with *AND*. For example, you can find businesses located in the United States, Mexico, or Canada that also offer cold storage or generic shipping services. This enables you to search for businesses that are categorized with different levels of specificity, while at the same time allowing a single inquiry to reference multiple different taxonomies.

Internationalization features

Services and business entities can have language-dependent names. There must be at least one name in any language, but you can also provide different translations for both business entities and your services.

A new type of taxonomy, `postalAddress`, enables you to specify addresses in different international formats.

Peer-based replication

As the number of UDDI registries increases, a simple file-based replication scheme is not appropriate anymore. The new peer-based replication features do not require that each UDDI registry has to communicate with all the others for replicating the information.

UDDI business registries on the Web

Currently, there are four organizations hosting nodes in the UDDI Business Registry. All four of the nodes are replicated and therefore contain the same information. Most of the companies also host a test registry in addition to the business registry. Those test instances can be used by anyone for test purposes. Web services found there are often subject to change and lack of availability. They are not meant for production use. Most of the companies also host V3 Beta test instances.

Table 7-1 specifies three URLs for each registry.² The first is the URL for the Web client that can be used by humans for exploring and publishing any information. Typically, you have unlimited access for reading information and need some sort of registration for publishing information.

Besides the Web client URL, you also find URLs for API access using a UDDI API implementation, such as UDDI4J, as described in “Java APIs for dynamic UDDI interactions” on page 141. There are always two different URLs for inquiry (reading information) and publishing (writing information). The latter one uses HTTPS as a secure connection to ensure that the publishing information is not corrupted or altered.

A current list of public UDDI Business Registry nodes are available online at:

<http://www.uddi.org/find.html>

Table 7-1 UDDI registries

Hosting organization	Access type	URL
IBM Business Registry	Web	http://uddi.ibm.com
	Inquiry	http://uddi.ibm.com/ubr/inquiryapi
	Publish	https://uddi.ibm.com/ubr/publishapi

² IBM also hosts a V3 beta registry (see <http://uddi.ibm.com> for more information).

Hosting organization	Access type	URL
IBM Test Registry	Web	http://uddi.ibm.com/testregistry/registry.html
	Inquiry	http://uddi.ibm.com/testregistry/inquiryapi
	Publish	https://uddi.ibm.com/testregistry/publishapi
IBM V3 Beta Test Registry	Web	http://uddi.ibm.com/beta/registry.html
	Inquiry	http://uddi.ibm.com/beta/v3inquiryapi
	Publish	https://uddi.ibm.com/beta/v3publishapi
	Security	https://uddi.ibm.com/beta/v3securityapi
	Custody	https://uddi.ibm.com/beta/v3custodyapi
Microsoft Business Registry	Web	http://uddi.microsoft.com/
	Inquiry	http://uddi.microsoft.com/inquire
	Publish	https://uddi.microsoft.com/publish
Microsoft Test Registry	Web	http://test.uddi.microsoft.com/
	Inquiry	http://test.uddi.microsoft.com/inquire
	Publish	https://test.uddi.microsoft.com/publish
Microsoft V3 Beta Test Registry	Web	http://uddi.beta.microsoft.com
	Inquiry	http://uddi.beta.microsoft.com/inquire
	Publish	https://uddi.beta.microsoft.com/publish
SAP Business Registry	Web	http://uddi.sap.com/
	Inquiry	http://uddi.sap.com/uddi/api/inquiry
	Publish	https://uddi.sap.com/uddi/api/publish
SAP Test Registry	Web	http://udditest.sap.com/
	Inquiry	http://udditest.sap.com/UDDI/api/inquiry
	Publish	https://udditest.sap.com/UDDI/api/publish
SAP V3 Beta Test Registry	Web	http://udditest.sap.com
	Inquiry	http://udditest.sap.com/UDDI/api/inquiry
	Publish	https://udditest.sap.com/UDDI/api/publish

Hosting organization	Access type	URL
NTT Business Registry	Web	http://www.ntt.com/uddi/
	Inquiry	http://www.uddi.ne.jp/ubr/inquiryapi
	Publish	https://www.uddi.ne.jp/ubr/publishapi

In general, all of the UDDI registry nodes have the same functionality, although there are significant differences in user interface and usability. Functions that are typically performed by users are:

- ▶ Find businesses, services, and tModels:
 - By name
 - By categorization
 - By key (UUID)
- ▶ Browse through businesses, services, and tModels:
 - Show services for businesses
 - Show binding informations for services
 - Show tModels for services
- ▶ Publish, change, and unpublish:
 - Business entities
 - Business relationships
 - Services
 - tModels

Web front ends for registries

Each of the UDDI registries previously listed has its own Web interface for human users. They have very different user interfaces with a different look and feel, and therefore, the usability varies. The functionality itself is the same for all of them, because they are all implementing the UDDI Version 2 features.

As an example, we introduce the Web client of the IBM UDDI Registry.

When accessing the site using <http://uddi.ibm.com>, you reach a page with several links to articles regarding UDDI. You can choose between the business registry, the V2 test registry and the V3 Beta test registry using the linked images on the right. After you have chosen the UDDI registry, you will find the tasks on the left-side menu, for example, *Find* and *Register*.

Note: A *reading* task, for example, finding businesses or services, does not require a registration. Whenever you want to *publish* some information about your business, service, or interfaces, you have to register. This is, in general, free and requires just your name, country of residence, and a valid e-mail address.

Finding information

If you want to find published UDDI information, you can choose between a set of finders, depending on what you are searching on, as shown in Figure 7-3. The simple approach is just to specify the type of entry you are looking for and a string that is contained in the name of that entry.

For more advanced searches, you can use one of the links named *Find a Business/Service/Technical Model*, which enables you to use categorization information and combinations of search criteria.

The screenshot shows the 'UDDI Find' interface. It has two main sections: 'Simple Search' and 'Advanced Search'. The 'Simple Search' section has a description: 'Simple Search allows you to search by NAME for a Business, Service, or Technical Model. You may use the '%' symbol as a wildcard that matches any character.' Below this, there is a 'Search For a' dropdown menu with 'Technical Model' selected, a 'Starting with' text input field, and a 'Find' button. The 'Advanced Search' section has a description: 'Advanced Search allows you to perform complex searches using multiple criteria as well as override default search behaviors.' Below this, there are three links: 'Find a Business', 'Find a Service', and 'Find a Technical Model'.

Figure 7-3 Options for searching the UDDI registry

After you have successfully submitted a search query, you can navigate through the results, such as displaying services that were published by a specific business, or showing services that implement a certain tModel specification.

Important: Unfortunately, there are a large number of experimental and test entries not only in the test registry, but also in the official business registry. So when locating services, be sure that you do not try to call a service with *access points* showing server names such as *localhost*, and be aware that there is no guarantee about availability of the services.

Figure 7-4 shows an example of a service found in UDDI with detailed information that might be interesting to the client of the service.

Service Details

The details of the selected service are shown below. Please use your browser's **Back** button to return to the previous page OR Press the **New Search** button to search again.

Service Information			
Key			
D8D76290-CA56-11D5-A64A-002035229C64			
Owning Business		Owner Key	
Abilizer		4B99A0C0-7FC2-11D5-91CE-002035229C64	
Service Name(s)			
Name			Language
Stock Quotes			en
Service Description(s)			
Description			Language
Obtain stock quotes.			en
Access Point(s)			
Protocol	Address	Description	Actions
http	http://services.xmethods.net/soap/urn:xmethods-delayed-quotes.wsdl	WSDL	Details
Service Locator(s)			
Code	Description	Type	
namespace	Namespace	UDDITYPE	
categorization	Categorization (taxonomy)	UDDITYPE	

Figure 7-4 Exploring details of a published service

First of all, there is a unique ID (UUID) specifying the service, as well as the business owning the service, including its key. You could also use these keys in the *advanced search* feature.

Names and descriptions of services can be specified in several languages. You will find the entries of all languages in the details view, including an ISO code of the language used. Two essential pieces of information are the *access points* and *service locators*. The first one is a pointer to concrete implementations of this service. The latter one describes the service using categorizations (taxonomies).

Note: The UDDI registry does not contain only Web services. For example, IBM also publishes other services, such as phone numbers and URLs for Web sites, where to order IBM products, and e-mail addresses.

After you have identified a Web service you want to use, you can follow the tModel link to download the WSDL file. This WSDL file includes all the necessary interface descriptions that can be used by tools such as Rational Application Developer to develop client applications that call the Web service statically. The binding template link provides the URL where you invoke the Web service.

Publishing information

The second task you might want to do using UDDI registries is to publish your company's information and services so that potential clients can use your services.

After you have selected the desired registry and have successfully logged in, you find an additional menu item on the left side called *Publish*. By clicking that, you reach a window with all the items that you ever published into the registry using this user name.

Publishing sequence

The typical order in which you publish your information is the following:

1. Define a business.

Describe your company or organization, and optionally assign some categorizations, describe your business domain, and add relationships to other business entities.

2. Define tModel or tModels.

Define technical models, referencing WSDL files for Web services or simpler tModels such as *mailto* or *ftp*. Note that a tModel does not always reference a WSDL file on a provider's site. tModels contain a unique key, a description, and classification locators. If a business provides public access to a WSDL file, it makes sense to specify that URL in the tModel description.

3. Define services and bindings.

Finally define your services. They consist of a name, key, and some descriptive information. You also immediately provide the binding information, which in general is composed of an access point, where the service can be activated, and binding details such as the protocols used to connect (for example, SOAP over HTTP). You also have to reference one or more tModels to describe your interface.

The path through the different steps is quite straightforward and not described in more detail here.

Accessing your services

There is an interesting fact to point out when accessing your information in the registry:

- ▶ When looking at the publishing window, you will immediately find your business entities, relationships, and tModels, including all the links to create new ones.
- ▶ You do not initially see the business services. This is because a service (other than tModels) can only exist for a specific business entity.
- ▶ Therefore, you have to first expand the business entry to see the definitions and links regarding services (Figure 7-5).

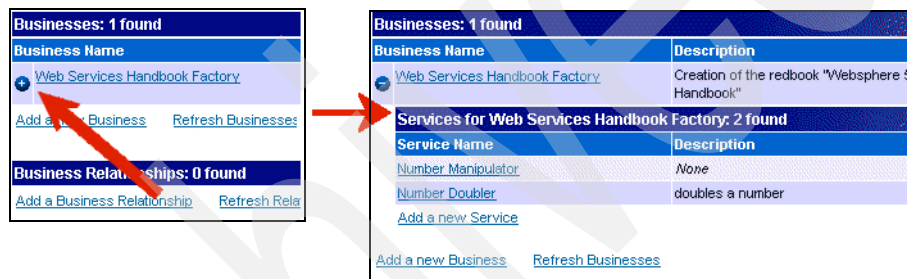


Figure 7-5 Expanding business entities to explore or define services

When you have finished defining your entities, services, and tModels, your publishing window looks like the one shown in Figure 7-6.

The screenshot displays the IBM UDDI Business Registry Version 2 web interface. The header includes the IBM logo, a search bar, and navigation links: Home, Products & services, Support & downloads, and My account. The breadcrumb trail indicates the path: IBM Corporation > Services/UDDI > Publish.

The main content area is titled "UDDI Business Registry Version 2" with the subtitle "Universal Description, Discovery, and Integration". It welcomes "Michael Schramm" and shows "Businesses: 1 found".

Business Name	Description	Actions
Web Services Handbook Factory	Creation of the redbook "Websphere 5 Web Services Handbook"	Edit Delete

Below the business list, it shows "Services for Web Services Handbook Factory: 2 found".

Service Name	Description	Actions
Number Manipulator	None	Edit Delete
Number Doubler	doubles a number	Edit Delete

There are links to "Add a new Service", "Add a new Business", and "Refresh Businesses".

It also shows "Business Relationships: 0 found" with links to "Add a Business Relationship" and "Refresh Relationships".

Finally, it shows "Technical Models: 2 found".

Technical Model Name	Description	Actions
tModel for doubling numbers	double a number	Edit Delete
http://michitest.wsdl/numberdoublerbinding/	None	Edit Delete

Links for "Add a new Technical Model" and "Refresh Models" are provided. At the bottom, there are links for "Help", "Terms of Use", and "Support". The footer contains links for "About IBM", "Privacy", "Legal", and "Contact".

Figure 7-6 Overview of published items

Java APIs for dynamic UDDI interactions

As previously described, it is possible to use UDDI repositories as a human user by using Web front ends. A second possibility is to explore them using client applications. Therefore, there is a standardized set of APIs defined for each version of UDDI registries.

There are many implementations of that API. For example, Microsoft provides a UDDI SDK that is a collection of client development components, sample code, and reference documentation that enables developers to interact programmatically with UDDI-compliant servers. This is available for Visual Studio .NET and separately for Visual Studio 6.0 or any other COM-based development environment.

The IBM UDDI Version 3 Client for Java is the preferred API for accessing UDDI using Java.

Another Java API that can be used to access UDDI repositories is UDDI4J. A number of companies, such as IBM, HP, and SAP, officially support the UDDI4J implementation.³ The following code examples use UDDI4J.

UDDI4J overview

UDDI4J is a Java class library that provides support for an API that can be used to interact with a UDDI registry. This class library provides classes that can be used to generate and parse messages sent to and received from a UDDI server.

Note: The UDDI4J API can be used against any of the registries (business registry, test registry, private UDDI registry).

The central class in this set of APIs is `org.uddi4j.client.UDDIProxy`. It is a proxy for the UDDI server that is accessed from client code. You can use this proxy to interact with the server and possibly find and download WSDL files. You might then want to use such APIs as WSDL4J or Apache Axis to interpret these WSDLs and invoke remote Web services. This section describes the first part, which is exploring and publishing to the UDDI.

Detailed information and download possibilities (including source code) can be found at:

<http://www.uddi4j.org>

The important packages are:

- ▶ `org.uddi4j.datatype`
This package contains classes used to send or receive UDDI information.
- ▶ `org.uddi4j.request`
This package contains messages sent to the server. These classes are typically not used directly; rather the `UDDIProxy` class uses these classes.
- ▶ `org.uddi4j.response`
This package represents response messages from a UDDI server.

³ Support for UDDI4J has been deprecated in IBM WebSphere UDDI Registry V3.

Prerequisites

Clients written using UDDI4J require a Java runtime environment of at least Version 1.2.2 and a SOAP transport, which at the moment can be one of the following:

- ▶ Apache Axis
- ▶ Apache SOAP 2.3
- ▶ HP SOAP

Using the library

You have to do some preparations to use the UDDI library. First, you have to specify a set of system properties where you define which SOAP implementation you are using and if there are proxies to cross.

If you not only want to explore a UDDI server but also want to publish information, you have to activate the HTTPS transport protocol by adding an implementation of JSSE to your class path. Refer to the documentation on the UDDI4J Web site for detailed information.

Writing UDDI clients

A typical application client to UDDI registries consists of three parts:

- ▶ Connect to the server by creating a proxy object.
- ▶ Find entities using this proxy.
- ▶ Publish entities using this proxy.

Creating a proxy object

When writing a UDDI client, you typically first create a `UDDIProxy` object by setting the server URLs to access, as shown in Figure 7-7.

The concrete values for the `inquiryURL` and `publishURL` are dependent on the UDDI registry you use and can be obtained from Table 7-1 on page 134.

```
// --- Add SSL support (only needed for publishing)
System.setProperty("java.protocol.handler.pkgs",
    "com.ibm.net.ssl.internal.www.protocol");
java.security.Security.addProvider(new com.ibm.jsse.JSSEProvider());

// --- Create UDDI proxy
UDDIProxy uddiProxy = new UDDIProxy();
uddiProxy.setInquiryURL(inquiryURL);
uddiProxy.setPublishURL(publishURL);

// --- Cache authorization token for publishing
AuthToken token = proxy.get_authToken("userid", "password")
```

Figure 7-7 Creating a UDDI proxy object

Finding information

After you have created a proxy object, you can easily find any kind of information, such as business entities, services, bindings, or tModels.

Finding business entities

The `find_business` method is used to find business entities by name. The method returns a `BusinessList` object, a collection that can be used to find specific information about all of the businesses that match the search parameter.

```
BusinessList UDDIProxy.find_business(
    java.util.Vector names,
    DiscoveryURLs discoveryURLs,
    IdentifierBag identifierBag,
    CategoryBag categoryBag,
    TModelBag tModelBag,
    FindQualifiers findQualifiers,
    int maxRows)
```

Parameters:

- ▶ `names`—A vector of names. You can use the % character as a wildcard.
- ▶ `discoveryURLs`—This is a list of URLs to be matched against the data associated with the `discoveryURLs` contents of registered business entity information. The returned `BusinessList` contains `BusinessInfo` structures matching any of the URLs passed (logical OR).
- ▶ `identifierBag`—This is a list of business identifier references. The result contains businesses matching any of the identifiers passed (logical OR).
- ▶ `categoryBag`—This is a list of category references. The result contains businesses matching all of the categories passed (logical AND).

- ▶ **tModelBag**—The registered business entity data contains binding templates that, in turn, contain specific tModel references. The tModelBag argument lets you search for businesses that have bindings that are compatible with a specific tModel pattern. The result contains businesses matching all of the tModel keys passed (logical AND). tModel key values must be formatted as URN-qualified UUID values prefixed with “uuid:”.
- ▶ **findQualifiers**—This can be used to alter the default behavior of search functionality.
- ▶ **maxRows**—This allows the requesting program to limit the number of results returned.

Finding services

To find a service using whatever search criteria, you use the find_service method:

```
ServiceList UDDIProxy.find_service(
    java.lang.String businessKey,
    java.util.Vector names,
    CategoryBag categoryBag,
    TModelBag tModelBag,
    FindQualifiers findQualifiers,
    int maxRows)
```

This function returns a ServiceList on success. In the event that no matches were located for the specified criteria, the ServiceList structure returned will contain an empty structure.

Parameters:

- ▶ **businessKey**—This optional uuid_key is used to specify a particular business entity instance. This argument can be used to specify an existing business entity in the registry or can be specified as "" (empty string) to indicate that all business entities are to be searched.
- ▶ **names**—This optional vector of names represents one or more partial names. A wildcard character % can be used to signify any number of any characters. Up to five name values can be specified. If multiple name values are passed, the match occurs on a logical OR basis within any names supplied.
- ▶ **categoryBag**—This is a list of category references. The returned ServiceList contains BusinessService structures matching all of the categories passed (logical AND by default).
- ▶ **tModelBag**—This is a list of tModel uuid_key values that represent the technical fingerprint of a BindingTemplate structure to find. If more than one tModel key is specified in this structure, only BusinessService structures that contain BindingTemplate structures with fingerprint information that matches *all* of the tModel keys specified will be returned (logical AND only).

- ▶ `findQualifiers`—This is used to alter the default behavior of search functionality.
- ▶ `maxRows`—This allows the requesting program to limit the number of results returned.

Finding tModels

To find tModels, use the `find_tModel` method. This method is for locating a list of tModel entries that match a set of specific criteria. The response will be a list of abbreviated information about tModels that match the criteria (`tModelList`).

```
TModelList UDDIProxy.find_tModel(
    java.lang.String name,
    CategoryBag categoryBag,
    IdentifierBag identifierBag,
    FindQualifiers findQualifiers,
    int maxRows)
```

Parameters:

- ▶ `name`—This string value represents a partial name. Because tModel data only has a single name, only a single name can be passed. A wildcard character (%) can be used. The returned `tModelList` contains `tModelInfo` elements for tModels whose name matches the value passed.
- ▶ `categoryBag`—This is a list of category references. The result contains tModels matching all of the categories passed (logical AND by default). `FindQualifiers` can be used to alter this logical AND behavior.
- ▶ `identifierBag`—This is a list of business identifier references. The result contains tModels matching any of the identifiers passed (logical OR).
- ▶ `findQualifiers`—This is used to alter the default behavior of search functionality.
- ▶ `maxRows`—This allows the requesting program to limit the number of results returned.

The three methods enable you to explore businesses, services, and tModels. As a typical dynamic client, you are not really interested in the name of a business or a service, but much more in the binding templates that tell you where you can access a service. A typical scenario would be to search for a binding using a tModel key, extract the access point, and invoke a service at that URL.

Finding bindings

You can find bindings using the `find_binding` method:

```
BindingDetail UDDIProxy.find_binding(  
    FindQualifiers findQualifiers,  
    java.lang.String serviceKey,  
    TModelBag tModelBag,  
    int maxRows)
```

This method returns a `BindingDetail` object that contains zero or more `BindingTemplate` structures matching the criteria specified in the argument list.

Parameters:

- ▶ `findQualifiers`—This collection can be used to alter the default behavior of search functionality.
- ▶ `serviceKey`—This is used to specify a particular instance of a `BusinessService` element in the registered data. Only bindings in the specific `BusinessService` data identified by the `serviceKey` passed will be searched.
- ▶ `tModelBag`—This is a list of `tModel` `uuid_key` values that represent the technical fingerprint to locate in a `BindingTemplate` structure contained within the `BusinessService` instance specified by the `serviceKey` value. If more than one `tModel` key is specified in this structure, only `BindingTemplate` information that exactly matches all of the `tModel` keys specified will be returned (logical AND).
- ▶ `maxRows`—This optional integer value allows the requesting program to limit the number of results returned.

This function returns a `BindingDetail` object on success. In the event that no matches were located for the specified criteria, the `BindingDetail` structure returned in the response will be empty and contain no `BindingTemplate` data.

In all the find methods just described, you can set the parameter `maxRows` to define the maximum number of results. There, you can also specify 0 in order to not limit the result. Keep in mind that, in the event of a large number of matches, an operator site might truncate the result set. If this occurs, the response message will contain the truncated attribute set to true.

Publishing information

If you want to register some information on a UDDI server, you have to use the authorization token obtained when creating the proxy.

Figure 7-8 shows how to publish a business entity. This example only fills the required name field. Of course in real scenarios, you might also consider adding categorization information and descriptions.

```
// create business entities and store them in a vector
Vector businessEntities = new Vector();
BusinessEntity be = new BusinessEntity("");
be.setName("IBM ITS0");
entities.addElement(be);
// save the entities on the UDDI server
BusinessDetail bd = proxy.save_business
(token.getAuthInfoString(), entities);
```

Figure 7-8 Publishing a business entity

The BusinessDetail object returned by the save method holds the unique identifiers (UUID) that were given to the new entities by the UDDI registry.

Similar to business entities, you can also create business services and tModels by using the classes BusinessService or TModel. Whenever you have to specify URLs or access points, you also find wrapper classes, such as OverviewURL or AccessPoint, that provide a very easy and intuitive way to model your registry entries.

For more details and examples, refer to the UDDI4J documentation and to several articles published at the library section of the developerWorks site:

<http://www.ibm.com/developerworks>

Private UDDI registries

The first implementation of UDDI was the *business registry*, which sometimes is also called the *UDDI cloud*. This cloud consists of nodes operated by IBM, Microsoft, SAP, and others that are replicated and hold exactly the same information. They can be accessed by everyone, both for exploring information and publishing information.

But in fact, there are other usage scenarios that seem to gain even more importance than the public registry, namely totally or partly private UDDI registries. There are a number of possible requirements that encourage companies and organizations to establish those UDDI registries outside the cloud.

Motivation for the use of private UDDI registries

There are a number of problems that arise in some environments when using public UDDI registries, especially the business registry.

Need for privacy

One requirement is that companies often do not want to show all of their interfaces to the whole world, and therefore, also open the possibility that everyone can (try to) communicate with their services.

Getting rid of UDDI pollution

Because the UDDI cloud is public and free to be used by everyone, it is obvious that there is often inaccurate, outdated, wrong, or misleading information in the registries. There is no concept of expiration dates for published information or any review mechanisms to ensure the quality of the content. This is a problem similar to Web sites on the Internet, with the main difference being that the users of Web sites are human and should be able to separate good or usable content from bad content. The clients of UDDI registries are often applications. This fact can lead to severe problems.

Standards and guidelines

In UDDI, you can publish businesses or services with very little information, and when entering URLs for access points, you can specify WSDL files, Web sites, or documents that describe a service in prose. This is allowable in UDDI and might be exactly what is required in some cases.

However, to make automatic (application client) inquiries (dynamic Web services) easier, you might want to set up some standards restricting the information that is set in specific places of the registry.

Possible scenarios for private UDDI registries

In this section, we list some scenarios where private registries are suitable.

Internal registry

A company or organization might want to consider a totally private registry, which resides inside the firewall and can only be accessed from inside the intranet, both programmatically and Web-based.

This scenario helps large organizations to provide internal services for other departments and divisions. It is very easy to mandate guidelines and restrictions, because no other company interacts with your registry.

e-marketplace UDDI registries

Another scenario could be a registry that is, in fact, totally public, but specializes in a very specific topic, or a group of companies, or an industry. Those registries would be perfect candidates for using specialized taxonomies, which have been supported since UDDI Version 2.

For example, the automotive industry might consider setting up a specialized registry with their own detailed categorization system.

Another advantage would be to restrict the usage to some special set of tModels. In order not to allow every company to publish their own service interfaces, the organization hosting the registry might want to consider defining standardized tModels and WSDLs that are supported by this registry.

Using those specialized registries would very much increase the possibility of automatically discovering and invoking Web services by application clients.

Extranet UDDI registries

A third scenario is a usage of UDDI where one company or organization hosts one registry for the communication between the owning company and the business partners, so the registry is on the Internet, but access is restricted to the owning company and partners with previously negotiated business contracts.

This enables a good compromise between the privacy of a company and the ability to communicate easily with its partners. All the advantages of private registries apply, keeping the registry clean and uniform.

Benefits of private UDDI registries

The usage scenarios of UDDI registrations are not restricted to the public business registry already discussed. There are many reasons why an organization might want to consider establishing a registry that is not part of the official registries. That makes it possible to restrict one or more of the following points:

- ▶ Who is allowed to explore the registry
- ▶ Who can publish information
- ▶ What kind of information is published (guidelines, standards)

Another important point about private registries is that the success rate for dynamic exploration performed by application clients increases dramatically.

Additional considerations for private UDDI registries

There are two considerations when thinking of any registry that is not part of the central business registry.

Propagation

Hosting a private registry does not necessarily mean that there is no propagation with the business registry. There might be scenarios where it makes sense to do one-way propagation from the private registry into the cloud. Especially in the e-marketplace scenario, this could make a lot of sense.

Securing APIs

In each case, you might consider enabling or disabling the inquiry and publishing APIs. So, in the partner scenario, it could be required that everyone is allowed to explore the Web services, but nobody would be able to publish a service except the hosting company itself.

WebSphere private UDDI registry

A private UDDI registry is shipped with WebSphere Application Server and can be installed on a WebSphere server.

Important: Installation and usage of the private UDDI registry is described in Chapter 23, “Implementing a private UDDI registry” on page 619:

- ▶ Installing the registry in a WebSphere Application Server
- ▶ Using the UDDI GUI for publish and find operations
- ▶ Using the UDDI Explorer of Application Developer to access the private UDDI registry

WebSphere Application Server V5.0.2 update

WebSphere Application Server Version 5.0.2 introduced the ability to add user-defined taxonomies, with available allowed values presented in the existing GUI taxonomy tree display.

WebSphere Studio Application Developer Version 5.1 has a Web Services Explorer user interface that also allows the addition and display of custom-checked taxonomies. The publisher of a custom taxonomy's categorization tModel can specify a *display name* for use in GUI implementations.

For more information, refer to:

http://publib.boulder.ibm.com/infocenter/wasinfo/topic/com.ibm.wasee.doc/inf/fo/ee/ae/rwsu_tax.html

WebSphere Application Server V5.1 update

WebSphere Application Server Version 5.1 introduced *UDDI Utility Tools* that provide these functions as commands and as a programmatic API (Figure 7-9):

- ▶ **Export**—Export a UDDI entity (or list of entities) from a registry into an entity definition file (XML format).
- ▶ **Import**—Import UDDI entities from an entity definition file or programmatically into a registry (add entities or update existing entities).
- ▶ **Promote**—Combine export and import to copy entities from one registry to another. Generation of an entity definition file is optional.
- ▶ **Delete**—Delete an entity or a list of entities.
- ▶ **Find**—Search for matching entities based on inquiry API objects (only programmatic API). The resulting entity list can be used in subsequent export, promote, and delete requests.

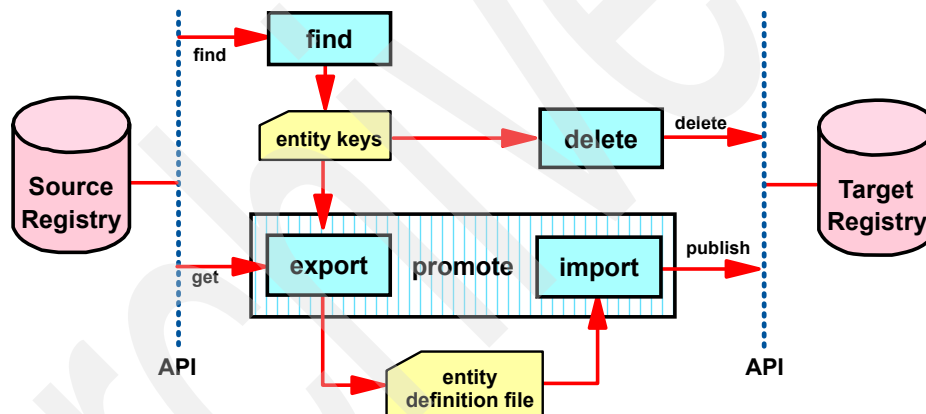


Figure 7-9 UDDI Utility Tools

WebSphere Application Server V6.0 update

WebSphere Application Server Version 6.0 introduces a V3-based UDDI repository. This includes the features that are new in the UDDI V3 standard (see “New features in UDDI Version 3” on page 129), as well as the following:

- ▶ **Version 2 UDDI inquiry and publish API compatibility**—Backward compatibility is maintained for the Version 1 and Version 2 SOAP Inquiry and Publish APIs.
- ▶ **UDDI administrative console extension**—The WebSphere Application Server V6 administrative console includes a section that enables administrators to manage UDDI-specific aspects of their WebSphere environment.

This includes the ability to set defaults for initialization of the UDDI node (such as its node ID) and to set the UDDI V3 policy values.

- ▶ **UDDI registry administrative interface**—A JMX administrative interface enables administrators to programmatically manage UDDI-specific aspects of the WebSphere environment.
- ▶ **Multi-database support**—The UDDI data is persisted to a registry database. In WebSphere Application Server V6, the databases that are supported are IBM DB2®, IBM Cloudscape™, and Oracle.
- ▶ **User-defined value set support**—This enables users to create their own categorization schemes or value sets, in addition to the standard schemes, such as NAICS, that are provided with the product.
- ▶ **UDDI Utility Tools**—UDDI Utility Tools enables the importing and exporting of entities using the UDDI Version 2 API.
- ▶ **UDDI user interface**—The UDDI user console supports the Inquiry and Publish APIs, providing a similar level of support for the Version 3 APIs as was offered for UDDI Version 2 in WebSphere Application Server Version 5.
- ▶ **UDDI Version 3 client**—The IBM Java Client for UDDI V3 is a Java client for UDDI that handles the construction of raw SOAP requests for the client application. It is a JAX-RPC client and uses Version 3 data types generated from the UDDI Version 3 WSDL and schema. These data types are serialized/deserialized to the XML that constitutes the raw UDDI requests.
- ▶ **UDDI Version 2 clients**—UDDI4J, JAXR, and EJB (a deprecated EJB interface for issuing UDDI V2 requests).

Summary

UDDI is a standard that provides the ability to use dynamic Web services, which means that the service requestor and service provider do not necessarily know about each other up front. An organization that wants to provide some service publishes this service in any UDDI registry.

There are two ways a client can find a Web service:

- ▶ One way is a human exploring the UDDI at design time, who might search for service or a WSDL, and use that information when programming the client.
- ▶ Another possibility would be a programmatic exploration by the client application, which allows dynamic binding and changing service providers at runtime.

There are three major points that are often not considered or are misunderstood when talking about UDDI registries:

- ▶ UDDI registries cannot only contain details of published Web services, but also details of other kinds of service, for example, services that cannot be used programmatically.
- ▶ There is an increasing need for specialized and partly private registries that are hosted by standards bodies or industries. In addition, internal intranet registries (with or without contact to business partners) are gaining more and more importance.
- ▶ When publishing or finding tModels, you never publish or store WSDL interface files. The tModel is just a logical link to a concrete interface. When a service references a tModel, you cannot directly discover the real interface.

More information

For more information about UDDI, consult the IBM UDDI Web site at:

<http://uddi.ibm.com>

For general UDDI information, refer to the Web site at:

<http://www.uddi.org>

For more information about the WebSphere private registry, consult the *WebSphere Application Server Information Center*, available at:

<http://publib.boulder.ibm.com/infocenter/ws60help/index.jsp>

Web Services Inspection Language

This chapter provides an introduction to the Web Services Inspection Language (WSIL or WS-Inspection) 1.0. An alternative as well as complement to UDDI, WS-Inspection is another service discovery mechanism that addresses a different subset of requirements using a distributed usage model.

We discuss the principal features and taxonomy, as well as its business application. The WS-Inspection specification is designed around an XML-based model for building an aggregation of references to existing Web service descriptions.

WS-Inspection 1.0 provides a notation serving these purposes. The WS-Inspection specification is a joint effort by IBM and Microsoft. It was a draft document at the time of writing this book, and represents the current status with regard to locating and inspecting services.

Overview

A centralized service registry, which is described in Chapter 7, “Introduction to UDDI” on page 123, is not the only model for Web services discovery. The simplest form of services discovery is requesting a copy of the service description (WSDL file) from the service provider. This is not very efficient because it requires a prior knowledge of the Web service. The most complex form is connecting to a UDDI registry that provides a more business-centric perspective.

Between these two extremes, there is a need for a distributed service discovery method that provides references to service descriptions at the service provider's point-of-offering. The Web Services Inspection Language provides this type of distributed discovery method by specifying how to inspect a Web site for available Web services. The WS-Inspection specification defines the locations on a Web site where you might find Web service descriptions.

The WS-Inspection specification does not define a service description language, which is left to other languages such as WSDL. WS-Inspection documents provide a method for aggregating different types of service descriptions. Within a WS-Inspection document, a single service can have more than one reference to a service description. For example, a single Web service might be described using both a WSDL file and within a UDDI registry. References to these two service descriptions can be put into a WS-Inspection document. If multiple references are available, it is beneficial to put all of them in the WS-Inspection document so that the document user can select the type of service description that they can understand and want to use.

The WS-Inspection specification contains two primary functions:

- ▶ It defines an XML format for listing references to existing service descriptions.
- ▶ It defines a set of conventions so that it is easy to locate WS-Inspection documents.

The WS-Inspection specification complements a UDDI registry by facilitating the discovery of services available on Web sites, which might not be listed yet in a UDDI registry. Figure 8-1 shows an overview of WS-Inspection.

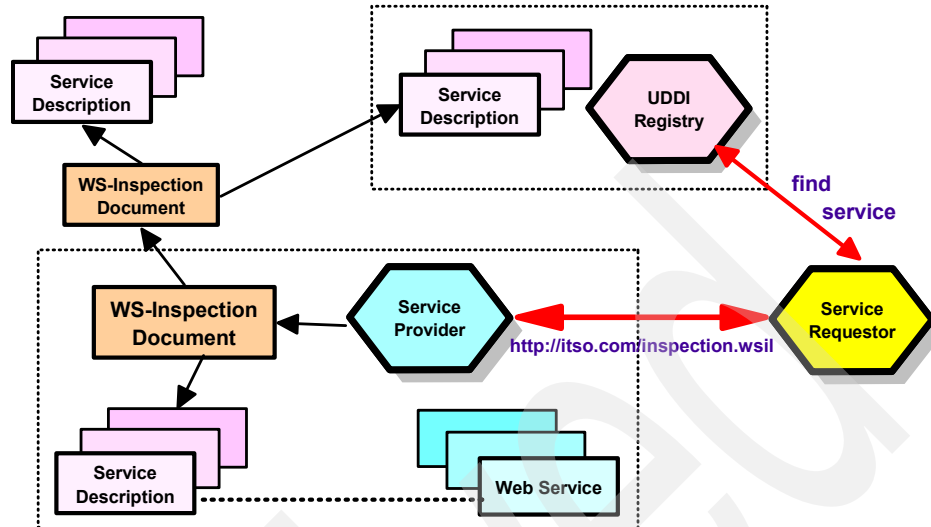


Figure 8-1 WS-Inspection overview

WS-Inspection document

The most important objectives of the inspection language are simplicity and extensibility. A WS-Inspection document is an aggregation of pointers to service description documents. A service descriptions can be any format, for example, a WSDL files, UDDI entries, or plain HTML. The WS-Inspection document is generally available at the point-of-offering (service provider) and allows the consumer to pick and choose from the available service descriptions using the access method that is most useful. A WSIL document can also contain references to other WSIL documents.

The WS-Inspection document contains an *inspection* element at the root that provides a wrapper to the two main elements:

- service** Contains the references to different types of service descriptions for the same Web service.
- link** Allows a WS-Inspection document to reference additional aggregation of services descriptions, such as other WS-Inspection documents or UDDI entries.

The optional *abstract* element is used as a container to provide a small textual description for human consumption. This element is allowed inside any other base element that allows child elements.

WS-Inspection document anatomy

Figure 8-2 shows the anatomy and the relationship among the different elements in a WS-Inspection document. The structure is quite simple and, therefore, it can be easily maintained.

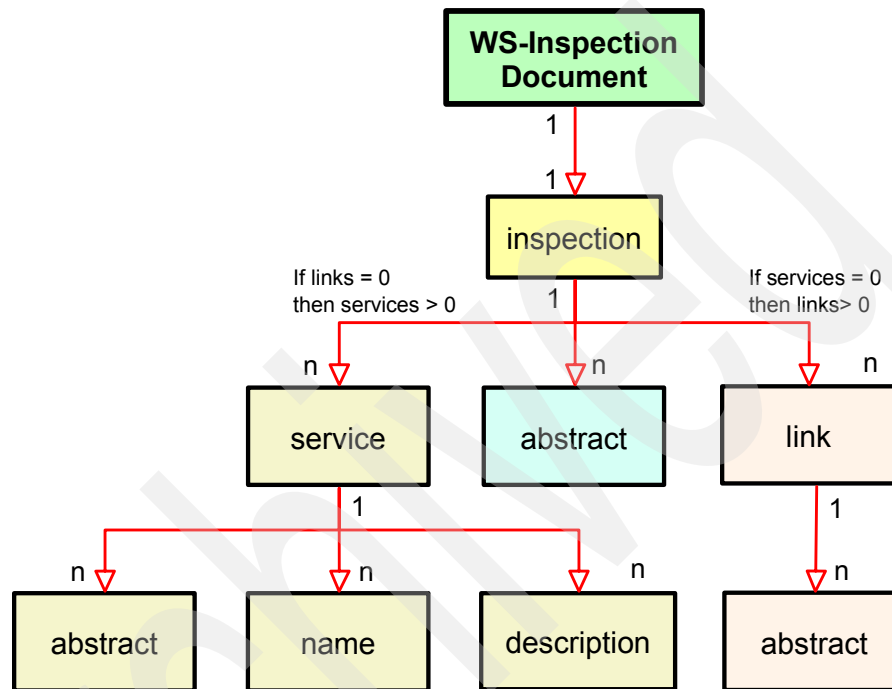


Figure 8-2 WS-Inspection document elements and relationship

The diagram should be read in the following way:

- ▶ One WS-Inspection document contains one inspection element.
- ▶ One inspection element contains zero or more abstract elements and zero or more links or services, or both, with one restriction. This restriction is that the inspection element must contain at least either one service or one link, but never none of them. For example, if there is no service, there has to be at least one link, and vice versa. Any other combination of links and services is possible.
- ▶ One service contains zero or more abstract, name, and description elements.
- ▶ One link contains zero or more abstract elements.

The containment relationship shown in the diagram directly maps to the XML Schema for WS-Inspection.

Example

Figure 8-3 contains a simple WS-Inspection document example. As we see, the syntax used in this file is not very complex, but rather easy to understand. We analyze this example in detail in this chapter, but the whole document is presented here to obtain a complete picture.

As an exercise, you can try identifying the different elements in Figure 8-2 while examining this example.

```
<?xml version="1.0"?>
<inspection xmlns="http://schemas.xmlsoap.org/ws/2001/10/inspection/"
<service>
  <abstract>A simple service with two descriptions</abstract>
  <description referencedNamespace="http://schemas.xmlsoap.org/wsdl/"
    location="http://itso.com/train.wsdl"/>
  <description referencedNamespace="urn:uddi-org:api_v2">
    <wsiluddi:serviceDescription
      location="http://itso.com/uddi/inquiryapi">
      <wsiluddi:serviceKey>
        4FA23580-5C39-14D5-9FCF-BB3200303F79
      </wsiluddi:serviceKey>
    </wsiluddi:serviceDescription>
  </description>
</service>
<service>
  <description referencedNamespace="http://schemas.xmlsoap.org/wsdl/"
    location="ftp://itso.com/tools/plane.wsdl"/>
</service>
<link
  referencedNamespace="http://schemas.xmlsoap.org/ws/2001/10/inspection/"
  location="http://itso.com/others/moreservices.wsil"/>
</inspection>
```

Figure 8-3 WS-Inspection document example

Namespaces

Table 8-1 shows the namespaces used in a WS-Inspection document.

Table 8-1 WS-Inspection namespaces

Prefix	Namespace URI	Definition
wsil	http://schemas.xmlsoap.org/ws/2001/10/inspection/	WS-Inspection namespace for the WS-Inspection framework.
wsilwsdl	http://schemas.xmlsoap.org/ws/2001/10/inspection/wsdl/	WS-Inspection namespace for the WS-Inspection WSDL binding.

Prefix	Namespace URI	Definition
wsiluddi	http://schemas.xmlsoap.org/ws/2001/10/inspection/uddi/	WS-Inspection namespace for the WS-Inspection UDDI binding.
uddi	urn:uddi-org:api	API and document namespace defined by the UDDI V1 specification.
uddiv2	urn:uddi-org:api_v2	API and document namespace defined by the UDDI V2 specification.
(other)	(various)	All other namespace prefixes are samples only. In particular, URIs starting with "http://itso.com" represent some application-dependent or context-dependent URI.

WS-Inspection and UDDI relationship

WS-Inspection and UDDI specifications address issues related to Web service discovery. Even so, these specifications were designed with different goals in mind, and thus exhibit different characteristics.

One of the fundamental points of a successful Web service is related to the discovery of the service. We illustrate how the two specifications can be used jointly to solve a variety of requirements.

The information can be extracted either straight from the source of information, WS-Inspection, or from a third-party UDDI registry. The two possibilities have some characteristic features.

WS-Inspection

- ▶ Retrieving the information directly from the source increases the possibilities of obtaining accurate information.
- ▶ The use of a distributed model allows the service descriptions to be stored at any location.
- ▶ They are very lightweight, easy to create, and easy to maintain.
- ▶ They allow the provider to present its services in a proactive fashion.
- ▶ They do not stipulate any particular format for the service and rely on other standards, including UDDI.
- ▶ They do not provide a rich mechanism to execute any focus discovery.

UDDI registry

- ▶ Retrieving the information indirectly provides the opportunity for more developed and advanced services and does not require that the original source be available or easily accessible.
- ▶ Uses a centralized model.
- ▶ Provides high-level functionality in the discovery (search facility).

Both systems address different sets of issues in the discovery problem space. Whereas the UDDI provides a high degree of functionality, the WS-Inspection document adopts it at a lower level. Therefore, both specifications should be understood as complementary technologies to be used either together or separately, depending upon the situation.

There is a whole range of mutual benefit in the combined use of the technologies. For example, a UDDI registry can be populated using the WS-Inspection documents found in an Internet search. Conversely, a UDDI registry can be discovered when a requestor retrieves a WS-Inspection document that references an entry in the repository. Therefore, they should not be viewed as competing mechanisms; for example, a business card does not compete with the yellow pages in disseminating personal information.

WS-Inspection definition

The WS-Inspection definition contains a detailed explanation of the core elements of the WS-Inspection language. The document contains an *inspection* element at the root, and references to individual service descriptions or links as descendants. The grammar of a WS-Inspection document is as follows:

```
<wsil:inspection>
  <wsil:abstract xml:lang="" (0 or 1)... /> (0 or more)
  <wsil:service> (0 or more)
    <wsil:abstract xml:lang="" (0 or 1) ... /> (0 or more)
    <wsil:name xml:lang="" (0 or 1) ... /> (0 or more)
    <wsil:description referencedNamespace="uri" location="uri" (0 or 1)>
      (0 or more)
      <wsil:abstract xml:lang="" (0 or 1) ... /> (0 or more)
      <!-- extensibility element --> (0 or 1)
    </wsil:description>
  </wsil:service>
  <wsil:link referencedNamespace="uri" location="uri" (0 or 1)/>
    (0 or more)
    <wsil:abstract xml:lang="" (0 or 1) ... /> (0 or more)
    <!-- extensibility element --> (0 or more)
  </wsil:link>
</wsil:inspection>
```

The *inspection* element must contain at least one *service* or one *link*.

To provide the possibility to include documentation, the specification uses the optional *abstract* element. This element contains a small textual description to be exploited by a human reader. The abstract element is allowed inside any base WS-Inspection language element that allows child elements. An optional `xml:lang` attribute specifies on the element the language in which the abstract element has been written.

Services

The *service* element provides the mechanisms to include a collection of description references for services. An inspection element can contain any number of service elements within it. Each *service* element may have one or more *abstract* elements and one or more *name* elements, and must have at least one *description* element, but can have more.

Service name

The *name* is used to associate the service with a particular name. The reason for this element is only for human consumption as the abstract element. In the same way used in the abstract element, an optional element, `xml:lang`, can be used to specify the language in which the name has been written.

Service description references

The most important and useful part of the inspection is the information contained within the *description* element. This element is used to provide pointers to service description documents of various forms. The service description element contains the following attributes:

referencedNamespace	Identifies the namespace to which the reference document belongs, such as WSDL or UDDI.
location	Provides the actual reference to the description. It has to be a valid URL.
extensible	Provides additional information that is needed to retrieve or process the service description. See “WS-Inspection bindings” on page 164 for more information.

In our example, the service definition is shown in Figure 8-4, where we specify two services, the first one about *trains* and the second one about *planes*.

```

<service>
  <abstract>A simple service with two descriptions</abstract>
  <description referencedNamespace="http://schemas.xmlsoap.org/wsdl/"
    location="http://itso.com/train.wsdl"/>
  <description referencedNamespace="urn:uddi-org:api_v2">
    <wsiluddi:serviceDescription
      location="http://itso.com/uddi/inquiryapi">
        <wsiluddi:serviceKey>
          4FA23580-5C39-14D5-9FCF-BB3200303F79
        </wsiluddi:serviceKey>
      </wsiluddi:serviceDescription>
    </description>
  </service>
<service>
  <description referencedNamespace="http://schemas.xmlsoap.org/wsdl/"
    location="ftp://itso.com/tools/plane.wsdl"/>
</service>

```

Figure 8-4 Service definition in our WS-Inspection document example

In the first service, we provide an abstract element explaining that it is a simple example that contains two descriptions:

- ▶ One description makes a reference to a WSDL document, which is specified using the namespace `http://schemas.xmlsoap.org/wsdl/` and the URL location `http://itso.com/train.wsdl`.
- ▶ Another description for the same Web service is provided using a UDDI registry Version 2 that we specify with the namespace `urn:uddi-org:api`. The location for the query is `http://itso.com/uddi/inquiryapi`, and the service key `4FA23580-5C39-14D5-9FCF-BB3200303F79` can be used to retrieve the service description.

In the second service, we provide a reference to a WSDL document, which is specified using the namespace `http://schemas.xmlsoap.org/wsdl/` and the URL location `ftp://itso.com/tools/plane.wsdl`.

Links

The *link* element allows WS-Inspection documents to reference additional aggregations of service descriptions, such as other WS-Inspection documents or UDDI repositories. Therefore, we can create a hierarchy of documents (see Figure 8-1 on page 157).

The link element helps service providers to create inspection documents with a large number of services elements pointing to the same location in the

information registry. Using the link, the providers are able to indicate to the consumers that they have to look at the indicated location for additional service description aggregations.

The `referencedNamespace` attribute identifies the namespace of the linked aggregation source, such as another inspection document or a UDDI registry, but there is no specification of how this link element has to be processed. This means that there is no predefined rule of how this information has to be used.

In our example (Figure 8-5), the link definition is where we specify one link to another WS-Inspection document providing the namespace and the location (`http://itso.com/others/moreservices.wsil`) of the document.

```
<link
  referencedNamespace="http://schemas.xmlsoap.org/ws/2001/10/inspection/"
  location="http://itso.com/others/moreservices.wsil"/>
```

Figure 8-5 Link references in our WS-Inspection document example

WS-Inspection bindings

We now introduce the characteristics of the extension elements related to the different bindings.

WSDL binding

The WS-Inspection language includes a binding for WSDL 1.1, providing the following capabilities:

- ▶ Indication of the type or types of WSDL bindings that appear in the referenced WSDL document
- ▶ Specification of which WSDL service is being referenced if several services exist within the same document
- ▶ Indication of whether or not a particular referenced WSDL document provided endpoint information

The use of the WSDL binding is optional if the referenced WSDL document has one service element or none. In these cases, a reference to a WSDL document may be made with only one description element. In cases where the WSDL document has more than one service, the binding must be used to indicate which service element is being referenced.

The grammar for the binding extension elements is shown in Figure 8-6 and the description of its element in Table 8-2.

```

<inspection
  xmlns:wsilwsdl="http://schemas.xmlsoap.org/ws/2001/10/inspection/wsd1/ ...>
<service ...>
  <description referencedNamespace="http://schemas.xmlsoap.org/wsd1/"
    location="uri">
    <wsilwsdl:reference endpointPresent="boolean"(0 or 1)>
      <wsilwsdl:referencedService>
        QName
      </wsilwsdl:referencedService> (0 or 1)
      <wsilwsdl:implementedBinding>
        QName
      </wsilwsdl:implementedBinding> (0 or more)
    </wsilwsdl:reference>
  </description>
</service>
</inspection>

```

Figure 8-6 WSDL binding extension elements grammar

Table 8-2 WSDL extensibility elements in WS-Inspection

Extension name	Explanation
wsilwsdl:reference	The container element for the rest of the binding information. The endpointPresent is a Boolean attribute that indicates whether the referenced WSDL contains endpoint information.
wsilwsdl:referencedService	Specifies which service is being referenced when there are more than one. The QName specified must be a valid service name in the referenced WSDL document.
wsilwsdl:implementedBinding	Used to indicate the type or types of binding that appear in a referenced WSDL document. The QName specified must be a valid binding name in the referenced WSDL document.

UDDI binding

The WS-Inspection language provides a set of usage patterns for references to business service or business entity information presented in a UDDI registry. This set of elements can refer to entries in a UDDI Version 1.0 or Version 2.0 registry, depending on the referenceNamespace used. The UDDI extension elements can be included in the description or link elements, in contrast to the WSDL extension elements, which are only in the description element.

The extension elements included in the link can only reference a UDDI business entity. Meanwhile, the extension elements included in the description can only reference a single UDDI business service.

The grammar for the binding extension elements is shown in Figure 8-7, and the description of its element in Table 8-3.

```
<inspection
(xmlns:wsiluddi://schemas.xmlsoap.org/ws/2001/10/inspection/uddi/ |
xmlns:wsiluddi://schemas.xmlsoap.org/ws/2001/10/inspection/uddiv2/) ... >
  <service ... >
    <description ... >
      <wsiluddi:serviceDescription location="uri"> (0 or 1)
        <wsiluddi:discoveryURL useType="nmtoken"> (0 or 1)
          <wsiluddi:serviceKey> (0 or 1)
        </wsiluddi:serviceDescription>
      </description>
    </service>
    <link ... >
      <wsiluddi:businessDescription location="uri"> (0 or 1)
        <wsiluddi:discoveryURL useType="nmtoken"> (0 or 1)
          <wsiluddi:businessKey> (0 or 1)
        </wsiluddi:businessDescription>
      </link>
    </inspection>
```

Figure 8-7 UDDI binding extension elements grammar

Table 8-3 UDDI extensibility elements in WS-Inspection

Element	Extension name	Explanation
description	wsiluddi:serviceDescription	Specifies a reference to a single UDDI service description. Must contain a serviceKey and might also contain a discoveryURL.
	wsiluddi:discoveryURL	Used to retrieve a UDDI business entity using HTTP GET to obtain a single service description using the serviceKey.
	wsiluddi:serviceKey	Used to retrieve the UDDI service description.
link	wsiluddi:businessDescription	Specifies a reference to a UDDI business entity. Must contain a discoveryURL, a businessKey, or both. The location attribute contains the UDDI registry inquiry interface.
	wsiluddi:discoveryURL	Specifies a valid URL to retrieve the UDDI business entity using HTTP GET.
	wsiluddi:businessKey	Specifies a business key that is used in combination with the location attribute of the business description to retrieve the UDDI business entity using HTTP POST.

WS-Inspection document publishing

The WS-Inspection grammar only provides a partial solution to the problem of advertising services. We need some rules to easily locate these WS-Inspection documents, because if not, their added value is lost.

► WS-Inspection document name

The WS-Inspection specification is not specific about the WS-Inspection document name. If a URL does not directly correspond to a WS-Inspection document, but the host name of the site is known, it should still be possible to access the WS-Inspection document. To accomplish this, the specification recommends that you name the document file `inspection.wsil`.

The file can be placed where the most common entry points to Web sites or applications deployed on the server are located. For instance, in the example used in this chapter, our document is placed in:

`http://itso.com/inspection.wsil`

► Linked WS-Inspection documents

They allow for a mechanism to inform users about services inspection-related documents using other types of content, such as HTML pages.

There are different possibilities when setting up WS-Inspection site, as shown in Table 8-4.

Table 8-4 WS-Inspection site setup

	Advantage	Disadvantage
One Inspection file per site: <code>http://itso.com/inspection.wsil</code> inspection.wsil is the recommended name in the specification.	► One access point to several services. ► Host name is sufficient to find the WSIL document.	► Administration can get complex. ► A large file can get stale over time.
Different file names per service: <code>http://itso.com/service1.wsil</code> <code>http://itso.com/service2.wsil</code>	► URI links to an exact service ► Easy to maintain	► Just the host name is not sufficient to find the WSIL document.
Different folders (URIs) per service: <code>http://itso.com/service1/inspection.wsil</code> <code>http://itso.com/service2/inspection.wsil</code>	► URI links to an exact service. ► Easy to maintain.	► URI/sub-folders must be created on the server. ► Just the host name is not sufficient to find the WSIL document.

Because the specification is not strict about the file name, nor about the location, it is up to the WS-Inspection site provider to define the setup. Looking at the described possibilities, we recommend using different file names for each service.

WS-Inspection examples

In this section, we describe two examples of bindings.

WSDL binding example

The first example shows the use of a WSDL binding, that is, the WSIL document points to a WSDL file on a server (Figure 8-8).


```

<?xml version="1.0"?>
<inspection
  targetNamespace="http://schemas.xmlsoap.org/ws/2001/10/inspection/"
  xmlns:wsiluddi="http://schemas.xmlsoap.org/ws/2001/10/inspection/uddi/"
  xmlns:wsilwsdl="http://schemas.xmlsoap.org/ws/2001/10/inspection/wsdl/"
  xmlns="http://schemas.xmlsoap.org/ws/2001/10/inspection/">
  <service>
    <abstract>The Flash Weather forecast service description</abstract>
    <name xml:lang="en-US">ITSO WS Flash Weather Service</name>
    <description referencedNamespace="http://schemas.xmlsoap.org/wsdl/"
      location="http://www.flashandthunder.com/ItsoFlashWeb/wsdl/
        itso/bean/WeatherForecast.wsdl">
      <wsilwsdl:reference endpointPresent="true">
        <wsilwsdl:implementedBinding xmlns:binding="http://bean.itso">
          intf:WeatherForecastSoapBinding
        </wsilwsdl:implementedBinding>
      </wsilwsdl:reference>
    </description>
  </service>
  <service>
    <abstract>The Thunder Weather forecast service description</abstract>
    <name xml:lang="en-US">ITSO WS Thunder Weather Service</name>
    <description referencedNamespace="http://schemas.xmlsoap.org/wsdl/"
      location="http://www.flashandthunder.com/ItsoThunderWeb/wsdl/
        itso/bean/WeatherForecast.wsdl">
      .....
    </description>
  </service>
</inspection>

```

Figure 8-8 WSIL example with WSDL binding

In this example, the WSIL document refers to two services by pointing to the WSDL files on two provider servers using URLs.

UDDI binding example

The second example shows the use of a UDDI binding, that is, the WSIL document points to a UDDI registry (Figure 8-9).

```

.....
<service>
  <name xml:lang="en-US">ITS0 WS Thunder Weather Service</name>
  <description referencedNamespace="urn:uddi-org:api_v2">
    <wsiluddi:serviceDescription location=
      "http://uddi.ibm.com/testregistry/inquiryapi">
      <wsiluddi:serviceKey>F3362720-0C6D-11D7-AB70-000629DC0A53
      </wsiluddi:serviceKey>
      <wsiluddi:discoveryURL useType="businessEntity">
        http://uddi.ibm.com/testregistry/inquiryapi
        ?businessKey=F5082FA0-0BCB-11D7-AB70-000629DC0A53
      </wsiluddi:discoveryURL>
      </wsiluddi:serviceDescription>
    </description>
  </service>
  <link referencedNamespace="urn:uddi-org:api_v2">
    <wsiluddi:businessDescription
      location="http://uddi.ibm.com/testregistry/inquiryapi">
      <wsiluddi:businessKey>F5082FA0-0BCB-11D7-AB70-000629DC0A53
      </wsiluddi:businessKey>
      <wsiluddi:discoveryURL useType="businessEntity">
        http://uddi.ibm.com/testregistry/inquiryapi
        ?businessKey=F5082FA0-0BCB-11D7-AB70-000629DC0A53
      </wsiluddi:discoveryURL>
      </wsiluddi:businessDescription>
    </link>
  </link>
  .....

```

Figure 8-9 WSIL example with UDDI binding

In this example, the <service> tag points to a single business service in the UDDI registry, while the <link> tag points to a business entity. The location attribute points to the address and API of the registry that is accessed.

Note that both entries are over-specified: The discoveryURL is optional in the <service> tag, and only the discoveryURL or the businesskey are required in the <link> tag.

WS-Inspection API

There is a WS-Inspection Java API called WSIL4J. This API can be used to locate and process WS-Inspection documents. The class library provides functionality to read and parse WS-Inspection documents and to generate new documents.

The primary classes are:

<code>org.apache.wsil.WSILDocument</code>	Interacts with the WS-Inspection documents.
<code>org.apache.wsil.client.WSILProxy</code>	Accesses specific service descriptions within a WS-Inspection document.

Currently, WSIL4J is an open-source project available from the Apache Software Foundation as a part of the Apache Axis work:

<http://xml.apache.org/axis/index.html>

Figure 8-10 contains an example of how to use this API. In this sample code, a WS-Inspection document is read and the service elements are searched for references to WSDL service descriptions. When a WSDL service description is found, its location is saved in a list.

```
...
// Create a new instance of a WS-Inspection document
WSILDocument document = WSILDocument.newInstance();
// Read and parse the WS-Inspection document
document.read(wsinspectionURL);
// Get the inspection element from the document
Inspection inspection = document.getInspection();
// Obtain a list of all service elements
Service[] services = inspection.getServices();
// Process each service element to find all WSDL document references
for (int serviceCount = 0; serviceCount < services.length; serviceCount++) {
    // Get the next set of description elements
    descriptions = services[serviceCount].getDescriptions();
    // Process each description to find the WSDL references
    for (int descCount = 0; descCount < descriptions.length; descCount++) {
        //If the referenced is WSDL, then save the location reference
        if(descriptions[descCount].
            getReferencedNamespace().equals(WSDLConstants.NS_URI_WSDL)) {
            // Add WSDL location to the list
            wsdlList.add(descriptions[descCount].getLocation());
        }
    }
}
...
```

Figure 8-10 WS-Inspection API example

Outlook

At the time this book was written, this technology was in its first use as a new service discovery mechanism. The population of WS-Inspection documents should greatly increase in coming months. There are few examples of published WS-Inspection documents on the Internet. One example is at the XMethod Web site:

<http://www.xmethods.net/inspection.wsil>

Based on the contribution of WSIL4J to Apache Axis, we may soon see a WS-Inspection API in Axis. This technology will be used for other applications, such as a Web service crawler. A service crawler would search through Web sites for WS-Inspection documents and then aggregate the service description references from multiple sites.

Consequently, and based on the current and future applications of the Web Services Inspection Language, the WS-Inspection documents will play a fundamental role in the overall development of the Web services technology.

Summary

In this chapter, we have described the Web Services Inspection Language and how this specification provides a simple method to discover any type of Web service description document. We analyzed its grammar and how these WS-Inspection documents can be found. We also saw that this technology does not try to replace present technologies regarding service discovery, but is a complementary discovery method that can be integrated to work together with UDDI repositories, for example.

Finally, we briefly reviewed the ways that the open-source WSIL4J API can facilitate the reading, parsing, and generation of WS-Inspection documents.

More information

The best source for more information is the Web Services Inspection Language (WS-Inspection) 1.0 specification, available at:

<http://www.ibm.com/developerworks/webservices/library/ws-wsilspec.html>

Web services security

To provide the basis for the need of Web service security, we first discuss general security and possible security exposures. Further, we provide detailed information about security concerns specific to Web services. We explain the available techniques and standards to set up a secure Web service environment.

We cover transport channel security solutions and then devote most of this chapter to the discussion of the WS-Security specification in IBM WebSphere Application Server Version 6.0.

Security overview

Since the early days of the Internet as a universal network open to anyone, information exchange has been a concern. Although it is worth noting that there is no absolute security, developments in this field were very quick and fruitful because they were driven by urgent business needs. Without an appropriate level of security, the commercial exploitation of the Internet would not be feasible.

Networks must be designed to provide a high level of security for information that travels across the Internet or privately managed intranets or extranets. Algorithms, such as third-party authentication, public key encryption, and digital signature, can provide a sufficient level of security. However, security does not only depend on algorithms, standards, and products. Companies are required to follow security best-practice recommendations.

Note: A company's security policy should reasonably cover the most important procedures, such as user management (adding/removing users and granting their rights and access levels), network usage guidelines (private mail, Web site access policy, and antivirus protection), user authentication procedures (user ID/password, key cards), system monitoring procedures, and procedures in case of attack.

A general security framework should address the following requirements:

- ▶ **Identification**—The party accessing the resource is able to identify itself to the system.
- ▶ **Authentication**—Authentication is the process of validating the user, whether a client is valid in a particular context. A client can be either an end user, a machine, or an application.
- ▶ **Authorization**—Authorization is the process of checking whether the authenticated user has access to the requested resource.
- ▶ **Integrity**—Integrity insures that information will not be changed, altered, or lost in an unauthorized or accidental manner.
- ▶ **Confidentiality**—No unauthorized party or process can access or disclose the information.
- ▶ **Auditing**—All transactions are recorded so that problems can be analyzed after the fact.
- ▶ **Non-repudiation**—Both parties are able to provide legal proof to a third party that the sender did send the information, and the receiver received the identical information. Neither involved side is “unable to deny.”

Some classifications also include *availability* to be a part of the above schema, meaning that a hostile attack cannot achieve denial-of-service by allocating too many system resources. In this chapter, we do not discuss this security aspect.

Web services security exposures

Web services security is one of the most important Web services subjects. When using Web services, similar security exposures exists as for other Internet, middleware-based applications, and communications.

To explain the Web services security exposures, let us use a bank teller scenario as an example (Figure 9-1). The bank teller (Web service client) connects over the Internet to the bank's data center (Web service provider). We assume there is no security applied at all, which is not realistic, but needed for the example.

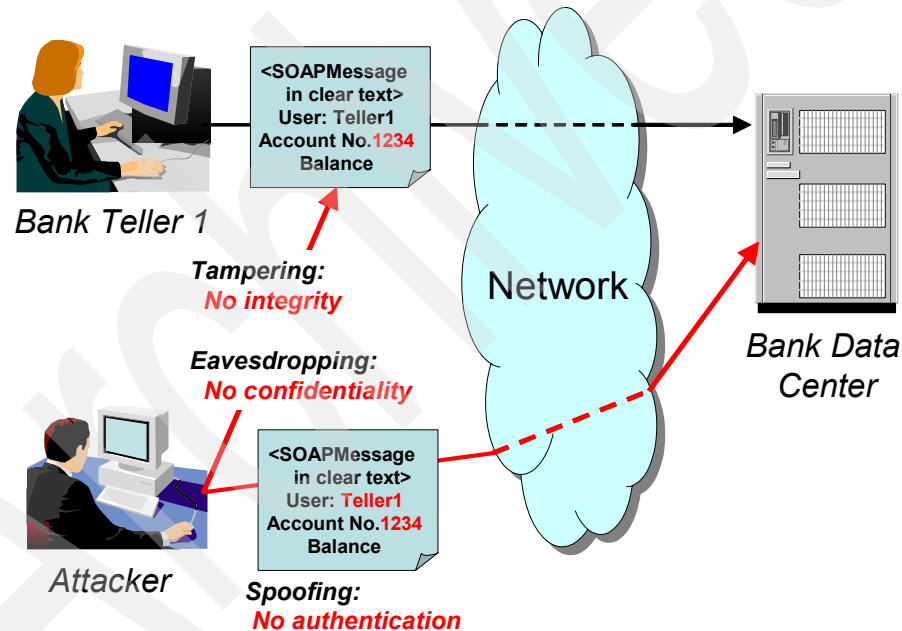


Figure 9-1 Common security exposures in a sample Web services application

The three major risk factors in this example are:

- **Spoofing—no authentication:** An attacker could send a modified SOAP message to the service provider, pretending to be a bank teller, to get confidential information, or to withdraw money from another customer's account.

By applying authentication to the Web service, this security exposure can be eliminated.

- ▶ **Tampering—no integrity:** The SOAP message is intercepted between the Web service client and server. An attacker could modify the message, for example, deposit the money into another account by changing the account number. Because there is no integrity constraint, the Web service server does not check if the message is valid and will accept the modified transaction.

By applying an integrity mechanism to the Web service, this security exposure can be eliminated.

- ▶ **Eavesdropping—no confidentiality:** An attacker can intercept the SOAP message and read all contained information. Because the message is not encrypted, confidential customer or bank information can go the wrong people. This exposure exists because the account number and balance information is sent over the network in plain text.

By applying a confidentiality mechanism to the Web service, this security exposure can be eliminated.

To prevent the described security exposures, the following mechanisms can be applied to secure a Web services environment (Figure 9-2):

- ▶ Message-level security—Web services security (WS-Security)
- ▶ Transport-level security—TLS/SSL

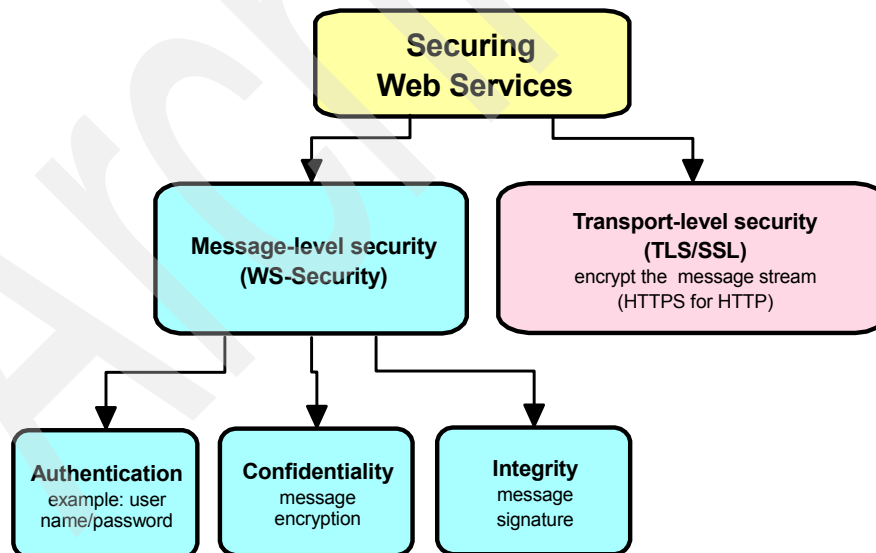


Figure 9-2 Securing Web services

Depending on the demanded level of application security, one or more of these security mechanisms can be applied. Also depending on other non-functional requirements, a combination of message-level security and transport-level security can be implemented.

The more security mechanisms implemented, which increase the security effect, the more influence other non-functional requirements are given. Therefore, when designing a Web services security solution, keep in mind that security has an impact on the following non-functional requirements:

- ▶ **System capacity**—Any applied security mechanism has impact on system resource usage (for example, CPU and memory usage). So, when planning a Web service environment, the required *security overhead* must be considered in the system capacity and volume planning.

The non-functional requirements, capacity and volume, cover the number of concurrent users and the number of transactions per second. This has influence on the required system infrastructure (hardware, network).

- ▶ **Performance**—Security mechanisms and functions also impact the application's response time. When defining the Web service system response time requirements, keep in mind that the response time will be affected when applying security.

The performance requirement for a system defines the response time for a main application operation (for example, less than one second for 90% of all transactions).

Note: Applying security is not only a question of feasibility; the additional system resources and the influence on the response time also must be considered.

We cover the WS-Security specification and the SSL mechanism in detail in the sections that follow.

WS-Security

The WS-Security specification provides message-level security, which is used when building secure Web services to implement message content integrity and confidentiality.

The advantage of using WS-Security over SSL is that it can provide end-to-end message-level security. This means that the message security can be protected even if the message goes through multiple services, called intermediaries.

Additionally, WS-Security is independent of the transport layer protocol; it can be used for any Web service binding (for example, HTTP, SOAP, RMI). Using WS-Security, end-to-end security can be obtained (Figure 9-3).

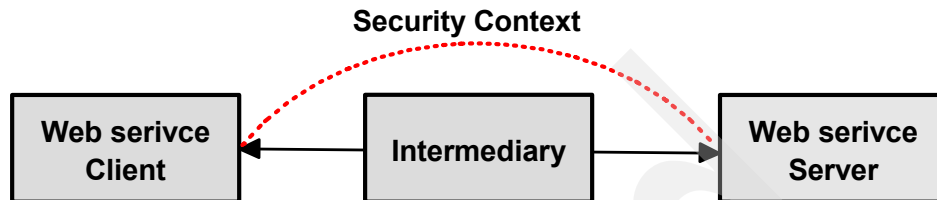


Figure 9-3 End-to-end security with message-level security

The WS-Security specification, Web Services Security: SOAP Message Security 1.0 (WS-Security 2004), is proposed by the OASIS WSS Technical Committee. This specification defines a standard set of SOAP extensions. The specification is flexible and is designed to be used as the basis for securing Web services within a wide variety of security models, including PKI, Kerberos, and SSL. It provides support for multiple security token formats, multiple trust domains, multiple signature formats, and multiple encryption technologies based on XML signature and XML encryption to provide integrity and confidentiality.

The specification includes security token propagation, message integrity, and message confidentiality. However, these mechanisms by themselves do not address all the aspects of a complete security solution. Therefore, WS-Security represents only one of the layers in a complex, secure Web services solution design.

Important: With WS-Security 1.0 the wire format changed in ways that are not compatible with previous WS-Security drafts. Also, interoperability between implementations based on previous drafts and Version 1.0 is not possible.

The WS-Security specification defines the usage of XML signature and XML encryption:

- ▶ Message integrity is provided by XML signature in conjunction with security tokens to ensure that modifications to messages are detected. For more information, refer to:
<http://www.w3c.org/Signature>
- ▶ Message confidentiality leverages XML encryption in conjunction with security tokens to keep portions of a SOAP message confidential. For more information, refer to:
<http://www.w3c.org/Encryption>

Evolution of the WS-Security specification

Figure 9-4 shows the evolution of the WS-Security specification.

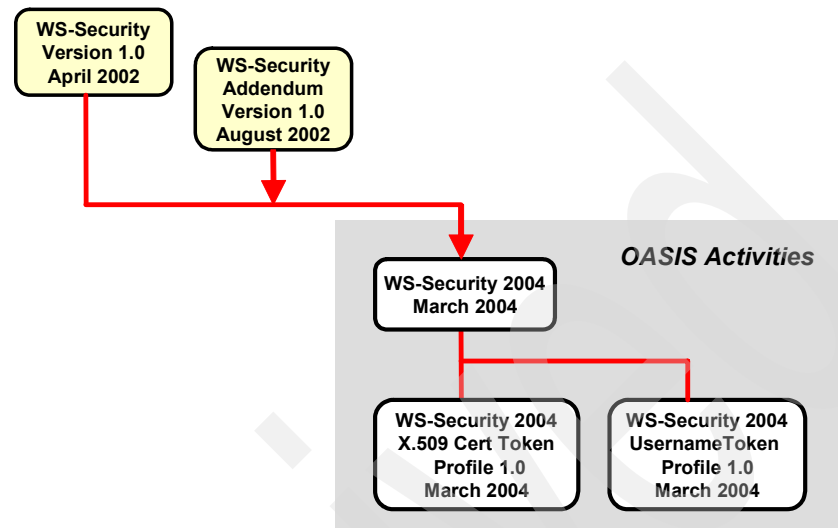


Figure 9-4 Evolution of Web services security

The first version of the WS-Security specification was proposed by IBM, Microsoft, and VeriSign in April 2002. After the formalization of the April 2002 specifications, the specification was transferred to the OASIS consortium:

<http://www.oasis-open.org>

In OASIS activities, the core specification and many profiles that describe the use of a specific token framework in WS-Security have been discussed. The latest specification and profiles of WS-Security were proposed in March 2004 as an OASIS standard.

The latest core specification, Web Services Security: SOAP Message Security 1.0 (WS-Security 2004) was standardized in March 2004. The two profiles, Web Services Security UsernameToken Profile 1.0 and Web Services Security X.509 Certificate Token Profile 1.0, were standardized at the same time.

There are other token profiles on which OASIS is currently working:

- ▶ Web Services Security: SAML Token Profile
- ▶ Web Services Security: Rights Expression Language (REL) Token Profile
- ▶ Web Services Security: Kerberos Token Profile
- ▶ Web Services Security: Minimalist Profile (MProf)
- ▶ Web Services Security: SOAP Message with Attachments (SwA) Profile

To read more about these standards, refer to:

- ▶ Specification: Web Services Security (WS-Security) Version 1.0 (April 2002):
<http://www.ibm.com/developerworks/webservices/library/ws-secure/>
- ▶ Web Services Security Addendum (August 2002):
<http://www.ibm.com/developerworks/webservices/library/ws-secureadd.html>
- ▶ Web Services Security: SOAP Message Security V1.0 (March 2004):
<http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf>
- ▶ Web Services Security: UsernameToken Profile V1.0 (March 2004):
<http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0.pdf>
- ▶ Web Services Security: X.509 Token Profile V1.0 (March 2004):
<http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0.pdf>

WS-Security support in WebSphere Application Server

The support of the April 2002 draft specification is provided in WebSphere Application Server Versions 5.0.2 and 5.1. WebSphere Application Server Version 6.0 supports the WS-Security 2004 specification and two token profiles (UsernameToken 1.0, X.509 Certificate Token 1.0).

The level of the security specification supported in WebSphere Application Server Version 6 is above these specifications, with the described changes in the OASIS erratas:

- ▶ Web Services Security: SOAP Message: Errata 1.0
<http://www.oasis-open.org/committees/download.php/9292/oasis-200401-wss-soap-message-security-1.0-errata-003.pdf>
- ▶ Web Services Security: UsernameToken Profile: Errata 1.0
<http://www.oasis-open.org/committees/download.php/9290/oasis-200401-wss-username-token-profile-1.0-errata-003.pdf>
- ▶ Web Services Security: X.509 Token Profile: Errata 1.0
<http://www.oasis-open.org/committees/download.php/9287/oasis-200401-x509-token-profile-1.0-errata-003.pdf>

Note: WebSphere Application Server V6.0 provides the runtime for the previously mentioned specifications. Rational Application Developer V6.0 provides functions to secure Web services based on these specifications. For more information, see Chapter 21, “Securing Web services” on page 445.

WS-Security road map

As previously mentioned, the WS-Security specification addresses only a subset of security services for all security aspects. A more general security model is required to cover other security aspects, such as logging and non-repudiation. The definition of those requirements is defined in a common Web services security model framework, a security white paper *Web Services Security Roadmap*, proposed by IBM and Microsoft. We describe this road map in the following section.

Web services security model framework

The Web services security model introduces a set of individual interrelated specifications to form a layering approach to security. It includes several aspects of security: identification, authentication, authorization, integrity, confidentiality, auditing, and non-repudiation. It is based on the WS-Security specification, co-developed by IBM, Microsoft, and VeriSign.

The Web services security model is schematically shown in Figure 9-5.

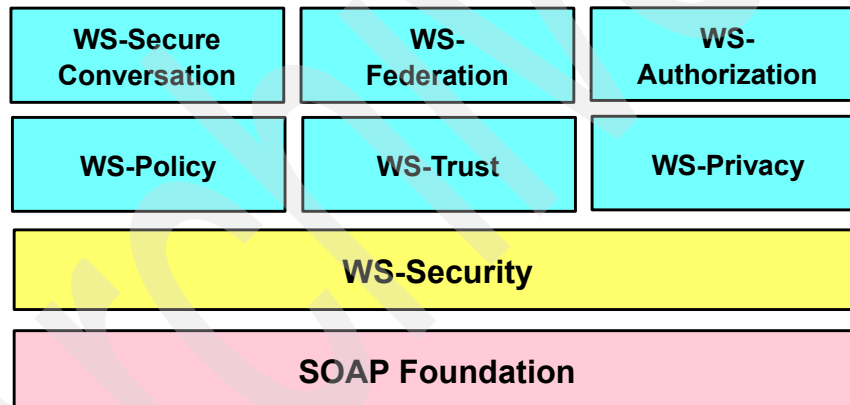


Figure 9-5 WS-Security road map

These specifications include different aspects of Web services security:

- ▶ **WS-Policy**—Describes the capabilities and constraints of the security policies on intermediaries and endpoints (for example, required security tokens, supported encryption algorithms, and privacy rules).
- ▶ **WS-Trust**—Describes a framework for trust models that enables Web services to securely interoperate, managing trusts and establishing trust relationships.
- ▶ **WS-Privacy**—Describes a model for how Web services and requestors state privacy preferences and organizational privacy practice statements.

- ▶ **WS-Federation**—Describes how to manage and broker the trust relationships in a heterogeneous federated environment, including support for federated identities.
- ▶ **WS-Authorization**—Describes how to manage authorization data and authorization policies.
- ▶ **WS-SecureConversation**—Describes how to manage and authenticate message exchanges between parties, including security context exchange and establishing and deriving session keys.

The combination of these security specifications enables many scenarios that are difficult or impossible to implement with today's more basic security mechanisms such as transport securing or XML document encryption.

When to use WS-Security

Here are some simple guidelines as to when WS-Security should be used:

- ▶ Multiple parts of message can be secured in different ways.
You can apply multiple security requirements, such as integrity on the security token (user ID and password) and confidentiality on the SOAP body.
- ▶ Intermediaries can be used.
End-to-end message-level security can be provided through any number of intermediaries.
- ▶ Non-HTTP transport protocol is used.
WS-Security works across multiple transports (also change of transport protocol) and is independent of the underlying transport protocol.
- ▶ User authentication is possible.
Authentication of multiple party identities is possible.

Example of WS-Security

This section provides examples of SOAP messages with WS-Security. Using WS-Security, the authentication mechanism, integrity, and confidentiality can be applied at the message level. In WebSphere Application Server V6.0, there are many options to apply these security mechanisms. In this section, we show the most typical scenarios of each mechanism as an introduction.

As an overview, Figure 9-6 shows an example of Web service security elements when the SOAP body is signed and encrypted.

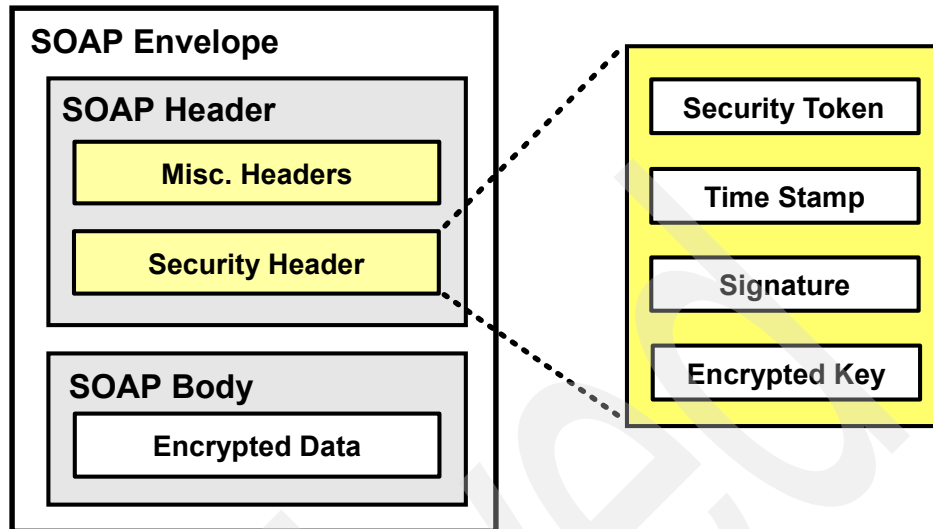


Figure 9-6 SOAP message security with WS-Security

Example 9-1 shows the sample SOAP message without applying WS-Security. As you can see, there is only a SOAP body under the SOAP envelope. Applying WS-Security, the SOAP security header will be inserted under the SOAP envelope.

Example 9-1 SOAP message without WS-Security

```

<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Header/>
  <soapenv:Body>
    <p821:getDayForecast xmlns:p821="http://bean.itso">
      <theDate>2004-11-25T15:00:00.000Z</theDate>
    </p821:getDayForecast>
  </soapenv:Body>
</soapenv:Envelope>
  
```

In the sections that follow, we show examples with WS-Security applied to the SOAP message.

Authentication

In Example 9-2, we show a SOAP message with authentication. As you can see, we have user name and password information as a <UsernameToken> tag in the message.

Example 9-2 SOAP message with authentication

```
<soapenv:Envelope xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Header>
    <wsse:Security soapenv:mustUnderstand="1"
      xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-
      wssecurity-secext-1.0.xsd">
      <wsse:UsernameToken>
        <wsse:Username>David</wsse:Username>
        <wsse:Password
          Type="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-
          username-token-profile-1.0#PasswordText">divaD</wsse:Password>
        </wsse:UsernameToken>
      </wsse:Security>
    </soapenv:Header>
    <soapenv:Body>
      <p821:getDayForecast xmlns:p821="http://bean.itso">
        <theDate>2004-11-25T15:00:00.000Z</theDate>
      </p821:getDayForecast>
    </soapenv:Body>
  </soapenv:Envelope>
```

When the user name token is received by the Web service server, the user name and password are extracted and verified. Only when the user name and password combination is valid, will the message be accepted and processed at the server.

Using the user name token is just one of the ways of implementing authentication. This mechanism is also known as *basic authentication*. Other forms of authentication are digital signature, identity assertion, LTPA token, and custom tokens (identity assertion, LTPA token, and custom tokens are extensions of Version 6.0; refer to “Extensions in WebSphere Application Server V6.0” on page 460).

These other mechanisms can be configured using Rational Application Developer V6.0. More information about using Rational Application Developer to implement authentication can be found in Chapter 21, “Securing Web services” on page 445.

Steps to enable a basic authentication

Here, we describe the steps to configure authentication in the client and server.

Client side

To insert the user name token into a SOAP message, a security token and its token generator must be specified in the client's WS-Security configuration:

- ▶ Specify a security token in the request generator configuration. In case of *basic authentication*, the security token type should be Username. This security token is sent inside the SOAP header to the server.
- ▶ Specify a token generator for the user name token in the request generator configuration. The role of the token generator is to grab the user name and password from the configuration file and generate the user name token with the user name and password. The token generator class for the user name token, UsernameTokenGenerator, is provided by the Web services security runtime as a default implementation.

Server side

To receive the client's user name token, a security token that is required by the server and a token consumer must be specified in the server's WS-Security configuration:

- ▶ Specify a security token that is required by the server. In case of basic authentication, the required security token type is Username (same as in the client configuration).
- ▶ Specify a token consumer in the request consumer configuration. The token consumer receives a security token in the request message and validates it. The token consumer class for the user name token, UsernameTokenConsumer, is provided by the Web services security runtime as a default implementation.
- ▶ Turn on global security in the WebSphere Application Server where the application is deployed.

More information about using Rational Application Developer to implement authentication is provided in Chapter 21, "Securing Web services" on page 445.

Integrity

Integrity is applied to the application to ensure that no one illegally modifies the message while it is in transit. Essentially, integrity is provided by generating an XML digital signature on the contents of the SOAP message. If the message data changes illegally, the signature would no longer be valid.

Example 9-3 shows a sample SOAP message with integrity. Here, the message body part is signed and added to the SOAP security header as signature

information. In WebSphere Application Server Version 6.0, multiple and arbitrary parts of the message can be signed, for example, a message body, security token, and timestamp.

Example 9-3 SOAP message with integrity

```
<soapenv:Envelope xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Header>
    <wsu:Timestamp
      xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-ws
        security-utility-1.0.xsd">
      <wsu:Created>2004-11-26T09:32:31.759Z</wsu:Created>
    </wsu:Timestamp>
    <wsse:Security soapenv:mustUnderstand="1"
      xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-
        wssecurity-secext-1.0.xsd">
      <wsse:BinarySecurityToken
        EncodingType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
          wss-soap-message-security-1.0#Base64Binary"
        ValueType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x5
          09-token-profile-1.0#X509"
        wsu:Id="x509bst_1080660497650146620"
        xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-ws
          security-utility-1.0.xsd">
        MIIBzzCCATigAwIBAgIEQZnQ7DANBgkqhkiG9w0BAQQFADAsMQswCQYDVQQGEwJVUzEMM
        AoGA1UECXMDSUJNMQ8wDQYDVQQDEwZDbG11bnQwHhcNMDDxMTE2MTAwNTMyWhcNMDDUwMj
        EOMTAwNTMyWjAsMQswCQYDVQQGEwJVUzEMMAoGA1UECXMDSUJNMQ8wDQYDVQQDEwZDbG1
        1bnQwZ8wDQYJKoZIhvcNAQEBBQADgY0AMIGJAoGBALrQpIVkTwPc72jTzST3d+fKTYLO
        qMAxq6YvfFweo6ZOH7r3+jAZu880Z7Ru6iMFvajpxrRjshesPnp66binS5vQqfmPAUxyh
        k+IMUAoFZQvG4byEwzDPmymys7M8Q4uZfUK7R/6PocAtwVCssx6zGsNcaaDkGYDC/5qK8+
        95y4ofAgMBAAEwDQYJKoZIhvcNAQEEBQADgYEArrrFUwSVHvWt05eVBImRu8t3cBbPd1Q
        ruB0uCHrNH0G9TwoCJE2JTeZV7+bCjfb17sD8K3mu/WIvS20FoTZWxEGnbxzHQ5aJb7ZC
        GQZ+ffclLCxWj+pG2Eg+BbWR2xStH3NJgPUPmpie16f5fkht16e+CDN9XXYm1qiYhR9Fk
        dI=
      </wsse:BinarySecurityToken>
      <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
        <ds:SignedInfo>
          <ds:CanonicalizationMethod
            Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#">
          <ec:InclusiveNamespaces
            PrefixList="wsse ds xsi soapenc xsd soapenv "
            xmlns:ec="http://www.w3.org/2001/10/xml-exc-c14n#" />
          </ds:CanonicalizationMethod>
          <ds:SignatureMethod
            Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1" />
          <ds:Reference
```

```

        URI="#wssecurity_signature_id_7018050865908551142">
        <ds:Transforms>
            <ds:Transform
                Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#">
                <ec:InclusiveNamespaces
                    PrefixList="xsi soapenc p821 xsd wsu soapenv "
                    xmlns:ec="http://www.w3.org/2001/10/xml-exc-c14n#" />
            </ds:Transform>
        </ds:Transforms>
        <ds:DigestMethod
            Algorithm="http://www.w3.org/2000/09/xmlsig#sha1"/>
        <ds:DigestValue>53Ed8o+4P7XquUGuRVm50AbQ4XY=</ds:DigestValue>
    </ds:Reference>
</ds:SignedInfo>
<ds:SignatureValue>
    SbhHeGPrsoyXbTPdIEcybfqvoEdZ1KiYjjvWZL/dvgqMS6/oi0cdR2Di08VNombj
    Hf9h/EAov+/zvt8i5enw5AziecKr6atLVNG4jKbuNORAhts242bBfybUks4YzduoW
    YcDU9EIXUCMTjRiMbWVvwgc1k4VhTUmKb3jN+yeRA=
</ds:SignatureValue>
<ds:KeyInfo>
    <wsse:SecurityTokenReference>
        <wsse:Reference URI="#x509bst_1080660497650146620"
            ValueType="http://docs.oasis-open.org/wss/2004/01/oasis-
            200401-wss-x509-token-profile-1.0#X509"/>
    </wsse:SecurityTokenReference>
</ds:KeyInfo>
</ds:Signature>
</wsse:Security>
</soapenv:Header>
<soapenv:Body>
    wsu:Id="wssecurity_signature_id_7018050865908551142"
    xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssec
    urity-utility-1.0.xsd">
        <p821:getDayForecast xmlns:p821="http://bean.itso">
            <theDate>2004-11-25T15:00:00.000Z</theDate>
        </p821:getDayForecast>
    </soapenv:Body>
</soapenv:Envelope>

```

A signature is created based on a key that the sender is authorized to have. Unauthorized sniffers do not have this key. When the receiver gets the message, it too creates a signature using the message contents. Only if the two signatures match does the receiver honor the message. If the signatures are different, a SOAP fault is returned to the sender.

Steps to enable integrity

Here, we describe the steps to configure integrity in the client and server.

Client side

To specify integrity for a part of a SOAP message, you have to specify the part that should be signed and the method of signing in the client's WS-Security configuration:

- ▶ Specify the parts of the message that have to be signed in the request generator configuration. The message parts can be specified by predefined keywords or XPath expressions. You can specify multiple parts that require a signature.
- ▶ In the most typical integrity example, a security token is inserted into the SOAP message, which will be used for signature verification by the server. In such an example, a token generator should be specified in the request generator configuration. This token generator's role is to generate a token for signature verification. In Example 9-3, an X.509 certificate token is inserted. The token generator for an X.509 certificate token, `X509TokenGenerator`, is provided by the Web services security runtime as a default implementation.
- ▶ Specify key related-information, which includes the location of the client's key, the type of key, and a password for protecting the key.
- ▶ Specify signing information, which defines how to sign to the specified part. You have to specify some options, such as a signature method algorithm and key-related information. Application Developer helps you to specify these options.
- ▶ If the client expects a response that includes integrity information by the server, the client also has to be configured to validate the integrity of the response message in the response consumer configuration.

Server side

To specify integrity for a part of a SOAP message, you have to specify the part that was signed and the way of verifying the signature in the server's WS-Security configuration:

- ▶ Specify the parts of the message that required a signature in the request consumer configuration. The message parts can be specified by predefined keywords or XPath expressions. You can specify multiple parts that require a signature.
- ▶ In a most typical integrity example, a security token is inserted into the SOAP message, which will be used for the signature verification by the server. In such an example, a token consumer should be specified in the request consumer configuration. This token consumer's role is to receive the token and extract the client certificate for signature verification. The token consumer for an X.509 certificate token, `X509TokenConsumer`, is provided by the Web services security runtime as a default implementation.

- ▶ Specify key-related information, which includes the location of the server's key, the type of key, and a password for protecting the key.
- ▶ Specify signing information, which defines how the signature should be verified. You have to specify some options, such as a signature method algorithm and key-related information.
- ▶ If the server sends a response that includes integrity information by the server, the server also has to be configured to sign the response message in the response generator configuration.

More information about using Application Developer to implement integrity is provided in Chapter 21, "Securing Web services" on page 445.

Confidentiality

Example 9-4 shows a sample SOAP message with confidentiality. Here, the message body part is encrypted and a security header with encryption information is added. In WebSphere Application Server V6.0, multiple and arbitrary parts of the message can be encrypted, for example, a message body and security token.

Example 9-4 SOAP message with confidentiality

```
<soapenv:Envelope xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Header>
    <wsu:Timestamp
      xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-ws
        security-utility-1.0.xsd">
      <wsu:Created>2004-11-26T09:34:50.838Z</wsu:Created>
    </wsu:Timestamp>
    <wsse:Security soapenv:mustUnderstand="1"
      xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-w
        ssecurity-secext-1.0.xsd">
      <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
              ValueType="http://docs.oasis-open.org/wss/2004/01/oasis-
                200401-wss-x509-token-profile-1.0#X509v3SubjectKeyIdentif
                  ier">
              Vniy7MUOXBumPoH1MNBdpiIWOPA=
            </wsse:KeyIdentifier>
          </wsse:SecurityTokenReference>
```

```

        </ds:KeyInfo>
        <CipherData>
          <CipherValue>
            0+2mTsRjU1iNlwANv1kGdzpkRV1GQc5epAT3p5Eg5UNAJ3H3YAX5VrdgMQmj1
            wzdSZLDEzBtcHPJq3c8c0AgmAy9EVdgcXIn/ZeV+80jMDn/HN2HfodYjURtIY
            Bg480SSkot0fy+YpBSXNR/MTfs1HT2H/Mjw/CyIbomWdQZhmE=
          </CipherValue>
        </CipherData>
        <ReferenceList>
          <DataReference>
            URI="#wssecurity_encryption_id_6866950837840688804"/>
          </DataReference>
        </ReferenceList>
      </EncryptedKey>
    </wsse:Security>
  </soapenv:Header>
  <soapenv:Body>
    <EncryptedData
      Id="wssecurity_encryption_id_6866950837840688804"
      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#tripledes-cbc"/>
      <CipherData>
        <CipherValue>
          OvLek01buZhFB11BNL4Kos195YHwYw0kSbMxkbI2pk7n117g0prPS2Ba2hyrXHABGQVmo
          sWpgqt+zijCPHUQCMmmm3qgFraK11DPMmwP94HvgxlgBmPw1Unt+WM4aKLNrHDnwwcQX5
          R07KT+fhFp4wxFEABwfHqzvTGNK3xRwJE=
        </CipherValue>
      </CipherData>
    </EncryptedData>
  </soapenv:Body>
</soapenv:Envelope>

```

Confidentiality is the process in which a SOAP message is protected so that only authorized recipients can read the SOAP message. Confidentiality is provided by encrypting the contents of the SOAP message using XML encryption. If the SOAP message is encrypted, only a service that knows the key for confidentiality can decrypt and read the message.

Steps to enable confidentiality

Here, we describe the simplified steps to enable confidentiality in the client and server.

Client side

To specify confidentiality for a part of a SOAP message, you have to specify the parts that should be encrypted and the method of encryption in the client's WS-Security configuration:

- ▶ Specify the parts of the message that have to be encrypted in the request generator configuration. The message parts can be specified by predefined keywords or XPath expressions. You can specify multiple parts that require encryption.
- ▶ Specify key-related information, which includes the location of the encryption key, type of key, and a password for protecting the key.
- ▶ Specify encryption information, which defines how to encrypt to the specified part. You have to specify options, such as an encryption method algorithm and key-related information. Application Developer helps you to specify these options.
- ▶ If a client expects a response that includes confidentiality by the server, the client also has to be configured to decrypt the server's encryption of the response message in the response consumer configuration.

Server side

To specify confidentiality required for part of a SOAP message, you have to specify the encrypted parts and the method of decryption in the server's WS-Security configuration:

- ▶ Specify the parts of the message that require decryption in the request consumer configuration. The message parts can be specified by predefined keywords or XPath expressions. You can specify multiple parts that require encryption.
- ▶ Specify key-related information, which includes the location of the decryption key, the type of key, and a password for protecting the key.
- ▶ Specify encryption information, which defines how to decrypt the specified part. You have to specify options, such as a encryption method algorithm and key-related information.
- ▶ For the message of Example 9-4, a token consumer should be specified in the request consumer configuration. This token consumer's role is to receive information for message decryption. The token consumer for an X.509 certificate token, X509TokenConsumer, is provided by the Web services security runtime as a default implementation.
- ▶ If a server sends a response that includes confidentiality, the server also has to be configured to encrypt of the response message in the response generator configuration.

More information about using Application Developer to implement confidentiality is provided in Chapter 21, “Securing Web services” on page 445.

Transport-level security

HTTP, the most used Internet communication protocol, is currently also the most popular protocol for Web services. HTTP is an inherently insecure protocol, because all information is sent in clear text between unauthenticated peers over an insecure network. It belongs to the group of protocols, such as SMTP, Telnet, and FTP, that were designed in the earlier stages of the Internet, when security seemed not to be an issue, and will eventually be replaced by transfer protocols that allow authentication and encryption.

To secure HTTP, transport-level security can be applied. Transport-level security is a well-known and often used mechanism to secure HTTP Internet and intranet communications. Transport-level security is based on Secure Sockets Layer (SSL) or Transport Layer Security (TLS) that runs beneath HTTP.

HTTPS allows client-side and server-side authentication through certificates, which have been either self-signed or signed by a certification agency.

For Web services bound to the HTTP protocol, HTTPS/SSL can be applied in combination with message-level security (WS-Security).

Unlike message-level security, HTTPS encrypts the *entire* HTTP data packet. There is no option to apply security selectively on certain parts of the message. SSL and TLS provide security features including authentication, data protection, and cryptographic token support for secure HTTP connections.

We do not cover HTTPS in more detail in this document. HTTPS is a well-known and well-documented protocol.

SOAP/HTTP transport-level security

Although HTTPS does not cover all aspects of a general security framework, it provides a security level regarding party identification and authentication, message integrity, and confidentiality. It does not provide authentication, auditing, and non-repudiation. SSL cannot be applied to other protocols, such as JMS. To run HTTPS, the Web service port address must be in the form `https://`.

Even with the WS-Security specification, SSL should be considered when thinking about Web services security. Using SSL, a point-to-point security can be achieved (Figure 9-7).

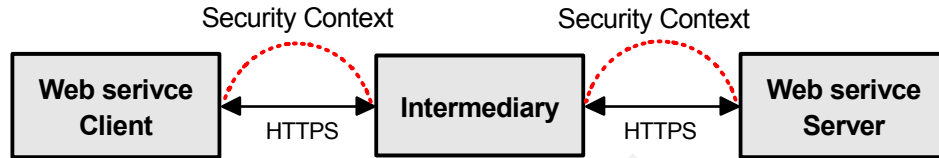


Figure 9-7 Point-to-point security with HTTPS

When to use transport-level security

Here are a few simple guidelines to help decide when transport-level security should be used:

- ▶ No intermediaries are used in the Web service environment.
With intermediaries, the entire message has to be decrypted to access the routing information. This would break the overall security context.
- ▶ The transport is only based on HTTP.
No other transport protocol can be used with HTTPS.
- ▶ The Web services client is a stand-alone Java program.

WS-Security can only be applied to clients that run in a J2EE container (EJB container, Web container, application client container). HTTPS is the only option available for stand-alone clients.

Summary

Web services technology enables a loosely coupled, language-neutral, platform-independent way of linking applications within organizations, across enterprises, and across the Internet. To achieve the target, however, it is essential for Web services to provide a sufficient level of security to support business transactions. Ensuring the integrity, confidentiality, and security of Web services through the application of a comprehensive security model is critical, both for organizations and their customers.

With WebSphere Application Server V6.0, Web services security can be applied as transport-level security and as message-level security. Highly secure client/server designs can be architected using these security levels.

Refer to Chapter 21, “Securing Web services” on page 445 for a detailed example of how to apply the different WS-Security features to a sample Web services application.

More information

Because Web services security is a quickly evolving field, it is essential for developers and designers to regularly check for recent updates. In this section, we provide some of the most important entry points for your exploration.

The XML Signature workgroup home page is available at:

<http://www.w3.org/Signature/>

The XML Encryption workgroup home page is available at:

<http://www.w3.org/Encryption/>

The WS-Security specification 1.0 is available at:

<http://www.ibm.com/developerworks/library/ws-secure/>

The white paper about the Web services security road map is available at:

<http://www.ibm.com/developerworks/webservices/library/ws-secmap/>

OASIS WS-Security 1.0 and token profiles is available at:

http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss

There are several commercial and non-commercial information sources that cover more general subjects, such as SSL encoding and the HTTPS protocol.

Web services interoperability

One of the key benefits of Web services technology, and the reason that it has gained widespread attention and adoption, is because of its promise of interoperability. In part because they are based on open, vendor-neutral standards, and because, for the very first time, *all* major vendors are recognizing and providing support for Web services, they hold the promise of complete cross-platform, cross-language interoperability.

Despite this promise and great potential, there is still a need to include this chapter on the subject of interoperability.

In this chapter, we explore interoperability of Web services between different platforms and implementation languages. We specifically discuss the work of the Web Services Interoperability Organization, introduced in Chapter 2, “Web services standards” on page 13.

In the enterprise Java environment, a number of SOAP engines exist. We discuss the evolution of SOAP engines in the WebSphere product line and how they interoperate.

There is also a certain level of disconnect between Java product vendors and the Microsoft .NET Framework. As a result, we discuss the specific areas of Web services where Java and .NET interoperability remains problematic.

Definition

Let us start with a definition of interoperable.

Definition: *Interoperable* means suitable for and capable of being implemented in a neutral manner on multiple operating systems and in multiple programming languages.

Web Services Interoperability Organization

An open industry organization, the *Web Services Interoperability Organization*, or *WS-I*, has been created to guide, recommend practices, and provide supporting resources to promote Web services interoperability across platforms, operating systems, and programming languages.

WS-I was founded specifically with the intent of facilitating interoperability of Web services between different vendor products and to clarify where gaps or ambiguities exist between the various standards and standards bodies.

There are various standards organizations—W3C, OASIS, and IETF, for example—all working to publish Web services standards. Each standard has been developed to address a specific Web services problem set. Developers charged with building a Web services solution are required to discover, interpret, and apply the rules of multiple Web services standards. Even within an enterprise, multiple development teams will likely interpret and apply the rules for the group of standards differently.

WS-I has formulated the concept of *profiles* to solve this problem and reduce complexity. Profiles consist of implementation guidelines for how related Web services specifications should be used together for best interoperability. To date, WS-I has finalized the Basic Profile V1.1, Attachments Profile V1.0, and Simple SOAP Binding Profile V1.0. Work on a Basic Security Profile is underway.

In addition, WS-I has also produced:

- ▶ Requirements in the form of use cases and usage scenarios
- ▶ A set of multivendor sample applications that implement the requirements and demonstrate the power of interoperability
- ▶ Testing tools that enable software development teams to test the conformance of a Web service implementation and its artifacts to the WS-I profiles

The WS-I Organization Web site is available at:

<http://www.ws-i.org/>

WS-I Basic Profile V1.1 and Simple SOAP Binding Profile V1.0

The WS-I Basic Profile has been split into two separate profiles as of V1.1. Conformance to WS-I Basic Profile V1.1 plus conformance to the Simple SOAP Binding Profile V1.0 is roughly equivalent to a combined conformance claim of WS-I Basic Profile V1.0 plus the published errata. In other words:

$$\begin{aligned} \text{Basic Profile V1.1} + \text{Simple SOAP Binding Profile V1.0} \\ = \text{Basic Profile V1.0} + \text{errata} \end{aligned}$$

The WS-I Basic Profile begins with a basis of the following 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

The Profile adds constraints and clarifications to those base specifications with the intent to promote interoperability. Some of the key constraints include:

- ▶ 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

WS-I Attachments Profile V1.0

The WS-I Attachments Profile V1.0 specifies guidelines for interoperability of Web service messages that contain attachments. Specifically, the Attachments Profile guides the use of SOAP messages that contain attachments using the

SOAP Messages with Attachments (SwA) specification. The SOAP Messages with Attachments specification is a W3C Note.

In general, the Java industry has adopted the SOAP Messages with Attachments (SwA) specification. The SAAJ (SOAP with Attachments API for Java) API models the MIME message format for SOAP as specified by SwA.

SwA is not without its problems however. The SwA solution breaks the Web services model to a certain extent. Among other issues, SwA does not work with WS-Security at this time. Because of this fundamental problem with SwA, W3C is moving in the direction of Message Transmission Optimization Mechanism (MTOM).

Another fundamental problem for interoperability is that Microsoft (despite being a founding WS-I member) is not planning on supporting SwA on any of its platforms (and therefore the WS-I Attachments Profile V1.0). For a detailed discussion of possible implementations for passing attachments to or from Microsoft platforms, refer to the recent article *Web Services, Opaque Data, and the Attachments Problem*, available at:

<http://msdn.microsoft.com/webservices/default.aspx?pull=/library/en-us/dnwebsrv/html/opaquedata.asp>

WS-I tools

WS-I provides tools that test Web service artifacts for compliance against the WS-I profiles. The testing tools use a man-in-the-middle approach to capture SOAP/HTTP messages. They also use WSDL and UDDI artifacts. Figure 10-1 shows the testing tool architecture.

These tools can be downloaded and used from the WS-I Web site. Alternatively, there are similar tools built into Rational Application Developer V6.0; see “Interoperability tools in Application Developer” on page 428 for a description of these tools.

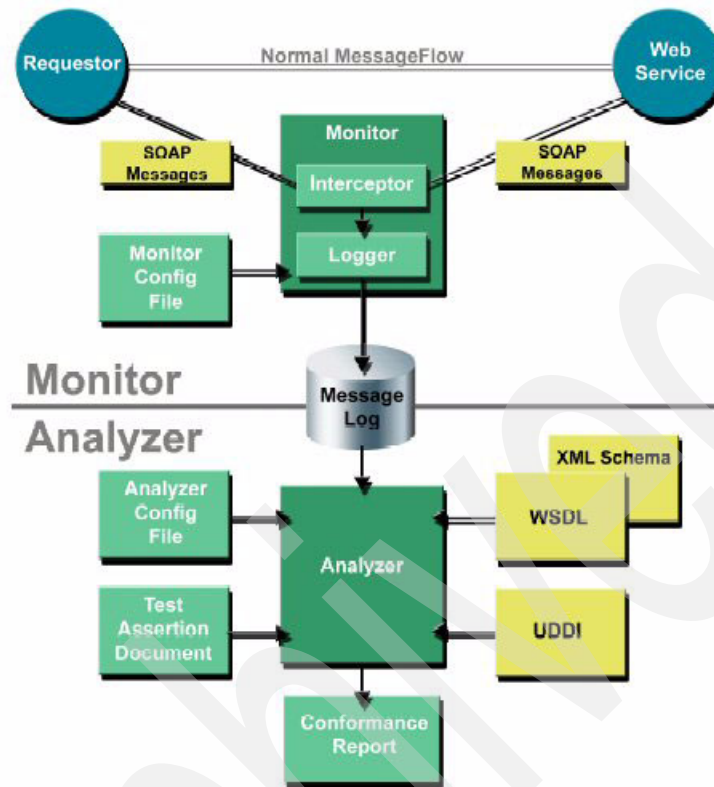


Figure 10-1 WS-I testing tools architecture

WS-I conformance claims

The WS-I Basic Profile V1.1 allows Web service artifacts to include a claim of conformance when they have been tested to conform to WS-I profiles. The standard refers to this mechanism in Section 2.4, located at:

http://www.ws-i.org/Profiles/BasicProfile-1.1-2004-08-24.html#conformance_claims

Currently, there is no good industry tooling support for this mechanism, and it is a best practice not to use it at this time.

Rational Application Developer V6.0 does not provide tooling support to add conformance claims to Web service artifacts, although this can be done manually. Web services that are required to interoperate with the Microsoft .NET Framework should not use conformance claims at this time (see “WS-I conformance claims” on page 202).

WebSphere interoperability

Table 10-1 provides a quick reference to the Web service runtimes and features relevant to Web services for each of the major WebSphere Application Server releases.

Table 10-1 WebSphere Application Server Web service support

WebSphere Application Server	Web service runtime	Web service features
Version 4.0 Version 5.0	IBM SOAP (based on Apache SOAP)	Not WS-I compliant
Version 5.0.2 Version 5.1	IBM WebSphere Apache Axis V1.0 IBM SOAP	WS-I Basic Profile V1.0 JAX-RPC V1.0 JSR109 V1.0 SAAJ V1.1 UDDI V2.0 WS-Security (OASIS Draft 13) SOAP/JMS support Web services caching Web services performance monitoring
Version 6.0	IBM WebSphere Apache Axis V1.0 IBM SOAP (deprecated)	WS-I Basic Profile V1.1 JAX-RPC V1.1 JSR109 V1.1 SAAJ V1.2 UDDI V3.0 WS-Security V1.0 WS-AtomicTransactions WS-Coordination JAXR support Multiple protocol/encodings (SOAP/JMS, EJB) Web services caching Web services performance monitoring

In general, Web services implemented in WebSphere Application Server are interoperable with each other, with the following exceptions:

- ▶ Web services implemented using the IBM SOAP engine are not WS-I compliant and, therefore, are unlikely to interoperate.
- ▶ Web services that implement WS-Security will not interoperate between WebSphere Application Server Versions 5.0.2/5.1 and Version 6, because the former implement a draft version of the specification that is not message-level compatible with WS-Security V1.0. Refer to “WS-Security support” on page 205 for more information.

Interoperability with .NET

The following discussion is based on Microsoft .NET Framework V1.1¹ and WSE (Web Services Enhancements) V2.0.

WS-I defines conformance in terms of a Web service instance and artifacts (for example, WSDL, SOAP/HTTP messages, UDDI entries). Where possible, the profile places requirements on artifacts (for example, WSDL descriptions and SOAP messages) rather than on the producing or consuming software's behaviors or roles. Artifacts are concrete, making them easier to verify.

This section discusses areas of the WS-I profiles that the Microsoft .NET Framework does not support, or does not strictly adhere to. It also discusses areas of functionality where interoperability issues still exist or where care should be taken.

RPC/literal WSDL

Although the WS-I Basic Profile V1.0 requirement allows for both document/literal and RPC/literal encoding styles, .NET does not provide explicit support for consuming a Web service that uses an RPC/literal binding.

Because the RPC/literal SOAP message format is a subset of the document/literal SOAP message format, it is possible (and the only approach currently available) to create a new WSDL description that allows interoperability. The Microsoft article *RPC/Literal and Freedom of Choice*, describes this issue:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnwebsrv/html/rpc_literal.asp

Tools are also available for converting RPC/literal WSDL files to wrapped document/literal at:

http://www.gotdotnet.com/team/tools/web_svc/default.aspx

Important: There is a restriction when converting RPC/literal WSDL to wrapped document/literal—it is not possible to support *operation overloading*. Wrapped document/literal requires the element name to be the same as the operation name. Because the type information of the arguments is not available in a SOAP document/literal message, the Web service provider must use a unique element name to map the message to an implementation.

¹ Microsoft .NET Framework V2.0 is currently in Beta1 phase and is not due for general release until 2Q2005.

WS-I conformance claims

WS-I Basic Profile V1.1 allows Web service artifacts to make conformance claims, as described in:

http://www.ws-i.org/Profiles/BasicProfile-1.1-2004-08-24.html#conformance_claims

The Microsoft .NET Framework tools fail to process a WSDL description that contains conformance claims. The `wsi:Claim` elements have to be manually removed from the WSDL.

We recommend that this mechanism is not used for Web services that have to interoperate with the Microsoft platform until this issue has been fixed.

SwA not supported

As described in “WS-I Attachments Profile V1.0” on page 197, Microsoft does not (and will not in the future) support the SOAP with Attachments specification.

Until further work is done on the MTOM specification, the current best practice when exchanging attachments is to transfer them “out of band” (for example, by passing a URI reference to the attachment). For MTOM, refer to:

<http://www.w3.org/TR/soap12-mtom/>

For more information about sending attachments in general, see “Using attachments” on page 316.

The MSDN article *Web Services, Opaque Data, and the Attachments Problem*, by Matt Powell, provides an excellent discussion of the options available (as of June 2004):

<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnwebsrv/html/opaquedata.asp>

WSDL import statements

In earlier versions of the Microsoft .NET Framework, import statements were not handled consistently by the .NET `wsdl.exe` tool. Some problems might still exist:

- ▶ The `wsdl:import` command works, but cannot be used to import schema definitions. Use an `xsd:import` command instead.
- ▶ The `xsd:import` command does not always work. Try using absolute URIs instead of relative URIs.

We recommend that WSDL definitions that use import statements are tested early for interoperability. Alternatively, keep WSDL definitions in a single file.

Mandatory header handling

The WS-I Basic Profile V1.1 has the following requirement:

R1025—A RECEIVER MUST handle messages in such a way that it appears that all checking of mandatory headers is performed before any actual processing.

By default, this is not the case with the Microsoft .NET Framework. A .NET Web service must be manually coded to check for the existence of mandatory headers before the main implementation of the Web service.

UTF-16 WSDL

The Microsoft .NET Foundation tools do not allow the consumption of Web services where the WSDL has been encoded using UTF-16 character encoding.

This is a requirement of WS-I Basic Profile V1.1:

R4003—A DESCRIPTION MUST use either UTF-8 or UTF-16 encoding.

Microsoft recommends either manually converting the WSDL to UTF-8², or using Notepad.exe to perform the conversion by opening the document and using the *Save As* dialog, saving in ANSI format.

User exception handling

When developing a Microsoft .NET Web service implementation, the platform does not natively allow for user-defined exceptions to be defined. The .NET platform does not add SOAP fault messages in the WSDL files, and all exceptions thrown by the Web Service are managed as simple SOAP server faults.

In addition, user-defined exceptions generated from WebSphere (or other platforms), or more correctly, types that are bound to fault messages in WSDL, are not represented as concrete classes in .NET. They are thrown as `SoapExceptions`, and the .NET code then has access to the fault code and detail as an XML document representation, rather than as an exception object.

This different implementation of exception management in the WSDL description, however, does not impact the interoperability between the two platforms.

² This assumes that extended characters are not used.

Object inheritance

When creating a Web service using the bottom-up approach, that is, creating a Web service using a Java artifact (for example, a JavaBean or EJB), there are some minor outstanding issues when (value type) object inheritance is used.

When Rational Application Developer generates a WSDL description from the Java interface, it must explicitly define all possible subclasses that may pass across the interface so that the Web service consumer is aware of them. The Application Developer GUI does not expose an option to add these extra classes to the WSDL definition, although the command-line tools do.³ There is an excellent article describing the issues involved, by Kyle Brown, available on developerWorks, *Web Services Value Type Inheritance and Interoperability*:

http://www.ibm.com/developerworks/websphere/techjournal/0401_brown/brown.html

As mentioned in the article, the WS-I Basic Profile V1.1 is silent on the issue of value type inheritance; however, IBM WebSphere Application Server and Microsoft .NET implementations will interoperate with this scenario.

Null and empty array handling

Using arrays is tricky when working with boundary conditions. In .NET, if an empty array is serialized, only the header XML element is created. If a null array is serialized, it is completely ignored and nothing is serialized as XML.

```
<Bean>                                <!-- example empty array -->
  <array/>
</Bean>

<Bean></Bean>                         <!-- example nil array -->
```

Conversely, JAX-RPC defines that an array should be mapped to a JavaBean. This created inconsistencies between .NET and WebSphere Application Server V5.x.

For WebSphere Application Server V6.0, the Java2WSDL and WSDL2Java emitters have been changed to use maps similar to .NET, and interoperability is achieved.

³ The *Web Services Value Type Inheritance and Interoperability* article uses WebSphere Studio Application Developer V5.1 and WebSphere Application Server V5.02, but is equally applicable for Rational Application Developer V6.0 and WebSphere Application Server V6.0.

Null primitives and dates

The Microsoft .NET Framework does not properly handle nillable primitive types. For example, we have a WSDL type defined as:

```
<element name="x" type="xsd:int" nillable="true" />
```

This type maps to a simple `int` type that cannot handle a null value. Nillable complex types are fine because they map to C# objects, which can be null.

In contrast, a JAX-RPC implementation will map this to a `java.lang.Integer` type (when it is declared as `nillable="true"`).

In addition, types defined as `java.util.Date` or `java.util.Calendar` in Java will be mapped to a `System.DateTime` data type in .NET. If the WSDL declares this as a nillable value, the Microsoft .NET platform will throw a `System.Format.Exception` when exposed to a null value. This is because `System.DateTime` is a `System.ValueType` and therefore cannot have a null value.

It is a recommended best practice to avoid nillable primitive and date types.

WS-Security support

Both IBM WebSphere Application Server V6.0 and Microsoft Web Service Extensions (WSE) V2.0 implement the finalized OASIS WS-Security V1.0 standard and will interoperate.

The WS-Security V1.0 wire format has changed over time and is not compatible with previous WS-Security drafts. Also, interoperability between implementations based on previous drafts and V1.0 is not possible.

WebSphere Application Server V5.0.2 and V5.1 are based on the April 2002 draft specification of WS-Security. Microsoft WSE V1.0 is also based on a draft specification. Patches are available for WebSphere Application Server V5.x to interoperate between the two.

WebSphere Application Server V6.0 and Microsoft WSE V2.0 both support the approved WS-Security V1.0 standard and can interoperate. Due to the wire format difference, WS-Security V1.0 implementations will not interoperate with draft WS-Security implementations.

Representation of arrays in WSDL

WS-I Basic Profile V1.1 defines the following requirement:

R2112—In a DESCRIPTION, elements SHOULD NOT be named using the convention ArrayOfXXX.

When a .NET Web service is implemented in a bottom-up manner, for example, a C#.NET class containing Web service annotations, the WSDL that is generated by .NET tooling will create an `xsd:complexType` definition for the array. For example, the following C#.NET method declaration will generate the WSDL definition shown in Figure 10-2:

```
[WebMethod]
public void stringArrayExample(String arg1, String [] lotsOfStrings)
```

```
<s:element name="stringArrayExample">
  <s:complexType>
    <s:sequence>
      <s:element minOccurs="0" maxOccurs="1" name="arg1" type="s:string" />
      <s:element minOccurs="0" maxOccurs="1" name="lotsOfStrings"
        type="s:ArrayOfString" />
    </s:sequence>
  </s:complexType>
</s:element>
<s:complexType name="ArrayOfString">
  <s:sequence>
    <s:element minOccurs="0" maxOccurs="unbounded" name="string"
      nillable="true" type="s:string" />
  </s:sequence>
</s:complexType>
```

Figure 10-2 WSDL generated by .NET for a string array

Because the WS-I Basic Profile requirement reads SHOULD NOT rather than MUST NOT, this is compliant WSDL. Unfortunately, it makes the WSDL less elegant and generally has the undesirable side-effect of causing an `ArrayOfString` class to be generated in a Java client⁴, rather than mapping to a `String[]` type. Because the .NET WSDL is still WS-I compliant, this behavior is noted as an inconvenience and does not affect interoperability.

⁴ WebSphere Application Server V6.0 emitters recognize this special case and generate a Java client with a `String[]` parameter when the IBM WebSphere runtime is used.

Summary

Overall interoperability between Web service implementations is good as long as the boundaries defined by the standards and WS-I profiles are adhered to. Interoperability will continue to improve as tooling and platforms mature.

We have attempted to provide resources, tools, and information that will aid the development community to design, implement, and deploy interoperable Web services. Although there are no guarantees of trouble-free integration, we hope that the information contained in this chapter will provide for a much more pleasant experience.

More information

For a description of the tools available in Rational Application Developer V6.0 that aid the production of WS-I compliant Web services, see Chapter 20, “Web services interoperability tools and examples” on page 427. This chapter also contains interoperability examples.

The WS-I Organization Web site, which includes profiles, sample application implementations, and compliance testing tools, is available at:

<http://www.ws-i.org/>

The WS-I Basic Profile V1.1 deliverable is available at:

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

Information about the IBM representation and contribution to the WS-I Organization is available at:

<http://www.ibm.com/developerworks/webservices/wsi/>

The W3C SOAP Messages with Attachments (SwA) specification is available at:

<http://www.w3.org/TR/SOAP-attachments>

A more in-depth discussion of Web services interoperability between IBM WebSphere and Microsoft .NET platforms can be found in the IBM Redbook *WebSphere and .NET Interoperability Using Web Services*, SG24-6395.

Another good source for .NET interoperability is the Microsoft paper *Building Interoperable Web Services (WS-I Basic Profile 1.0)*, available at:

<http://download.microsoft.com/download/8/d/8/d8d28b77-2ab2-4c8e-b389-e23684f12035/WSI-BP.pdf>

Web services architectures

This chapter focuses on some of the architectural concepts that need to be considered on a Web services project. We define and discuss *service-oriented architecture* (SOA) and the relationship between SOAs and Web services.

From a different perspective, we take a look at the Web services protocol stack to give an overview of the standards that make up a basic Web service, as well as the additional current and forthcoming standards that build on top of basic Web service functionality to provide *quality of service* (QoS) characteristics, such as security, reliability, and transaction and operational management of Web services.

Web services are not an architecture as such—they are a collection of technologies. We look at a range of message exchange patterns that can be implemented using Web services and how they can affect a Web services project's architecture.

To complete our architectural view of Web services, we describe the SOAP processing model and how the use of intermediary SOAP nodes and Web service gateways can be used.

Service-oriented architecture

Definition of a service-oriented architecture: A service-oriented architecture consists of a set of business-aligned services that collectively fulfill an organization's business process goals and objectives. These services can be choreographed into composite applications and can be invoked through Internet-based open standards and protocols.

The characteristics and requirements of a service-oriented architecture (SOA) have been introduced in Chapter 1, “Web services introduction” on page 3. To provide the minimum requirements for an SOA, the components shown in Figure 11-1 must exist in the organization:

- ▶ Services
- ▶ Enterprise service bus (ESB)
- ▶ Service directory—An organization-level WSDL repository
- ▶ Internet gateway—Optionally enables internal services to be exposed to the Internet
- ▶ Business process choreography tools—Optionally provide service composition facilities

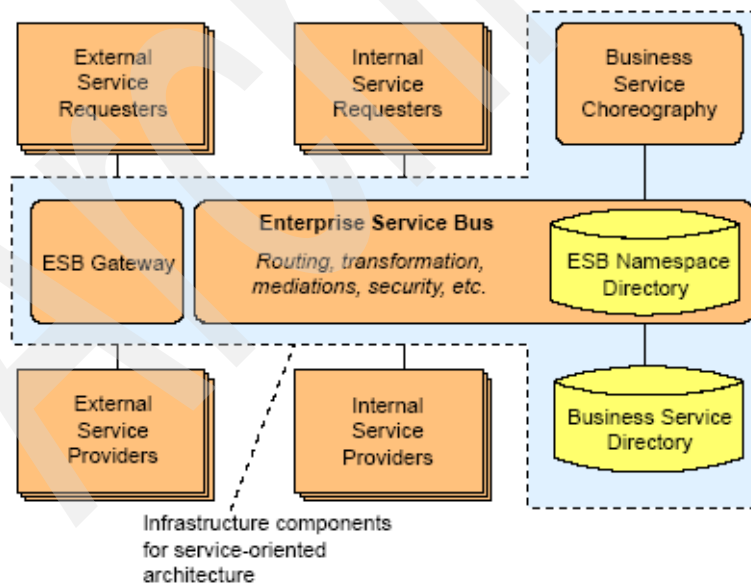


Figure 11-1 Components of a service-oriented architecture

The existence of these components at an enterprise or organizational level transforms a collection of essentially independent point-to-point services into an architecture that exhibits the characteristics—loose coupling, location transparency, component reuse, service composition—of a service-oriented architecture.

A more detailed discussion of service-oriented architectures and enterprise service buses are outside the scope of this redbook. We encourage you to refer to these additional resources:

- ▶ An excellent series of articles titled *Migrating to a service-oriented architecture* discusses the various virtues of service-oriented architectures and is available on the IBM developerWorks Web site:
<http://www.ibm.com/developerworks/library-combined/.backup/ws-migratesoa>
- ▶ A useful reference site for resources on the subject of SOAs is available at:
<http://www.ibm.com/software/solutions/webservices/resources.html>
- ▶ Also of interest are the IBM Redbooks:
 - *Patterns: Implementing an SOA Using an Enterprise Service Bus*, SG24-6346
 - *Patterns: Service-Oriented Architecture and Web Services*, SG24-6303

Enterprise service bus

Although the concept of a service-oriented architecture does not necessarily map directly to a software product, the concept of an enterprise service bus certainly does. Vendors have released a number of enterprise service bus products based on existing enterprise application integration (EAI) products.

The functions of an enterprise service bus are:

- ▶ Communication middleware supporting a variety of communication paradigms, qualities of service (such as security, guaranteed delivery, performance, transactional), platforms, and protocols
- ▶ The ability to transform message formats between consumer and provider
- ▶ The ability to convert the transport protocol between consumer and provider

Web services versus service-oriented architectures

The service-oriented architecture has been used under various guises for many years. It can and has been implemented using a number of different distributed computing technology, such as CORBA or messaging middleware. The effectiveness of service-oriented architectures in the past has always been limited by the ability of the underlying technology to interoperate across the enterprise.

Web services technology is an ideal technology choice for implementing a service-oriented architecture:

- ▶ Web services are standards based. Interoperability is a *key business advantage* within the enterprise and is *crucial* in B2B scenarios.
- ▶ Web services are widely supported across the industry. For the very first time, *all* major vendors are recognizing and providing support for Web services.
- ▶ Web services are platform and language agnostic—there is no bias for or against a particular hardware or software platform. Web services can be implemented in any programming language or toolset. This is important because there will be continued industry support for the development of standards and interoperability between vendor implementations.
- ▶ This technology provides a migration path to gradually enable existing business functions as Web services are needed.
- ▶ This technology supports synchronous and asynchronous, RPC-based, and complex message-oriented exchange patterns.

Conversely, there are many Web services implementations that are not a service-oriented architecture. For example, the use of Web services to connect two heterogeneous systems directly together is not an SOA. These uses of Web services solve real problems and provide significant value on their own. They may form the starting point of an SOA.

In general, an SOA has to be implemented at an enterprise or organizational level in order to harvest many of the benefits.

For more information about the relationship between Web services and service-oriented architectures, or the application of IBM Patterns for e-business to a Web services project, refer to *Patterns: Service-Oriented Architecture and Web Services*, SG24-6303.

Web services protocol stack

Like most other distributed system technologies, Web services are built from a core foundation that provides basic communications-level interoperability. Built on top of these foundations are additional aspects or services that may or may not be required in any given project, depending on the nature of the project and other factors such as the non-functional requirements and architectural constraints. These additional facets are layered on top of each other, or wrapped around each other, much like the layers of an onion. Figure 11-2 is a diagram of the Web services *onion*, showing how the different aspects rely on each other.

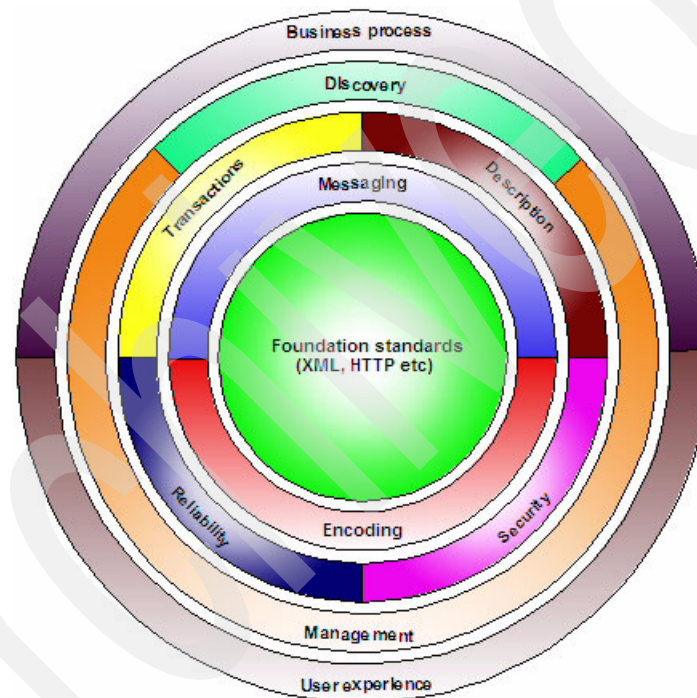


Figure 11-2 Web services "onion"

Figure 11-3 attempts to provide a snapshot of the rapidly changing landscape of Web services-related standards and specifications. It is not intended to be a strictly correct stack diagram—it just attempts to show the various standards efforts in terms of the general category to which they belong. For example, the W3C SOAP 1.2 standard depends on both the W3C XML InfoSet standard and, generally, the IETF RFC2616 HTTP/1.1 standard. Conversely, the transaction standards are in no way dependant on the reliability layer.

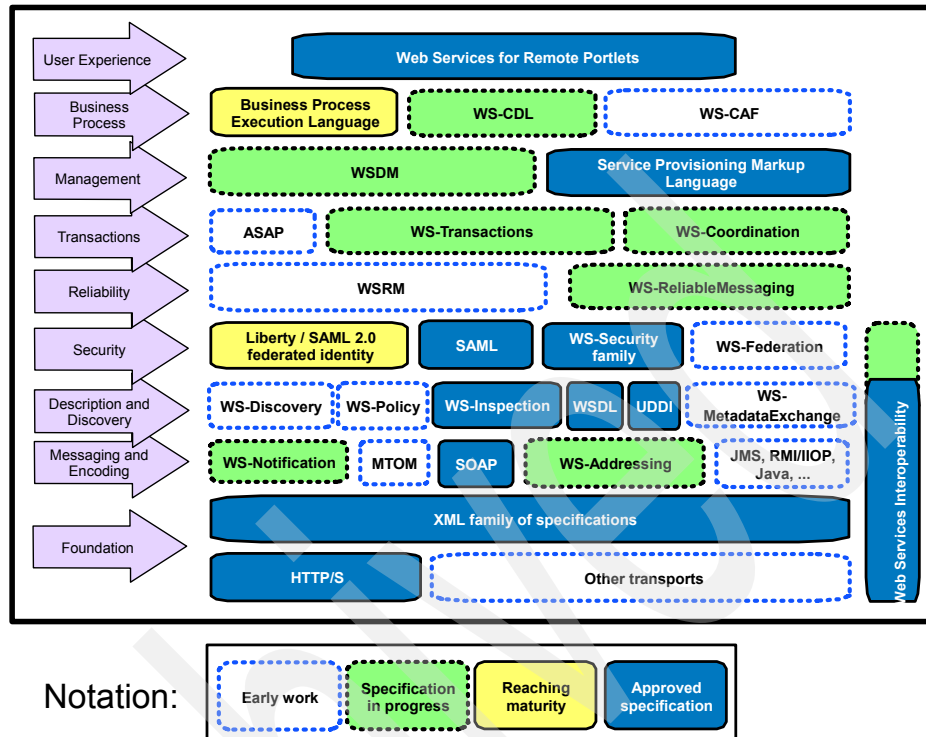


Figure 11-3 Web services protocol stack

We provide an overview of the standards that relate to Web services and the organizations that administer them in Chapter 2, “Web services standards” on page 13.

Detailed information about the core Web services standards, SOAP, WSDL, and UDDI, can be found in Chapter 3, “Introduction to SOAP” on page 39, Chapter 4, “Introduction to WSDL” on page 69, and Chapter 7, “Introduction to UDDI” on page 123, respectively.

Given the current momentum behind Web services and the pace at which standards are evolving, it is also useful to refer to an online compilation of Web services standards. An online compilation is available on the IBM developerWorks Web site at:

<http://www.ibm.com/developerworks/views/webservices/standards.jsp>

Message exchange patterns

Web services is a very flexible message-oriented concept—there are no restrictions and very little documented in the standards regarding message exchange patterns (sometimes referred to as interaction patterns).

It is important to remember that a Web service is not constrained to use SOAP over HTTP/S as the transport mechanism. In fact, the definition of a Web service, found at <http://www.w3.org/TR/ws-arch/#what-is>, according to the W3C Web Services Architecture Working Group is as follows.

Definition: “A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, *typically conveyed using HTTP with an XML serialization* in conjunction with other Web-related standards.”
[Emphasis added.]

To a certain extent, a message exchange pattern is implied by the underlying transport protocol, or perhaps put another way, some transport protocols are better adapted to some message exchange patterns than others. For example, when SOAP/HTTP is used as the SOAP binding, a response is implicitly returned for each request. An asynchronous transport such as SOAP/JMS is probably more proficient at handling a publish-subscribe message exchange pattern.

The remainder of this section discusses some of the common message exchange patterns in the context of Web services and how they can affect your architectural decisions regarding component placement and SOAP bindings:

- ▶ One-way
- ▶ Asynchronous two-way
- ▶ Request-response
- ▶ Workflow-oriented
- ▶ Publish-subscribe
- ▶ Composite

One-way

In this very simple message exchange pattern, messages are pushed in one direction only. The source does not care whether the destination accepts the message (with or without error conditions). The service provider implements a Web service to which the requestor can send messages.

Depending on your reliability and interoperability requirements and your SOAP implementation's capabilities, it might be more appropriate to use a messaging-based transport such as IBM WebSphere MQ. This decouples the consumer component from the producer component, thereby increasing the availability of the consumer component in a distributed environment. It also provides fire-and-forget capabilities so that the requestor component does not have to wait for the provider to process the message (Figure 11-4).

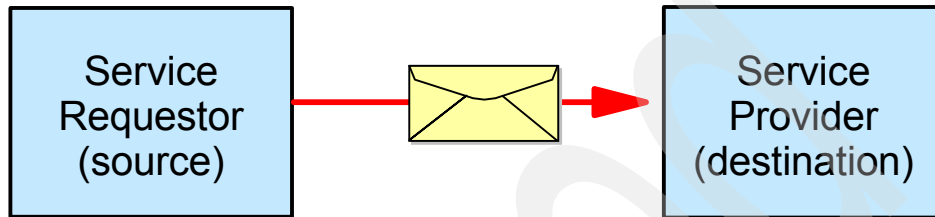


Figure 11-4 One-way message exchange pattern

An example of a one-way message exchange pattern is a resource monitoring component. Whenever a resource changes in an application (the source), the new value is sent to a monitoring application (the destination).

Asynchronous two-way¹

In this message exchange pattern (Figure 11-5), the service requestor expects a response, but the messages are asynchronous in nature (for example, where the response might not be available for many hours). Both sides must implement a Web service to receive messages. In general, the Web service provided by the service 2 provider component has to relate a message it receives to the corresponding message that was sent by the service 1 requestor component.

Technically, this message exchange pattern is the same as one-way with the additional requirement that there has to be a mechanism to associate response messages with their corresponding request message. This can be done at the application level, or by the use of SOAP header information.

The WS-BusinessActivity, ASAP, or WS-TX standards might be useful in this scenario.

¹ This might also be referred to as the “basic callback” message exchange pattern.

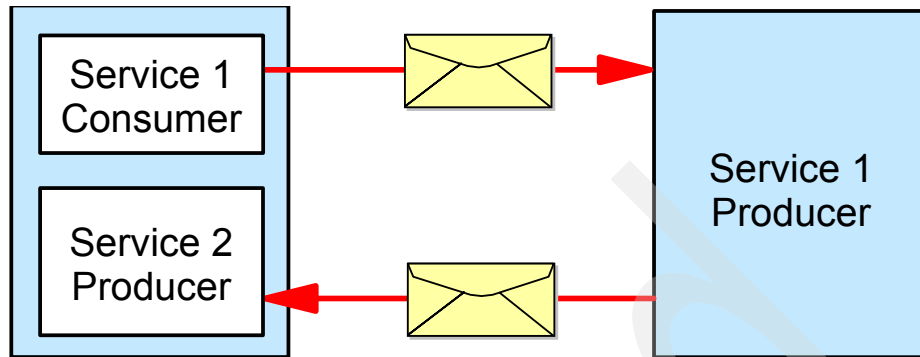


Figure 11-5 Asynchronous two-way message exchange pattern

An example of an asynchronous two-way message exchange pattern could be a message delivery system where a message is sent (by service 1 requestor) requesting a read receipt. The message delivery system (service 1 provider) delivers the message, and after it has been read, returns an asynchronous reply by sending a corresponding message to the service 2 provider.

Request-response

Probably the most common message exchange pattern, a remote procedure call (RPC) or request-response pattern, involves a request message and a synchronous response message (Figure 11-6). In this message exchange pattern, the underlying transport protocol provides an implicit association between the request message and the response message.

Note: The request-response message exchange pattern is not related to the RPC SOAP binding style. This is a separate concept related to the encoding of the SOAP message.

In situations where the message exchange pattern is truly synchronous, such as when an end user is waiting for a response, there is little point in decoupling the consumer and producer. In this situation, the use of SOAP/HTTP as a transport provides the highest level of interoperability. In cases where reliability or other quality of service requirements exist (such as prioritization of requests), alternative solutions might have to be sought.

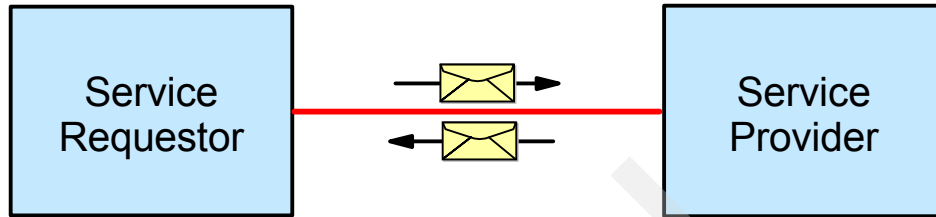


Figure 11-6 Request-response message exchange pattern

There are numerous examples of this message exchange pattern, for example, requesting an account balance on a bank account.

Workflow-oriented

A workflow message exchange pattern can be used to implement a business process where multiple service producers exist. In this scenario, the message that is passed from Web service to Web service maintains the state for the workflow. Each Web service plays a specific role in the workflow (Figure 11-7).

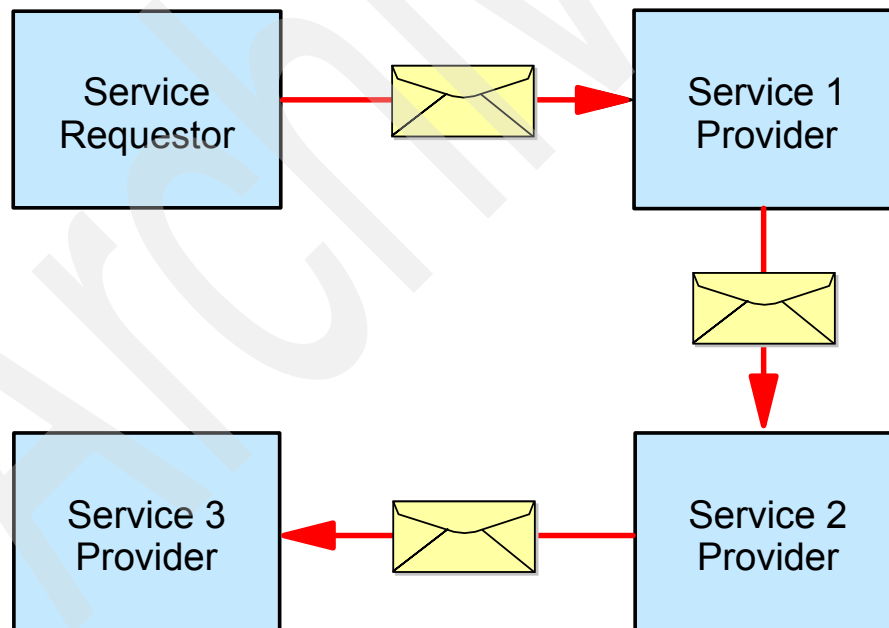


Figure 11-7 Workflow-oriented message exchange pattern

This message exchange pattern is inflexible and does not facilitate reuse—the workflow or *choreography* has been built into each of the Web services, and the

individual Web services no longer exhibit the self-contained property. In some complex B2B scenarios, it is not possible to have any one organization control the process. In this case, a workflow-oriented message pattern might have to be used, possibly in conjunction with a control mechanism, such as an agreed business process workflow definition embedded (or referenced) in the message itself.

Publish-subscribe

The publish-subscribe message exchange pattern, also known as the event-based or notification based pattern, is generally used in situations where information is being *pushed* out to one or more parties (Figure 11-8).

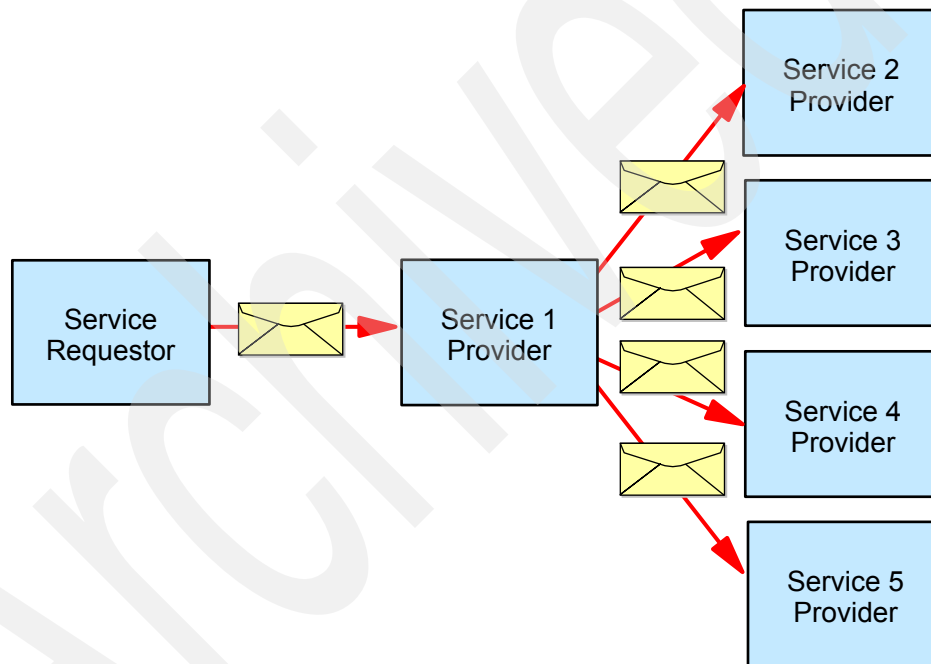


Figure 11-8 Publish-subscribe message exchange pattern

Implementation of the observer pattern² at the application level is one possible architecture. Alternatively, the service 1 provider component could publish SOAP messages to a messaging infrastructure that supports the publish-subscribe paradigm.

² As described in *Design Patterns: Elements of Reusable Object-Oriented Software*, by Erich Gamma et al.

If business logic is not required at all at the service 1 provider component, the service description (WSDL) could simply declare a JMS (or similar) SOAP binding. This would, of course, require consumers to support this type of SOAP binding.

The WS-Notification family of standards is being developed to support this type of message exchange pattern in an interoperable manner (see “WS-Notification” on page 21).

An example of a publish-subscribe message exchange pattern is a news syndication system. A news source publishes an article to the service 1 provider Web service. The service 1 provider Web service, in turn, sends the article to all interested parties.

Composite

The composite message exchange pattern is where a Web service is composed by making requests to other Web services. The composite service producer component controls the workflow and will generally also include business logic (Figure 11-9).

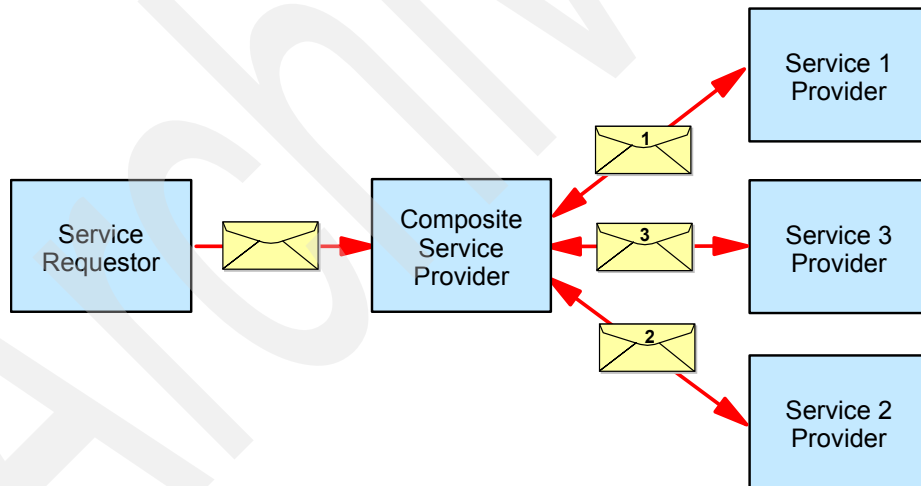


Figure 11-9 Composite message exchange pattern

This is a more flexible architecture than the workflow-oriented message exchange pattern, because all of the Web services are self-contained. The composite service producer component might be implemented in the conventional manner, or could be implemented using a business process choreography engine such as IBM WebSphere Business Integration Server Foundation Version 5.1.

An example of a composite message exchange pattern might be an online ordering system, where the service consumer represents a business partner application placing an order for parts. The composite service provider component represents the ordering system that has been exposed as a Web service to consumers and business partners through the Internet. The ordering system might be implemented as a business process choreography engine. The business process might involve using the service 1 to check for the availability of parts in the warehouse, service 2 to verify the credit standing of the customer, and service 3 to request delivery of the parts to the customer. Some of these services might be internal to the company, and others might be external.

SOAP processing model

At an application level, a typical Web service interaction occurs between a service consumer and a service provider, optionally with a lookup to a service registry (Figure 11-10.)

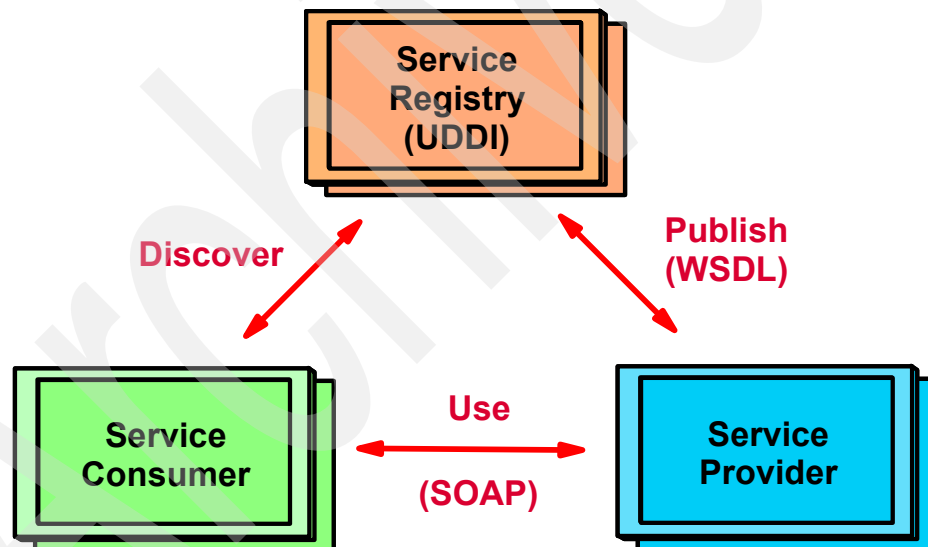


Figure 11-10 Single Web service interaction

At the infrastructure level, additional intermediary SOAP nodes might be involved in the interaction (Figure 11-11).

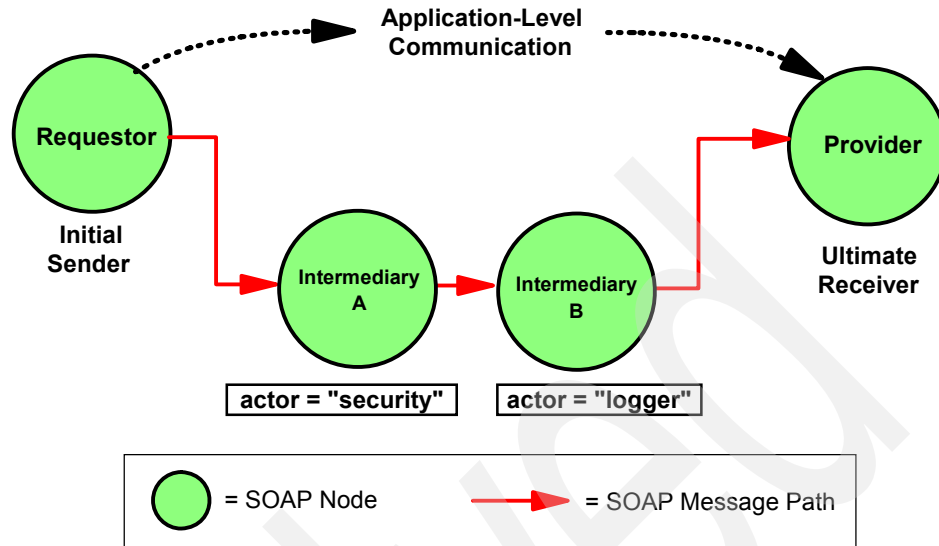


Figure 11-11 SOAP processing model

These intermediary nodes might handle quality of service and infrastructure functions that are non-application specific. Examples include message logging, routing, prioritization, and security. Intermediaries might be physically co-located with the service requestor or provider, or alternatively, might be located somewhere in-between. In general, intermediaries should not alter the meaning of the message body or influence the business semantics of the message.

A typical situation where you need to use intermediary SOAP nodes is where you have an existing Web service implementation within your enterprise that you now want to expose externally. There might be new requirements associated with requests originating from outside of your organization, such as additional interoperability requirements, increased security requirements, auditability of requests, or contractual service-level agreements. These requirements can be implemented using an intermediary SOAP node, or a *Web service gateway*.

Web service gateways

Definition: A Web service gateway is a middleware component that bridges the gap between Internet and intranet environments during Web service invocations.

A Web service gateway implements the facade or adaptor pattern³ to existing Web services. It provides some or all of the following functions and can be used at an organization's network boundary, or internally as an intermediary SOAP node:

- ▶ Provides automatic publishing of (modified) WSDL files to an external UDDI or WSIL registry
- ▶ Provides automatic protocol/transport mappings
- ▶ Provides security functions
- ▶ Provides mediation of message structure
- ▶ Implements a proxy server for Web service communications through a firewall
- ▶ Provides auditing of SOAP messages
- ▶ Provides operational management and reporting of published interfaces
- ▶ Provides Web service threat detection and defense

For more information about Web service gateways, refer to Chapter 22, “Web services and the service integration bus” on page 559.

³ As described in *Design Patterns: Elements of Reusable Object-Oriented Software*, by Erich Gamma et al.

Summary

In this chapter, we discussed the abstract concepts of a service-oriented architecture and another term, the enterprise service bus, and how they relate to Web services.

We attempted to provide some clarity regarding the fast moving landscape of Web services standards and references to where you can find more information about the various standards.

Web services are no different from other distributed technologies in terms of the ways in which components can interact with each other. We presented a collection of well-known message exchange patterns and some of the architectural issues that can be involved in implementing them using Web services technology.

We also introduced the concept of intermediary SOAP nodes and Web service gateways and the motivation for using them.

More information

For more information about selecting and applying IBM Patterns for e-business to a Web services or SOA project, refer to *Patterns: Service-Oriented Architecture and Web Services*, SG24-6303.

The development of composite Web services using a Business Process Execution Language (BPEL)-based choreography engine is discussed in detail in *Using BPEL Processes in WebSphere Business Integration Server Foundation Business Process Integration and Supply Chain Solutions*, SG24-6324.

More information about the IBM WebSphere Business Integration Server Foundation V5.1 product can be found on the IBM Web site at:

<http://www.ibm.com/software/integration/wbisf/>

An introduction to the IBM Web Services Gateway is provided in the developerWorks article *An introduction to Web Services Gateway*, available at:

<http://www.ibm.com/developerworks/webservices/library/ws-gateway/>

Best practices

In this chapter, we describe some best practices for Web services and service-oriented architectures. The best practices are broad in their scope because they cannot take any problem domain or solution into account. However, they can serve as a high-level check list when designing and implementing Web services.

Generic best practices

In this section, we describe some best practices that apply to any Web service solution, independent of the product vendor and the problem domain.

Be WS-I compliant

Being WS-I compliant means that your application follows the industry's conformance standards with regards to interoperability. If the development tool you are using supports the development of WS-I compliant Web services¹, you should turn this feature on and follow its advice. (You can find more information about WS-I in Chapter 10, "Web services interoperability" on page 195.)

However, conforming to WS-I does not mean that your application will be interoperable in any case, because some other party might not be WS-I compliant. Also, there are some ambiguities in the WS-I profiles.

Use simple data types

Even though Web services were designed with interoperability in mind, it is best to use *simple* data types where possible. By simple, we mean integers and strings. In addition, compound data types (comparable with structs in C, or records in Pascal) and arrays of simple types are simple.

Anything that does not fall into this pattern should be used carefully. In particular, the Java collection classes and similarly complex data types should be avoided altogether because there might be no proper counterparts at the client side.

Avoid nillable primitives

Nillable primitive types are allowed for Web services, but there are interoperability issues when using them. The best advice is to not use them at all and use dedicated flags to signal the condition that a value does not exist.

Avoid fine-grained Web services

Web services use a very simple, yet powerful format for their main protocol: XML. While being able to read and structure XML documents with just any simple text editor eases the use of SOAP, the process of automatically creating and interpreting XML documents is more complex.²

¹ As does Rational Application Developer.

² This process is also referred to as "marshalling" and "demarshalling."

Therefore, there is always a point where the complexity of dealing with the protocol is higher than performing the actual computation. To avoid this problem, design Web services that perform *more complex* business logic. This can also mean that your Web service allows for bulk processing instead of multiple invocations with one parameter only.

Avoid Web services for intra-application communication

This best practice is closely related to the previous practice. Intra-application communication (that is, communication *within* an application) is generally not exposed to any third-party clients. Therefore, there is no need to allow for an interoperable interface in this case. However, try to take into consideration that this might change in the future.

Use short attribute, property, and tag names

This is another practice that is closely related to the previous practices. As each attribute, property, and tag name is transmitted verbatim, the length of a message is directly dependent on the length on the attribute and property names. The general guideline is the shorter the attribute, property, and tag names are, the shorter the transmitted message and the faster the communication and processing.³

Avoid deep nesting of XML structures

This is yet another practice that is closely related to the previous practices. Because parsing of deeply nested XML structures increases processing time, deeply nested compound data types should be avoided. This also increases comprehension of the data type itself.

If you are familiar with CORBA or EJBs, apply what you learned there—CORBA applications share many concepts with Web services. In fact, a service-oriented architecture can be implemented with CORBA as well. Almost all best practices that you learned when designing and implementing CORBA-based solutions apply also in the domain of Web services.

³ There are means of using Web services without this drawback: WebSphere Application Server allows for direct invocation of EJBs (see “Multiprotocol binding” on page 418 for more details), and it is being investigated whether ASN.1 can be used for Web services bindings (see <http://java.sun.com/developer/technicalArticles/WebServices/fastWS/> for more details).

Apply common sense (also known as being defensive)

If a standard or specification is not clear enough, try to implement your Web service such that it can handle any of the interpretations you can think of. An example from a different, although not less instructive, domain is the following excerpt from the TCP/IP specification (RFC 793):

Postel's Law: *Be conservative in what you do, be liberal in what you accept from others.*⁴

<http://www.ietf.org/rfc/rfc0793.txt?number=793>

WebSphere Application Server best practices

This section lists some best practices for Web service design and deployment when WebSphere Application Server is the platform of choice for your Web service implementation.

Use the WebSphere Web services engine

The WebSphere SOAP runtime includes many performance improvements that can help with the performance of your application.

Use caching of Web services as provided by the platform

WebSphere Application Server provides an excellent caching framework that allows for caching of information at various levels. Among these, you can also cache Web service requests, thus save processing time. The cache is easy to set up and can be used on any existent Web service. In addition, caching can be also turned on at the client, thus allowing for even more performance improvements.

More information about caching of Web services can be found in Chapter 24, “Web services caching” on page 633.

⁴ The complete statement in the RFC is: “Robustness Principle—TCP implementations will follow a general principle of robustness: be conservative in what you do, be liberal in what you accept from others.”

Summary

In this chapter, we described some best practices when designing and implementing Web services and service-oriented architectures. Most of these are not tied to any specific vendor product. However, because there are always product-dependent best practices, we included some that can be exploited when using the WebSphere Application Server product.

More information

For more information, see the following papers and sites:

- ▶ Web services interoperability:
<http://www.ws-i.org>
- ▶ The paper *Best practices for Web services*, available at:
<http://www.ibm.com/developerworks/library/ws-bestcol.html>
- ▶ The paper *Performance patterns for distributed components and services*, parts 1 and 2, available at:
<http://www.ibm.com/developerworks/library/ws-interface1/>
<http://www.ibm.com/developerworks/library/ws-interface2/>
- ▶ The paper *Performance Best Practices for Using WebSphere Application Server Web Services*, available at:
<http://www.sys-con.com/websphere/article.cfm?id=394>



Part 2

Implementing and using Web services

In this part, we describe practical examples using the IBM WebSphere and Rational products to create and use Web services.

We introduce the products and a simple Web service example that we then implement in various stages using the products.



IBM products for Web services

This chapter introduces the IBM WebSphere Application Server Version 6 and IBM Rational Application Developer for WebSphere Software Version 6 products for Web services development. We provide an general overview of the products and talk in detail about the Web services-specific product functions.

WebSphere Application Server Version 6

As the foundation of the WebSphere software platform, WebSphere Application Server Version 6.0 is one of the industry's premier Java-based application platforms, integrating enterprise data and transactions for the dynamic e-business world. Each configuration available delivers a rich application deployment environment with application services that provide enhanced capabilities for transaction management, as well as the security, performance, availability, connectivity, and scalability expected from the WebSphere family of products.

WebSphere Application Server is J2EE 1.4 compatible and provides Web services support above and beyond the specification. Further, there are new features for rapid development and deployment, which help reduce development cycle time and maximize the ability to use existing skills and resources.

WebSphere Application Server Version 6 has three product offerings:

- ▶ **WebSphere Application Server V6 Express**—Offers the entry point to e-business, an out-of-the-box solution for managing simple, dynamic Web sites with an easy-to-use Web application server and a development environment. The Express product is fully J2EE 1.4 compliant.
- ▶ **WebSphere Application Server V6**—The core of the WebSphere portfolio, this product is an industry-leading J2EE and Web services application server, delivering a high-performance and extremely scalable transaction engine for dynamic e-business applications.
- ▶ **WebSphere Application Server V6 Network Deployment**—Provides an operating environment with advanced performance and availability capabilities in support of dynamic application environments. In addition to all of the features and functions within the base WebSphere Application Server, this configuration delivers advanced deployment services that include clustering, edge-of-network services, Web services enhancements, and high availability for distributed configurations.

In addition, the Network Deployment package contains the Web services gateway. In WebSphere Application Server Version 6, the gateway is now fully integrated in the service integration technology. Detailed information about the gateway can be found in “Web services and the service integration bus” on page 559.

What is new in WebSphere Application Server Version 6

Version 6 includes support for J2EE 1.4, Web services enhancements (WS-AtomicTransaction, WS-Addressing, WS-I Basic Profile 1.1), WebSphere

platform messaging with fully integrated Java messaging engine, JavaServer Faces (JSR 127), and standardized logging (JSR 47).

Other enhancements provide the solid base for a robust infrastructure, including a unified clustering framework and fail-over for stateful session EJBs.

Additional programming model extensions include Service Data Objects (SDO), activity session, asynchronous beans, dynamic query, application profiling, JTA support, WorkArea service, and scheduler, among others. These features have been part of the WebSphere Server Foundation V5.1 product.

WebSphere Application Server Version 6 supports and complies with the Web service specifications and standards listed in Table 13-1.

Table 13-1 Web services standards in WebSphere Application Server Version 6

Specification or standard	Short description
JAX-RPC (JSR-101) 1.1	Additional type support xsd:list Fault support Name collision rules New APIs for creating services isUserInRole()
Web Services for J2EE 1.1 (JSR109, JSR921) ¹	Moved to J2EE 1.4 schema types Migration of Web services client DD moving to appropriate container DDs Handlers support for EJBs Service endpoint interface (SEI) is a peer to LI/RI
SAAJ 1.2	APIs for manipulating SOAP XML messages SAAJ infrastructure now extends DOM (easy to cast to DOM and use)
WS-Security	WSS 1.0 WS-I Security Profile
WS-I Basic Profile 1.1	Attachments support
WS-Transaction	WS-AtomicTransaction
JAXR 1.0	Java client API for accessing UDDI (Version 2 only) and ebXML registries
UDDI v3	Includes both the registry implementation and the client API library

Rational software development products

Rational software development products are part of the IBM Software Development Platform.

Note: The IBM Software Development Platform is a set of integrated tools, best practices, and services that support a proven end-to-end process for the application development life cycle.

Rational software development products fit into a tools framework that supports structured application development, including modeling, proven design practices and patterns, and an iterative development process that helps ensure that applications meet user requirements.

The Rational software development products are based on Eclipse 3.0 and provide a comprehensive and productive application development environment for creating and maintaining J2EE-compliant enterprise application systems. It includes many features not available in Eclipse.

Because the products are built on the Eclipse platform, development teams can adapt and extend the development environment with best-of-breed plug-in tools from IBM, IBM Business Partners, and the Eclipse community to match their needs and maximize developer productivity.

The Rational software development products are:

- ▶ Rational Web Developer for WebSphere Software
- ▶ Rational Application Developer for WebSphere Software
- ▶ Rational Software Architect
- ▶ WebSphere Studio Application Developer Integration Edition
- ▶ WebSphere Studio Enterprise Developer

For general information about Rational products, go to:

<http://www.ibm.com/software/rational>

Rational Web Developer for WebSphere Software Version 6

Rational Web Developer (Web Developer for short) is the follow-on product to WebSphere Studio Site Developer Version 5.1.2. It is an entry-level IDE for Web and Java developers and primarily used for building JSP and servlet-based Web applications, Java applications, and Web services. It provides visual development with JavaServer Faces components and Enterprise Generation Language (EGL) for generating Java code. For detailed information go to:

<http://www.ibm.com/software/awdtools/developer/web/index.html>

Rational Application Developer for WebSphere Software Version 6

Rational Application Developer (Application Developer for short) is the follow-on product to WebSphere Studio Application Developer Version 5.1.2. It allows for more advanced J2EE development, including Enterprise JavaBeans (EJB) components. It supports portal-based and UML-based development and contains IBM Rational ClearCase® LT for version control. For detailed information, go to:

<http://www.ibm.com/software/awdtools/developer/application/index.html>

Rational Software Architect

The capabilities of Rational Web Developer and Rational Application Developer are also incorporated into Rational Software Architect (Software Architect for short), which adds support for UML 2 modeling, patterns, model transforms, code generation, C/C++ development, and Java application structural review and control. For detailed information, go to:

<http://www.ibm.com/software/awdtools/architect/swarchitect/index.html>

WebSphere Studio Application Developer Integration Edition V5.1

Another version of Rational Application Developer is the WebSphere Studio Application Developer Integration Edition, which helps enable accelerated development and integration of composite business applications that deploy to the IBM WebSphere Business Integration Server Foundation. It provides a broad portfolio of rich application and technology adapters and J2EE-based visual workflow tools. For detailed information, go to:

<http://www.ibm.com/software/integration/wsadie/>

WebSphere Studio Enterprise Developer Version 5.1.2

WebSphere Studio Enterprise Developer (Enterprise Developer for short) is based on WebSphere Studio Application Developer and adds support for COBOL and PL/I development and for the development of applications that target legacy back-end systems, such as IBM CICS® and the IBM @server® zSeries® family of servers. It also provides EGL code generation that outputs COBOL source code. For detailed information, go to:

<http://www.ibm.com/software/awdtools/studioenterprisedev/>

Web services support in Rational Application Developer

For this book, we used the rapid application development features of Rational Application Developer to develop the Web services applications. Some of the examples could also be performed with Rational Web Developer; only when EJBs are involved, Application Developer must be used.

Web services tooling

This section gives an overview of the Web services-specific functions in Rational Application Developer, which provides tools to assist with the following aspects for Web services development:

- ▶ **Create service provider**—Use the Application Developer tooling to create, deploy, test, and publish Web services bottom-up from existing Java beans, enterprise beans, DADX files, and URLs, and top-down from WSDL. When generating a Web service, the wizards support the automatic generation of additional artifacts, such as a JavaBean proxy to easily access the Web service, and a test client.

To create a Web service, either graphical wizards or command-line tools can be used. Chapter 16, “Develop Web services with Application Developer V6.0” on page 273 shows how to use the wizards to create a Web service, and Chapter 19, “Command-line tools, Ant, and multiprotocol binding” on page 395 describes and shows the usage of the command-line tools.

- ▶ **Create service consumer**—Use the Web services client tools (again, GUI wizard or command line) to create a client for any Web service. Only the WSDL file is needed to create a Web service client.
- ▶ **Secure**—The Web Service wizards and deployment descriptor editors assist you with configuring Web services security (WS-Security) for the WebSphere Application Server environment. In Chapter 21, “Securing Web services” on page 445, we show how to apply security to a Web service.
- ▶ **Run**—Run Web services provider and consumer components in WebSphere Application Server or Tomcat test environments. The deployment and administration for the WebSphere test environment is integrated in Application Developer.
- ▶ **Test**—Web services can be tested, running locally or remotely. For local tests, the WebSphere test environment can be used. The WebSphere test environment contains a complete WebSphere Application Server runtime environment. Rational Application Developer provides different functions to test Web services. See “Testing Web services” on page 338.

- ▶ **Discover**—Browse Universal Description, Discovery, and Integration registries (UDDI) or Web Services Inspection Language (WSIL) sites to find Web services for integration. The IBM Web Services Explorer provides the all necessary functions to discover a Web service.
- ▶ **Publish**—Publish Web services to a UDDI V2 or V3 Business Registry, using the Web Services Explorer.
- ▶ **Build skeletons**—Generate JavaBean and EJB skeletons from WSDL files. This can be helpful during the development and test phase of a project. For example, when the service is defined (WSDL), but not running at the service provider site, and the client needs to be tested, a test service provider can be created to emulate the provider.
- ▶ **Validate**—Use the WSDL and DADX validators to check for structural and semantic problems in these types of files. This feature is useful when receiving a service WSDL file from a service provider to check that the files are valid.
- ▶ **Compliance**—Different WS-I profile compliance tests and levels can be defined for the Web services development environment. Application Developer can check compliance for the Simple SOAP Basic 1.0 and the Attachment Profile 1.0.

When creating or changing Web services, the WS-I compliance tester will analyze the service, and depending on the configuration, ignore, suggest, or require profile compliance. This can be defined in the Web services preferences, as described in “Web services configuration settings” on page 241.

- ▶ **WSDL support**—Application Developer provides wizards and functions to easily work with WSDL files:
 - Use the graphical editor to create a WSDL file from a template and to add WSDL elements (service, port, port types, messages).
 - Create WSDL documentation; this creates HTML documentation for the WSDL file, similar to a JavaDoc document.
 - Validate WSDL file for WS-I compliance.
- ▶ **Web services-specific navigation**—Application Developer now organizes Web services together in the Project Explorer in a *Web Services* group. This makes is easier to find and work with Web services.

Table 13-2 shows the supported Web services-related technologies and specifications in Application Developer Version 6.

Table 13-2 Specifications supported in Rational Application Developer Version 6

Technology or specification	Version or level supported
HTTP/HTTPS	1.0 and 1.1
JMS	1.1
SOAP	1.1
SOAP Attachments	1.0
UDDI	2.0 and 3.0
WSDL	1.1
WSIL	1.0
WS-Security	OASIS Standard 1.0
WS-I Basic Profile	1.1.2
WS-I Simple SOAP Binding Profile	1.0.3
WS-I Attachments Profile	1.0
JAX-RPC	1.0 for J2EE 1.3 1.1 for J2EE 1.4
Web Services for J2EE	WSEE 1.0 (JSR109) for J2EE 1.3 WSEE 1.1 (JSR 921) for J2EE 1.4

Web services runtime environments

Rational Application Developer supports three Web service provider runtime environments:

- **IBM WebSphere runtime environment**—This is the recommended runtime environment for production use. Only the WebSphere runtime environment is full supported by IBM. It includes specialized serializers and deserializers for complex objects, JSR 109 support for enterprise Web services (EJBs), and SOAP over JMS support.
- **IBM SOAP runtime environment**—This was the only supported runtime environment in previous releases of WebSphere Studio (Version 5.0 and earlier). It should only be used for backward compatibility, and it supports Apache SOAP 2.3.

Currently, DB2 Web services from SQL statements (DADX files) still require the SOAP runtime.

- **Apache Axis 1.0 runtime environment**—This is the third version of the Apache SOAP implementation. Apache Axis evolved from the Apache SOAP implementation (which began as IBM SOAP4J).

The Apache Axis runtime is not suggested for WebSphere production environments, but can be used for Apache Tomcat servers.

Table 13-3 shows the supported runtime and server configurations.

Table 13-3 Supported runtime and server configurations

SOAP runtime	Server configuration
IBM WebSphere	WebSphere Application Server V5.0.2, V5.1, V6.0
IBM SOAP	Apache Tomcat V3.2, V4.0, V4.1 WebSphere Application Server V5.0, V5.1
Apache Axis 1.0	Apache Tomcat V4.0, V4.1, V5.0 WebSphere Application Server V5.0, V5.1, V6.0

Web services configuration settings

In this section, we explain the Web service-specific configuration settings in the Application Developer workbench. These options influence the behavior and generated artifacts of the Web service tooling.

Note: To develop Web services with Rational Application Developer, the Web services development capabilities must be enabled. Enabling this capability activates the required functions to develop Web services and the Web services options section in the workbench preferences page.

To enable the Web service capabilities, open the Application Developer preferences page (Figure 13-1):

- Select *Windows* → *Preferences*.
- Expand *Workbench* and select *Capabilities*.
- In the right pane, expand *Web Services Developer* and select both *Component Test for Web Services* and *Web Services Development*.
- Click *OK* to activate the changes and to close the preferences window.

Note: The Web services preferences section opens when reopening the preferences page, after the Web services capabilities have been enabled.

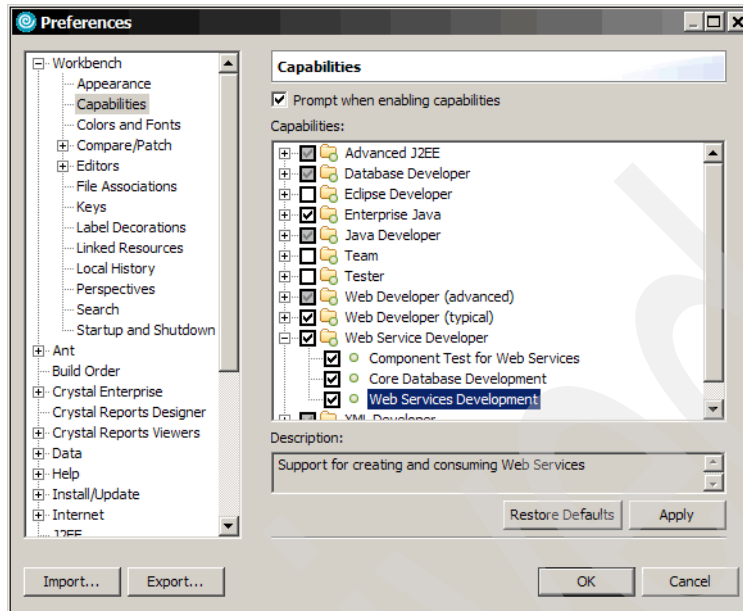


Figure 13-1 Enable Web services development capabilities

Web services preferences

To change the Web services preferences, select *Window* → *Preferences* and expand *Web Services* (Figure 13-2).

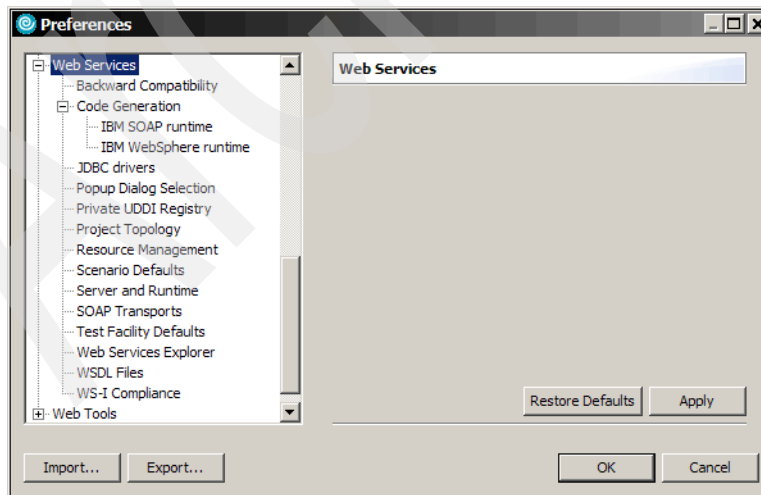


Figure 13-2 Web services preferences

We now provide an overview of the Web services preferences:

- ▶ **Backward Compatibility**—This option only applies when using the IBM SOAP runtime environment. Enabling this option will use WebSphere Application Server Version 4 mapping styles.
- ▶ **Code Generation**—Change the behavior of the code generation wizards for the SOAP and the WebSphere runtime.
 - When enabling *Disable data binding and use SOAPElement* for the WebSphere runtime, the `noDataBinding` option will be used for code generation. This is needed when a consistent mapping mechanism is required for an entire service endpoint, for example, when JAX-B (Java API for XML Binding) or Service Data Objects (SDO) have to be used for the binding.
 - When enabling *Do not overwrite loadable Java classes*, no duplicate classes are generated when the original class is available, for example, in a utility JAR file.
- ▶ **JDBC drivers**—This setting is only used for DADX validation in the Application Developer Linux version. It must be defined to provide the location to the correct JDBC driver JAR file.
- ▶ **Popup Dialog Selection**—Change the behavior of the pop-up dialogs for the Web Service wizards. These preferences can be used to disable some of the dialog steps in the wizard, for example, publishing to UDDI.
- ▶ **Private UDDI Registry**—Specify the delimiter for the used UDDI category data column and string. This is helpful when working with a private UDDI registry where self-defined tags are used.
- ▶ **Project Topology**—Specify whether the Web service server and client are generated into separate enterprise applications. This setting is highly recommend (select *Generate Web service and Web service client in different EAR projects*). Having the service and the client in the same project can cause conflicts and makes the production assembly and deployment more complex. The default sequence of client types (Web, Java, EJB, Application Client) can also be changed on this page.
- ▶ **Resource Management**—Specify the file and folder overwrite and creation behavior when creating a Web service. For example, select *Overwrite files without warning* as the default (if you regenerate the same service multiple times).
- ▶ **Scenario Defaults**—Specify initial selections for the Web Service wizard. For example, select *Start Web service in Web project*, *Generate a proxy*, and *Test the Web service*. These options will be preselected when running the wizard.
- ▶ **Server and Runtime**—Specify the default server (WebSphere v6.0 Server), the Web service runtime (IBM WebSphere), and the J2EE Version (1.4).

- ▶ **SOAP Transports**—Specify the default transport (HTTP or JMS).
- ▶ **Test Facility Defaults**—Specify the default settings for the test facility. We suggest to that you clear the option *Launch the sample when generated*, and leave the Web service sample JSPs at the top.
- ▶ **Web Services Explorer**—When using the IBM SOAP runtime and testing services that exchange array types, *Ignore schema for SOAP arrays* should be enabled on this page.
- ▶ **WSDL Files**—Specify the default target name space for WSDL files created with the WSDL editor.
- ▶ **WS-I Compliance**—Specify the Application Developer behavior in respect to WS-I compliance checks. Application Developer can validate the compliance for WS-I Basic Profile 1.0 and WS-I Attachment Profile 1.0. Recommended selections are *Suggest compliance* for Basic Profile (WS-I SSBP compliance level) and *Ignore compliance* for Attachment Profile (WS-I AP compliance level).

Be sure to click *Apply* when making changes and close the dialog by clicking *OK*.

Summary

In this chapter, we introduce the IBM WebSphere Application Server and Rational Application Developer products. We talked about the products in general and provided an overview of the Web services-specific product features.

We use most of the Web services-specific product functions in the sample applications that we develop in this book.

More information

The WebSphere family of products are regularly updated to cope with a business environment that changes at a high pace. Therefore, the Internet represents one of the best sources of up-to-date information.

For information about the IBM WebSphere family of products, refer to:

<http://www.ibm.com/software/websphere/>

For IBM Rational product information, refer to:

<http://www.ibm.com/software/rational/>

Sources of additional information about particular products can be found at the end of the corresponding chapters.



Sample application: Weather forecast

In this chapter, we describe the base Java code of a weather forecast application, which we use to demonstrate Web services technology.

Note: The weather forecast application has been slightly rewritten from the previous version (SG24-6891):

- ▶ It uses the embedded Cloudscape database or DB2 for storing weather information.
- ▶ The logic has been restructured into service, business, data, and back-end modules.

Weather forecast application components

The weather forecast application is the example that we use in the following chapters to demonstrate how to create a Web service with different tools and programs. The weather forecast is an application that simulates weather forecast predictions.

This weather forecast application is composed of three main modules: business module, data module, and back-end module. It also provides two different implementations, as a session EJB in an EJB project and as a JavaBean in a Web project.

Figure 14-1 shows the components of the weather forecast application.

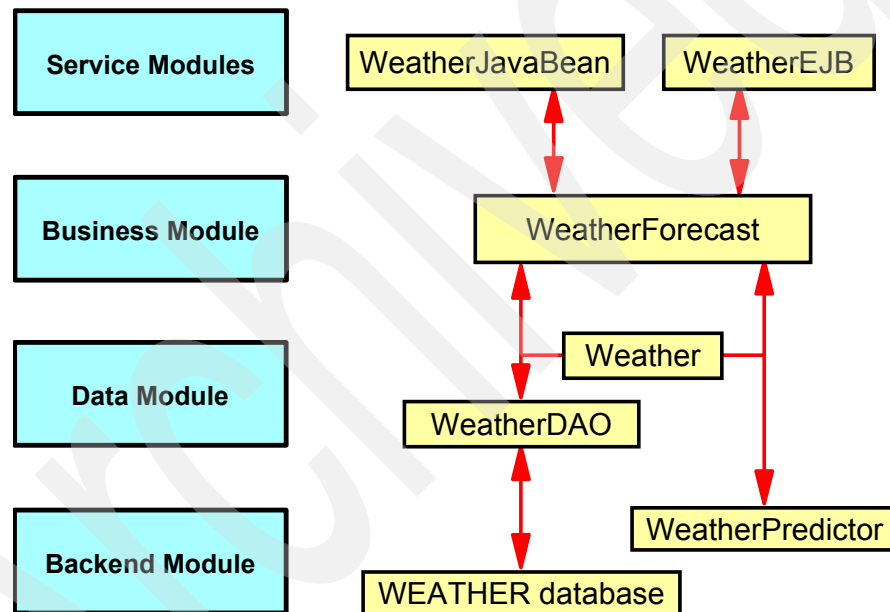


Figure 14-1 Weather forecast application components

Service modules

The service modules contain the implementations of the weather forecast application that we will turn into Web services:

- ▶ The **WeatherJavaBean** class as part of a Web module
- ▶ The **WeatherEJB** session bean as part of an EJB module (we also have a duplicate **WeatherJMS** session bean for a JMS Web service)

Business module

The business module is implemented by the **WeatherForecast** class and used by the service modules. The **WeatherForecast** class offers the business logic of our example by providing four principal functions:

- ▶ Return the weather forecast prediction for one specific day (Weather object)
- ▶ Return the weather forecast prediction for a period of time (Weather[] array)
- ▶ Return the temperature prediction for a period of time (int[] array)
- ▶ Load new weather information into the database (no return value)

The functionality offered by the Weather forecast application is described by the **IWeather** interface with the method signatures shown in Table 14-1.

Table 14-1 Weather forecast application interface

Method summary	
itso.objects.Weather	Get weather information for a specific day: getDayForecast(java.util.Calendar theDate)
itso.objects.Weather[]	Get forecast for a specific period of time: getForecast(java.util.Calendar startDate, int days)
int[]	Get temperatures for a specific period of time: getTemperatures(java.util.Calendar startDate, int days)
void	Set a forecast for a specific day: setWeather(itso.objects.Weather dayWeather)

Data module

The data module is implemented by the **WeatherDAO** and **Weather** class:

- ▶ The **WeatherDAO** class contains all the functionality to store and retrieve weather information from the database, in form of Weather objects.
- ▶ The **Weather** class provides the content information of a weather prediction:
 - Date of the prediction (Calendar)
 - Weather condition: sunny, partly cloudy, cloudy, rainy, stormy (String)
 - Wind direction in an eight-point compass (String)
 - Wind speed in kilometers per hour (int)
 - Temperature in degrees Celsius (int)
 - Database flag, information comes from the database or not (boolean)

The **Weather** class is the data transfer object (DTO) that holds the weather information and is passed from module to module.

Back-end module

The back-end module contains the **WEATHER** database with the **ITS0.SANJOSE** table, where the weather information for San Jose is stored.

The back-end module is supported by the **WeatherPredictor** class. This class is responsible for making a simulated prediction about weather conditions using random values, when no information for the requested day is contained in the database.

Information flow

Figure 14-2 shows the internal flow of the system information for one of the query methods.

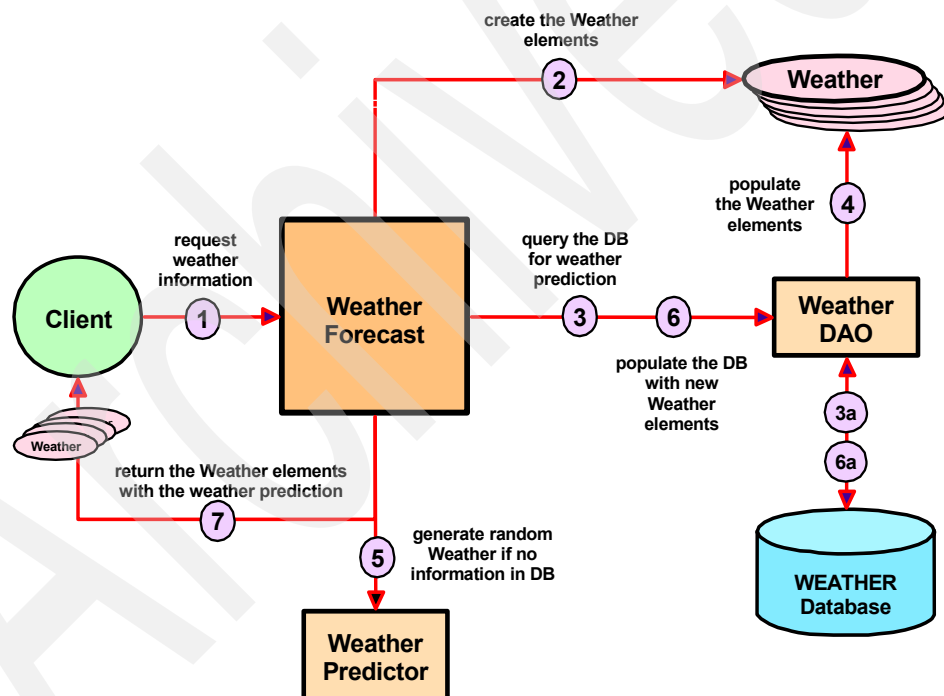


Figure 14-2 Weather forecast information flow

The steps for this flow are:

1. A client requests weather information from the WeatherForecast bean.
2. The WeatherForecast bean creates a Weather element (or elements) for the response to the client's weather request.
3. The WeatherForecast queries the weather prediction from the WEATHER database using the WeatherDAO bean.
4. The WeatherDAO bean populates the Weather element (or elements) based on the information present at that moment in the database.
5. The weather information that is not in the database is requested from the WeatherPredictor.
6. The database is populated by the queries with the new Weather element (or elements) generated by the WeatherPredictor.
7. The WeatherForecast returns the Weather element (or elements) to the client.

Note: The WeatherPredictor class uses a random number algorithm to populate weather information. This makes our example very simple, but enables us to concentrate on the important Web services aspects instead of trying to write a sophisticated back-end application.

Figure 14-3 shows the internal flow of the system information for the load method.

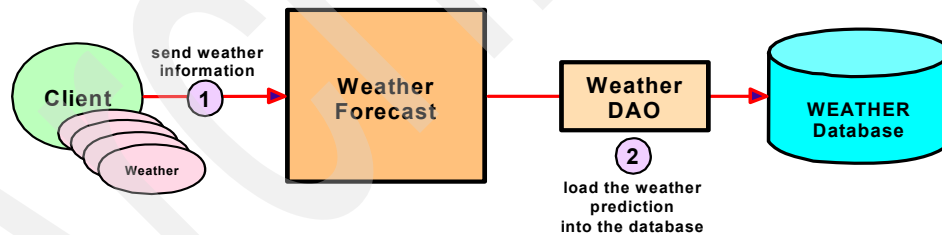


Figure 14-3 Weather forecast load flow

The steps for the load flow are:

1. A client sends weather information to the WeatherForecast bean to load the database.
2. The WeatherForecast bean populates the database with the Weather element using the WeatherDAO class.

Weather forecast application implementation

Figure 14-4 shows the implementation of the weather forecast application with packages and modules.

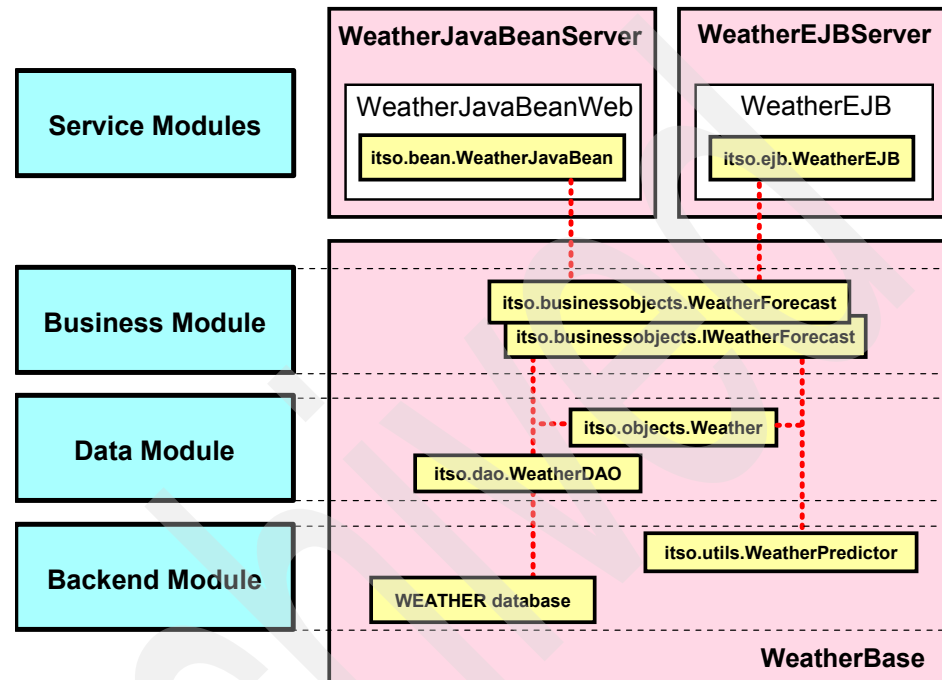


Figure 14-4 Weather forecast application implementation

The service modules are implemented as enterprise applications:

- ▶ **WeatherJavaBeanServer**—Contains the **WeatherJavaBeanWeb** Web module (with the **WeatherJavaBean** JavaBean)
- ▶ **WeatherEJBServer**—Contains the **WeatherEJB** EJB module (with the **WeatherEJB** session bean) and the **WeatherEJBClientClasses** module (the EJB client classes required in an EJB client—not shown in Figure 14-4)
- ▶ **WeatherEJBJMSServer** (not shown)—Contains the **WeatherEJBJMS** EJB module (a copy of the **WeatherEJBServer** module with the **WeatherJMS** session bean for a JMS Web service)

The other modules are all contained in the **WeatherBase** utility module that is contained in all the enterprise applications.

Weather database

The WEATHER database contains one table, ITS0.SANJOSE, with five columns:

- ▶ WEATHERDATE DATE—Date of weather prediction, primary key
- ▶ CONDITION VARCHAR (20)—Condition: sunny, partly cloudy, cloudy, rainy, stormy
- ▶ WINDDIR VARCHAR (20)—Wind direction: N, NE, E, SE, S, SW, W, NW
- ▶ WINDSPEED INTEGER—Wind speed (kilometers/hour)
- ▶ TEMPERATURE INTEGER—Temperature (degree Celsius)

We provide two implementations of the WEATHER database, DB2 and Cloudscape.

Data source

The weather forecast WeatherDAO class uses a resource reference to look up the data source used to connect to the WEATHER database. All the modules that use the weather forecast base project must have a resource reference defined in the deployment descriptor:

- ▶ Name—WeatherDataSourceReference
- ▶ Type—javax.sql.DataSource
- ▶ JNDI name—jdbc/weather

The data source can be configured in the enterprise application deployment descriptor and is automatically deployed to the WebSphere Application Server Version 6 test environment. We provide sample data source definitions for DB2 and Cloudscape; you can choose which data source to use by setting one of the data source JNDI names to jdbc/weather.

Extract of the source code

In this section, we list part of the source code to provide you with a better understanding of the implementation.

The source code is available in \SG246461\samcode_setup\JavaBase if you want to see the complete source of the base Java classes (see Appendix C, “Additional material” on page 701).

The instructions to install the base enterprise applications in Rational Application Developer are available in “Installing the base weather forecast application” on page 705.

Data transfer object: Weather

Example 14-1 shows the Weather class.

Example 14-1 Weather class (extract)

```
package itso.objects;
import .....;

public class Weather implements Serializable {

    private String    windDirection = null;
    private int       windSpeed = 0;
    private int       temperatureCelsius = 0;
    private String    condition = null;
    private Calendar  date = null;
    private boolean    dbflag = false;

    // Constructors
    public Weather() { this(Calendar.getInstance()); }
    public Weather(Calendar theDate) { date = (Calendar)theDate.clone(); }

    // toString
    public String toString() {
        SimpleDateFormat sdf = new SimpleDateFormat("EEE. MMM d, yyyy zzz");
        sdf.setTimeZone(date.getTimeZone());
        return "Weather: " + sdf.format(date.getTime()) + ", " + condition
            + ", wind: " + windDirection + " at " + windSpeed + "km/h "
            + ", temperature: " + temperatureCelsius + " Celsius ";
    }

    // getter and setter methods not shown
}
```

Business object: WeatherForecast

Example 14-2 shows the WeatherForecast class.

Example 14-2 WeatherForecast class (extract)

```
package itso.businessobjects;
import itso.dao.WeatherDAO;
import itso.objects.Weather;
import .....;

public class WeatherForecast implements IWeatherForecast {
    public WeatherForecast() { super(); }

    public Weather getDayForecast(Calendar theDate) throws Exception {
```

```

        Weather[] forecasts = getForecast(theDate, 0);
        if (forecasts != null && forecasts.length == 1)
            return forecasts[0];
        else
            throw new Exception("Invalid forecast returned.");
    }

    public Weather[] getForecast(Calendar startDate, int days) throws Exception
    {
        if (startDate == null)
            startDate = Calendar.getInstance(); // defaults to today
        Weather[] iWeather = executeQuery(startDate, days);
        return iWeather;
    }

    public int[] getTemperatures(Calendar startDate, int days) throws Exception
    {
        if (startDate == null)
            startDate = Calendar.getInstance(); // defaults to today
        Weather[] iWeather = getForecast(startDate, days);
        int[] temperatures = new int[iWeather.length];
        for (int i = 0; i < iWeather.length; i++) {
            temperatures[i] = iWeather[i].getTemperatureCelsius();
        }
        return temperatures;
    }

    public void setWeather(Weather dayWeather) throws Exception {
        if (dayWeather == null)
            throw new Exception("Please provide valid weather elements");

        WeatherDAO weatherDAO = new WeatherDAO();
        // delete information, if old exists
        weatherDAO.deleteWeather(dayWeather.getDate());
        // insert updated information
        weatherDAO.insertWeather(dayWeather);
        WeatherLogger.log("Inserted: "+dayWeather);
    }

    private Weather[] executeQuery(Calendar startdate, int days) {
        WeatherDAO weatherDAO = new WeatherDAO();

        List result = new ArrayList();
        Calendar dateLoop = (Calendar) startdate.clone();

        for (int i = 0; i <= days; ++i) {
            Weather w = weatherDAO.getWeather(dateLoop);
            if (w == null) {
                // nothing found for today, we have to perform simulation
                w = new Weather(dateLoop);
            }
        }
    }

```

```

        WeatherPredictor.calculateWeatherValues(w);
        WeatherLogger.log("Predicted: "+w);

        try {
            // insert into database
            setWeather(w);
        } catch (Exception e) {
            System.err.println("some error occured while performing an
                update... check logs");
            e.printStackTrace();
        }
    } else {
        WeatherLogger.log("Retrieved: "+w);
    }
    result.add(w);

    // proceed to next day
    dateLoop.add(Calendar.DAY_OF_MONTH, 1);
}
return (Weather[]) result.toArray(new Weather[0]);
}
}

```

The `WeatherForecast` class is also the Web service implementation in a Web project.

Data access: WeatherDAO

Example 14-3 shows the `WeatherDAO` class.

Example 14-3 WeatherDAO class (extract)

```

package itso.dao;
import itso.objects.Weather;
import .....;

public class WeatherDAO {
    public static final String _JNDI_NAME =
        "java:comp/env/WeatherDataSourceReference";

    public int insertWeather(Weather w) {
        Connection con = getConnection();
        PreparedStatement pm = null;
        int rs = 0;
        try {
            pm = con.prepareStatement("INSERT INTO ITSO.SANJOSE(WEATHERDATE,
                CONDITION, TEMPERATURE, WINDDIR, WINDSPEED) VALUES(?,?,?,?,?)");
            Date sqlDate = new Date(w.getDate().getTime().getTime());

```



```

        // ..... similar to getWeather
    }
    return rs;
}

public boolean deleteWeather(Calendar day) {
    Connection con = getConnection();
    PreparedStatement pm = null;
    int rs = 0;
    try {
        pm = con.prepareStatement("DELETE FROM ITSO.SANJOSE WHERE
                                   WEATHERDATE = ?");
        //..... similar to getWeather
    }
    return (rs > 0);
}

public Weather getWeather(Calendar day) {
    Connection con = getConnection();
    PreparedStatement pm = null;
    Weather result = null;
    try {
        pm = con.prepareStatement("SELECT * FROM ITSO.SANJOSE
                                   WHERE WEATHERDATE = ?");
        Date sqlDate = new Date(day.getTime().getTime());
        pm.setDate(1, sqlDate);
        ResultSet rs = pm.executeQuery();
        while (rs.next()) {
            result = new Weather();
            Calendar theDate = Calendar.getInstance();
            theDate.setTime(rs.getDate("WEATHERDATE"));
            result.setDate(theDate);
            result.setCondition(rs.getString("CONDITION"));
            result.setTemperatureCelsius(rs.getInt("TEMPERATURE"));
            result.setWindDirection(rs.getString("WINDDIR"));
            result.setWindSpeed(rs.getInt("WINDSPEED"));
            result.setDbflag(true);
        }
    } catch (SQLException e) {
        e.printStackTrace(System.err);
        result = null;
    } finally {
        try {
            if (pm != null) pm.close();
            if (con != null) con.close();
        } catch (Exception e) { e.printStackTrace(System.err); }
    }
    return result;
}

```

```

private Connection getConnection() {
    Connection con = null;
    try {
        InitialContext ic = new InitialContext();
        DataSource ds = (DataSource) ic.lookup(_JNDI_NAME);
        con = ds.getConnection();
    } catch (NamingException e) { e.printStackTrace(System.err); }
    } catch (SQLException e) { e.printStackTrace(System.err); }
    return con;
}
}

```

Predictor: WeatherPredictor

Example 14-4 shows the WeatherPredictor class.

Example 14-4 WeatherPredictor class (extract)

```

package itso.utils;
import itso.objects.Weather;
import java.util.Random;

public class WeatherPredictor {
    // possible conditions for weather in general and wind directions
    private static final String[] possibleConditions = new String[] { "sunny",
        "partly cloudy", "cloudy", "rainy", "stormy" };
    private static final String[] possibleWinds = new String[] { "N", "NE",
        "E", "SE", "S", "SW", "W", "NW" };

    // random number generator
    private static final Random random =
        new Random(System.currentTimeMillis());

    public static void calculateWeatherValues(Weather w) {
        w.setWindDirection(possibleWinds[random.nextInt(possibleWinds.length)]);
        w.setWindSpeed(random.nextInt(40) + 1);
        w.setTemperatureCelsius(random.nextInt(50) - 10);
        w.setCondition(possibleConditions[random
            .nextInt(possibleConditions.length)]);
    }
}

```

JavaBean Web service: WeatherJavaBean

Example 14-5 shows the WeatherJavaBean class.

Example 14-5 WeatherJavaBean class (extract)

```
package itso.bean;
import itso.businessobjects.IWeatherForecast;
import itso.businessobjects.WeatherForecast;
import itso.objects.Weather;
import java.util.Calendar;

public class WeatherJavaBean implements IWeatherForecast {

    public Weather getDayForecast(Calendar theDate) throws Exception
    { return new WeatherForecast().getDayForecast(theDate); }

    public Weather[] getForecast(Calendar startDate, int days) throws Exception
    { return new WeatherForecast().getForecast(startDate, days); }

    public int[] getTemperatures(Calendar startDate, int days) throws Exception
    { return new WeatherForecast().getTemperatures(startDate, days); }

    public void setWeather(Weather dayWeather) throws Exception {
        WeatherForecast wfc = new WeatherForecast();
        wfc.setWeather(dayWeather);
    }
}
```

EJB Web service: WeatherEJB

Example 14-6 shows the WeatherEJB class.

Example 14-6 WeatherEJB class (extract)

```
package itso.ejb;
import itso.businessobjects.IWeatherForecast;
import itso.businessobjects.WeatherForecast;
import itso.objects.Weather;
import java.util.Calendar;

public class WeatherEJBBean implements javax.ejb.SessionBean, IWeatherForecast
{
    // EJB callback methods not shown (ejbCreate, etc)

    // all business methods identical to WeatherJavaBean
}
```

Summary

In this chapter, we covered the weather forecast example base code that is used to create a Web service using the different tools or products. We discussed the different components and how they interact to provide the weather forecast functionality.

In the chapters that follow, we examine in detail the information required to create and use Web services using the different tools available in Rational Application Developer and WebSphere Application Server.

Development overview

This chapter provides a general introduction to Web services development. We present different paths for developing a Web service and the different types of clients that can be implemented. Throughout this chapter, we do not focus on a specific development tool. Instead, the descriptions are generic and more detailed examples follow in the subsequent chapters.

This chapter contains the following topics:

- ▶ Building a new Web service
- ▶ Building a new Web service client

Overview

The development process for Web services is very similar to the development process of any other software. There are four main phases in developing a Web service: build, deploy, run, and manage.

- ▶ The *build* phase includes development and testing of the Web service application, including the definition and functionality of the service.
- ▶ The *deploy* phase includes publication of the service definition, the WSDL document, and deployment of the runtime code of the Web service.
- ▶ The *run* phase includes finding and invoking the Web service.
- ▶ The *manage* phase includes the management and administration of the Web service. This includes performance measurement and maintenance of the Web service.

Figure 15-1 depicts the complete development process. Using different problem domains, the terms used within this picture would change; however, the general view would not.

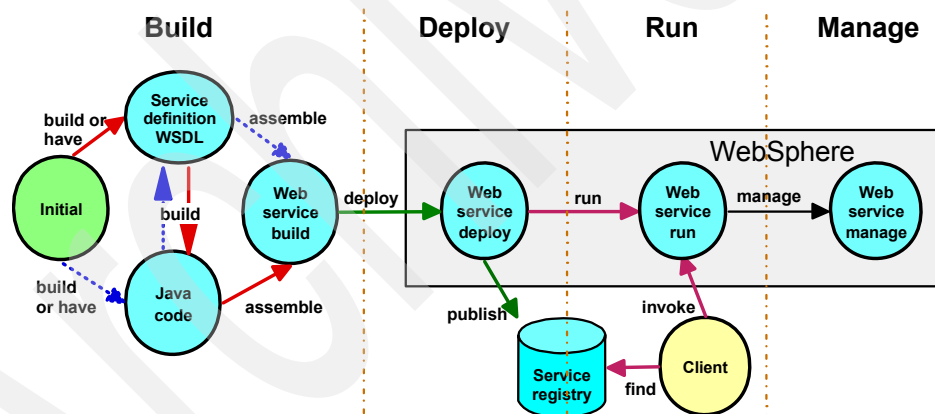


Figure 15-1 Web services development

The remainder of this section describes the four development phases in more detail.

Build phase

The build phase, which includes testing and debugging, is the first phase in the development process of a new Web service. Because Web services can be written from scratch and use already existing legacy applications, there are two possible paths to be followed:

- ▶ The red (solid) path—From the initial state, we build or already have Java code. Using this Java code, we build the service definition (WSDL document) with the business methods that we want to expose. After we have generated the WSDL document, we assemble the Web service application. This approach is called bottom-up development.
- ▶ The blue (dashed) path—From the initial state, we build or already have a service definition, a WSDL document. Using this WSDL document, we build or adapt the Java code to implement that service. After we have implemented the code, we assemble the Web service application. This approach is called top-down development.

Deploy phase

The second phase of a Web service development process is deployment. In this phase, we deploy the Web service to an application server. Deploying a Web service makes it accessible by clients. However, these clients have to be aware of the newly installed Web service, and thus, the next step in this phase is to publish the Web service. The publication can be done through a private or public UDDI registry, using a WSIL document, or by directly providing the information about the new service to consumers, for example, through e-mail. A combination of all these publishing methods is also possible. After the service has been published, it can be called by clients.

Run phase

The third phase is the runtime. In this phase, the Web service is operative and is invoked by clients that require the functionality offered by this service.

Manage phase

The final phase is the management phase where we cover all the management and administration tasks of the Web service.

The manage phase can include measurement tools that are used for monitoring the Web service and to accumulate performance data. In most real-life applications, clients would require a certain quality of service. Also, tools for authorization and statistics (billing) would be required.

Building a new Web service

In this section, we describe the different paths to use in the generation of a Web service. We find three principal approaches, depending on the elements that we use to start the creation of the service. A Web service can be implemented from:

- ▶ An existing application (bottom-up)—Transforming an existing application into Web services includes the generation of service wrappers to expose the business functionality.
- ▶ An existing service definition, WSDL, to generate a new application (top-down)—Generating a new application includes using a specific programming language and model.
- ▶ An existing group of already generated Web services to provide a new combination of functionality (multiple services)—Composing a new Web service might include the use of workflow technologies.

Bottom-up

The bottom-up approach is the most common way to build a Web service. We start with a business application that is already developed, tested, and running on the company systems. Figure 15-2 shows the bottom-up path.

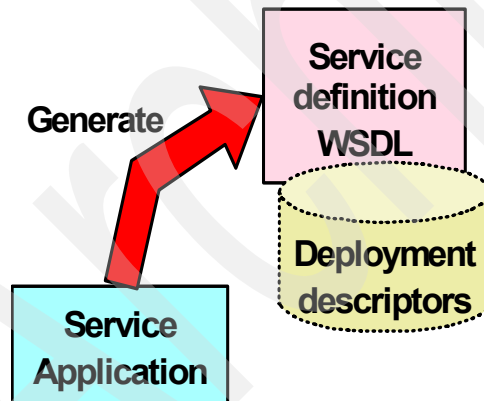


Figure 15-2 Bottom-up path

We start from the service application and create the WSDL document for it, providing all the functionality that we desire to externally expose as a Web service. Under certain conditions, we can create the WSDL using tools that create the WSDL for us and we can use the generated WSDL as a starting point for our Web service enablement. Depending of the implementation model (J2EE, for example), we also have to generate the deployment descriptors.

Top-down

The top-down approach is commonly used when we have a standard service definition and we want to implement this definition to provide the requested service.

The service definition might, for instance, come from an industry-sector agreement and might be implemented by any number of providers, for example, when a group of airlines agree on how to provide their plane schedules. In that case, there is a strict definition of what the providers receive and how they have to respond. Therefore, the providers have to adapt their current systems to follow the new specification.

Figure 15-3 shows the top-down path.

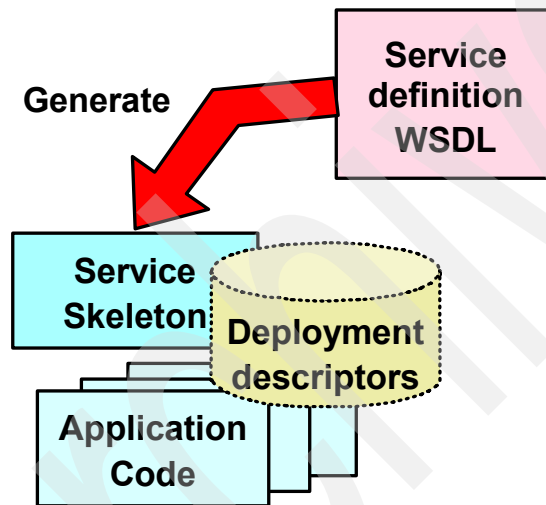


Figure 15-3 Top-down path

The process of creating a Web service using this path can be divided into the following three steps:

- ▶ Find the service interface—We localize the service definition to use it as the entry point for the implementation. We obtain the WSDL document through a proprietary channel from the service provider (e-mail, for example), or through a WSIL document, or by searching an UDDI registry.
- ▶ Generate the implementation skeleton—Using the service definition, we generate a skeleton with the methods and parameters that we have to fill in to implement the Web service.

- Develop the new Web service—Using the skeleton, we complete all the methods with the appropriate logic. Depending on the amount of code that we can reuse from other applications and the complexity of the service definition, we develop more or less new code. In a J2EE environment, we also generate deployment descriptors. Finally, we test the new Web service to check that everything runs smoothly.

Composing a Web service

The multiple services approach is commonly used when we have a collection of Web services running in one or more systems and we want to provide new functionality reusing the existing features provided by these Web services. In this path, we create a new integrated Web service combining some of the individual characteristics provided by the existing Web services and also other business modules. Figure 15-4 shows this path.

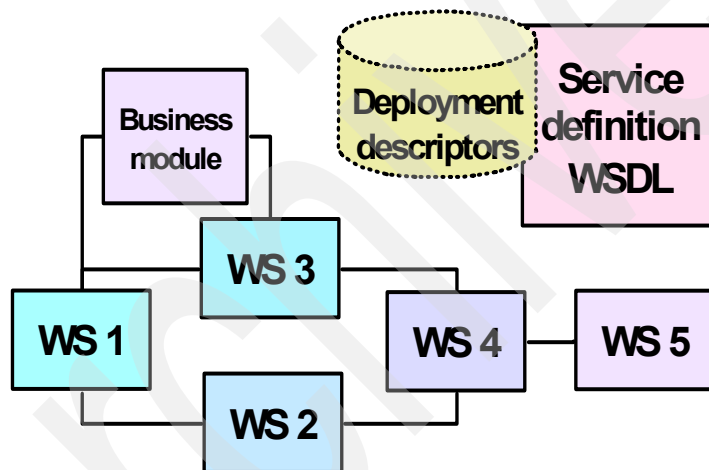


Figure 15-4 Multiple services path

The individual Web services, *WS n*, are linked sequentially or in a graph. Therefore, the output of one service is the input for the following service or business module. We can also create different outputs at runtime depending on the flow characteristics and the input data.

The multiple services path can have both previous approaches, bottom-up and top-down, in its implementation. The most common is the bottom-up alone or combined with the generation of one or more top-down services to complete the sequence. Conversely, we can consider the creation of a top-down Web service dividing the resulting service into multiple sub-services that can be useful for other applications.

Types of Web services implementation

The base for the J2EE implementation of a Web service can be varied from different modules and applications. These possibilities in the creation of a Web service depend on the tools and products that we are using in the generation process and the facilities that they provide to easily complete the creation task. Table 15-1 lists the tools and which Web services implementation types they facilitate on a particular path.

Table 15-1 Web services implementation facilities by products

Tool and runtime	Path	Implementation			
		JavaBean	EJB	ISD	URL
Rational Application Developer V6.0 (IBM WebSphere)	Bottom-up	Yes	Yes	No	No
	Top-down	Yes	Yes	No	No
	Multiple services (business process) implemented as an EJB service				
Rational Application Developer V6.0 (Apache Axis 1.0)	Bottom-up	Yes	Yes	Yes	Yes
	Top-down	Yes	Yes	No	No
Rational Application Developer V6.0 (IBM SOAP)	Bottom-up	Yes	Yes	Yes	Yes
	Top-down	Yes	Yes	No	No

ISD = Web service deployment descriptor (points to implementation code)

URL = Uniform Resource Locator (for a servlet)

Building a new Web service client

Web services for J2EE specifies three different types of clients (see “JAX-RPC client programming styles” on page 95). In this section, we explain the client types from a more general point of view.

The task to build or generate a Web service client (also known as a service requestor) depends on the methods of how the client is binding to a Web service server. The client uses a local service stub or proxy to access the remote server and service. The WSDL document is used to generate or set up this particular stub or proxy. The stub or proxy knows at request time how to invoke the Web service based on the binding information.

The methods of binding to a service are defined by the time when the various elements of binding information are available and used, namely at build time versus at runtime. This results in three client types:

- ▶ Static client
- ▶ Dynamic client with known service type
- ▶ Dynamic client with unknown service type

Note: In this context, the term *binding information* is used in a general sense and is not limited to the <wsdl:binding> section of a WSDL document.

Static client

The static client has a static binding created at build time. This is made possible, because in the development phase, we know the interface, the binding method, and the service endpoint of the Web service that we are going to invoke. We also decide that we only use this client for that specific service and nothing else. Therefore, in these cases, the best solution is to use a static client.

No public, private, or shared UDDI registry or WSIL document is involved in the runtime process. Figure 15-5 shows the process to generate a static client.

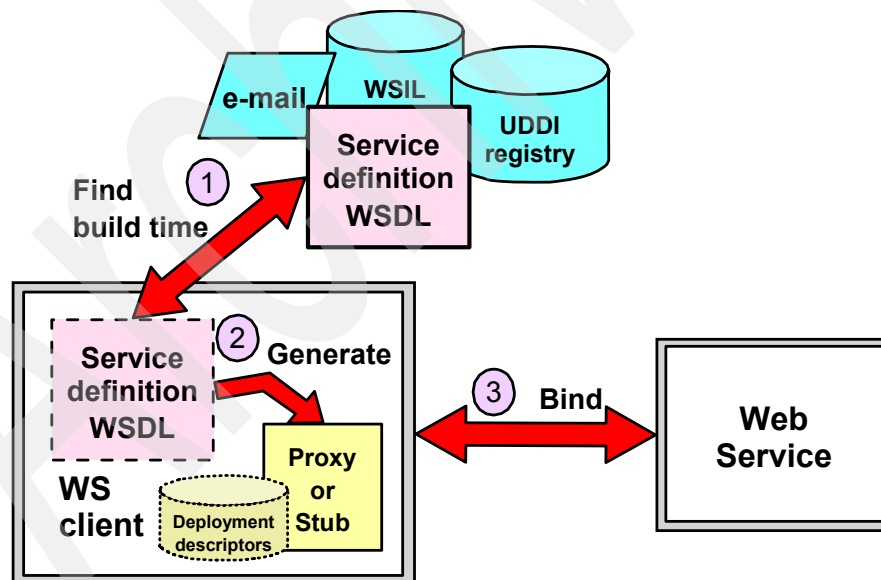


Figure 15-5 Static client path

The steps to build a static service client are:

1. Manually find the service definition or WSDL—We obtain the WSDL document through a proprietary channel from the service provider (e-mail, for example) or through a WSIL or looking in a UDDI registry. An important point is that we store the WSDL document into a local configuration file. Therefore, we have all the information previously defined before we start to code the client.
2. Generate the service proxy or stub—Using the information contained in the WSDL document, we generate the proxy or stub and probably the deployment descriptors. This stub is a local representation of the remote Web service. Depending on the tool or product we use in this step, the generation is performed automatically.
3. Test the client—We test the code to check that the client correctly operates and binds to the Web service.

Dynamic client with known service type

The dynamic client has a dynamic binding that is only known and decided on at runtime. This client is used when we know the interface of the service but not the implementation (the service endpoint). This means that the operation, the parameters associated with the operation, and the way to bind to the service are already known, but the address where this service is provided is not known.

An example of this is when an industry defines a service interface and the partner companies implement the specification. In that case, we only want to have one client that can dynamically change between the different providers based on certain criteria (performance, cost, and so forth).

The use of a public, private, or shared UDDI registry is involved in the process to dynamically provide to the client a range of entry points available at a specific time. Figure 15-6 shows the process to generate such a dynamic client.

Note: Instead of a UDDI registry, Web Services Inspection Language (WSIL) can also be used to create dynamic clients.

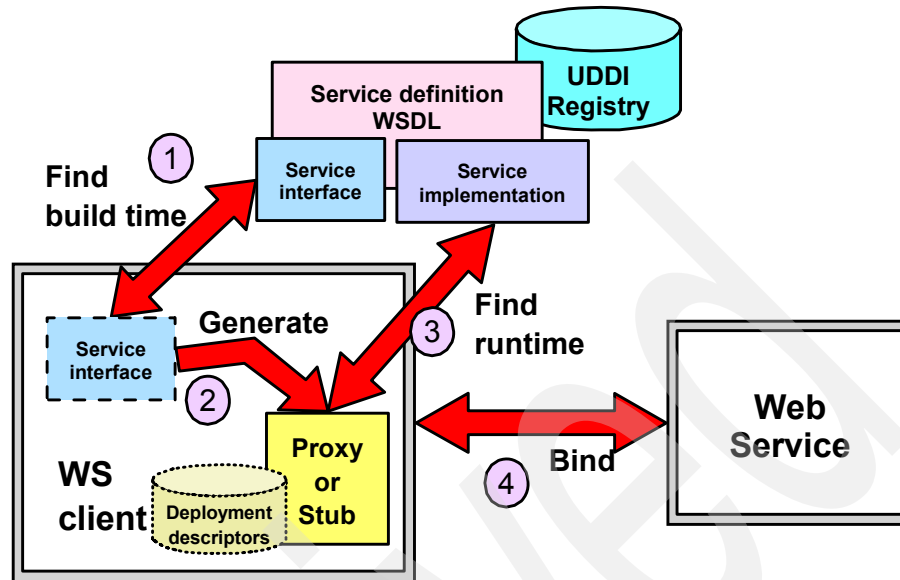


Figure 15-6 Dynamic client path

The steps to build a dynamic service client are:

1. **Manually find the service interface definition of the WSDL**—We obtain the service interface definition, the types, the messages, the port type with the operations, and the binding of the WSDL document through the same mechanism used with the static client. In this case, we are only interested in the service interface, which is what we load into a local configuration file. Therefore, the information about how to invoke the Web service is previously defined before we start to code the client.
2. **Generate the generic service proxy or stub**—Using the information contained in the service interface, we generate the generic proxy or stub and probably the deployment descriptors. This generic proxy or stub can be used to access any implementation of the service interface used in the generation process.
3. **The proxy or stub (or a helper class) contains the necessary code to locate a service implementation by searching a UDDI registry.** This UDDI lookup code is currently not generated by tools and must be hand-coded using the UDDI4J API.
4. **Test the client**—We test the code to check that the client correctly operates and binds to any Web services that implement the service interface dynamically.

Dynamic client with unknown service type

There are other, more dynamic ways to connect to a Web service. For example, at build time, we do not know the binding to connect to a Web service, or the binding is decided at runtime from among different possibilities. The steps to create this client are the same as for the previously described client, with the only difference being that the proxy is created at runtime with the binding information collected just before the connection. To create such a client, we can use the Web Services Invocation Framework (WSIF), which is deprecated by WebSphere Application Server Version 6.

An even more dynamic way is when we do not know anything about the service in advance, because it is in a dynamic invocation interface (DII). In these cases, the service client obtains the service definition WSDL document from a UDDI registry or WSIL document at runtime. There is no proxy code generation at build time; it is generated at runtime to bind and invoke the Web service. This kind of binding requires the presence of a user interface that can provide the input data and understand the meaning of the output.

This last path hardly has any real business application. However, it can be used to implement generic test clients.

Types of client implementations

The base for the Java implementation of a Web services client can vary depending on where the client can reside and what kind of module or application is built.

From the point of view of where the client can reside, the possibilities are:

- ▶ Stand-alone J2SE application client
- ▶ J2EE application in some middleware container (EJB container, Web container, application client container)

From the point of view of what kind of module or application is built, the possibilities are:

- ▶ Java class—Stand-alone J2SE or application client container
- ▶ JavaBean—Web container, or called by others
- ▶ EJB session bean—EJB container
- ▶ Servlet or JavaServer Page (JSP)—Web container
- ▶ Java applet—Browser

Summary

In this chapter, we presented the development concepts to create a Web service and a Web service client. We studied the different paths to follow or choose when we want to create a Web service. We also learned how to select these paths, depending on the starting situation in the case of Web services or the functional objectives in the case of the Web service clients.

More information

For a more comprehensive discussion of the topic, refer to the document *Web Services Development Concepts 1.0*, available at:

<http://www.ibm.com/software/solutions/webservices/pdf/WSDC.pdf>

Develop Web services with Application Developer V6.0

In this chapter, we explore some of the features of IBM Rational Application Developer for WebSphere Software Version 6 (Application Developer for short) as they relate to Web services.

If you are not familiar with the basic steps of how to create Web applications using Application Developer, refer to the Redbooks *Rational Application Developer Version 6 Programming Guide*, SG24-6449, and *EJB 2.0 Development with WebSphere Studio Application Developer*, SG24-6819 (although this publication is about Version 5.0).

This chapter goes through the following scenarios for creating Web services:

- ▶ Creating a Web service in a bottom-up way, using HTTP and JMS protocols
- ▶ Creating a Web service in a top-down fashion
- ▶ Creating a handler for a Web service
- ▶ Sending binary data as attachments
- ▶ Investigating atomic transactions

Overview

Rational Application Developer provides a toolbox for discovering, creating, and publishing Web services that are created from JavaBeans, DADX files, Enterprise JavaBeans, and URLs. You can also use the Web service tools to create a skeleton JavaBean and a sample application from a WSDL document. For more information about the Web service-related functions in Application Developer, refer to “Web services support in Rational Application Developer” on page 238.

The development path that you would typically follow to create and publish a Web service is as follows:

- ▶ Create router projects.
- ▶ Create or import an artifact to be turned into a Web service.
- ▶ Create a Web service.
- ▶ Create a proxy and a test client.
- ▶ Publish a business entity and a Web service to a registry.

Web tools assist you in developing Web applications that you can configure as a Web service. Web applications are developed in a Web project, and server tools enable you to use the unit test environment to test and deploy your Web services.

More information about the Web services development process can be found in Chapter 15, “Development overview” on page 261.

Selected scenarios

In the sections that follow, we focus on the following development scenarios:

- ▶ Bottom-up development by generating a Web service from a JavaBean using HTTP as the transport for the SOAP messages
- ▶ Bottom-up development by generating a Web service from a session EJB using HTTP as the transport for the SOAP messages
- ▶ Bottom-up development by generating a Web service from an EJB using JMS as the transport for SOAP messages
- ▶ Top-down development by generating a JavaBean from an existing WSDL
- ▶ Bottom-up generation of a Web service from a URL
- ▶ Create and configure service handlers
- ▶ Show how to handle binary data (attachments)
- ▶ Call a Web service as part of an atomic transaction

For other types of Web service generation, refer to the online documentation of Rational Application Developer.

Preparation

Before developing the Web services, Application Developer must be configured properly:

- ▶ For installation instructions, see “Installing Rational Application Developer V6.0” on page 654.
- ▶ For configuration instructions, see “Setting up Rational Application Developer” on page 657. This includes setting up preferences for Web services, as described in “Web services configuration settings” on page 241 and “Web services preferences” on page 242.
- ▶ The base code must have been imported, configured, and tested, as described in “Installing the base weather forecast application” on page 705.

Creating a Web service from a JavaBean

In this section, we create a Web service from an existing JavaBean, a wrapper around the weather forecast application introduced in Chapter 14, “Sample application: Weather forecast” on page 247.

We create the Web service using a bottom-up development method, which is a fast and easy way to get started with creating Web services.

We describe how to use the Application Developer wizard to create a Web service that returns information from the weather forecast service based on the JavaBean implementation. The wizard guides us through the process of creating the WSDL document and the proxy classes, deploying the service on the test server, and generating a sample application to test the Web service.

In this example, we create the server side of the Web service, publish it to a WebSphere Application Server V6 test environment, generate a test application, and test it using the Web Services Explorer and the generated test application.

Web Service wizard

To start the Web Service wizard on our Java bean:

- ▶ Switch to the Web perspective Project Explorer.
- ▶ Navigate to the JavaBean by expanding *Dynamic Web Projects* → *WeatherJavaBeanWeb* → *Java Resources* → *JavaSource* → *itso.bean*.
- ▶ Select the WeatherJavaBean.
- ▶ From the context menu, select *Web Services* → *Create Web service*.

Note: If you cannot find *Web Services* on the context menu, verify that the Web service capability is enabled. See “Web services configuration settings” on page 241.

We proceed now page-by-page through the wizard and explain each page. Later, when we create more Web services, we do not explain each step in detail.

Web Services page

On the Web Services page, select the following options (Figure 16-1):

- ▶ **Web service type:** *Java bean Web Service*
Here, we specify what type of Web service we create. The choices are:
 - DADX Web Service—Create a Web service from a stored procedure or SQL statement.
 - Skeleton Java bean Web Service—Create a skeleton from a WSDL file.
 - EJB Web Service—Create a Web service from an enterprise bean.
 - Skeleton EJB Web Service—Create a session EJB skeleton from a WSDL file.
 - Java bean Web Service—Create a Web service from a JavaBean.
 - ISD Web Service—Create a Web service from a Web service deployment descriptor.
 - URL Web Service—Create a Web service that returns the content from a URL (for example, a servlet).
- ▶ Select *Start Web service in Web project*. If you do not select this option, you have to manually start the Web service. This option also enables other options on this page.
- ▶ Clear *Launch Web Services Explorer to publish this Web service to a UDDI Registry*, because we do not want to publish our service to a UDDI.
- ▶ Select *Generate a proxy*. The wizard generates client proxy classes enabling simple method calls in a client program to call the Web service. In most cases, these proxy classes implement the JAX-RPC API.
- ▶ For Client proxy type, select *Java proxy*. The other proxy type is *Web Service User-Defined Function* to invoke a Web service from a stored procedure or SQL statement.
- ▶ Select *Test the Web service*. This option lets you test the Web service using the Web Services Explorer before the proxy is generated. It also enables you to create a test client (a set of JSPs) in a client project.

- ▶ Clear *Monitor the Web service*. This option lets you monitor your Web service using the TCP/IP Monitor by routing the traffic through the monitor and configuring the monitor for the server on which the service is deployed. For information about how to use the TCP/IP Monitor, refer to “TCP/IP Monitor” on page 368.
- ▶ Select *Overwrite files without warning*, or you might get warning pop-up messages. Select this option to rerun the wizard for the same Web service.
- ▶ Select *Create folders when necessary*. A number of folders are created for the results of the generation.
- ▶ Clear *Check out files without warning*. This option applies to the team environment.
- ▶ Click *Next* to proceed to next page.

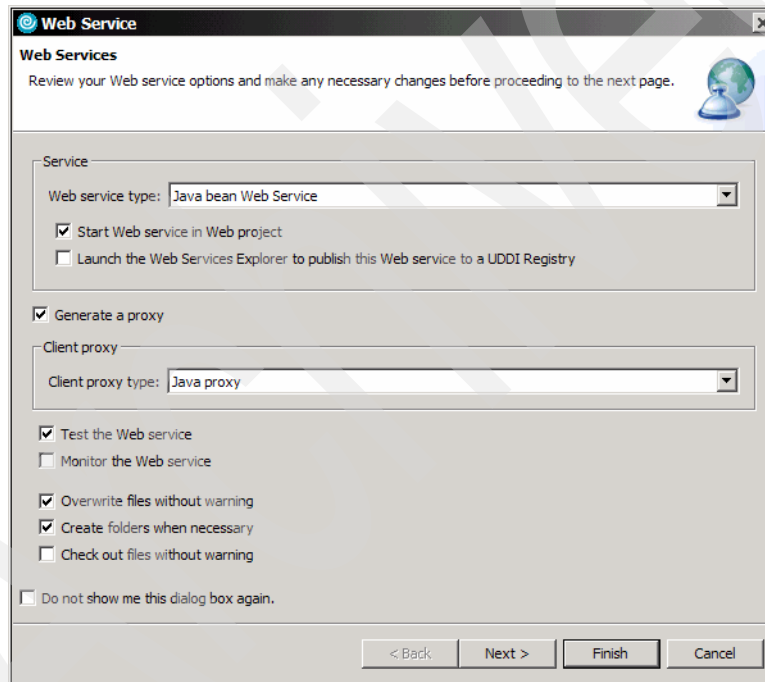


Figure 16-1 Web Service wizard: Web Services

Object Selection page

On the Object Selection page, you can specify from which JavaBean the Web service is generated (Figure 16-2). Ensure that the Bean selected is `itso.bean.WeatherJavaBean`. If it is not selected, use *Browse classes* or *Browse files* to select the weather JavaBean.

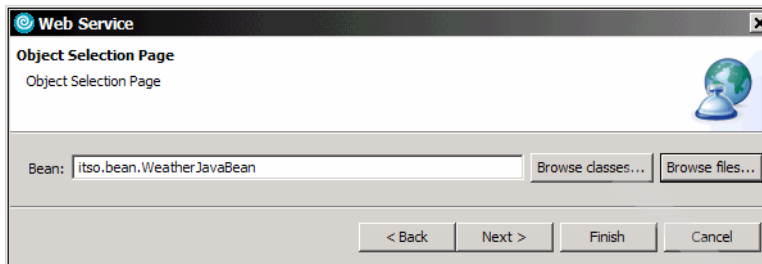


Figure 16-2 Web Service wizard: Object Selection

Service Deployment Configuration page

On the Service Deployment Configuration page, specify the deployment settings for the service and the generated test client (Figure 16-3).

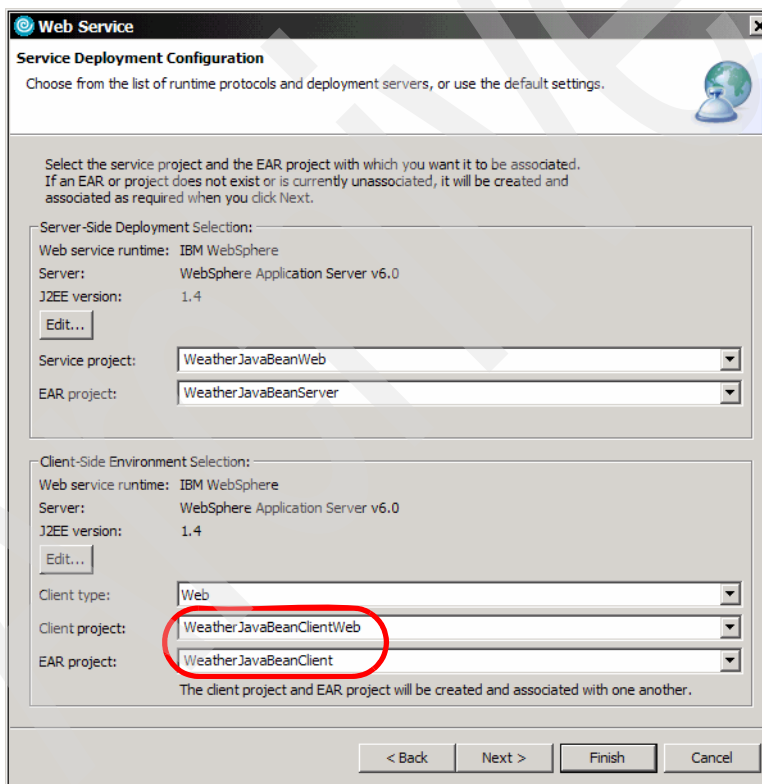


Figure 16-3 Web Service wizard: Service Deployment Configuration

Both the weather service and the test client will be deployed on WebSphere. Therefore, make sure that the following is selected for both server-side and client-side:

- ▶ Web service runtime: IBM WebSphere
- ▶ Server: WebSphere Application Server v6.0
- ▶ J2EE Version: 1.4

For the Client type, specify *Web*. Other choices are *EJB*, *Application Client* (running in a J2EE client container), and *Java* (stand-alone class).

Make sure that the test client is created as a new enterprise application by specifying the client-side projects, as shown in Figure 16-3:

```
Client project: WeatherJavaBeanClientWeb
EAR project: WeatherJavaBeanClient
```

Important: Always verify the generated project names. Application Developer inserts default names that might not be your choice.

If you generate the client code into the wrong project (for example, a server project), it is very hard to back out the changes unless you work in a team environment with a repository.

Service Endpoint Interface Selection page

On this page, it is possible to use an existing service endpoint interface (SEI), but not select the option and have the wizard generate the interface (Figure 16-4).

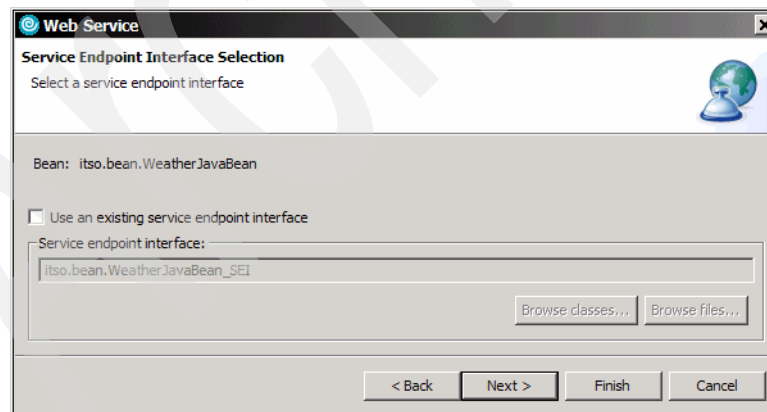


Figure 16-4 Web Service wizard: Service Endpoint Interface Selection

Web Service Java Bean Identity page

The identify page shows the Web service URI and WSDL name and gives us these options (Figure 16-5):

- ▶ Specify the name of the WSDL file (leave the default name).
- ▶ Select which methods to expose. In this example, we expose all methods.
- ▶ Select *Document/Literal* for Style And Use, because this is WS-I compliant. See “Messaging modes” on page 55 for more about styles and encodings.
- ▶ Select *No Security* for the Security Configuration. We explore security options in Chapter 21, “Securing Web services” on page 445.
- ▶ Select if the service exclusively uses the WSDL 1.1 Mime attachment or the swRef method. Our example does not use attachments, so just leave it as is.
- ▶ Select if custom package to namespace mapping is required. In this example, the default mappings are fine.

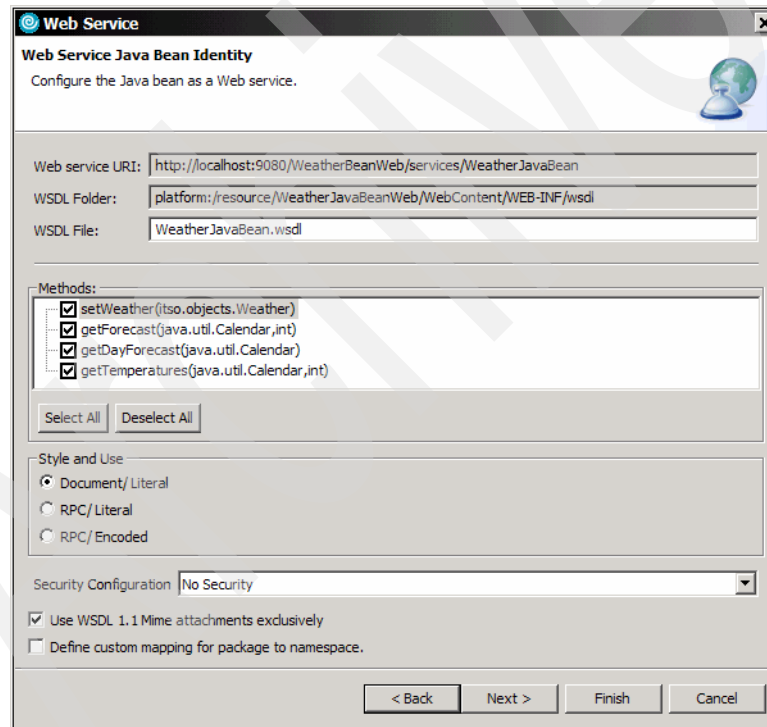


Figure 16-5 Web Service wizard: Identity

Click *Next*. The SEI, helper classes, the WSDL file, and the Web service deployment descriptor are generated into the service project.

Web Service Test page

On this page, you can launch the Web Services Explorer to test the Web service before generating the proxy (Figure 16-6). This page is only available if *Test the Web Service* has been selected on the first wizard page.

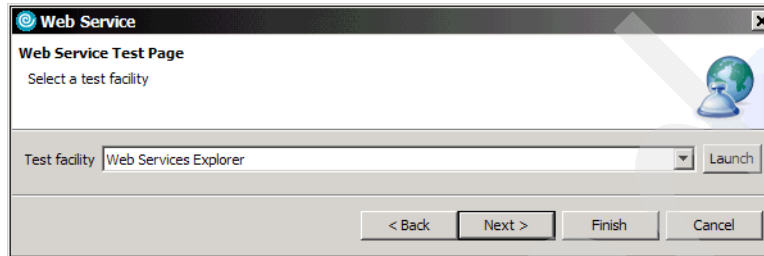


Figure 16-6 Web Service wizard: Test

For now, we will not launch the Web Services Explorer. We can always launch it later, as described in “Testing with the Web Services Explorer” on page 286.

For more information about using the Web Services Explorer, refer to “Web Services Explorer” on page 339.

Web Service Proxy page

On this page, select the options shown in Figure 16-7.

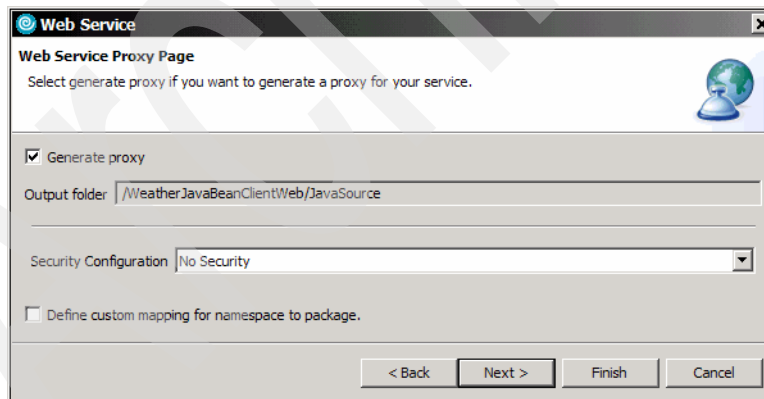


Figure 16-7 Web Service wizard: Proxy

Select the following options:

- ▶ *Generate proxy.* This creates proxy classes for the Web service into the client project.

- ▶ Select *No Security*. We explore secure Web services in Chapter 21, “Securing Web services” on page 445.
- ▶ On this page, it is also possible to customize the mapping between namespaces and Java packages on the client side.
- ▶ Click *Next*, and the Web service is stored in the server.

Web Service Client Test page

Use this page to specify how to test the generated proxy. You can test the generated proxy with Web service sample JSPs, the Web Services Explorer, or the Universal Test Client (Figure 16-8). This page is only available if *Test the Web Service* has been selected on the first wizard page.

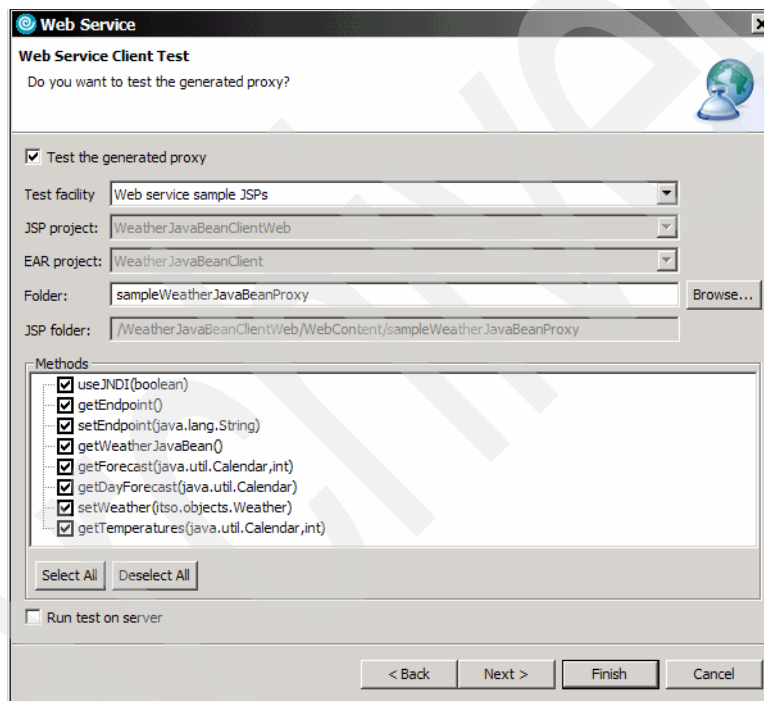


Figure 16-8 Web Service wizard: Client Test

If you choose to generate the JSP client, you can specify which methods should be included and which folder should contain the JSP files (leave the defaults).

You can also have the wizard start the JSP client. Clear *Run test on server*, and then we can start the test client also manually.

For more information about testing with the JSP sample application, refer to “Web services sample test JSPs” on page 345.

Web Service Publication page

Leave both options cleared on this page, because the weather service will not be published to an UDDI registry (Figure 16-9).

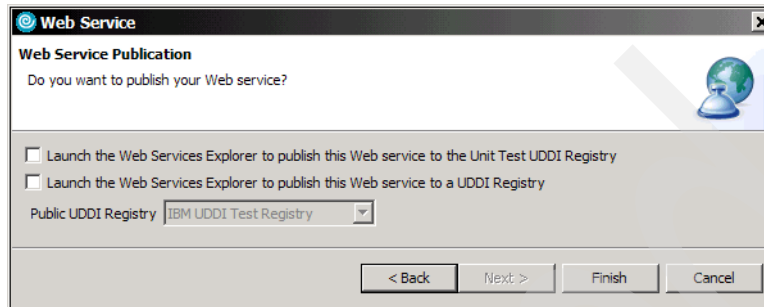


Figure 16-9 Web Service wizard: Publication

Click *Finish* to complete the Web Service wizard.

Notice in the console view that the server has been updated with the client project.

When the wizard is finished, a Web browser opens with the JSP sample application, and you can now test the newly created Web service.

Generated files

Let us have a closer look at the files generated by the wizard (Figure 16-10).

Files generated in the server project

According to the settings made during the run of the wizard, the following files in the WeatherJavaBeanWeb project have been created:

- ▶ Service endpoint interface (SEI): `itso.bean.WeatherJavaBean_SEI.java` is the interface defining the methods exposed in the Web service.
- ▶ WSDL file: `/WebContent/WEB-INF/wsdl/WeatherJavaBean.wsdl` describes the Web service. A copy of this file is also placed in the `WebContent/wsdl` folder.
- ▶ Deployment descriptor: `webservices.xml`, `ibm-webservices-ext.xml` and `ibm-webservices-bnd.xml`. These files describe the Web service according to the Web services for J2EE style (JSR 109). The JAX-RPC mapping is described in the `WeatherJavaBean_mapping.xml` file. For more information about JSR 109 deployment, refer to Chapter 6, "Web Services for J2EE" on page 103.

- ▶ Data mapping files: The files in the `itso.objects` package perform the data conversion from XML to Java objects and back.
- ▶ A servlet is defined in the Web deployment descriptor to invoke the JavaBean.

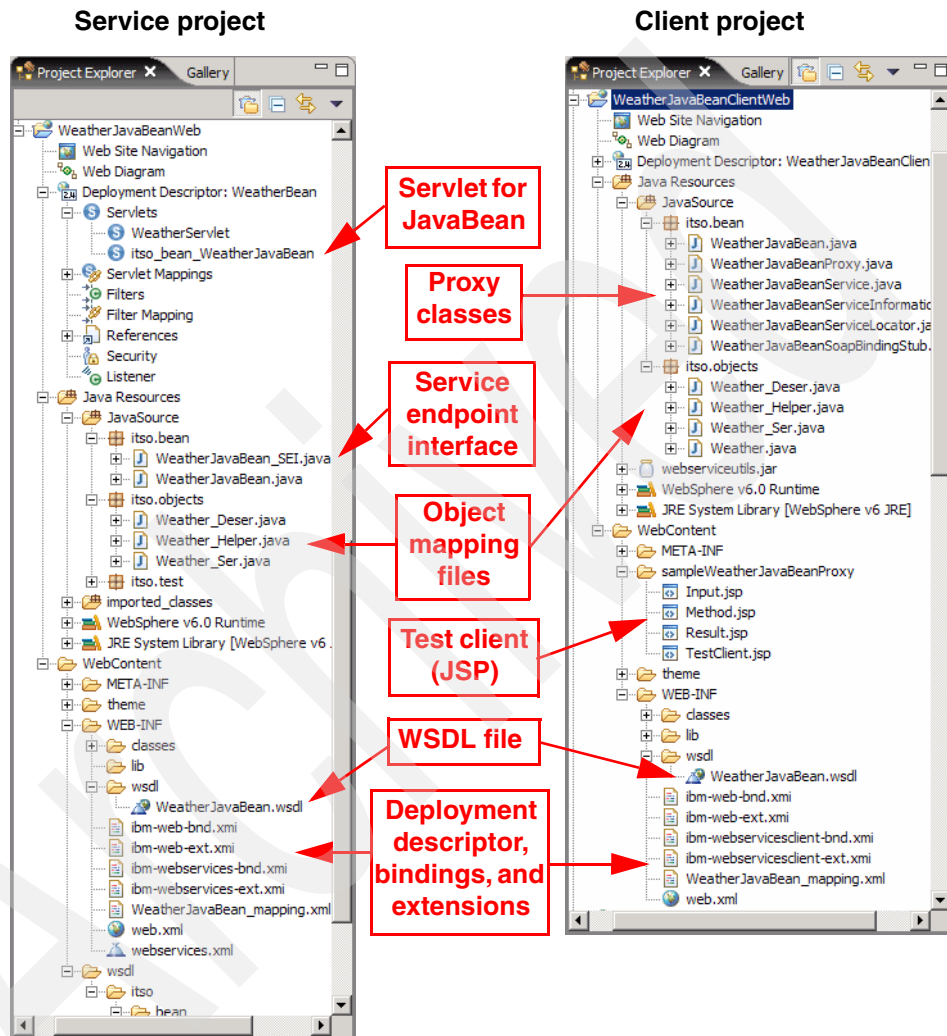


Figure 16-10 Wizard-generated files in the Project Explorer

Note: In the Project Explorer, you also find a separate Web Services section with Services and Clients (Figure 16-14 on page 289).

Files generated in the client project

If the creation of a client-side proxy is selected, two packages are generated in the WeatherJavaBeanClientWeb project:

- ▶ `itso.bean` contains the proxy classes. These classes are used by the client to make remote calls as per JSR 101 or JSR 109. With the help of these classes, the client can instantiate local representations of the remote classes. The generated test JSPs also use these proxy classes.
- ▶ `itso.objects` contains the Java to XML and XML to Java data mapping logic.
- ▶ Test client: JSPs to test each method exposed as a Web service. The test client is generated in the `WebContent/sampleWeatherJavaBeanProxy` folder.
- ▶ Deployment descriptor: `web.xml` and extensions. These files describe the Web service in the client according to the Web services for J2EE style (JSR 109). The JAX-RPC mapping file is also there.

Note that the `webservicesclient.xml` file (used in Version 5) is not generated; the information is included in the Web deployment descriptor `web.xml`.

- ▶ A copy of the WSDL file (in `WEB-INF\wsdl`).

Proxy classes

Let us have a closer look at the proxy classes:

- ▶ **WeatherJavaBean**—This is a copy of the service endpoint interface (SEI) containing the methods of the Web service.
- ▶ **WeatherJavaBeanServiceLocator**—This class contains the address of the Web service (`http://localhost:9080/WeatherBeanWeb/services/WeatherJavaBean`) and methods to retrieve the address and the SEI:

```
getWeatherJavaBeanAddress()  
getWeatherJavaBean()  
getWeatherJavaBean(URL)
```

- ▶ **WeatherJavaBeanService**—This is the interface implemented by the `WeatherJavaBeanServiceLocator`, consisting of the three methods listed above.
- ▶ **WeatherJavaBeanSoapBindingStub**—This class implements the SEI for the client. An instance of this class is returned by the `getWeatherJavaBean` methods of the locator.
- ▶ **WeatherJavaBeanServiceInformation**—This class contains descriptive information about the Web service and is not used at execution.
- ▶ **WeatherJavaBeanProxy**—This is a helper class that includes the APIs of JSR 101 and JSR 109 to get the SEI from the locator. Clients can call the Web services methods (for example, `getDayForecast`), and the internal method `_initWeatherJavaBeanProxy` returns the SEI for execution.

Testing the Web service

The Web service is now installed and running in the server, and we can test it using multiple methods:

- ▶ Web Services Explorer
- ▶ Test client JSPs
- ▶ Universal Test Client

We briefly show the first two methods here. For more details about testing Web services, see Chapter 17, “Test and monitor Web services” on page 337.

Testing with the Web Services Explorer

To start the Web Services Explorer, select the `WeatherJavaBean.wsdl` file (in `WeatherJavaBeanWeb/WEB-INF/wsdl`) and *Web Services* → *Test with Web Services Explorer* (context).

A Web Browser view opens with the WSDL file selected. It shows the operations (methods) that can be invoked and the endpoint (Figure 16-11).

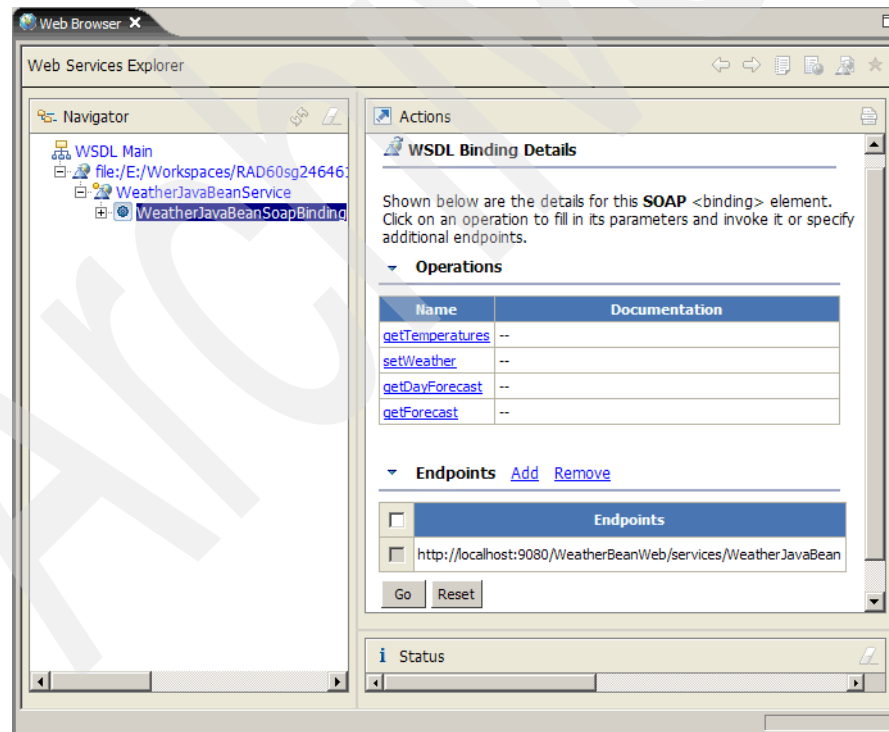


Figure 16-11 Web Services Explorer: Operations

Do the following:

- ▶ Select a method, for example, `getDayForecast`.
- ▶ You are prompted to enter the date parameter.
- ▶ Click *Browse* to select a date from the calendar.
- ▶ Click *Go* to execute the Web service. The Status pane displays the results (Figure 16-12).

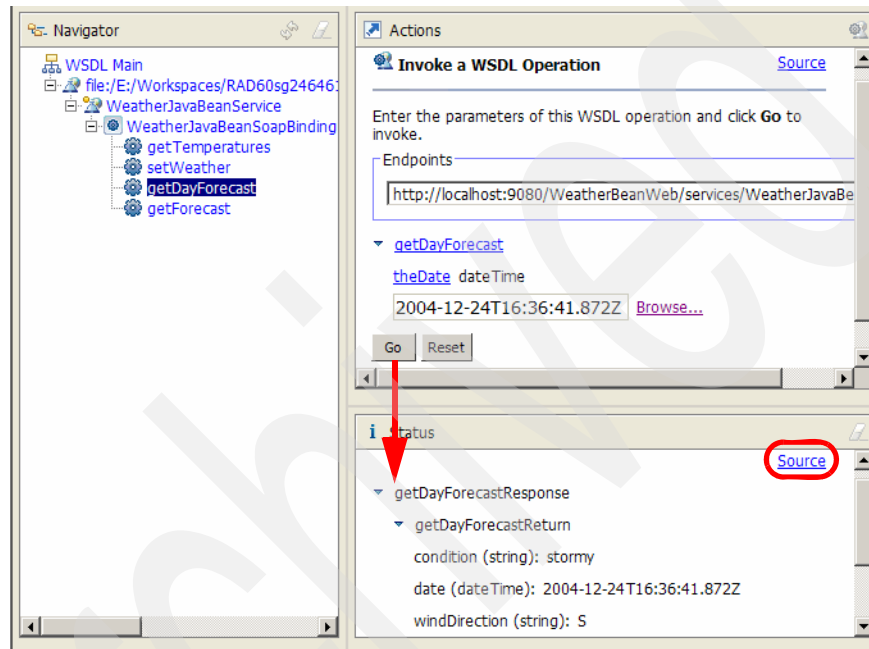


Figure 16-12 Web Services Explorer: Execution

- ▶ Click *Source* to see the SOAP input and output messages (double-click the Status pane header to maximize it).
- ▶ Execute other methods to validate the Web service functionality.
- ▶ Close the browser.

For more information about the Web Services Explorer, see “Web Services Explorer” on page 339.

Testing with the test client JSPs

The test client JSPs have been generated into the `WeatherJavaBeanClientWeb` project:

- ▶ Select the `TestClient.jsp` (in `WebContent/sampleWeatherJavaBeanProxy`) and *Run* → *Run on Server* (context).

- ▶ When prompted, select the *WebSphere Application Server v6.0* and *Set server as project default*. Click *Finish*.
- ▶ The test client opens in a Web Browser (Figure 16-13). Select a method, enter the parameter or parameters, click *Invoke*, and the results are displayed.

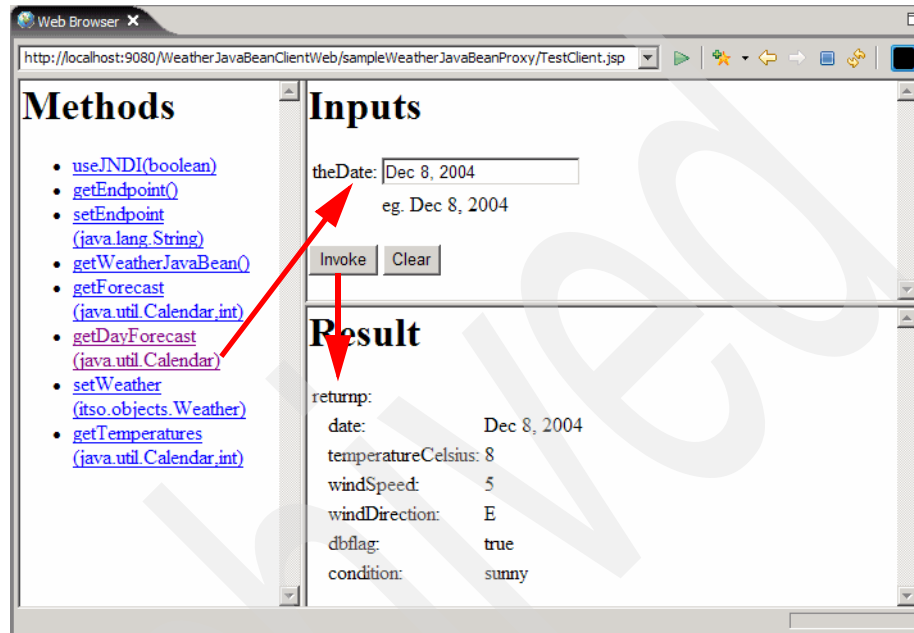


Figure 16-13 Test client run

For more information about the test client, see “Web services sample test JSPs” on page 345.

Creating Web service clients

Having the JavaBean Web service generated and running enables us to write real Web service clients.

Web service clients can execute in different environments:

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

Stand-alone Java client

A stand-alone client is a Java program that invokes the Web service using the generated proxy classes. A stand-alone client can only use the JSR 101 API, that is, without using JNDI to find the service locator.

Generating proxy classes

To generate the Java client proxy classes using the wizard, perform these steps:

- ▶ Create a Java project named `WeatherClientStandalone` to contain the client code (*File* → *New* → *Project* → *Java* → *Java Project*).
- ▶ Client proxy classes are generated from the WSDL file. You can locate the WSDL file in the project or in the Web Services section under Services (Figure 16-14).

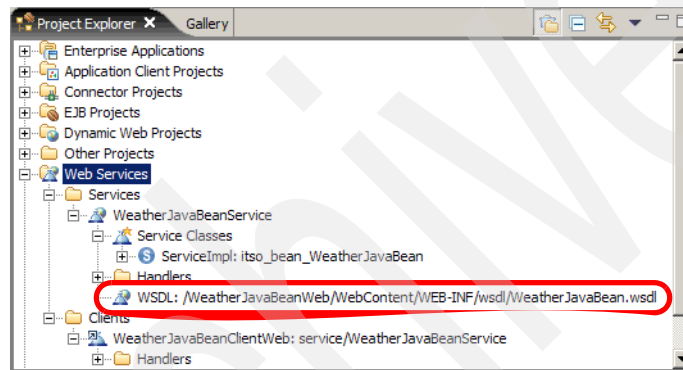


Figure 16-14 Web Services in Project Explorer

- ▶ Select the `WeatherJavaBean.wsdl` file and *Generate Client* (context). This starts the Web Service wizard with a set of pages that relate to clients.
- ▶ On the Web Services page, make sure *Java proxy* is selected as the client proxy type. Leave other options to their default settings.
- ▶ On the Web Service Selection page, the `WeatherJavaBean.wsdl` is preselected.
- ▶ On the Client Environment Configuration page, select:
 - Client type: *Java*
 - Client project: `WeatherClientStandalone` (use the pull-down menu)
- ▶ On the Web Service Proxy page, select *No Security*.
- ▶ On the Web Service Client Test page (if it appears), clear *Test the generated proxy*. We do not want sample JSPs generated.
- ▶ Click *Finish*.

The wizard generates the proxy classes (itso.bean package) and helper classes (itso.objects package). Expand *Other Projects* → WeatherClientStandalone to locate the generated code.

Creating the Java client

To continue creating the Java application, perform these steps:

- ▶ In the WeatherClientStandalone project, create a package (itso.client) and a class (WeatherJavaClient) in the new package.
- ▶ Import or copy the sample code into the WeatherJavaClient.java file. The client code is shown in Example 16-1 and is available in the sample code:

SG246461\sampcode\clients\standalone\WeatherJavaClient.java

Example 16-1 Stand-alone Java client source (compressed)

```
package itso.client;

import itso.bean.WeatherJavaBeanProxy;
import itso.objects.Weather;
import java.text.SimpleDateFormat;
import java.util.Calendar;

public class WeatherJavaClient {

    private static String toString(Weather w) {
        SimpleDateFormat sdf = new SimpleDateFormat("EEE. MMM d, yyyy zzz");
        sdf.setTimeZone(w.getDate().getTimeZone());
        return "Weather: " + sdf.format(w.getDate().getTime()) + ", "
            + w.getCondition() + ", wind: " + w.getWindDirection() + " at "
            + w.getWindSpeed() + "km/h " + ", temperature: "
            + w.getTemperatureCelsius() + " Celsius ";
    }

    public static void main(String[] args) {
        System.out.println("WEATHER FORECAST - STANDALONE CLIENT");
        System.out.println("=====");
        Weather w = null;
        Calendar today = Calendar.getInstance();
        try {
            System.out.println("Getting WeatherForecast proxy ...");
            WeatherJavaBeanProxy proxy = new WeatherJavaBeanProxy();
            //proxy.setEndpoint("http://.../services/WeatherJavaBean");
            //proxy.setEndpoint("http://.../services/WeatherEJB");
            System.out.println("Using endpoint: "+proxy.getEndpoint());
            System.out.println("Invoking getDayForecast ...");
            w = proxy.getDayForecast(today);
            System.out.println("Todays weather: " + toString(w));
            System.out.println("Raise temperature by 5 degrees ...");
        } catch (Exception e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

```

        w.setTemperatureCelsius( w.getTemperatureCelsius() + 5 );

        System.out.println("Invoking setWeather ...");
        proxy.setWeather(w);
        System.out.println("Invoking getDayForecast ...");
        w = proxy.getDayForecast(today);
        System.out.println("Warmer weather: " + toString(w));
    } catch (Exception e) { e.printStackTrace(); }
    System.out.println("End");
}
}

```

- Open the WeatherJavaBeanProxy class and change the `_useJNDI` value to `false` (a stand-alone client cannot use JNDI):

```
private boolean _useJNDI = false;
```

Running the Java client

Before running the client, make sure that the test server and the Web service are running. Select the WeatherJavaClient class and *Run* → *Java Application*. The console output from a successful run should look similar to Example 16-2.

Example 16-2 Stand-alone Java client run

```

WEATHER FORECAST - STANDALONE CLIENT
=====
Getting WeatherForecast proxy ...
Using endpoint: http://localhost:9080/WeatherBeanWeb/services/WeatherJavaBean
Invoking getDayForecast ...
Today's weather: Weather: Thu. Dec 9, 2004 GMT, sunny, wind: SW at 45km/h ,
                  temperature: 21 Celsius
Raise temperature by 5 degrees ...
Invoking setWeather ...
Invoking getDayForecast ...
Warmer weather: Weather: Thu. Dec 9, 2004 GMT, sunny, wind: SW at 45km/h ,
                  temperature: 26 Celsius

```

Web JavaServer Faces client

To illustrate how a Web service can be invoked from a rich HTML client, we create a JavaServer Faces (JSF) page that invokes the weather forecast Web service.

Preparation

Create a dynamic Web project (*File* → *New* → *Dynamic Web Project*) named WeatherClientJSF. Click *Show Advance* and enter WeatherClientEAR as the EAR project. Click *Finish*.

Select the WebContent folder and *New* → *Faces JSP File* (context). Enter WeatherReport as the name and click *Finish*. The Page Designer opens.

Locating the Web service

Application Developer provides tooling to invoke a Web service from a JSF page:

- ▶ In the Page Data view, select *New* → *Web Service* (context). The Web Service Discovery dialog opens. The purpose is to locate the WSDL file and generate the Java proxy and helper beans.
- ▶ The WSDL file can be found in a registry by URL (known location) or by searching the workspace for running Web services. We locate the Web service in the workspace, and then select it (Figure 16-15).

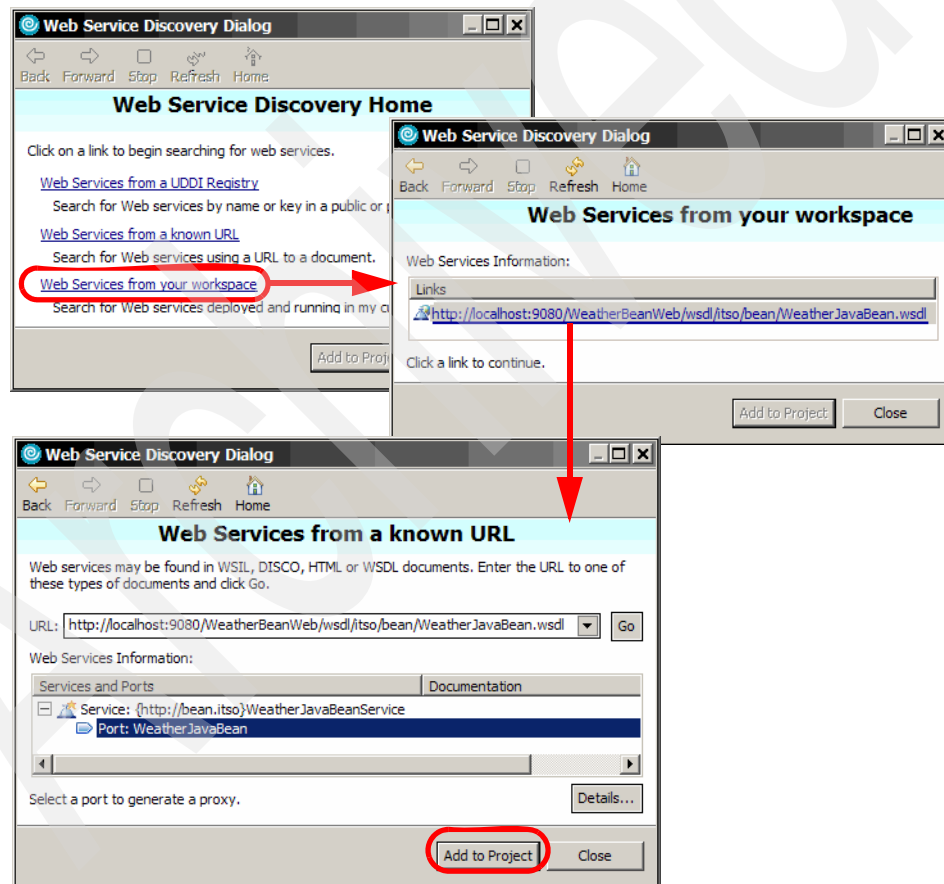


Figure 16-15 Web Service Discovery: Locating a Web service

- Click *Add to Project*. We are prompted to select the method to invoke, for example, `getForecast` (Figure 16-16).

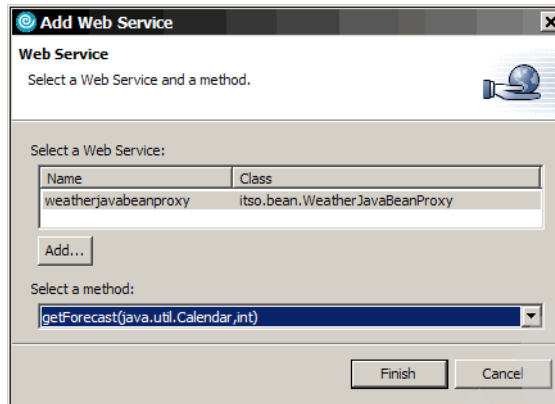


Figure 16-16 Selecting the Web service method

- Click *Finish*. Client proxy classes and helper beans are generated. The proxy appears in the Page Data view with a result bean and a parameter bean (Figure 16-17).

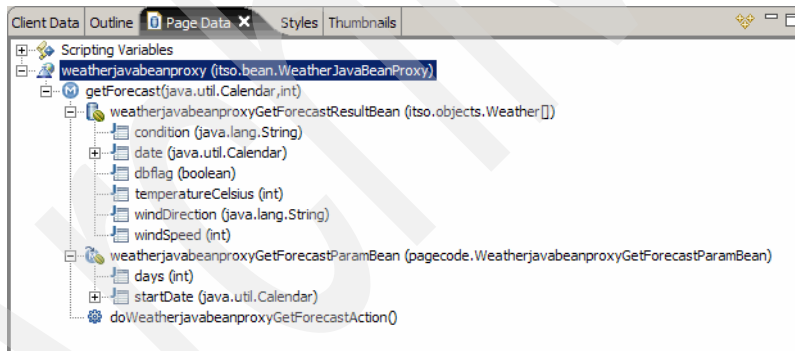


Figure 16-17 Web service proxy, result bean, and parameter bean

- We use these beans to build the JSF page.

Building the JSF page

We construct the layout of the JSF page in a few steps (Figure 16-18):

- Change the heading to a suitable text (Weather Report From Web Service).

- ▶ Expand the parameter bean and drag/drop the days property under the heading (we use the current date as the date). In the dialog box, change the label to Number of Days. Click *Options* to generate a button labeled Invoke Web Service. Click *Finish*.
- ▶ Drag/drop the result bean under the input form. In the dialog box, click *None*, and then select only date.time, temperatureCelsius, condition, windDirection, and windSpeed. Order the fields in this sequence using the arrows on the right. Change the labels as desired, for example, change Time to Date. Click *Finish*.
- ▶ Change the border of the data table to 1 in the Properties view.

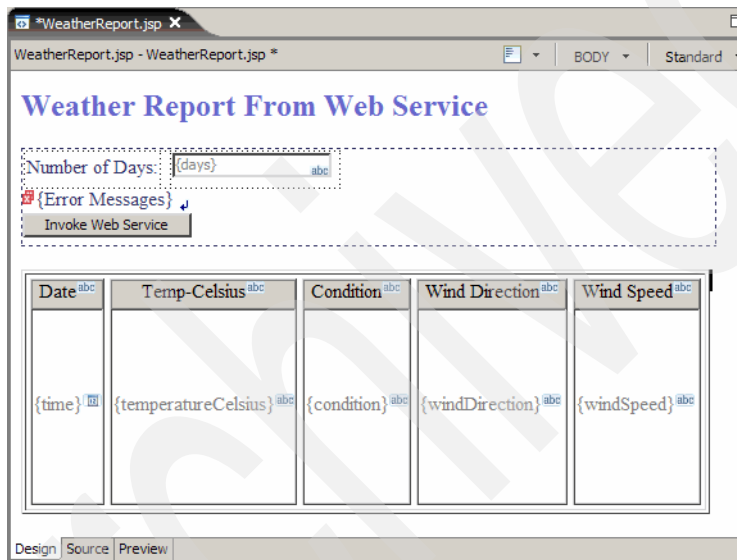


Figure 16-18 JSF weather report layout

Adding the logic

When the *Invoke Web Service* button is clicked, we execute the Web Service:

- ▶ Select the *Invoke Web Service* button. In the Quick Edit view, select *Command*, click in the right pane, and enter the logic:


```
java.util.Calendar today = java.util.Calendar.getInstance(); // today
getWeatherjavabeanproxyGetForecastParamBean().setStartDate(today);
doWeatherjavabeanproxyGetForecastAction();
return "";
```
- ▶ The `doWeatherjavabeanproxyGetForecastAction` method was generated into the page code. This method executes the Web service and stores the result in the result bean.

- You can open the page code (WeatherReport.java) by selecting *Edit Page Code* anywhere in the Page Designer, and then look for the method code:

```
public String doWeatherjavabeanproxyGetForecastAction() {
    try {
        weatherjavabeanproxyGetForecastResultBean =
            getWeatherjavabeanproxy().getForecast(
                getWeatherjavabeanproxyGetForecastParamBean()
                    .getStartDate(),
                getWeatherjavabeanproxyGetForecastParamBean().getDays());
    } catch (RemoteException e) {
        logException(e);
    }
    return null;
}
```

- Save the JSF page and the page code.

Running the JSF application

To run the application, select WeatherReport.jsp and *Run* → *Run on Server*. You might have to restart the project or server if the run fails. Figure 16-19 shows a sample run.

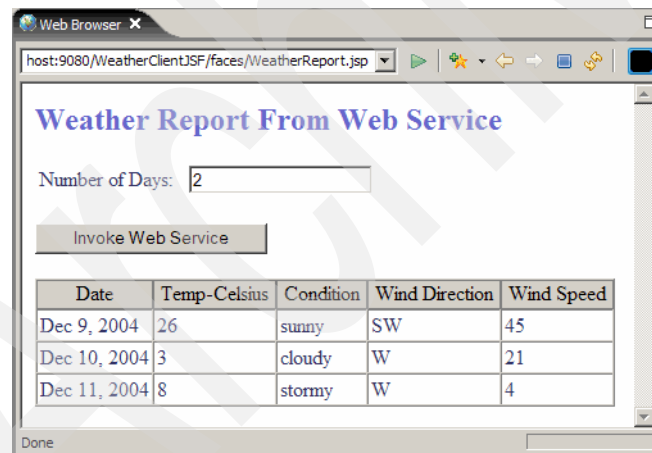


Figure 16-19 JSF Web service run

Improving the application

We can display a graphical weather forecast symbols instead of text:

- Create a folder named images under WebContent.
- Select the images folder and *Import* → *File system*.

- ▶ Locate the folder \SG246461\sampcode\weather-images, select all five images, and click *Finish*.
- ▶ In Page Designer, delete the {condition} field in the Condition column.
- ▶ Drag an Image component from the Palette view into the Condition column.
- ▶ Select the image in Page Designer. In the Properties view, click the icon next to the File property and select *Bind* in the pop-up menu. In the dialog box, expand the weatherjavabeanproxy and select the condition property of the result bean and click *OK*.
- ▶ The binding is inserted into the File property. Change the value to point to the imported images:


```
old: #{varweatherjavabeanproxyGetForecastResultBean.condition}
new: /images/#{varweatherjavabeanproxyGetForecastResultBean.condition}.jpg
```
- ▶ Rerun the application (Figure 16-20). Now, we see the images.

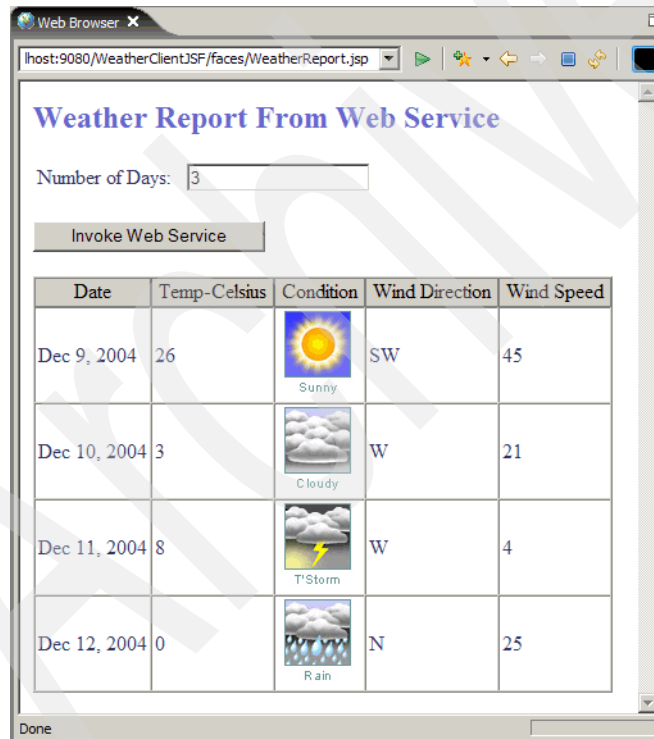


Figure 16-20 JSF Web service run (improved)

The sample code of this application (JSP, page code, parameter bean) is in \SG246461\sampcode\client\jsf.

Creating a Web service from a session bean

We provide a session bean, WeatherEJB, which implements the interface IWeatherForecast. In this section, we create a Web service from the session bean contained in the enterprise application WeatherEJBServer.

For this example, we use SOAP over HTTP. A Web router project is required with a servlet to route the client requests to the session EJB. We have to create this Web project in advance.

We do not create a client project this time. We can use the Web Services Explorer for testing, and we can use one of our existing clients and reroute the call to the session EJB Web service.

Running the Web Service wizard

The process of creating a Web service from a session bean is very similar to creating a service from a JavaBean, so the instructions in this section are shorter than in the previous section:

- ▶ Create a Dynamic Web Project with the name WeatherEJBRouterWeb in the WeatherEJBServer EAR project. Click *Show Advanced* to select the EAR project as WeatherEJBServer. Click *Finish*.
The router project is used to route service calls to the session bean and should not be used for anything else.
- ▶ In the Project Explorer, expand *EJB Projects* → *WeatherEJB* → *Deployment Descriptor* → *Session Beans*.
- ▶ Select the WeatherEJB and *Web Services* → *Create Web service* (context) to start the wizard.
- ▶ On the Web Services page, the *Web service type* is preselected as *EJB Web Service*. Change the options:
 - Select *Start Web service in Web project*.
 - Clear *Generate a proxy*.
 - Clear *Test the Web service*.
 - Select *Overwrite files without warning*.
 - Select *Create folders when necessary*.
- ▶ On the Object Selection page, only the WeatherEJB is shown.
- ▶ On the Service Deployment Configuration page, verify that the service project is WeatherEJB and the EAR project is WeatherEJBServer. The runtime should be IBM WebSphere, WebSphere Application Server V6.0, and J2EE V1.4.
- ▶ On the Web Service EJB Configuration page, enter WeatherEJBRouterWeb as the router project. Verify that the transport is *SOAP over HTTP*.

- ▶ On the Web Service Java Bean Identify page, accept all defaults. We will publish all methods of the session bean and use *Document/Literal*.
- ▶ Click *Finish*; the only remaining page is for UDDI publishing.

When the wizard finishes, the created Web service is deployed and started on the test server. You can use the Web Services Explorer to test the Web service:

- ▶ Select the WeatherEJB.wsdl file (in the EJB project under META-INF) and *Web Services* → *Test with Web Services Explorer*.
- ▶ Select a method, enter parameters, and click *Go*.

Generated code

You will find the code generated by the wizard in the following projects:

- ▶ WeatherEJB
 - Mapping classes: `itso.objects`
 - WSDL file: `META-INF/wsdl/WeatherEJB.wsdl`
 - Mapping file: `META-INF/WeatherEJB_mapping.xml`
 - Deployment descriptor: `META-INF/webservices.xml` and extensions
- ▶ WeatherEJBClientClasses (under Other Projects)
 - Service endpoint interface: `itso.ejb.WeatherEJB_SEI`
 - Mapping classes: `itso.objects`
- ▶ WeatherEJBRouterWeb
 - Router servlet with the name WeatherEJB mapped to the URL `services/WeatherEJB`
 - A copy of the WSDL file in `WebContent/WEB-INF/wsdl` and in `WebContent/wsdl`

Running clients against the EJB Web service

In the stand-alone client (`WeatherJavaClient` in `WeatherClientStandalone`), activate the commented line to change the endpoint and rerun the application:

```
proxy.setEndpoint
    ("http://localhost:9080/WeatherEJBRouterWeb/services/WeatherEJB");
```

In the JSF client, you can modify the `getWeatherjavabeanproxy` method in `WeatherReport.java` and change the endpoint by adding:

```
weatherjavabeanproxy.setEndpoint
    ("http://localhost:9080/WeatherEJBRouterWeb/services/WeatherEJB");
```

Creating a Web service top-down from WSDL

In the top-down approach, a skeleton implementation of a Web service is created from an existing WSDL file. Application Developer can generate a skeleton JavaBean or a skeleton session EJB. The process for both is very similar. Note that for an EJB skeleton, the EJB project must not be empty.

In this section, we use Application Developer to create a server-side JavaBean skeleton from an existing WSDL file.

For this scenario, we create a new enterprise application, `WeatherTopDownServer`, with a Web project, `WeatherTopDownServerWeb`, containing the service skeleton and implementation.

Creating the Web project and importing the WSDL

Start this sample by creating the `WeatherTopDownServerWeb` project in an enterprise application named `WeatherTopDownServer`.

Import the base WSDL file (a copy of the `Weather.javabean.wsdl` file) into a new folder `WebContent\wsdl` from:

```
\SG246461\sampcode\servers\topdown\wsdl\WeatherTopDown.wsdl
```

Creating the skeleton JavaBean

With the WSDL file imported into the workspace, we can run the Web Service wizard to create the required skeleton Java code:

- ▶ Select the `WeatherTopDown.wsdl` file and *Web Services → Generate Java bean skeleton*.
- ▶ Select *Generate a proxy*, and select *Test the Web service*.
- ▶ For the Service Deployment Configuration client, select *Web* as client type, `WeatherTopDownServerWebClient` as the client project, and `WeatherClientEAR` as the EAR project.
- ▶ On the Web Service Client Test page, clear *Run test on server* (there is nothing to test yet). Click *Finish*.

When the wizard finishes, the `WeatherJavaBeanSoapBindingImpl` skeleton class opens in the Java editor.

Generated code

The generated code in the `WeatherTopDownServerWeb` consists of:

- ▶ `itso.bean` package—`WeatherJavaBean` interface and `WeatherJavaBeanSoapBindingImpl` skeleton class

- ▶ itso.objects package—Mapping classes
- ▶ WEB-INF\wsdl—New WSDL file with updated address
- ▶ WEB-INF—Deployment descriptor (webservices.xml)

The generated code in the WeatherTopDownServerWebClient consists of the proxy classes, mapping classes, and the sampleWeatherJavaBeanProxy test client.

JavaBean skeleton

From the information in the WSDL file, the wizard generates the code shown in Figure 16-21. To implement the functionality for our Web service, we just have to replace the content of the methods in the generated skeleton with code that implements the weather forecast application.

```
package itso.bean;

public class WeatherJavaBeanSoapBindingImpl
    implements itso.bean.WeatherJavaBean {
    public itso.objects.Weather[] getForecast(java.util.Calendar startDate,
        int days) throws java.rmi.RemoteException {
        return null;
    }
    public itso.objects.Weather getDayForecast(java.util.Calendar theDate)
        throws java.rmi.RemoteException {
        return null;
    }
    public void setWeather(itso.objects.Weather dayWeather)
        throws java.rmi.RemoteException {
    }
    public int[] getTemperatures(java.util.Calendar startDate, int days)
        throws java.rmi.RemoteException {
        return null;
    }
}
```

Figure 16-21 Generated skeleton JavaBean Web service

Using the top-down generation of a JavaBean skeleton is the preferred method when generating new Web services. Using this method, you start with designing your service on an abstract level using the WSDL editor in Application Developer, and from the WSDL file, you can use the wizard to generate the required skeleton code to implement the service.

Implementing the JavaBean skeleton Web service

You can implement the Web service in many ways:

- ▶ Write a new implementation.
- ▶ Copy an existing implementation.
- ▶ Call an existing Web service and process the results into the required format.

You can browse the Internet or the UDDI Business Registry for weather forecast Web services. Some Web services are free, for demonstration purposes, others are available at a cost. Here are some Web sites to explore:

- ▶ Business Registry
<http://www.ibm.com/services/uddi>
- ▶ Search for weather
<http://www.xmethods.com>

For example, the *Global Weather* service (found on xmethods) returns an XML string that has the data (in **bold red**) required for our Web service:

<http://www.webservicex.com/globalweather.asmx>

```
<?xml version="1.0" encoding="utf-16"?>
<CurrentWeather>
  <Location>San Jose, San Jose International Airport...</Location>
  <Time>Dec 10, 2004 - 01:53 PM EST / 2004.12.10 1853 UTC</Time>
  <Wind> from the ENE (070 degrees) at 9 MPH (8 KT):0</Wind>
  <Visibility> 3 mile(s):0</Visibility>
  <SkyConditions> overcast</SkyConditions>
  <Temperature> 55.9 F (13.3 C)</Temperature>
  <DewPoint> 53.1 F (11.7 C)</DewPoint>
  <RelativeHumidity> 90%</RelativeHumidity>
  <Pressure> 30.26 in. Hg (1024 hPa)</Pressure>
  <Status>Success</Status>
</CurrentWeather>
```

By parsing the XML, we could generate the result for the `getDayForecast` method.

Creating a composed Web service

To illustrate the concept of a composed Web service (a Web service that calls another Web service), we call the session EJB Web service to implement the skeleton JavaBean. The `WeatherTopDownServerWeb` project (our service) becomes now a client (calling another service).

Creating the proxy classes

Select the WeatherEJB.wsdl file (in WeatherEJB\ejbModule\META-INF\wsdl) and *Web Services* → *Generate Client*. In the Web Service Client wizard:

- ▶ Clear *Test the Web service*.
- ▶ In the Client Environment Configuration page, select *Web* as the client type and WeatherTopDownServerWeb as the client project.
- ▶ Click *Finish*.

The WeatherEJBProxy class and its companions are generated into the itso.ejb package.

Implementing the skeleton

The methods of the skeleton class invoke the existing EJB Web service for processing. In a real-life implementation, the result objects of the called Web service might not match the required Web service results and processing would be required.

Example 16-3 shows a subset of our implementation. For the complete code, see \SG246461\sampcode\servers\topdown\implementation.

Example 16-3 Implemented skeleton JavaBean Web service

```
public class WeatherJavaBeanSoapBindingImpl implements ...WeatherJavaBean {
    public itso.objects.Weather[] getForecast(java.util.Calendar startDate,
        int days) throws java.rmi.RemoteException {
        System.out.println("Calling the EJB Web service");
        return new WeatherEJBProxy().getForecast(startDate, days);
    }
    public itso.objects.Weather getDayForecast(java.util.Calendar theDate)
        throws java.rmi.RemoteException {
        System.out.println("Calling the EJB Web service");
        itso.objects.Weather w = new WeatherEJBProxy().getDayForecast(theDate);
        w.setTemperatureCelsius( w.getTemperatureCelsius() - 3);
        w.setWindSpeed( w.getWindSpeed() + 5);
        System.out.println("Changed the weather!");
        return w;
    }
    public void setWeather(itso.objects.Weather dayWeather)
        throws java.rmi.RemoteException {
        System.out.println("Calling the EJB Web service");
        new WeatherEJBProxy().setWeather(dayWeather);
    }
}
```

Test the Web service by executing the TestClient.jsp that was generated into the WeatherTopDownServerWebClient project.

Web services and JMS binding

Using the Application Developer, it is possible to develop Web services that uses SOAP over JMS transport instead of SOAP over HTTP.

In this example, we use the WeatherJMS bean in the WeatherEJBMS project to create a SOAP over JMS Web service using the default messaging provider in WebSphere Application Server Version 6.

To execute the example, you must have JMS configured on the test server, as documented in “Setting up messaging for the JMS Web service” on page 713. You can configure the server before or after you run the wizard.

Tip: Configure the server before running the wizard and remember the JNDI names specified for the activation specification, the queue connection factory, and the queue.

Creating an EJB router project

To use the wizard for SOAP over JMS, we must create an EJB project that will function as the router project for the session bean. This EJB project will contain the message-driven bean (MDB) that listens to the JMS queue.

To create an EJB router project, perform these steps:

- ▶ Select *File* → *New* → *Project* → *EJB* → *EJB Project*.
- ▶ Enter WeatherEJBMSRouter as the name.
- ▶ Click *Show advanced* and select WeatherEJBMS as the EAR project.
- ▶ Clear *Create an EJB Client JAR Project to hold the client interfaces and classes*. Select *Create a default stateless session bean* (otherwise, deployment fails if the project is empty).
- ▶ Click *Finish*. Do not save the diagram with the default session bean.

Running the Web Service wizard

The process of creating a Web service using SOAP over JMS is very similar to the previous samples, so only JMS-specific details are highlighted in this example.

To run the Web Service wizard, perform these steps:

- ▶ In the Project Explorer, expand *EJB Projects* → *WeatherEJBMS* → *Deployment Descriptor* → *Session Beans*.

- ▶ Select the WeatherJMS session EJB and *Web Services* → *Create Web service* (context) to start the wizard.
- ▶ On the Web Services page, select *EJB Web Service* (preselected), *Start Web service in Web project*, *Generate a proxy of type Java proxy*, *Test the Web service*, *Overwrite files without warning*, and *Create folders when necessary*.
- ▶ On the Object Selection Page, the WeatherJMS bean is selected.
- ▶ On the Service Deployment Configuration page, WeatherEJBJMS is the service project. For the client-side:
 - Select *Web* as the type.
 - Enter WeatherEJBJMSSClientWeb as the client project.
 - Enter WeatherEJBJMSSClient as the client EAR project.

The EAR project and the client Web project will be created.

For both the server and the client, the runtime should be IBM WebSphere, WebSphere Application Server V6.0, and J2EE 1.4.

- ▶ On the Web Service EJB Configuration page, select the WeatherEJBJMSRouter project we created earlier as the router project and select *SOAP over JMS* as the transport.
- ▶ Selecting *SOAP over JMS* opens the JMS URI Properties section (Figure 16-22).

Select transports:
☐ SOAP over HTTP
☒ SOAP over JMS

JMS URI Properties

JMS destination: queue

Destination JNDI name: jms/weatherQ

JMS connection factory: jms/weatherQCF

Port component name: WeatherJMS

MDB deployment mechanism: JMS ActivationSpec

ActivationSpec JNDI name: eai/weatherAS

Listener input port name:

Initial context factory name:

JNDI provider URL:

Delivery Mode: 1

Request message lifetime:

JMS request message priority: 4

Connection factory userid:

Connection factory password:

These JNDI names must match what is specified in the server configuration.

Figure 16-22 SOAP over JMS properties

- Select *queue* for JMS destination, and enter **jms/weatherQ** as the JNDI name and **jms/weatherQCF** as the connection factory JNDI name,
- Select *JMS ActivationSpec* and enter **eai/weatherAS** as the JNDI name.
- Accept the port component name as WeatherJMS.
- Leave all the other default values.
- ▶ On the Web Service Java Bean Identify page, select all the methods and *Document/Literal*.
- ▶ Skip the Web Service Test page; it is not possible to use the Web Services Explorer to test a JMS Web service.
- ▶ On the Web Service Proxy page, select *Generate a proxy*. The proxy goes into the WeatherEJBJMSSClientWeb project.
- ▶ On the Web Service Client Test page, select *Web service sample JSPs* and *Run test on server*.
- ▶ On the Web Service Publication page, click *Finish*.

Note: Be patient. It takes a while to start the projects.

Web service address

Open the WeatherJMS.wsdl file or the generated WeatherJMSServiceLocator. The address of the JMS Web service is:

```
<wsdlsoap:address location=
  "jms:/queue?destination=jms/weatherQ&
  connectionFactory=jms/weatherQCF&targetService=WeatherJMSJMS" />
```

Testing the JMS Web service

When the wizard finishes, you can test the Web service using the generated test client:

- ▶ Run one of the methods and watch for a result.
- ▶ If the test fails, select the server in the Servers view and *Restart Project* → *WeatherEJBJMSServer* (context).
- ▶ Rerun the methods in the test client. Now, the Web service works.

Deleting the default session bean

You can delete the generated DefaultSession bean from the WeatherEJBJMSSRouter project (expand the deployment descriptor).

Creating a J2EE application client for SOAP over JMS

To illustrate a managed client running in a J2EE client container, we create a Java program that invokes the JMS Web service:

- ▶ Create a J2EE application client project by selecting *File* → *New* → *Project* → *J2EE* → *Application Client Project*. Enter `WeatherClientAppJMS` as the name. Click *Show Advanced* and select the `WeatherClientEAR` EAR project. Clear *Create a default Main class*. Click *Finish*. You get an error, because there is no main class.
- ▶ Create a package (`itso.client`) and a class named `WeatherJMSClient`.
- ▶ Select the `MANIFEST.MF` file and *Open with* → *JAR Dependency Editor*. For the main class, click *Browse*, locate the `WeatherJMSClient`, and click *OK*. Save and close, and the error is gone.

Running the Web Service wizard

Select the `WeatherJMS.wsdl` file in the `WeatherEJBjms` project (under `ejbModule/WETA-INF/wsdl`) and *Web Services* → *Generate Client* (context) to start the Web Service Client wizard:

- ▶ Clear *Test the Web service*.
- ▶ In the Client Environment Configuration page, select *Application Client* as the type and `WeatherClientAppJMS` as the client project.
- ▶ Click *Finish* to generate the proxy classes into the `WeatherClientAppJMS` project.

Completing the Java program

Open the `WeatherJMSClient` program. Complete the code with the example in:

```
\SG246461\sampcode\client\jms\WeatherClientAppJMS.java
```

Running the managed client

Select the `WeatherJMSClient` and *Run* → *Run*. In the Run configuration, select *WebSphere v6.0 Application Client*, and then click *New* (Figure 16-23):

- ▶ Enter `WeatherJMSClient` as the name.
- ▶ For the enterprise application, select `WeatherClientEAR`.
- ▶ For the application client module, select `WeatherClientAppJMS`.
- ▶ Select *Enable application client to connect to a server*.
- ▶ Select *Use specific server*.
- ▶ Click *Apply* and then click *Run*.

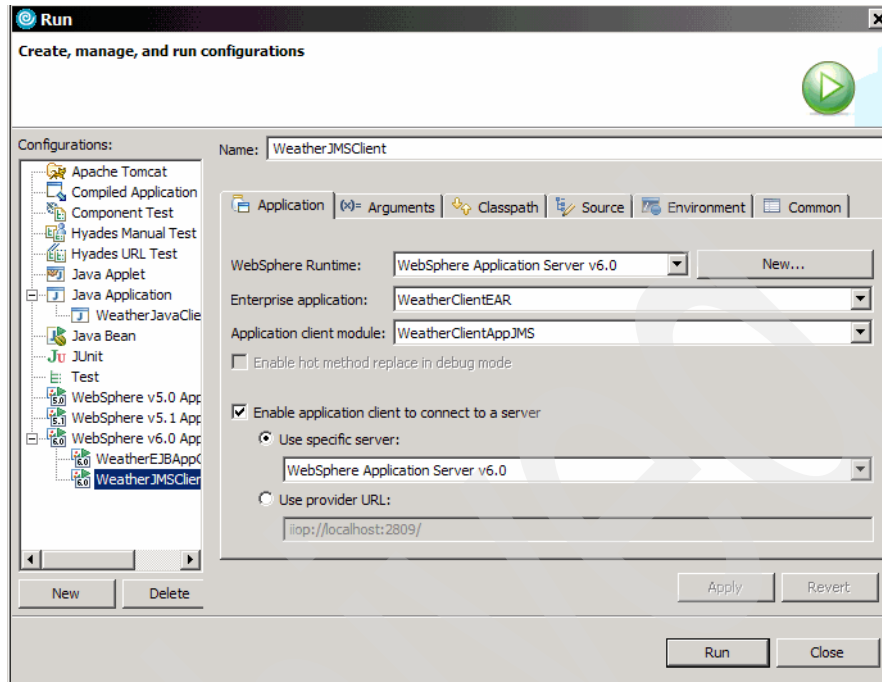


Figure 16-23 Application client run configuration

The console should show the application starting and displaying the results:

```
WEATHER FORECAST - JMS APPLICATION CLIENT
=====
Getting WeatherForecast proxy ...
Using endpoint: jms:/queue?destination=jms/weatherQ&connectionFactory=
                jms/weatherQCF&targetService=WeatherJMSJMS
Invoking getDayForecast ...
Today's weather: Weather: Sat. Dec 11, 2004 GMT, cloudy, wind: NE at 14km/h
, temperature: 46 Celsius
Raise temperature by 5 degrees ...
.....
End
```

Running the managed client outside of Application Developer

To run the application client, we have to export the enterprise application that contains the client:

- ▶ Select the WeatherClientEAR project and *Export* → *EAR file*.
- ▶ Enter a destination directory on your hard disk. Clear all options.
 \SG246461\sampcode\zSolution\JMSClient

- Click *Finish*.

The `launchClient` command of the Application Server is used to start a client. From the export directory, issue this command:

```
C:\<RAD60_HOME>\runtimes\base_v6\bin\launchClient.bat WeatherClientEAR.ear  
<RAD60_HOME> is where Application Developer is installed.
```

Running the managed client from a remote machine

To run the managed client from a remote machine where a WebSphere Application Server Version 6 client is installed, the endpoint address must be provided to the proxy class by the application using code such as:

```
proxy.setEndpoint( proxy.getEndpoint() +  
                    "&jndiProviderURL=iio://<hostname>:2809" );
```

Creating a Web service from a URL

The purpose of a URL Web service is to provide a mechanism to easily incorporate remotely running Web content into a client application. The content returned by the service is the content from the original location, typically HTML. The client can, if required, further process the markup returned from the content provider.

The Application Developer wizard generates a Web service and, optionally, a client proxy from an existing servlet, which can be located anywhere on the Web. During that process, the implementer can specify the syntax of the servlet parameters.

Note: The generated Web service from a URL requires the IBM SOAP runtime engine, which is no longer supported in WebSphere Application Server Version 6.

If you have configured a Tomcat or WebSphere Application Server Version 5.x server, you can create and test a URL Web service. We provide a sample servlet in:

```
\SG246461\sampcode\servers\url
```

Because this is old technology, we do not provide detailed information. A Version 5 example is documented in *WebSphere Version 5.1 Application Developer 5.1.1 Web Services Handbook*, SG24-6891.

Using Web service handlers

Handlers can be used to process or even modify SOAP messages either before or after the message is sent through the network. A handler is a component associated with a Web service or a specific Web service port and can be either general or specific in nature, depending on the purpose of the handler. Examples of handlers are encryption, decryption, logging, auditing, caching, and authorization.

All handlers must implement the `javax.xml.rpc.handler.Handler` interface or extend the existing `GenericHandler`. When a handler is called, it is passed an instance of a `MessageContext`, which can be used to access the SOAP envelope using the SOAP with Attachments API for Java (SAAJ). The `MessageContext` can also be used to pass objects between handlers so that they can share information specific to a request or response.

The life cycle of a handler is similar to what we know from the life cycle of a servlet:

- ▶ The runtime calls the handler's `init` method with configuration in the `HandlerInfo` object to initialize the handler.
- ▶ Depending on the processing, the `handleRequest`, `handleResponse`, or `handleFault` method is called.
- ▶ At the end, the runtime calls the handler's `destroy` method.

In this section, we create a general logging server-side handler for the weather forecast JavaBean service (WeatherJavaBeanWeb project) using the handler tool in Application Developer. For the client side (WeatherJavaBeanClientWeb project), we create a simple timing handler, giving us the total time for calling the service.

Creating a server-side Web service handler

On the server side, we create a simple logging handler that will print the SOAP request and the SOAP response to an output stream.

Make sure that the JavaBean Web service is installed and running in the server. You can import the solution enterprise applications, WeatherJavaBeanServer and WeatherJavaBeanClient.

To create a server-side handler, perform these steps:

- ▶ In the Project Explorer, expand *Web Services* → *Services*. Select the WeatherJavaBeanService (of the WeatherJavaBeanWeb project) and *Configure Handlers* (context).

- ▶ Make sure the Service Description in the drop-down menu is WeatherJavaBeanService. In case several Web services are deployed, they will be shown in the drop-down menu.
- ▶ Add a handler by clicking *Add* and enter the following values:
 - Class name: itso.handler.LoggingHandler
 - Name: LoggingHandler
 - Port name: WeatherJavaBean
- ▶ If multiple handlers are defined for the service, use the *Move Up* and *Move Down* buttons to change the order of processing.
- ▶ Select *Generate skeleton classes for new handlers*.
- ▶ Click *Finish* (Figure 16-24).

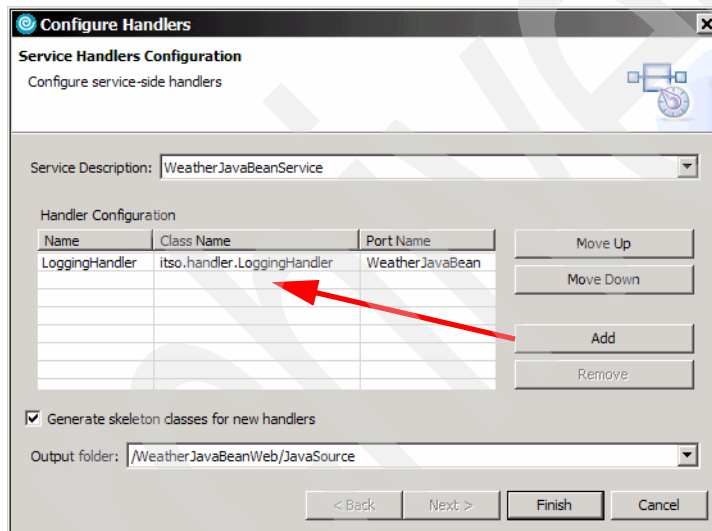


Figure 16-24 Handler configuration

When the Java skeleton has been generated by the wizard, the handler implementation is opened automatically in the editor. To add the logging functionality to the handler, add the `logMessage` method to the class (Figure 16-25).

```

private void logMessage(MessageContext context, String msg,
                        java.io.OutputStream output) {
    javax.xml.soap.SOAPMessage message =
        ((javax.xml.rpc.handler.soap.SOAPMessageContext)context).getMessage();
    try {
        output.write(msg.getBytes());
        message.writeTo(output);
        output.write("\r\n".getBytes());
    } catch (Exception e) {
        throw new javax.xml.rpc.JAXRPCException(e);
    }
}

```

Figure 16-25 Handler logging method

Add code to the `handleRequest` and `handleResponse` methods that call the `logMessage` method. Delete the `handleFault` method (Figure 16-26).

```

public boolean handleRequest(MessageContext context) {
    logMessage(context, "REQUEST : ", System.out);
    return true;
}

public boolean handleResponse(MessageContext context) {
    logMessage(context, "RESPONSE: ", System.out);
    return true;
}

// public boolean handleFault(MessageContext context)

```

Figure 16-26 Implementing `handleRequest` and `handleResponse`

The `LoggingHandler` is added to the handlers under *Web Services* → *Services* → *WeatherJavaBeanService* (you might have to double-click *Handlers*).

Testing the Web service handler

Restart the service project in the server by selecting the server and *Restart Project* → *WeatherJavaBeanServer*.

Run the generated test client in *WeatherJavaBeanClientWeb* and invoke one of the methods. If the handler is working correctly, the SOAP request and response are printed to the console view (Figure 16-27).

```

[12/13/04 13:03:27:935 PST] 00000063 SystemOut      0 REQUEST :
<soapenv:Envelope ...><soapenv:Header/><soapenv:Body><p821:getDayForecast
xmlns:p821="http://bean.itso"><theDate>2004-12-12T08:00:00.000Z</theDate></p
821:getDayForecast></soapenv:Body></soapenv:Envelope>

[12/13/04 13:03:27:985 PST] 00000063 SystemOut      0 RESPONSE:
<soapenv:Envelope ..."><soapenv:Header/><soapenv:Body> <p821:getDayForecastR
esponse ...><getDayForecastReturn><condition>rainy</condition><date>2004-12-
12T08:00:00.000Z</date><windDirection>N</windDirection><windSpeed>25</windSp
eed><temperatureCelsius>0</temperatureCelsius><dbflag>1</dbflag></getDayFore
castReturn></p821:getDayForecastResponse></soapenv:Body></soapenv:Envelope>

```

Figure 16-27 Logging handler output

The handler configuration can be modified using the Web Services Editor by opening the LoggingHandler (under *Web Services* → *Services*), or by opening the Web service deployment descriptor (*WebContent/WEB-INF/webservices.xml*). The LoggingHandler is visible on the Handlers page in the editor.

Note: A realistic logging handler should use a logging framework for its output. The output file could be configured as an initialization parameter that is processed in the `init` method.

Creating a client-side Web service handler

For the client, we create a simple handler that will print the time it takes to call the Web service. The duration of the call is calculated as the time difference between the calls to methods `handleRequest` and `handleResponse`.

To create a client-side handler, perform these steps:

- ▶ In the Project Explorer, expand *Web Services* → *Clients* → *WeatherJavaBeanClientWeb*. Select *WeatherJavaBeanClientWeb* and *Configure Client Handlers* (context).
- ▶ Verify the Client Service Reference as `service/WeatherJavaBeanService`.
- ▶ Add a handler by clicking *Add* and enter `itso.handler.TimingHandler` as the class name and *TimingHandler* as the name.
- ▶ Select *Generate skeleton classes for new handlers*.

Implement the `handleRequest` and `handleResponse` methods to measure the time it takes to call the service (Figure 16-28). Add `import java.util.Date` to resolve the errors.


```

public boolean handleRequest(MessageContext context) {
    // save the time when the request is sent
    Date startTime = new Date();
    context.setProperty("beginTime", startTime);
    return true;
}

public boolean handleResponse(MessageContext context) {
    Date endTime = new Date();
    Date startTime = (Date) context.getProperty("beginTime");
    long duration = endTime.getTime() - startTime.getTime();
    System.out.println("JavaBean Web service elapsed time: " + duration);
    return true;
}

```

Figure 16-28 Handler code to measure the response time

Test the handler by running one of the methods in the client. The elapsed time is displayed in the console view.

The handler configuration can be modified using the Web Deployment Descriptor Editor by opening the TimingHandler (under *Web Services* → *Clients*), or by opening the Web deployment descriptor (WebContent/WEB-INF/web.xml). The TimingHandler is visible on the WS Handlers page in the editor. Note that you have to select a service reference from the pull-down menu.

Using handlers in a stand-alone client

In the previous sample, the client handler was configured using the deployment descriptor of the Web application.

For a stand-alone client, such as the client in “Creating Web service clients” on page 288, a deployment descriptor does not exist. Instead, the handler must be configured and registered programmatically using the JAX-RPC API (Figure 16-29). The complete Java stand-alone client can be found in:

\SG246461\sampcode\clients\standalone\WeatherJavaClientWithHandler.java

- ▶ Import the WeatherJavaClientWithHandler into the WeatherClientStandAlone project (itso.client package).
- ▶ Create an itso.handler package and copy the TimingHandler into that package.
- ▶ Select the WeatherJavaClientWithHandler and *Run* → *Java Application*.

```

import javax.xml.namespace.QName;
import javax.xml.rpc.handler.HandlerInfo;
import javax.xml.rpc.handler.HandlerRegistry;

.....

System.out.println("Registering the TimingHandler");
Map config = new HashMap();
QName[] header = null;
HandlerInfo handlerinfo = new HandlerInfo(TimingHandler.class,
                                           config, header);
WeatherJavaBeanService service = new WeatherJavaBeanServiceLocator();
HandlerRegistry registry = service.getHandlerRegistry();
List handlerChain = registry.getHandlerChain(
    new QName("http://bean.itso", "WeatherJavaBean") );
handlerChain.add(handlerinfo);
.....
WeatherJavaBean proxy = service.getWeatherJavaBean();

```

Figure 16-29 Programmatic registration of client handlers

Note: You cannot use the WeatherJavaBeanProxy class because it creates a new instance of the WeatherJavaBeanServiceLocator.

When running the stand-alone Java client, the output looks similar to this:

```

WEATHER FORECAST - STANDALONE CLIENT WITH HANLDER
=====
Registering the TimingHandler
Getting WeatherForecast proxy ...
Using endpoint:
    http://localhost:9080/WeatherBeanWeb/services/WeatherJavaBean
Invoking getDayForecast ...
JavaBean Web service elapsed time: 1172
Todays weather: Weather: Mon. Dec 13, 2004 GMT, rainy, wind: SW at 13km/h ,
                temperature: 60 Celsius
Raise temperature by 5 degrees ...
Invoking setWeather ...
JavaBean Web service elapsed time: 90
Invoking getDayForecast ...
JavaBean Web service elapsed time: 30
Warmer weather: Weather: Mon. Dec 13, 2004 GMT, rainy, wind: SW at 13km/h ,
                temperature: 65 Celsius

End

```

Handlers and handler chains

If multiple handlers are defined for a service, they represent an ordered list of handlers, also known as a handler chain. Chained handlers are invoked in the order in which they are configured (remember the *Move Up* and *Move Down* buttons in the wizard), and when one handler has finished its processing, it passes the result to the next handler in the chain.

The steps for executing a chain can be described as:

- ▶ The `handleRequest` methods of all handlers in the chain on the client are executed in the order configured.
- ▶ The `handleRequest` methods of all handlers in the chain on the server are executed in the order configured.
- ▶ When all `handleRequest` methods have completed, the message is delivered to the endpoint service.
- ▶ When the service has completed its work, the runtime invokes the `handleResponse` method of the handlers in the server chain in reverse order.
- ▶ When the server processing finishes, the `handleResponse` methods are executed in the client in reverse order.

For all handlers, if processing completes without error, the `handle` method returns `true` to pass the message to the next handler in the chain. Returning `false` means that the processing in the chain has been terminated.

Closing comments about using handlers

Using handlers is a good way of preprocessing and postprocessing SOAP messages, but some issues must be kept in mind when using handlers:

- ▶ Response time might increase because a new layer of processing is added.
- ▶ There are restrictions for what kind of changes to the SOAP message are allowed. A simple rule is that handlers should deal mainly with headers and should not change the operation (method to be invoked) and message structure in the body of the message.

Using attachments

Today, Web services are getting more and more complex and are evolving from simple requests with simple parameters to full scale services that also handle complex objects.

The transport protocol used today, SOAP, is a text-based protocol using XML. Text-based protocols are preferable for several reasons and have already proved their success in the context of Web services. However, reliance on textual encoding has a dark side when it comes to non-textual data, and XML itself does not offer any efficient solution for including binary data. The W3C XML Schema specifies that binary data should be base 64 encoded, but this would lead up to a 50% increase in data size compared to non-encoded data.

So why use binary data at all? The answer is that there are several good reasons for using binary data, for example, multimedia applications. The use of binary data has been around for a long time and will most likely remain popular.

Several methods for supporting binary data in Web services have been proposed. We discuss the most well-known of them:

- ▶ **Base 64 encoding**—Base 64 encoding is a well-known standard that has been around for quite some time, used especially when sending binary data within e-mail messages. By splitting the data into 6-bit chunks, the data can be encoded into a character stream that fits perfectly into XML for including non-structured data. However, because the data size increases by 33%, this method should only be used for small parts of data. Despite its size and efficiency problems, using base 64 encoding has some benefits, such as working with higher-level Web services specifications and interoperability among most platforms.
- ▶ **SOAP with Attachments (SwA)**—This method sends binary data with SOAP in a MIME multipart message. SOAP with Attachments is currently only a W3C note, but at the time of writing, the WS-I Organization has created a Profile based on SwA known as Attachments Profile 1.0, which complements the WS-I Basic Profile 1.1 to add support for interoperable SOAP messages with attachments. Currently, W3C has no intention to further support SwA, but is working on MTOM, which also aims to fix some of the problems with SwA, such as not working well with the WS-I specifications. Another problem with SwA is interoperability, because Microsoft no longer supports SOAP with Attachments, which makes interoperability with the .NET platform difficult regarding binary data.
- ▶ **WS-Attachments with Direct Internet Message Encapsulation (DIME)**—WS-Attachments with DIME is a simple and efficient solution for dealing with attachments. DIME itself is a simple mechanism for packaging a collection of data with the SOAP message. The protocol is very similar to MIME for Web

services, but is a faster and more efficient protocol compared to MIME. Because there are no recommendation for DIME or WS-Attachments, this solution shares some of the same problems as SOAP with Attachments, such as working with higher-level Web services specifications.

- **Message Transmission Optimization Mechanism (MTOM)**—Message Transmission Optimization Mechanism is, at the time of writing, a proposed recommendation at W3C.

How do I use attachments today?

Choosing the best method for dealing with binary data depends on the requirements. Base 64 encoded data within the XML message is a very useful method for small pieces of binary data, such as security tokens in form of hashes, keys, and certificates. Because the encoded data is part of the SOAP message, you can also apply higher-level specifications, such as WS-Security, without any problems.

However, if speed and efficiency are required, attachments are the only practical option, and you should consider using SOAP with Attachments or even the DIME protocol, but you must also keep in mind that for both mechanisms, you will lose the ability to apply higher-level WS-I specifications. With the Attachments Profile and SwA implementation on all your platforms, you should consider using SwA, because this will make interoperability easier as long as no Microsoft platform is accessing the service and security is not an issue. The DIME mechanism should only be used when integrating with existing DIME and WS-Attachment implementation, because it does not provide a great degree of interoperability and is mainly supported by Microsoft, which will switch to MTOM in the near future.

Implementing attachments

Rational Application Developer provides tools to assist with creating Web services, and in relation to attachments, it supports the following specifications: SOAP with Attachments, JAX-RPC Version 1.1, WS-I Basic Profile 1.1, and WS-I Attachments Profile V1.0.

In this section, we extend the weather forecast application with a function that returns an image as a result of a query. The returned image reflects the weather condition for the current day. If an image is passed as parameter to the Web service, the weather forecast is written on the image, and the modified image is returned to the client.

At the time of writing, the most popular type of attachment in Java is available through the JAX-RPC and SOAP with Attachments API for Java (SAAJ).

Because JAX-RPC is a more high-level abstraction compared to SAAJ, most aspects regarding the used protocol are hidden when using JAX-RPC. When using JAX-RPC, attachments are represented by Java classes.

In the following section, we provide a step-by-step example of how to build an interoperable Web service using attachments based on JAX-RPC 1.1, WS-I Basic Profile 1.1, and Attachments Profile 1.0—standards supported by WebSphere Application Server Version 6 and Rational Application Developer Version 6.

With JAX-RPC, it is easy to use attachments in a Web service, because you only have to define the WSDL document using MIME binding in the input or output and then use Application Developer to generate the required Java artifacts.

Our example

To show the use of attachments, we add a `getDayForecastImage` method to the forecast application that takes a date and an image as input and returns an image as output.

If the input image is `null`, a predefined image showing the weather condition is returned. If an input image is supplied, the weather result is written into the image and the updated image is returned.

Creating the WSDL document

Creating the WSDL part of the attachment is not as straightforward as the tool. At the time of writing, it does not support attachments when starting from Java.

Figure 16-30 shows excerpt from the `WeatherAttachment.wsdl` file, highlighting some important lines. The complete file is available in:

```
\SG246461\samrcode\servers\attachment\wsdl
```

```

<wsdl:types>
  <schema xmlns="http://www.w3.org/2001/XMLSchema" ...
    ...
    <import namespace="http://ws-i.org/profiles/basic/1.1/xsd"
      schemaLocation="swaref.xsd"/>
    ...
</wsdl:types>

<wsdl:message name="getDayForecastImageRequest">
  <wsdl:part name="theDate" type="xsd:dateTime"/>
  <wsdl:part name="bgImage" type="xsd:hexBinary"/>
</wsdl:message>

<wsdl:message name="getDayForecastImageResponse">
  <wsdl:part name="weatherImage" type="ws-i:swaRef"/>
</wsdl:message>

<wsdl:binding name="WeatherAttachmentSoapBinding"
  type="impl:WeatherAttachment">
  <wsdlsoap:binding style="document" ..... />
  ....
  <wsdl:operation name="getDayForecastImage">
    <wsdlsoap:operation soapAction=""/>
    <wsdl:input name="getDayForecastImageRequest">
      <mime:part>
        <wsdlsoap:body parts="theDate" use="literal"/>
      </mime:part>
      <mime:multipartRelated>
        <mime:part>
          <mime:content part="bgImage" type="image/jpeg"/>
        </mime:part>
      </mime:multipartRelated>
    </wsdl:input>
    <wsdl:output name="getDayForecastImageResponse">
      <mime:multipartRelated>
        <mime:part>
          <wsdlsoap:body parts="weatherImage" use="literal"/>
        </mime:part>
      </mime:multipartRelated>
    </wsdl:output>
  </wsdl:operation>
</wsdl:binding>
...

```

Figure 16-30 Excerpt from the WSDL attachment document

According to WSDL 1.1, we can specify in the binding part of the WSDL whether a message part goes into the SOAP body or as a MIME attachment. For the `getDayForecastImage` method, the input `theDate` goes into the SOAP body, as normal, and the `bgImage` with content type `image/jpeg` is a MIME part. The output `weatherImage` is of type `swaRef`, which is a reference to the attachment in the SOAP message. The `swaRef` type is introduced by WS-I Attachments Profile 1.0 and is a way to describe the use of MIME types in WSDL documents.

The `swaRef` type is an interoperable way to mark references to attachments in the descriptions; the actual reference is not known until runtime.

Importing the enterprise application

We provide a skeleton enterprise application, `WeatherAttachmentServer`, that contains the `WeatherAttachmentWeb` project in:

```
\SG246461\sampcode\servers\attachment\EAR
```

Import this EAR file. Select *Allow nested projects overwrites*. Select the `WeatherBase.jar` file as the utility project. Change the new project name from `WeatherBase1` to `WeatherBase`.

The `WeatherAttachmentWeb` project contains the `WeatherAttachment.wsdl` file and the `swaref.xsd` file (in `wsdlstart`), a number of GIF files (in `images`), and the data source reference `WeatherDataSourceReference` in the deployment descriptor.

Generating the service endpoint interface

From the WSDL file, we can create the service endpoint interface (SEI):

- ▶ Select the `WeatherAttachment.wsdl` file and *Web Services* → *Generate Java bean skeleton*.
- ▶ Clear *Start Web service in Web project*.
- ▶ Verify that the target projects are `WeatherAttachmentWeb` and `WeatherAttachmentServer`.

The skeleton implementation class `WeatherAttachmentSoapBindingImpl` opens in the editor. Example 16-4 shows the `WeatherAttachment` service endpoint interface that is generated.

Example 16-4 Attachment SEI

```
package itso.bean;

public interface WeatherAttachment extends java.rmi.Remote {
    public itso.objects.Weather getDayForecast(java.util.Calendar theDate)
        throws java.rmi.RemoteException;
}
```



```

    public javax.activation.DataHandler getDayForecastImage(
        java.util.Calendar theDate, java.awt.Image bgImage)
        throws java.rmi.RemoteException;
}

```

From the interface, we can see, that all MIME types in the WSDL document are mapped to Java types. Table 16-1 shows all mappings according to JAX-RPC and the JavaBeans Activation Framework. If a used MIME type is not in the table, the JAX-RPC implementation will map it to `javax.activation.DataHandler`.

Table 16-1 JAX-RPC mappings of MIME types

MIME type	Java type
image/gif, image/jpeg	<code>java.awt.Image</code>
text/plain	<code>java.lang.String</code>
multipart/*	<code>javax.mail.internet.MimeMultipart</code>
text/xml, application/xml	<code>java.xml.transport.Source</code>

Notice that the return type is `javax.activation.DataHandler`, which must be used to get hold of the attachment in the SOAP response.

Implementing the server

Update the `WeatherAttachmentSoapBindingImpl` class with the code provided in:

```

\SG246461\samcode\servers\attachment\implementation

```

Generating the client proxy

Select the `WEB-INF/wsdl/WeatherAttachment.wsdl` file and *Web Services* → *Generate Client*. Clear *Test the Web service*. For the client, specify:

- ▶ Client type: Java
- ▶ Client project: `WeatherAttachmentClient`
(This project will be created.)

The client project shows up under Other Projects in the Project Explorer. It contains the proxy and helper classes. The `WeatherAttachmentProxy` class contains the `getDayForecastImage` method:

```

    public javax.activation.DataHandler getDayForecastImage
        (java.util.Calendar theDate, java.awt.Image bgImage)
        throws java.rmi.RemoteException{

```

Implementing clients

We provide a stand-alone GUI client, `WeatherAttachmentGUIClient`. This client invokes the Web service using either SAAJ or JAX-RPC, passing the current date and an image to the server. The GUI displays the resulting image, together with the weather forecast as text (Figure 16-31).

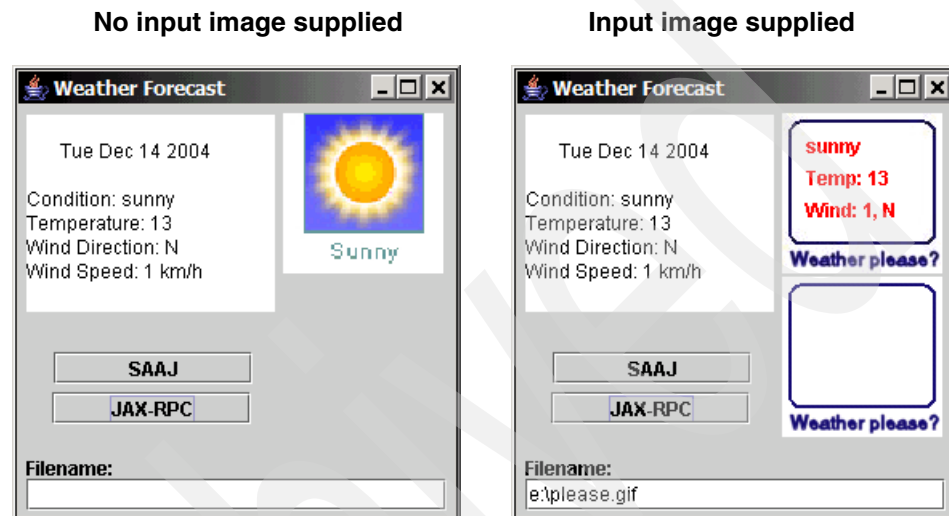


Figure 16-31 GUI client working with attachments

SAAJ client

As mentioned earlier, SAAJ is closer to the protocol, which makes it perfect for understanding the mechanism behind SOAP with Attachments.

Sending a SOAP message with an attachment involves these steps:

- ▶ Create a SOAP connection and SOAP message objects.
- ▶ Retrieve the message body from the message object.
- ▶ Create an XML operation element in the body.
- ▶ Add the date input parameter under the operation.
- ▶ Create a data handler using the input image.
- ▶ Create the attachment.
- ▶ Associate the attachment with the message.
- ▶ Call the service.

The weather service replies with the image created as an attachment. The client retrieves the image from the response body using a data handler.

Example 16-5 shows the client code for the SAAJ implementation.

Example 16-5 Using the SAAJ API for attachments

```
SOAPConnection connection =
    SOAPConnectionFactory.newInstance().createConnection();
SOAPMessage message = MessageFactory.newInstance().createMessage();
SOAPPart part = message.getSOAPPart();
SOAPEnvelope envelope = part.getEnvelope();
SOAPBody body = envelope.getBody();
SOAPBodyElement operation = body.addBodyElement
    (envelope.createName("getDayForecastImage", "tns", "http://bean.itso"));
operation.setEncodingStyle("http://schemas.xmlsoap.org/soap/encoding/");
SOAPElement theDate = operation.addChildElement("theDate", "");
SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ss");
theDate.addTextNode(sdf.format(today.getTime()));

if (bgImage != null) {
    DataHandler dh = new DataHandler( new FileDataSource(fileName) );
    AttachmentPart attachment = message.createAttachmentPart(dh);
    attachment.setContentId("bgImage=" + attachment.getContentId());
    message.addAttachmentPart(attachment);
}

// call the weather forecast using SAAJ
SOAPMessage result = connection.call(message, ENDPOINT_URL);

SOAPPart resultPart = result.getSOAPPart();
SOAPEnvelope resultEnvelope = resultPart.getEnvelope();
SOAPBody resultBody = resultEnvelope.getBody();
if (!resultBody.hasFault()) {
    Iterator iterator = result.getAttachments();
    if (iterator.hasNext()) {
        DataHandler dh = ((AttachmentPart) iterator.next()).getDataHandler();
        weatherImage = javax.imageio.ImageIO.read(dh.getInputStream());
    }
}
```

JAX-RPC client

In Example 16-6, you can see how much shorter the client becomes when using the proxy classes generated by Application Developer from the WSDL file.

Using the proxy, you simply pass an image object to the call, and it is automatically converted into an attachment.

Example 16-6 Using JAX-RPC for attachments

```
WeatherAttachmentProxy proxy = new WeatherAttachmentProxy();
proxy.setEndpoint(ENDPOINT_URL);
```

```
// get the image for todays forecast
DataHandler result = proxy.getDayForecastImage(today, bgImage);

// fetch the returned image
weatherImage = javax.imageio.ImageIO.read(result.getInputStream());
```

Implementing and running the GUI client

The GUI client, WeatherAttachmentGUIClient, can be imported into an itso.client package from:

```
\SG246461\sampcode\clients\attachment\gui
```

Before running the client, copy the please.gif into a suitable directory:

```
From: \SG246461\sampcode\clients\attachment\images\please.gif
To:   C:\please.gif
```

Select the WeatherAttachmentGUIClient and *Run* → *Java Application*.

Displaying SOAP messages (TCP/IP Monitor)

In Example 16-7, you can see the SOAP request with an attachment. The request comes in two parts: The first part is a normal SOAP request, and the second part is the attachment.

Example 16-7 SOAP input message with attachment

```
-----=_Part_0_147100310.1103048296527
Content-Type: text/xml; charset=UTF-8
Content-Transfer-Encoding: binary
Content-Id: <781710019222.1103048296517.IBM.WEBSERVICES@<host>>

<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Header/>
  <soapenv:Body>
    <getDayForecastImage>
      <theDate>2004-12-14T18:18:14.605Z</theDate>
    </getDayForecastImage>
  </soapenv:Body>
</soapenv:Envelope>
-----=_Part_0_147100310.1103048296527
Content-Type: image/jpeg
Content-Transfer-Encoding: binary
Content-Id: <bgImage=771783468694.1103048296517.IBM.WEBSERVICES@<host>>
```

Example 16-8 shows the response to this request. Because the Web service returns a data handler, the SOAP body part contains a reference to the attachment, which the client can use to retrieve the data.

Example 16-8 SOAP response message with attachment

```
-----=_Part_279300_1497286637.1103048298170
Content-Type: text/xml; charset=UTF-8
Content-Transfer-Encoding: binary
Content-Id: <2161077430253.1103048298170.IBM.WEBSERVICES@<host>>

<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Header/>
  <soapenv:Body>
    <getDayForecastImageResponse>
      <weatherImage>
        cid:weatherImage=2151079314413.1103048298170.IBM.WEBSERVICES@<host>0
      </weatherImage>
    </getDayForecastImageResponse>
  </soapenv:Body>
</soapenv:Envelope>

-----=_Part_279300_1497286637.1103048298170
Content-Type: application/octet-stream
Content-Transfer-Encoding: binary
Content-Id: <weatherImage=2151079314413.1103048298170.IBM.WEBSERVICES@<host>>
```

Generating WSDL from Java

In Application Developer V6.0, you cannot generate a good WSDL file from the Java skeleton or implementation of the attachment example.

In the Web Service wizard, you have the option *Use WSDL 1.1 Mime attachments exclusively*. However, the generated WSDL files show errors in both cases:

- ▶ With the option selected, the input image maps to type `tns3:Image`, and the output image to `tns3:DataHandler`, both of which cannot be resolved.

Note: This works in Application Developer V6.0.1. The WSDL file is valid, and a generated client proxy has the same signature as our example.

- ▶ With the option cleared, both images map to `swaRef`, which cannot be resolved.

It is possible to import the `swaref.xsd` file into the project and add:

```
<import namespace="http://ws-i.org/profiles/basic/1.1/xsd"
      schemaLocation="swaref.xsd"/>
```

Then, change the generated types from `type="tns3:swaRef"` to `type="wsi:swaRef"`, and the errors disappear.

When generating proxy classes from the corrected WSDL file, the signature becomes:

```
public DataHandler getDayForecastImage(Calendar theDate,
                                         DataHandler bgImage)
    throws java.rmi.RemoteException
```

This might work by rewriting the client code.

Web Services Atomic Transaction

In this section, we experiment with WS-AtomicTransaction in a very simple example.

WebSphere Application Server Version 6 implements the Web Services Atomic Transaction (WS-AT) specification. This specification enables Web service applications to participate in global transactions distributed across a heterogeneous Web service environment.

Here is an extract of the *WebSphere Application Server Information Center*. Search for *Atomic Transaction* and *WS-AtomicTransaction* to find the description of the WebSphere support:

- ▶ The Web Services Atomic Transaction for WebSphere Application Server provides transactional quality of service to the Web services environment. This enables distributed Web service applications, and the resources they use, to take part in distributed global transactions.
- ▶ The WS-AT support is an interoperability protocol that introduces no new programming interfaces for transactional support. Global transaction demarcation is provided by standard J2EE use of the JTA UserTransaction interface. If a Web service request is made by an application component running under a global transaction, a WS-AT CoordinationContext is implicitly propagated to the target Web service, if the appropriate application deployment descriptors have been specified.
- ▶ If WebSphere Application Server is the system hosting the target endpoint for a Web service request that contains a WS-AT CoordinationContext, WebSphere automatically establishes a subordinate JTA transaction in the target runtime environment that becomes the transactional context under which the target Web service application executes.

Figure 16-32 shows a transaction context shared between two WebSphere application servers for a Web service request that uses WS-AT.

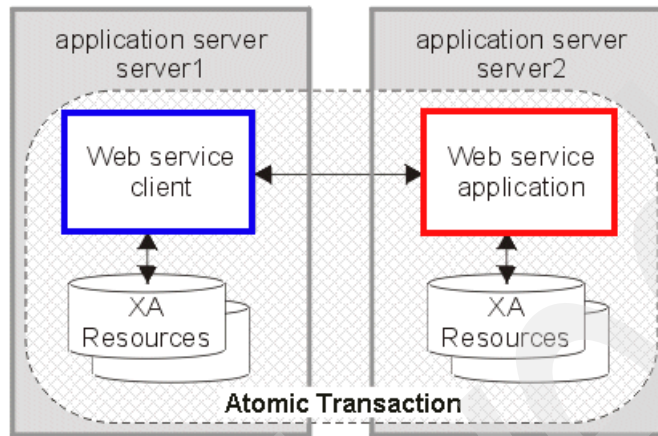


Figure 16-32 WS-AtomicTransaction support in Application Server Version 6

WS-AT is a two-phase commit transaction protocol and is suitable for short duration transactions only. WS-AT is well suited for distributed transactions within a single enterprise.

Because the purpose of an atomic transaction is to coordinate resource managers that isolate transactional updates by holding transactional locks on resources, it is generally not recommended that WS-AT transactions be distributed across enterprise domains. Inter-enterprise transactions typically require a looser semantic than two-phase commit.

Application developers do not have to explicitly register WS-AT participants. The WebSphere Application Server runtime takes responsibility for the registration of WS-AT participants in the same way as the registration of XA resources in the JTA transaction to which the WS-AT transaction is federated. At transaction completion time, all XA resources and WS-AT participants are atomically coordinated by the WebSphere Application Server transaction service.

If a JTA transaction is active on the thread when a Web service request is made, the transaction is propagated across on the Web service request and established in the target environment. This is analogous to the distribution of transaction context over IIOP, as described in the EJB specification. Any transactional work performed in the target environment becomes part of the same global transaction.

There are no specific development tasks required for Web service applications to take advantage of WS-AT; however, there are some application deployment descriptors that have to be set appropriately.

Deployment descriptors for atomic transactions

To enable WS-AT, deployment descriptors must be updated for Web and EJB modules. Here is an extract of such changes:

- ▶ In a Web module that invokes a Web service, specify *Send Web Services Atomic Transaction on requests* to propagate the transaction to the target Web service.
- ▶ In a Web module that implements a Web service, specify *Execute using Web Services Atomic Transaction on incoming requests* to run under a received client transaction context.
- ▶ In an EJB module that invokes a Web service, specify *Use Web Services Atomic Transaction* to propagate the EJB transaction to the target Web service.
- ▶ In an EJB module, bean methods must be specified with transaction type *Required*, which is the default, to participate in a global atomic transaction.

Implementing a simple atomic transaction

For this example, we use the top-down Web service implemented in “Creating a Web service top-down from WSDL” on page 299. Figure 16-33 shows the projects and beans involved in this application:

- ▶ The `TestClient.jsp` invokes the JavaBean Web service in the Web project and passes a `Weather` object for insertion into the database.
- ▶ The JavaBean Web service invokes the EJB Web service to perform the insert operation.

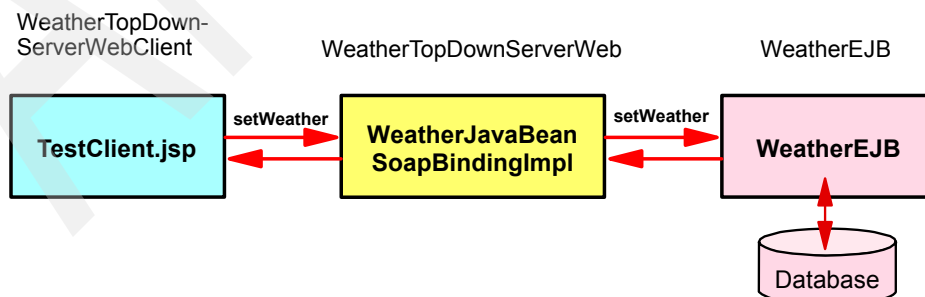


Figure 16-33 Top-down Web service example

To experiment with atomic transactions, we expand the JavaBean Web service to also update the database so that we have two applications, each updating the database by inserting a record (Figure 16-34).

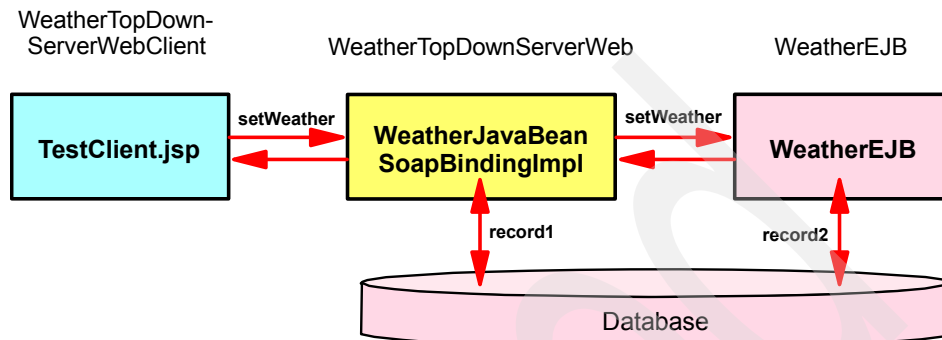


Figure 16-34 Top-down Web service example for atomic transaction

Changes in the EJB Web service

The `setWeather` method in the `WeatherEJB` is modified to optionally throw an exception to simulate an error:

```

public void setWeather(Weather dayWeather) throws Exception {
    WeatherForecast wfc = new WeatherForecast();
    wfc.setWeather(dayWeather);
    //throw new java.rmi.RemoteException("WeatherEJB insert failed");
}
  
```

Changes in the JavaBean Web service

We expand the JavaBean Web service (`WeatherJavaBeanSoapBindingImpl`) in `WeatherTopDownServerWeb` to start a global transaction, insert a weather object into the database, change the date to the next day, call the EJB Web service (to insert another weather object), and commit the changes:

- To display the weather in a nice format, open the `Weather` class in the `itso.objects` package and add the `toString` method copied from the `Weather` class in the `WeatherBase` project.
- Add an `itso.dao` package and copy the `WeatherDAO` from the `WeatherBase` project. We use this DAO to insert the weather record.
- Edit the `setWeather` method of the `WeatherJavaBeanSoapBindingImpl` bean:

```

public void setWeather(itso.objects.Weather dayWeather)
    throws java.rmi.RemoteException {
    UserTransaction userTransaction = null;
    try {
        // start transaction
  
```

```

InitialContext context = new InitialContext();
userTransaction = (UserTransaction)
                    context.lookup("java:comp/UserTransaction");
userTransaction.begin();

// insert record1 into database
System.out.println("Date: "+dayWeather.getDate().getTime());
WeatherDAO dao = new WeatherDAO();
dao.insertWeather(dayWeather);
System.out.println("Initial database weather: "+dayWeather);

System.out.println("Changing the weather before calling EJB");
dayWeather.getDate().add(Calendar.DAY_OF_MONTH, 1);
dayWeather.setWindSpeed( dayWeather.getWindSpeed() + 2);
System.out.println("New database weather: "+dayWeather);

// original method code - next two lines
System.out.println("Calling the EJB Web service");
new WeatherEJBProxy().setWeather(dayWeather);

// commit or rollback
userTransaction.commit();
//throw new java.rmi.RemoteException("Simulated abort");

} catch (java.rmi.RemoteException re) {
    try {
        userTransaction.rollback();
    } catch (Exception e) {}
    throw new java.rmi.RemoteException("Top-down service failed: "
                                        + re.getMessage());
} catch (Exception e) {
    throw new java.rmi.RemoteException("Top-down transaction error: "
                                        + e.getMessage());
}
}
}

```

This code is available in \SG246461\sampcode\ws-at.

Note: The WeatherDAO uses connections with automatic commit. By using a global transaction around the JDBC access, autocommit is disabled, and the commit is performed when the global transaction ends. The global transaction will be carried from the JavaBean Web service to the EJB Web service when we activate the atomic transaction support.

- ▶ Add the JDBC resource reference for the WEATHER database by editing the deployment descriptor of WeatherTopDownServerWeb. On the References page, add a resource reference named WeatherDataSourceReference (javax.sql.DataSource) with application authentication and with a JNDI name of jdbc/weather. You can duplicate the reference by looking at the WeatherJavaBeanWeb project.

Testing the modified application

At this point, we can test the application by restarting the enterprise applications (WeatherEJBServer and WeatherTopDownServer) in the server and running the TestClient.jsp of the WeatherTopDownServerWebClient project (Figure 16-35):

- ▶ Make sure to use a date that does not exist in the database so that you can see if records are inserted.
- ▶ You can clean the database with commands such as:
db2 "delete from itso.sanjose where weatherdate >= '01/18/2005'"
- ▶ After the run, the ITS0.SANJOSE table displays the records. There should be two new records.

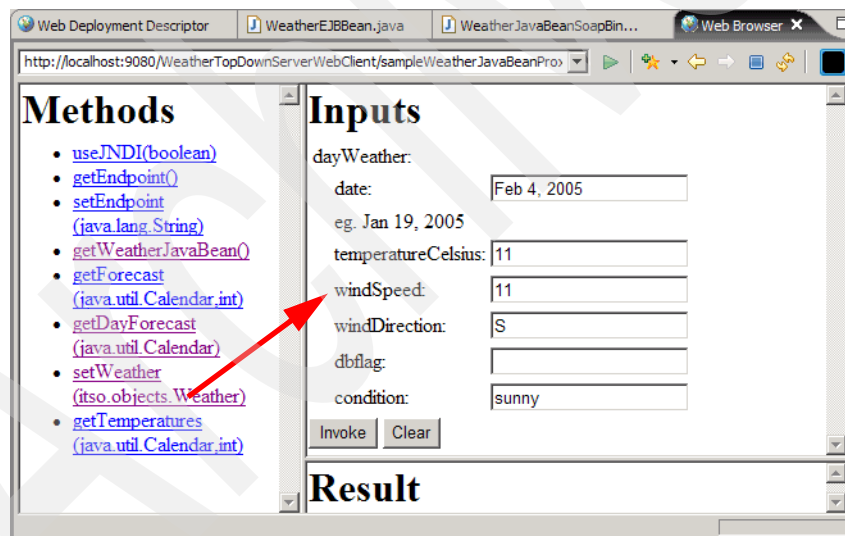


Figure 16-35 Executing the test client for atomic transactions

Activating atomic transaction support

To activate the atomic transaction support, we have to update the deployment descriptor of the Web application:

- ▶ Open the deployment descriptor of WeatherTopDownServerWeb and select the `its0_bean_WeatherJavaBeanSoapBindingImpl` servlet on the Servlets page.
- ▶ Scroll down and select *Send Web Services Atomic Transactions on requests* (Figure 16-36).

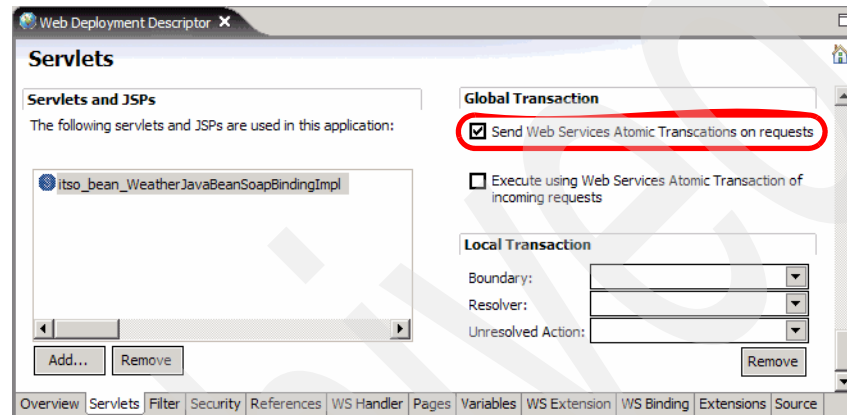


Figure 16-36 Activating atomic transaction in the deployment descriptor

Testing atomic transactions

To test atomic transactions, perform these steps:

- ▶ Always use a date where no data exists in the database, or delete existing records first.
- ▶ To see the SOAP messages, start a TCP/IP Monitor (see “TCP/IP Monitor” on page 368) and change the address in the WSDL files to port 9081:
 - `WeatherEJB.wsdl` in `WeatherTopDownServerWeb/WebContent/WEB-INF/wsdl` (or directly in the `WeatherEJBServiceLocator` class)
 - `WeatherTopDown.wsdl` in `WeatherTopDownServerWebClient/.....` (or directly in the `WeatherJavaBeanServiceLocator` class)
- ▶ Run the `TestClient` with the changed application. Two records are inserted.

Simulating an error in the EJB Web service

To simulate an error in the EJB Web service, perform these steps:

- ▶ Edit the `setWeather` method of the `WeatherEJB` and activate the statement:

```
throw new java.rmi.RemoteException("WeatherEJB insert failed");
```
- ▶ Run the client, and no records are inserted in the database. The Web service returns the exception:

```
exception: java.rmi.RemoteException: Top-down service failed:
javax.transaction.TransactionRolledbackException: ; nested exception is:
java.rmi.RemoteException: WeatherEJB insert failed
```

- ▶ Reset the `setWeather` method of the `WeatherEJB` and deactivate the statement:

```
//throw new java.rmi.RemoteException("WeatherEJB insert failed");
```

Simulating an error in the JavaBean Web service

To simulate an error in the JavaBean Web service, perform these steps:

- ▶ Edit the `setWeather` method of the `WeatherJavaBeanSoapBindindImpl` bean and perform a rollback by changing the statements:

```
// userTransaction.commit()           <=== deactivate
throw new java.rmi.RemoteException("Simulated abort");
```

- ▶ Run the client, and no records are inserted in the database. The Web service returns the exception:

```
exception: java.rmi.RemoteException: Top-down service failed:
Simulated abort
```

Be sure to reset the client code and the port in the service locator classes.

SOAP message for atomic transaction

The SOAP message passed from the JavaBean to the EJB Web service carries the WS-AT information in the header:

```
<soapenv:Envelope .....>
<soapenv:Header>
  <wscoor:CoordinationContextType soapenv:mustUnderstand="1">
    <wscoor:Expires>Never</wscoor:Expires>
    <wscoor:Identifier>com.ibm.ws.wstx:00000101912...</wscoor:Identifier>
    <wscoor:CoordinationType>http://schemas.xmlsoap.org/ws/2004/10/wsat
      </wscoor:CoordinationType>
    <wscoor:RegistrationService
      xmlns:wscoor="http://schemas.xmlsoap.org/ws/2004/10/wscoor">
      <wsa:Address
        xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing">
```

```

        http://9.65.1.154:9080/_IBMSYSAPP/wscoor/services/
        RegistrationCoordinatorPort
    </wsa:Address>
    <wsa:ReferenceProperties
        xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing">
        <websphere-wsat:txID xmlns:websphere-wsat=
            "http://wstx.Transaction.ws.ibm.com/extension">
            com.ibm.ws.wstx:00000101912...</websphere-wsat:txID>
        <websphere-wsat:instanceID xmlns:websphere-wsat=
            http://wstx.Transaction.ws.ibm.com/extension">
            com.ibm.ws.wstx:00000101912...</websphere-wsat:instanceID>
        </wsa:ReferenceProperties>
    </wscoor:RegistrationService>
    </wscoor:CoordinationContextType>
</soapenv:Header>
<soapenv:Body>
    .....
</soapenv:Envelope>

```

Note that this messages has information from the WS_Coordination, WS-Addressing, and WS-AtomicTransaction specifications.

Summary

In this chapter, we described how to use Rational Application Developer Version 6 to develop Web services and Web service clients for JavaBean and EJB Web services, using both SOAP over HTTP and SOAP over JMS.

In addition, we showed how to develop JAX-RPC handlers, how to work with attachments, and how to use the WebSphere atomic transaction support.

Archived

Test and monitor Web services

In this chapter, we talk about the techniques and functions provided by IBM WebSphere Application Server and Rational Application Developer to test and monitor Web services.

First, we describe the Rational Application Developer test facilities, and then we show how to monitor Web services with WebSphere Application Server functions.

Throughout this chapter, we use the weather forecast application as generated from the WeatherJavaBean. The weather forecast application must be running, and the database must have been created as described in “Setting up the WEATHER database” on page 708. All samples in this chapter use the test data created with the JavaBean weather forecast application.

Testing Web services

Web services testing has the same approach as testing any other distributed application. The simplified goal of testing is to verify that the application, running on a server, is acting as expected; that is, does the response on the client side match the expected return value?

Because each distributed application has several layers, there are a number of places to define and run tests. Depending on the actual Web service implementation (JavaBean, EJB), the well-known application development test approaches should be used. Applying test techniques is to verify the functions of the Web service implementation and all its components.

Testing modes

There are two kinds of test modes: *automated* and *manual*. As a best practice, the approach should be to implement all tests as automated whenever possible. Automated tests can help the development process to be more efficient.

Automated tests have following advantages:

- ▶ Consistent approach—Exact defined and documented tests; reproducible tests with reliable tests results.
- ▶ Repeatable process—Easy to rerun tests scripts. This prevents stress and workload for developers as caused by manual tests and increases development efficiency.
- ▶ Sharable—Test definitions and scripts have to be created only once and can be shared within development teams. This helps to create a consistent test environment, and all team members use the same tests to verify the application functions.

With Application Developer, automated and manual tests can be accomplished. We talk about the test features in the following sections:

- ▶ Web Services Explorer—Manual test
- ▶ Web services test client JSPs—Manual test
- ▶ Universal Test Client—Manual test
- ▶ Web services component test—Automated test

Testing approaches

In addition to manual and automated tests, there are different test approaches:

- ▶ **Class-level test**—Used to test the functions of a single class and to test interactions between class methods and other classes.
- ▶ **Method-level test**—This test focuses on testing only specific class methods. Method-level tests can be defined for methods that do not rely on a class or application context and can be called in isolation with other methods.
- ▶ **Component test**—A component is a particular function or group of related functions. To test a component means to test all coherent modules that make up the component as a group to verify all parts work together.
- ▶ **Regression test**—This test has the purpose to ensure that changes made to the application (changed functions, fixes) do not adversely affect the correct functionality inherited from the previous version.
- ▶ **Subsystem-level test**—Subsystem-level testing focuses on verifying the interfaces between the component-under-test (CUT) and other subsystems. These tests exercise the interactions between the different objects of the subsystem. When defining subsystem-level tests in a multilayer environment, tested subsystems are called tiers.
- ▶ **Performance test**—Performance tests still do not have the appropriate focus in development projects. But the success of projects, and good applications, is highly coupled to well-performing and stable applications. The performance testing gets more complicated, the more layers an application is built on (system availability and performance). Performance tests should be used throughout all application development stages. It should start in the development stage (class/components), and we highly recommend testing before an application can be called production ready.

The common best practice for test is: *Begin early with testing and test often.*

Tip: All known Java/J2EE test approaches should be applied for Web services development, respectively for the actual Web services implementation and all sub-layers (JavaBeans, EJBs).

Web Services Explorer

In this section, we discuss the Web Services Explorer test functions. We show some best practices about how to use the Web Services Explorer to test Web services. Also, the Web Services Explorer provides more functions than performing Web services tests; refer to the Rational Application Developer online documentation for more information.

Important: The Web Services Explorer can only be used to test HTTP bound Web services. It is not possible to test SOAP/JMS Web services. To test SOAP/JMS services, use the Web services sample test JSPs (see “Web services sample test JSPs” on page 345).

In the sections that follow, we explain the Web Services Explorer test functions and how to use them.

Starting the Web Services Explorer

To test a Web service available in Application Developer, we recommend that you start the Web Services Explorer from the service entry in the Web services project structure, or from the service WSDL file:

- ▶ Start the Web Services Explorer from the services project structure.

In the Project Explorer, expand *Web Services* → *Services*, and then select the desired Web service (for example, *WeatherJavaBeanService*) and *Test with Web Services Explorer* (Figure 17-1).

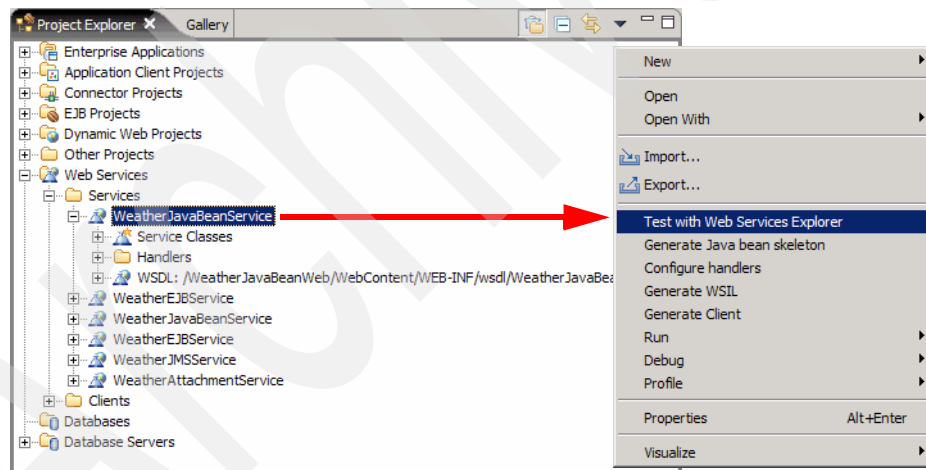


Figure 17-1 Start the Web Services Explorer from a Web service entry

- ▶ Start the Web Services Explorer from a service WSDL file.

Select a WSDL file (under Dynamic Web Projects) *WeatherJavaBeanWeb/*WebContent/*WEB-INF/wsdl/WeatherJavaBean.wsd1* and *Web Services* → *Test with Web Services Explorer*.

Working with the Web Services Explorer

The first time you start the Web Services Explorer it takes a while, but then the WSDL page of the explorer opens. The methods and the endpoint of the Web service are listed for selection (Figure 17-2).

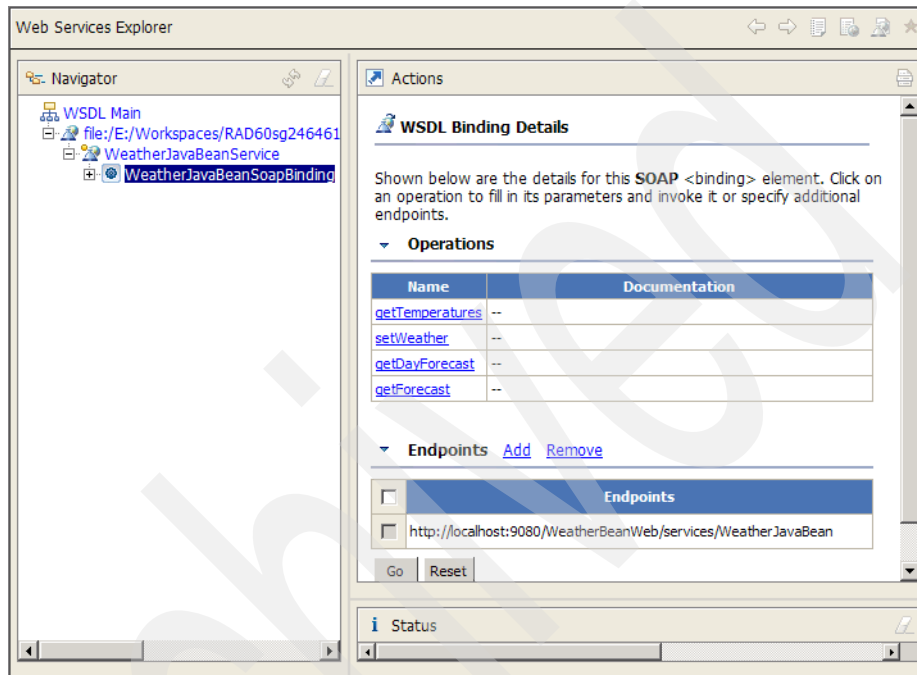


Figure 17-2 Web Services Explorer: WSDL page

Changing the endpoint

If the Web service to test is not running at the endpoint defined in the WSDL document, the endpoint address can be changed with the Web Services Explorer:

- ▶ Click *Add* for endpoints. An entry is added to the list.
- ▶ Overtyp the new entry with the specific details (Figure 17-3).
- ▶ The changed endpoint address can be used when testing Web service operations with the Web Services Explorer. To activate an endpoint, select the check box and click *Go*.
- ▶ Remove the endpoint to test with the default endpoint.

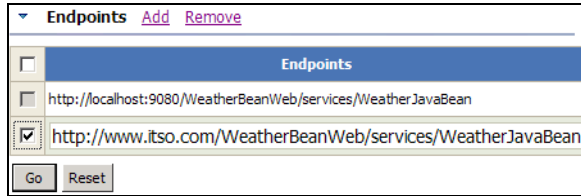


Figure 17-3 Add another endpoint address

Running a method

To test the Web service select one of the methods, either in the Navigator pane (left) or in the Actions pane (right). You are prompted for the parameters.

For example:

- ▶ Select the `getTemperatures` method. You are prompted for the `startDate` and the days.
- ▶ The date/calendar format is quite complex. However, click *Browse* and select a date from the calendar that is displayed.
- ▶ Click *Go* to invoke the Web service (Figure 17-4).

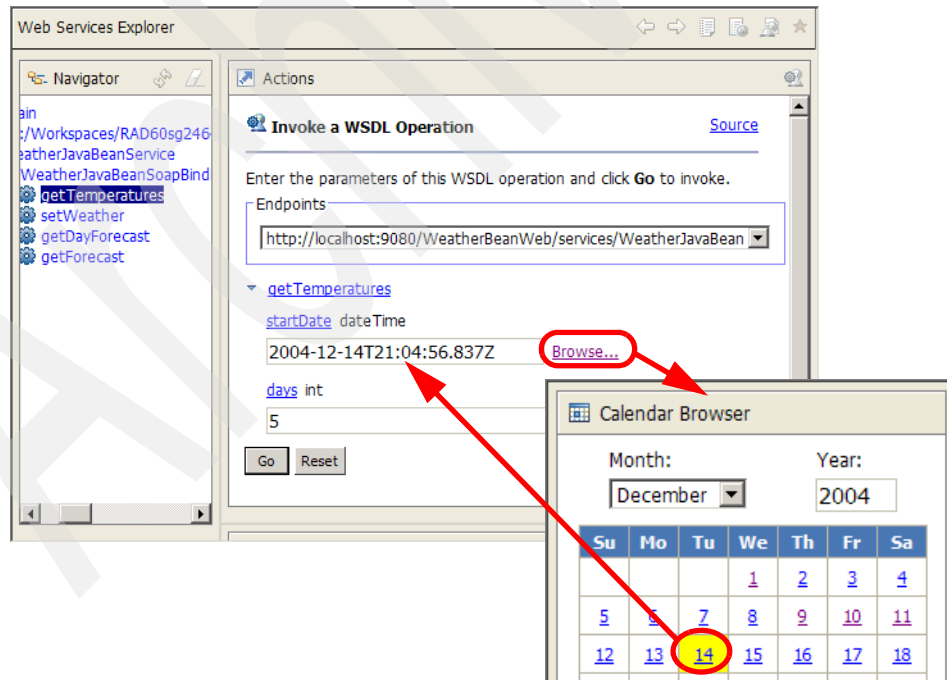


Figure 17-4 Run a Web service operation with parameters

- The results are displayed in the Status pane (Figure 17-5).

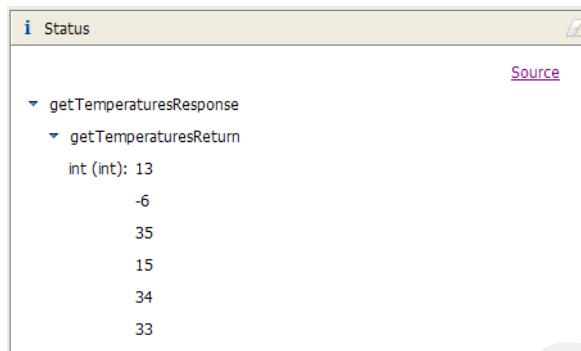


Figure 17-5 Web service result: Formatted

- In the Actions pane, you can also submit a call in SOAP message format. Click *Source* (top right) and edit the SOAP message before sending it (Figure 17-6).

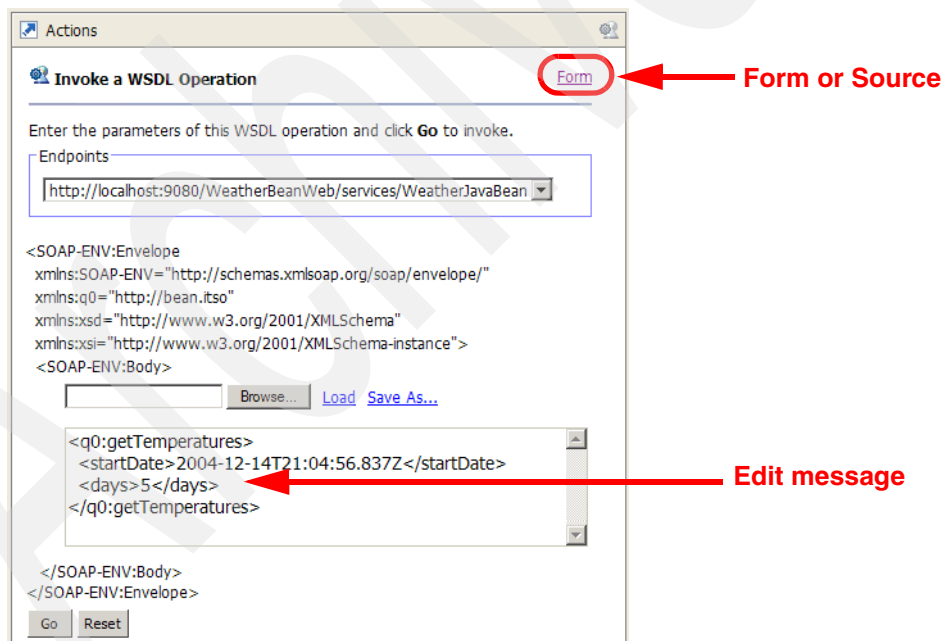


Figure 17-6 Display and edit the SOAP message in source format

- The message body can be saved to a file and loaded from a file. This function is very helpful when testing Web service operations with many parameters.

Viewing the operation results in SOAP format

The result of a Web service call are displayed in the Explorer's Status pane. You can switch to the Source view as well and display the SOAP input and output messages (Figure 17-7).

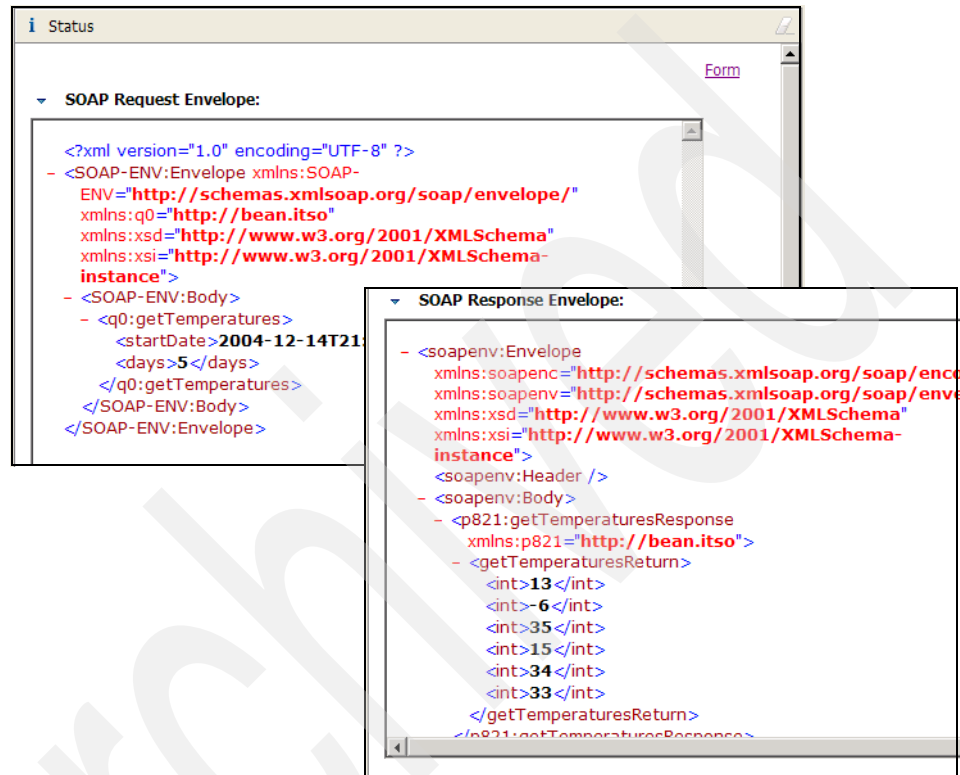


Figure 17-7 Web service results: SOAP messages

Tip: You can double-click the title bar (Navigator, Actions, Status) to maximize that pane. Double-click again to return to the normal view.

Clearing results

Before calling another Web service operation with the Web Services Explorer, we recommend that you clear the status view by clicking the Eraser icon  in the Status bar.

Web services sample test JSPs

The Web Service wizard of Application Developer can create test JavaServer Pages (JSP) for a Web service. This function is part of generating client-side code, such as proxy classes, into a client Web project:

- ▶ The test client can be generated by the Web Service wizard when generating server and client code (see “Web Service wizard” on page 275, and especially, Figure 16-8 on page 282).
- ▶ The test client can be generated by the Web Service Client wizard when generating a client from a WSDL file.

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

Generating client-side code

The Web Service Client wizard generates proxy classes and, optionally, test client JSPs. For example, we can regenerate the client code for the JavaBean Web service:

- ▶ In the Project Explorer, expand *Web Services* → *Services*, and select the WeatherJavaBeanService and *Generate Client* (context).
- ▶ In the Web Services page, select *Java proxy* and *Test the Web service* (Figure 17-8).

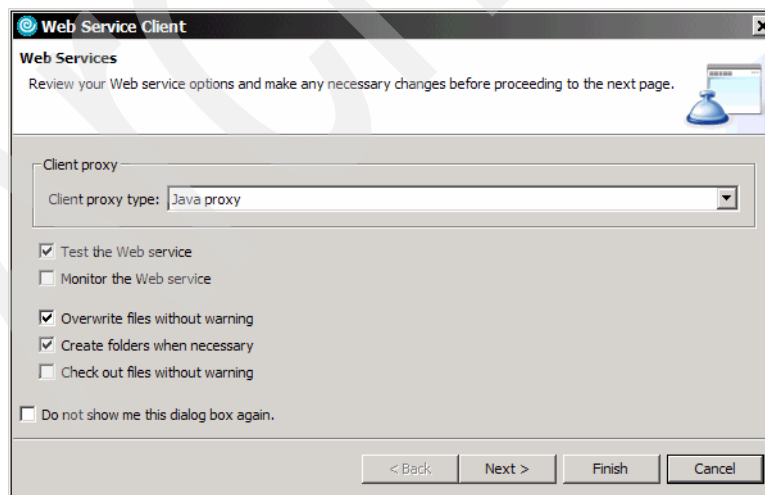


Figure 17-8 Set the client proxy type

- ▶ In the Web Service Selection page, the WSDL is preselected.
- ▶ In the Client Environment Configuration page, select *Web* as the client type, and make sure that the projects are set to `WeatherJavaBeanClientWeb` and `WeatherJavaBeanClient`.

Note that you can enter names of new projects and they will be created and added to the selected server.

To change the Web service runtime or server, click *Edit* (Figure 17-9).

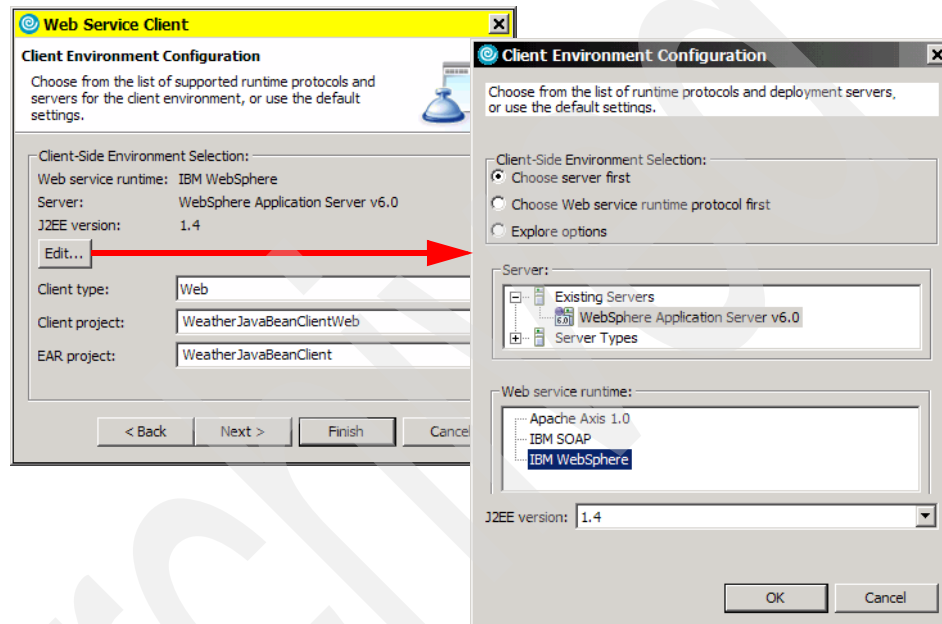


Figure 17-9 Define the client environment configuration

- ▶ On the Web Service Proxy page, you can specify security options (see Chapter 21, “Securing Web services” on page 445) and custom mappings between namespaces and packages.

If you select *Define custom mapping for namespace to package*, the next dialog box step provides the mapping configuration.

- ▶ On the Web Service Client Test page, select *Test the generated proxy*, *Web service sample JSPs*, and *Run test on server*.

You can also change the name of the generated folder and you can select which methods should be generated into the test client. In addition to the Web service methods, utility methods to manipulate the proxy can be generated (Figure 17-10). Click *Finish*.

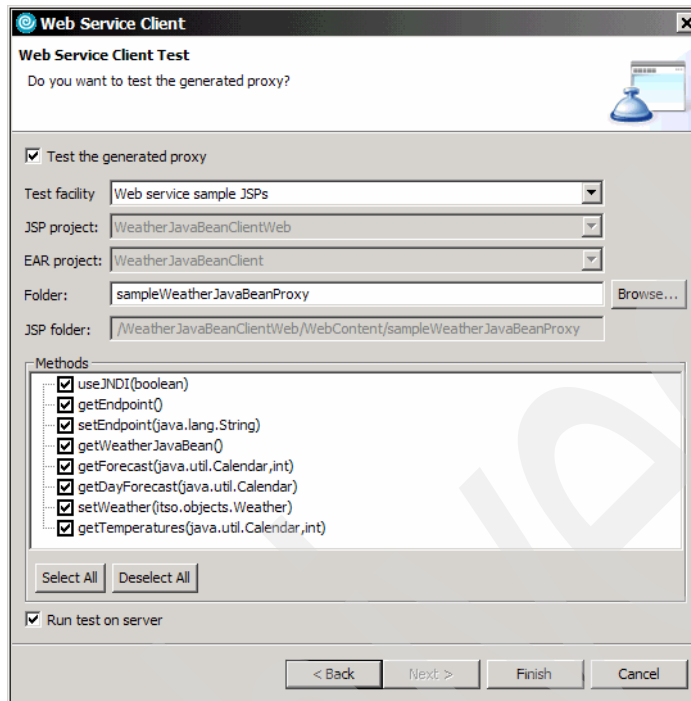


Figure 17-10 Define the operations for the JSP code

Testing with test client JSPs

The test client JSPs start when the client wizard finishes. To restart the test client application, select the generated `TestClient.jsp` file in the client Web project, and select *Run on Server* (see Figure 17-11).

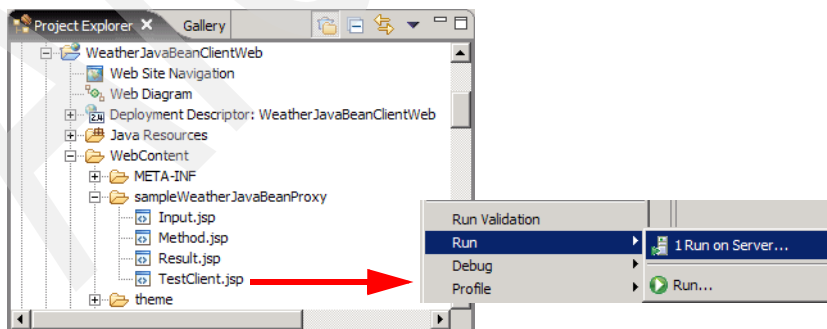


Figure 17-11 Start the generated sample Web application

The `TestClient.jsp` is a combination of three frames, a **Methods** frame to select a method, an **Inputs** frame to enter parameter values, and a **Result** frame to display the result of the execution (Figure 17-12).

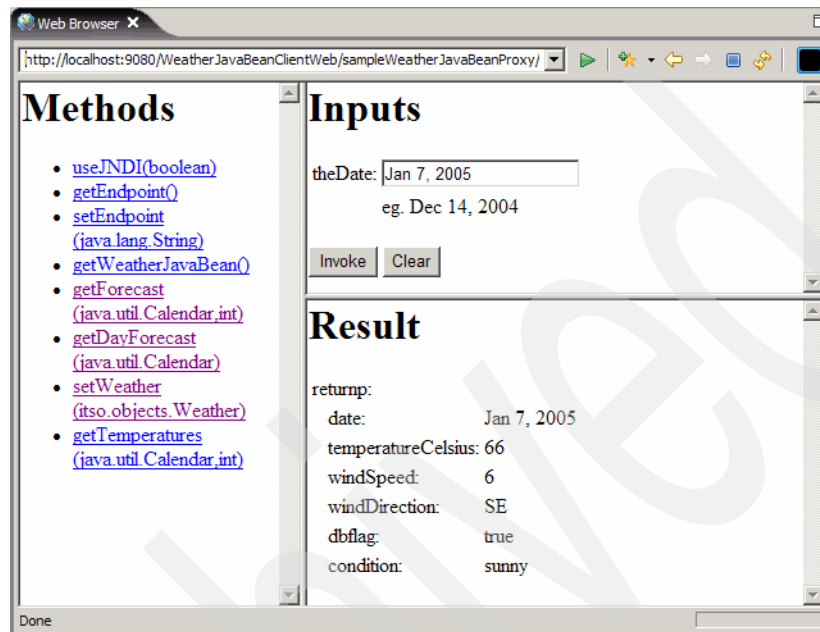


Figure 17-12 Generated Web service test JSPs

Utility methods

In addition to calling Web service operations, you can invoke these utility methods:

- ▶ `getEndpoint`—Display the current endpoint.
- ▶ `setEndpoint`—Change the endpoint to another server or another port (for example, for monitoring).
- ▶ `useJNDI`—Set to false to use the JSR 101 convention.

Formatting of results

Note that functions that return arrays of JavaBeans might not format the results. For example, execute the `getForecast` method and the result displays as objects:

```
itso.objects.Weather@61a607dc
```

Our original `Weather` class in the `WeatherBase` project has a `toString` method that formats a `Weather` object. You can copy the `toString` method from the original class into the generated `Weather` class in the `WeatherJavaBeanClientWeb` project.

Rerun the test client using the modified Weather class (Figure 17-13).

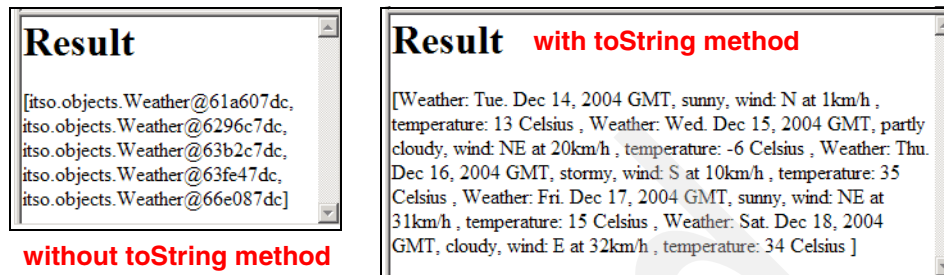


Figure 17-13 Formatting arrays of JavaBeans

Universal Test Client

The main function of the Universal Test Client (UTC) is to test EJBs without having to write client code. UTC has been improved over time and enables you to test almost any program.

Using UTC, we can load Java classes, create instances, and run methods. For Web services, we can load the generated proxy class and execute Web service functions.

Starting the Universal Test Client

You can start UTC in two ways:

- ▶ Select a class or EJB and *Launch Universal Test Client* (context).
- ▶ Select the server and *Run Universal Test Client* (context).

The first option is preferable because it loads the selected class and creates an instance of the class. The second option does not start with any instances and instances must be created by executing UTC functions.

Testing a Web service with the Universal Test Client

To test our example with UTC, select the WeatherJavaBeanProxy class in the WeatherJavaBeanClientWeb project and *Launch Universal Test Client* (context):

- ▶ UTC opens (be patient the first time), and the proxy object is displayed under Objects.
- ▶ Expand the proxy object to see its methods (Figure 17-14). Notice the icon after the name; clicking it removes the object.

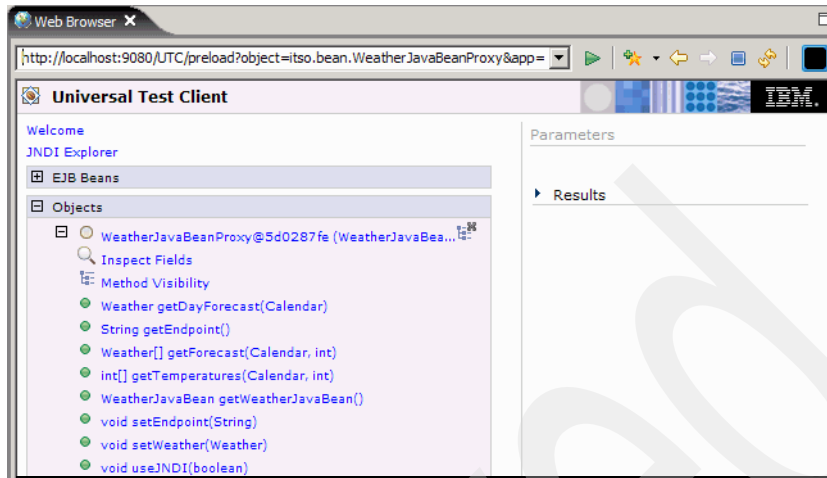


Figure 17-14 Universal Test Client: Proxy object

- Select a method, for example, `getDayForecast`, and you are prompted for parameters. The `Calendar` objects are automatically set to today. Click *Invoke*, and the result opens (Figure 17-15).

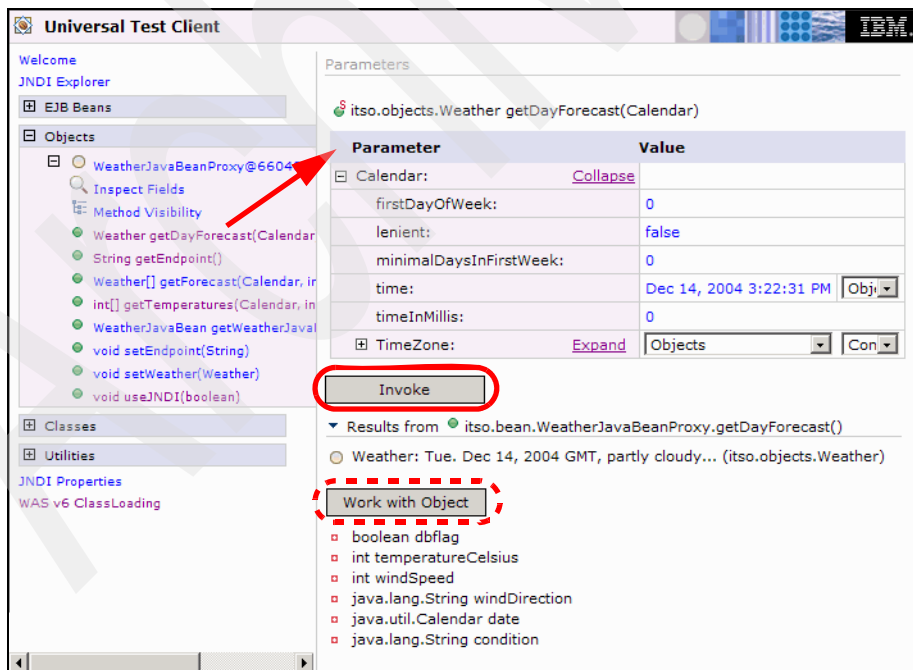


Figure 17-15 Universal Test Client: Invoking a method

- ▶ To work with the result object, click *Work with Object*. The result Weather object appears under Objects, and you can run any of its methods.
- ▶ If you run a method that returns an array (getForecast), all result objects are displayed (Figure 17-16).

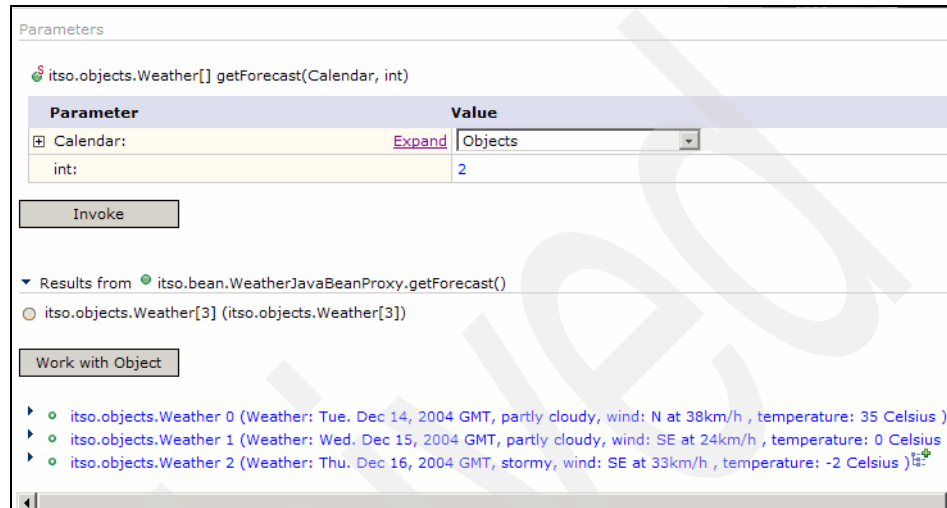


Figure 17-16 Universal Test Client: Array result

- ▶ Clicking *Work with Object* adds the whole array under Objects. You can also click the icons at the end of each object to add an individual object.
- ▶ Manipulating Calendar objects is not easy. You can retrieve a Calendar object from a result Weather object (getDate) and *Work with Object*. Then, you can use the add method to change the date. Finally, use the modified Calendar object in a Web service call.

Testing an EJB with the Universal Test Client

UTC is great for testing EJBs. You can load EJBs into UTC and then work with the home and component interfaces, for example:

- ▶ Select a session bean (WeatherEJB) in the Project Explorer in the deployment descriptor of the WeatherEJB project and select *Run on Server*.
- ▶ The WeatherEJBHome appears in UTC under EJB Beans. Execute the create method and click *Work with Object*.
- ▶ Execute the methods of the WeatherEJB.

Web services component test

Application Developer provides wizards to easily create and work with component tests for Web services. A Web service component test can be defined for Web service operations.

The Application Developer component test functions can be used to test a Web service located on either a production server or in the local runtime test environment.

The tests created by the component test wizard consist of a Java test script based on the JUnit framework and associated test data that is maintained in test data tables. The Application Developer features comply with the *UML Testing Profile* standard and use the JUnit testing framework.

For the UML Testing Profile specification, refer to:

<http://www.omg.org/cgi-bin/apps/doc?ptc/04-04-02.pdf>

After a test has been created, it can be reused throughout the whole development process. This test should be rerun whenever changes to the Web service application are made to ensure non-regression of the component.

To isolate a component-under-test (CUT) from the behavior of other components or sublayers, it is possible to create stubs. Stubs can also be generated to wrap other Web services.

Note: To use the component test functions, the IBM Rational Agent Controller must be installed and started. For installation instructions, see “Installing the Agent Controller” on page 656.

The component test functions are not included in Rational Web Developer.

Creating a Web service component test

To create a component test for a Web service, the Web service description document (WSDL) is required. Application Developer wizards can create a component test for a Web service by analyzing the service WSDL file.

Application Developer creates a regular Java 2, Standard Edition (J2SE) interface to access the Web service with the component test. For the interface creation, the WSDL2Java and Java2WSDL tools are used (see “Command-line tools, Ant, and multiprotocol binding” on page 395). The test pattern used for Web services is the same as the scenario-based testing for J2SE; however, there are no specific to Web service patterns.

Enabling the component test

To work with component tests, the workbench capabilities for Web Services Developer (see “Web services configuration settings” on page 241) and the Workbench *Tester* must be enabled. Select *Window* → *Preferences* → *Workbench* → *Capabilities* and select *Tester* (Figure 17-17).

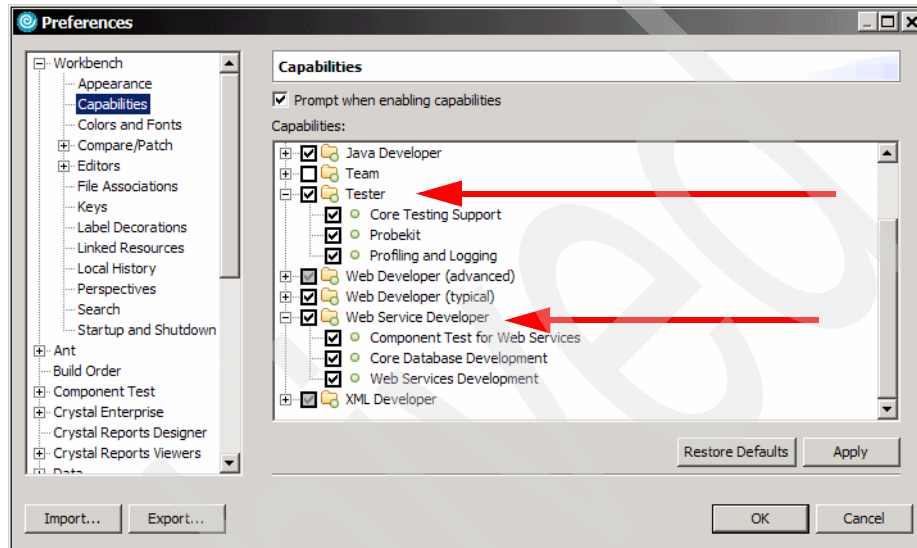


Figure 17-17 Enable Workbench Tester capabilities

Test perspective

Application Developer provides a Test perspective to create and work with component tests. To open the perspective, select *Window* → *Open Perspective* → *Other* → *Test*.

The steps to accomplish a Web service component test are:

- ▶ Create a Component Test project.
- ▶ Create a Web service component test.
- ▶ Define input and result values.
- ▶ Run the test.
- ▶ Verify the test results.

We describe each step in detail in the sections that follow.

Creating a Component Test project

A Component Test project is used to store the tests. To create the project, perform these steps:

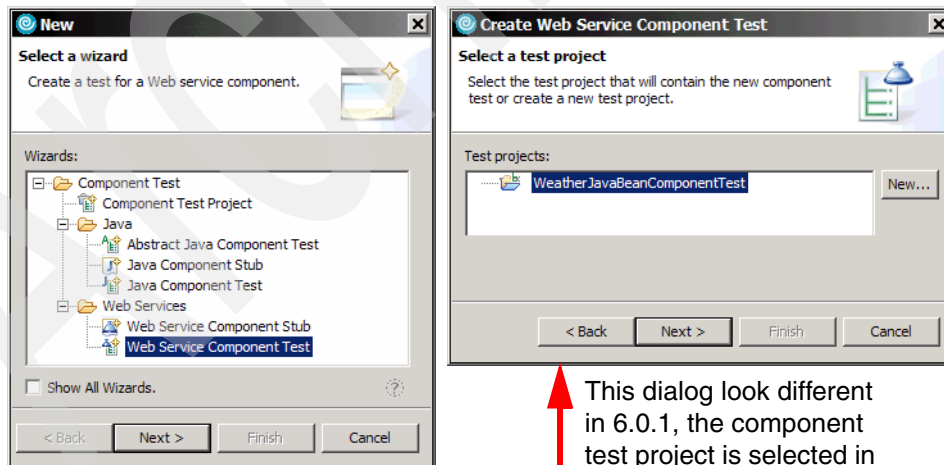
- ▶ Select *File* → *New* → *Project* → *Component Test* → *Component Test Project*. Click *Next*.
- ▶ Enter a project name, for example, *WeatherJavaBeanComponentTest*, and click *Next*.
- ▶ Define the test scope. For Web service component tests, the test scope is not necessary. The test scope is only required for J2SE or EJB tests, where the CUT (component-under-test) has to be loaded in the workspace. For the Web service test, the WSDL file contains all the required data and information. Selecting projects in this window does not affect the created component test.
- ▶ Click *Finish*.

Creating a Web service component test

Component tests are created using the Component Test wizard:

- ▶ In the Test perspective, select the *WeatherJavaBeanComponentTest* project and *New* → *Component Test* (context).
- ▶ Expand *Component Test* and select *Web Service Component Test*. Click *Next*.

Select the *WeatherJavaBeanComponentTest* project (Figure 17-18).



This dialog look different in 6.0.1, the component test project is selected in a pull-down

Figure 17-18 Component Test wizard: Web service and project

- Select the Web service under test:
 - Click *Browse* to locate the WSDL file. Expand the WeatherJavaBeanWeb project and select the WeatherJavaBean.wsdl file. Click *OK*.
 - Select the port type; WeatherJavaBean should be preselected.

Note: You have to select *Apache Axis* as runtime; otherwise, the port type list is empty.

- Clear *Use defaults* and enter `itso.weather.test` as the package.
- Click *Next* (Figure 17-19).

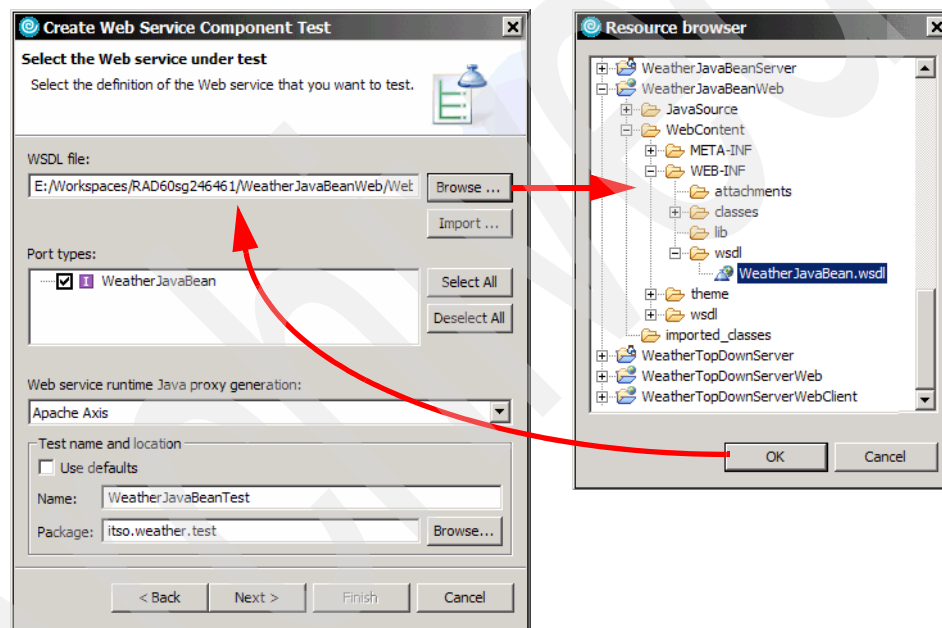


Figure 17-19 Define the Web service under test

- Select *Scenario-based Web Service Testing* (only in V6.0.1) and click *Next*.
- Define a test scenario (Figure 17-20):
 - When running the component test with the bindings defined in the WSDL file, select the empty argument method; otherwise, select the port method with the URL argument to change the service provider location.
 - Select the no-argument method and click *Add*.
 - For Testable operations, select the `getTemperatures` method and click *Add*. Multiple methods can be selected in sequence.

- It is possible to change the variable name for the ports. Select the port locator method in the test scenario list, click *Rename*, and overtype the object name. For this example, we do not change the object name.

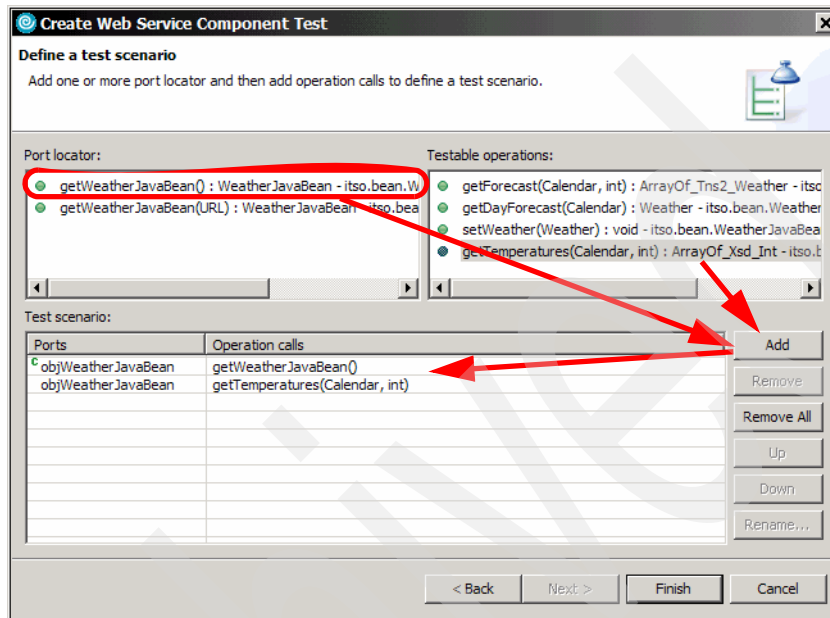


Figure 17-20 Define the component test scenario

- Click *Finish* to have the wizard create the component test artifacts.

A `WeatherJavaBeanTest` class appears under Test Suite in the project. Other artifacts are not visible in the Test perspective. In the Web perspective, you can see that proxy and helper classes were generated as well.

After the component test has been created, the test suite editor opens for the created test suite definition. Close this editor; it is not necessary for a Web service-based test for the next steps.

Defining the test data

The next step is to define the input and output values for the component test. The component test data is stored in *test data tables* (TDTs). To edit the contents of TDTs, the Application Developer editor must be used.

The component tester reads the input data for Web service operation test calls from the TDT and uses the expected values from the TDT to verify the Web service call return data. Refer to the Application Developer online help for more detailed information about test data tables.

Testing the suite implementation class

Open the test suite implementation class by selecting the WeatherJavaBeanTest class and *Open Behavior* (context). The generated Java code opens in the editor.

Click anywhere inside the test_WeatherJavaBean method to fill the Test Data Table view at the bottom (Figure 17-21).

```
package itso.weather.test;

import junit.framework.*;

public class WeatherJavaBeanTest extends TestCase {

    public void test_WeatherJavaBean() throws javax.xml.rpc.ServiceException,
        java.rmi.RemoteException {
        itso.bean.WeatherJavaBean objWeatherJavaBean = null;
        objWeatherJavaBean = new WeatherJavaBeanServiceLocator()
            .getWeatherJavaBean();
        java.util.Calendar startDate = null;
        int days = 0;
        itso.bean.ArrayOf_Xsd_Int retValue = null;
        retValue = objWeatherJavaBean.getTemperatures(startDate, days);
    }
}
```

In 6.0.1 there are additional lines of code to set the Web service URL.

In 6.0.1 the result is defined as int[].

Action	JType	In	Expected
objWeatherJavaBean = new WeatherJavaBeanServiceLoc...	xy		
objWeatherJavaBean	itso.bean.WeatherJavaB...		
<expected exception>	Throwable		
retValue = objWeatherJavaBean.getTemperatures(startD...	xy		
startDate	java.util.Calendar		
days	int		
retValue	itso.bean.ArrayOf_Xsd_Int		
<expected exception>	Throwable		

Figure 17-21 Test Data Table view for weather forecast component test

Defining the input data

The getWeatherJavaBean method does not require parameters. The required parameters for the getTemperatures operation are the date for the weather forecast (java.util.Calendar) and the number of forecast days (int). We have to define the parameter values:

- ▶ Define the forecast day input parameter:
 - The startDate parameter must be of the type Calendar, which is an abstract class. We must use a subclass, such as, GregorianCalendar. Double-click the type field (java.util.Calendar), and select *use subclass*. In the pop-up dialog box, select GregorianCalendar, and click OK (Figure 17-22).

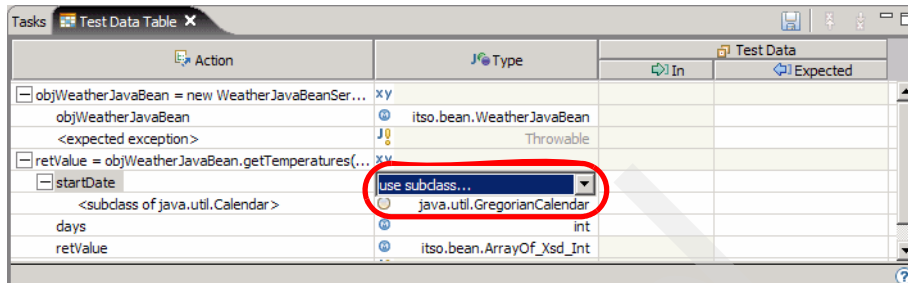


Figure 17-22 Selecting a subclass of Calendar

- Double click the `java.util.GregorianCalendar` field, and select the constructor with the parameters `year`, `month`, and `date`. Enter arguments 2004, 8, and 17 into the test data column to set the date to September 17 (months start with 0).
- For the `days` parameter, enter 2 as the value to retrieve a total of three days (Figure 17-23).

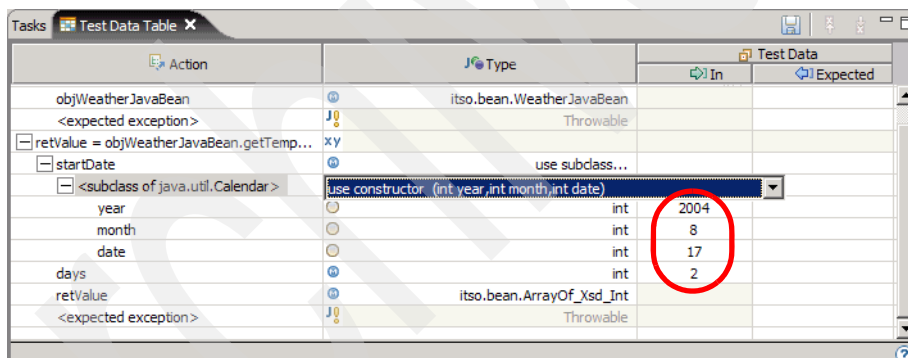


Figure 17-23 Define the input values for the getTemperature method

Defining the return data

The `getTempratures` operation returns an integer array. With the given input data (`days = 2`), the operation returns three integer values.

The WEATHER database was loaded with data for September 17-19 with temperature values of 23, 20, and 17. See “Setting up the WEATHER database” on page 708 for instructions to load the database.

We could not manage to set the return values for the `ArrayOf_Xsd_Int`. Therefore, we added three lines of code to the Java test case to retrieve the return values:

```
retValue = objWeatherJavaBean.getTemperatures(startDate, days);
int value1 = retValue.get_int(0); <=== not necessary in 6.0.1
int value2 = retValue.get_int(1); <=== not necessary in 6.0.1
int value3 = retValue.get_int(2); <=== not necessary in 6.0.1
```

Next, we defined the expected values for the three temperatures (Figure 17-24).

Action	Type	In	Expected
objWeatherJavaBean = new WeatherJa...	itsio.bean.WeatherJavaBean		
<expected exception>	Throwable		
retValue = objWeatherJavaBean.getTe...	use subclass...		
startDate	use constructor (int year,int month,int date)		
year	int	2004	
month	int	8	
date	int	17	
days	int	2	
retValue	itsio.bean.ArrayOf_Xsd_Int		
<expected exception>	Throwable		
value1 = retValue.get_int(0)	int		23
<expected exception>	Throwable		
value2 = retValue.get_int(1)	int		20
<expected exception>	Throwable		
value3 = retValue.get_int(2)	int		17
<expected exception>	Throwable		

Figure 17-24 Set the expected return values for the test operation

Changes in the test data table are not saved automatically. When changing data in the test data table, the diskette icon on the Test Data Table view changes from disabled (gray), to enabled (light blue). This is the only indicator for changes in the TDT. To save the changes in the data table, either click the diskette icon  or press Ctrl-S.

Tip: To limit the size of the test data table, you can include statements in the test case to disable or enable the data table for some lines of code:

```
ComponentTest.disableTestDataTable();
// lines of code that do not show in the data table
ComponentTest.enableTestDataTable();
```

Running the test

To run the test, select the WeatherJavaBeanTest suite and *Run* → *Component Test*. The test progress is only shown by the Workbench background indicator (at the right bottom of the Workbench).

The test is finished when the background task indicator disappears and a new file is created in the component test project Run folder.

Verifying the test results

The test results can be viewed within Application Developer, or can be exported as HTML pages.

Expand all the levels for the newly created execution artifact in the Run folder, and select the entry Individual Test #0. This opens the Test Data Comparator view, which shows the detailed Web services test results. The Actual column shows the effective test return values (Figure 17-25).

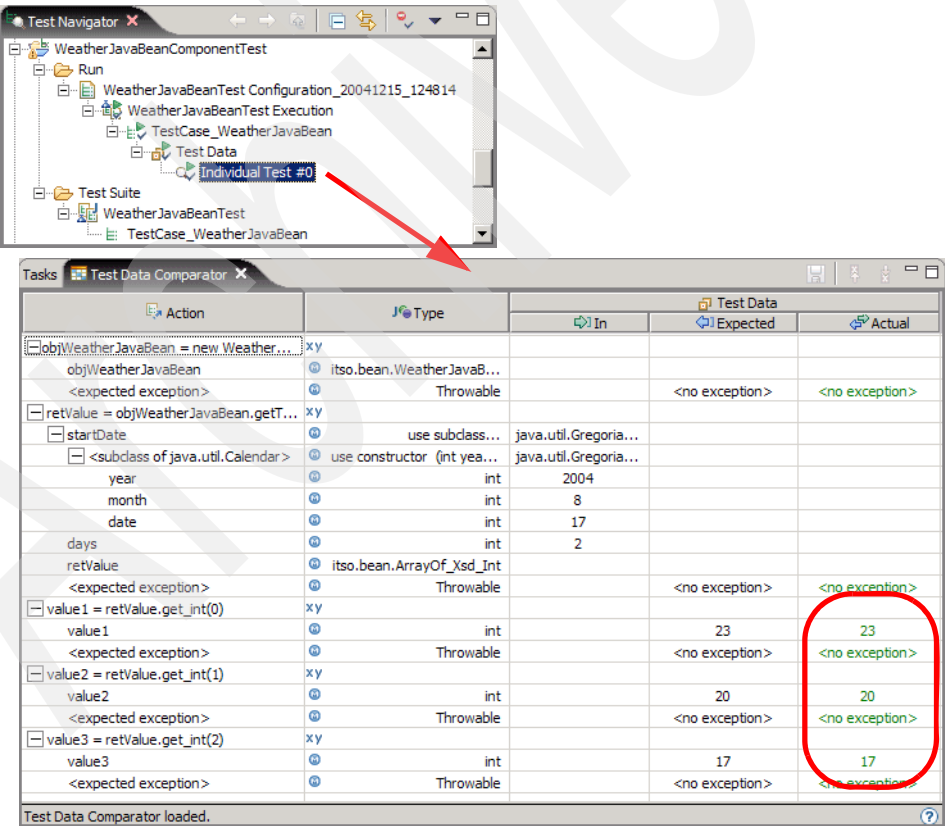
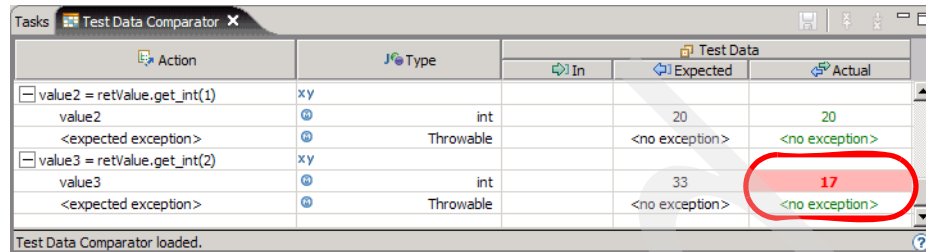


Figure 17-25 Test Data Comparator for a successful test run

To demonstrate a failing test run, change one expected return value, and run the test again. Figure 17-26 shows the Test Data Comparator view with a failed test run.



Action	J2E Type	Test Data		
		In	Expected	Actual
value2 = retValue.get_int(1)	xy			
value2	int		20	20
<expected exception>	Throwable		<no exception>	<no exception>
value3 = retValue.get_int(2)	xy			
value3	int		33	17
<expected exception>	Throwable		<no exception>	<no exception>

Test Data Comparator loaded.

Figure 17-26 Test Data Comparator for a failed test run

To get an quick overview of a test run result (successful or failed), the Test Navigator view shows an indication icon of the results (Figure 17-27).

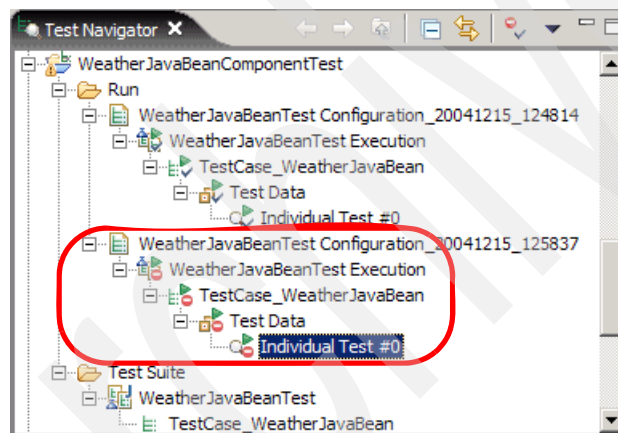


Figure 17-27 Successful and failed test indication icons

Exporting the test results as HTML

To export test results as HTML, select *File* → *Export*, and select *Component test execution results*. In the Export dialog box (Figure 17-28):

- ▶ Select the test results to be exported.
- ▶ Select a destination folder.
- ▶ Select all or only failed execution results.
- ▶ Select *Only one HTML file* (instead of one file per test).
- ▶ Click *Finish*.

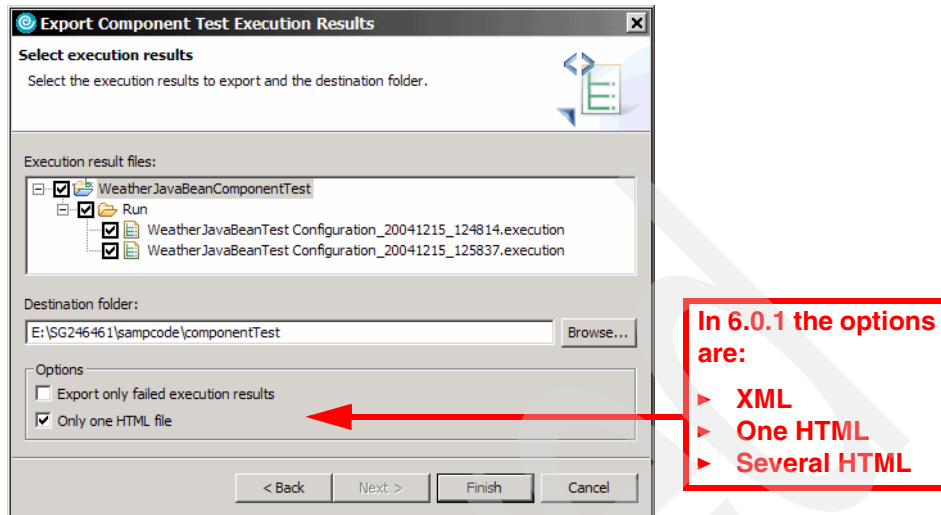


Figure 17-28 Export execution test results as HTML files

To view the exported test results, open the generated HTML file. The file displays an overview with links and passed and failed executions, followed by individual results. Click a link to jump to that test. Figure 17-29 shows the overview part.

Report files	E:\Workspaces\RAD60sg246461\WeatherJavaBeanComponentTest\Run\WeatherJavaBeanTest Configuration_20041215_124814.execution E:\Workspaces\RAD60sg246461\WeatherJavaBeanComponentTest\Run\WeatherJavaBeanTest Configuration_20041215_125837.execution		
File creation date	Wed, 15 Dec 2004 13:18:58.958		
Pass	1	Inconclusive	0
Fail	1	Error	0
Total	2	Success	50 %

Figure 17-29 Execution test results as an HTML file

The presentation of the detailed test results in the Report files is similar to the test data table representation in Application Developer.

Tip: A component test project should be shared in a repository. This helps to share test suites and test cases in a development team. Further, when sharing a component test project, important test results can be saved (versioned) for later reviews.

Monitoring with Tivoli Performance Viewer

In this section, we describe the Web services monitoring functions provided with WebSphere Application Server Version 6. This functionality is provided with IBM Tivoli® Performance Viewer. Starting with WebSphere Application Server Version 6, Tivoli Performance Viewer is fully integrated in the WebSphere administration console. Tivoli Performance Viewer enables administrators and programmers to monitor the overall health of WebSphere Application Server without leaving the administrative console.

We cover the following subjects in this section:

- ▶ Enabling and starting Tivoli Performance Viewer
- ▶ Monitoring Web service requests

For more information about the Tivoli Performance Viewer, refer to the *WebSphere Application Server Information Center*.

Enabling and starting Tivoli Performance Viewer

To collect data with Tivoli Performance Viewer, the Performance Monitoring Infrastructure (PMI) service must be enabled. The PMI monitoring level *Basic* is enabled by default in Application Server Version 6.

To effectively enable or disable the PMI, the appropriate server has to be restarted.

The first step is to verify that the PMI is enabled for the test server. Start the administration console by selecting the server in the Servers view and *Run administrative console*. After the console opens, log in with your regular user ID and perform these steps:

- ▶ Select *Servers* → *Application servers*, and then select *server1*.
- ▶ Select *Performance Monitoring Infrastructure (PMI)* in the Performance section.
- ▶ Verify that *Enable Performance Monitoring Infrastructure (PMI)* is selected (Figure 17-30). By default, *Basic* monitoring is enabled.

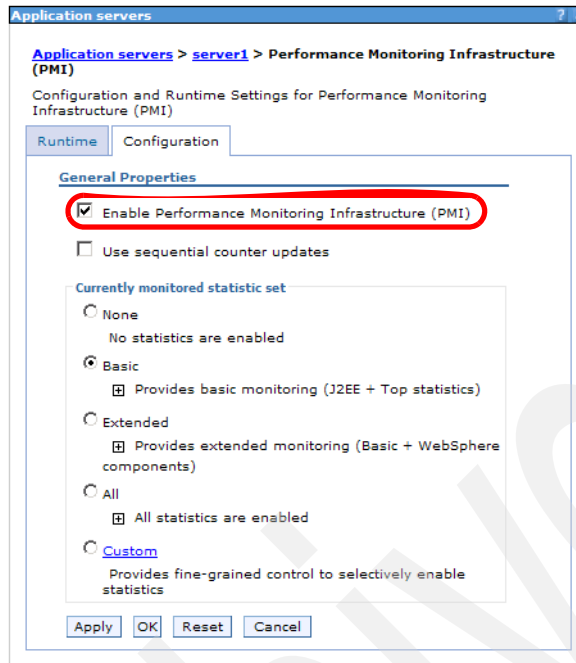


Figure 17-30 Verify that PMI is enabled

Enabling Web service PMI metrics

To have WebSphere report PMI data for Web services, the monitoring statistics have to be changed. There are two statistic sets that can be used. Either the level *All* can be enabled, or the *Custom* level set is defined with the appropriate measure settings.

For this example, we define the Web services metrics through the *Custom* set metric level so that we receive only Web services-specific data in Tivoli Performance Viewer:

- ▶ Select *Custom*, and a dialog box opens to configure the settings.
- ▶ Select *Web services* in the left pane.
- ▶ Select all the counters by clicking .
- ▶ Click *Enable* (Figure 17-31).

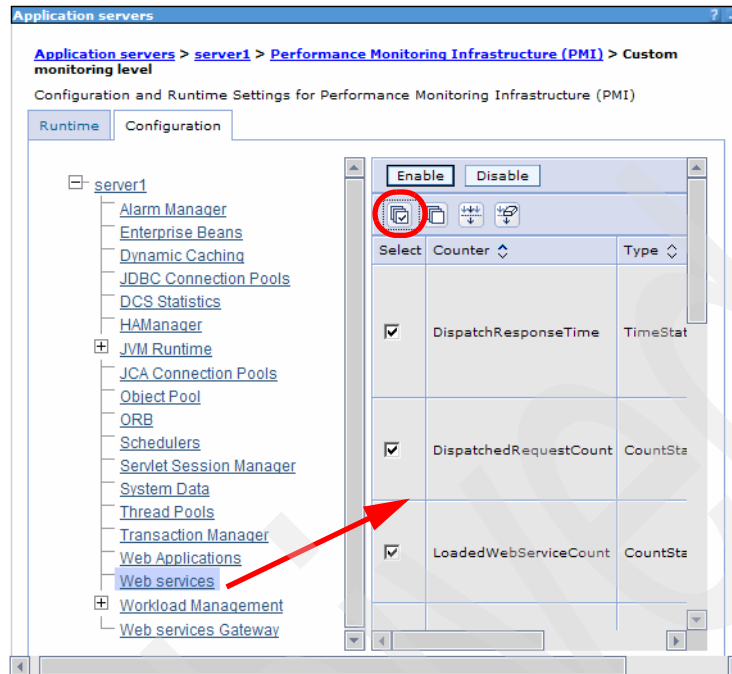


Figure 17-31 Enable Web services metrics in the custom monitoring level

After enabling the counters, save the configuration (click *Save*) and restart the server in the Servers view.

Monitoring Web service requests

When the server is ready, you can verify in the administrative console that monitoring is enabled. Select the *server1* and *Performance Monitoring Infrastructure (PMI)*. In the Runtime tab, select *Custom*, and then select *Web services*, and you can see in the Status column (right side) that the counters are enabled.

Using Tivoli Performance Viewer

In the administration console, select *Monitoring and Tuning* → *Performance Viewer* → *Current Activity*.

- ▶ Select the check box next to *server1* and click *Start Monitoring*.
- ▶ Run some of the Web services that you have created in Chapter 16, “Develop Web services with Application Developer V6.0” on page 273.
- ▶ Click *server1* to open the Tivoli Performance Viewer window.

To view the Web services monitoring data in the Tivoli Performance Viewer, expand the *server1* → *Performance Modules* tree and select *Web services*.

Tivoli Performance Viewer provides several types of measured data for Web services. Table 17-1 shows and explains the counters.

Table 17-1 Tivoli Performance Viewer Web services counters

Counter name	Provided data
LoadedWebServiceCount	The number of loaded Web services
ReceivedRequestCount	The number of requests that are received by a service
DispatchedRequestCount	The number of dispatched requests that target service implementation
ProcessedRequestCount	The number of requests that are dispatched with corresponding replies that are returned successfully
ResponseTime —milliseconds	The average time between the receipt of a request and the return of the reply
RequestResponseTime —milliseconds	The average time between the receipt of a request and the dispatch for processing of the request
DispatchResponseTime —milliseconds	The average time between the dispatch of a request and the receipt of reply; Enabled
ReplyResponseTime —milliseconds	The average time between the dispatch of the reply and the return of the reply
PayloadSize —bytes	The average payload size of a received request and reply
RequestPayloadSize —bytes	The average payload size of a request
ReplyPayloadSize —bytes	The average payload size of a reply

Click *View Module(s)* to display the counters (Figure 17-32).

The top of the viewer shows the graphical representation of the data. Select the counters that you want to show in the graph.

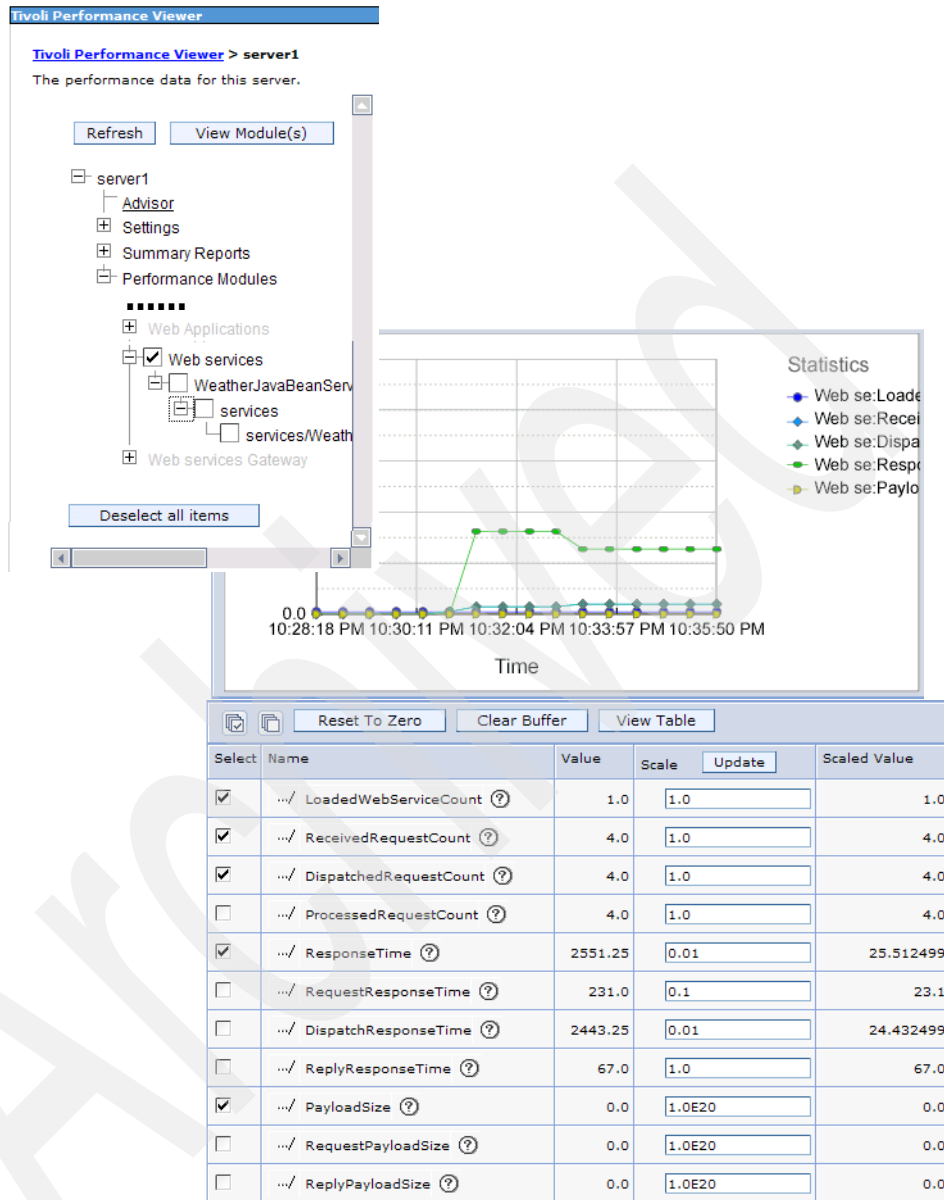


Figure 17-32 Tivoli Performance Viewer: Data

Tip: If the graph does not open, install the Scalable Vector Graphics (SVG) plug-in, available at:

<http://www.adobe.com/svg/viewer/install/main.html>

TCP/IP Monitor

To monitor Web services communicating over TCP/IP, the TCP/IP Monitor can be used. TCP/IP Monitor is a trace facility build into the Application Developer workbench. We cover the configuration, start, and use of the TCP/IP Monitor in this section.

TCP/IP Monitor is the recommended monitoring tool when using Application Developer. When monitoring TCP/IP communications in a WebSphere Application Server environment, the *tcpmon* tool should be used (see “WebSphere Application Server TCP/IP Monitor” on page 372).

The TCP/IP Monitor is a simple server running inside Application Developer and can monitor all TCP/IP traffic for specific ports. Therefore, all TCP/IP-based communication between a Web service server and client can be traced.

To configure and use the TCP/IP Monitor, Application Developer provides these features:

- ▶ TCP/IP Monitor preferences page
- ▶ TCP/IP Monitor view

Defining a TCP/IP Monitor configuration

We configure the monitor in the Application Developer Preferences:

- ▶ Select *Window* → *Preferences* → *Internet* → *TCP/IP Monitor* (Figure 17-33).

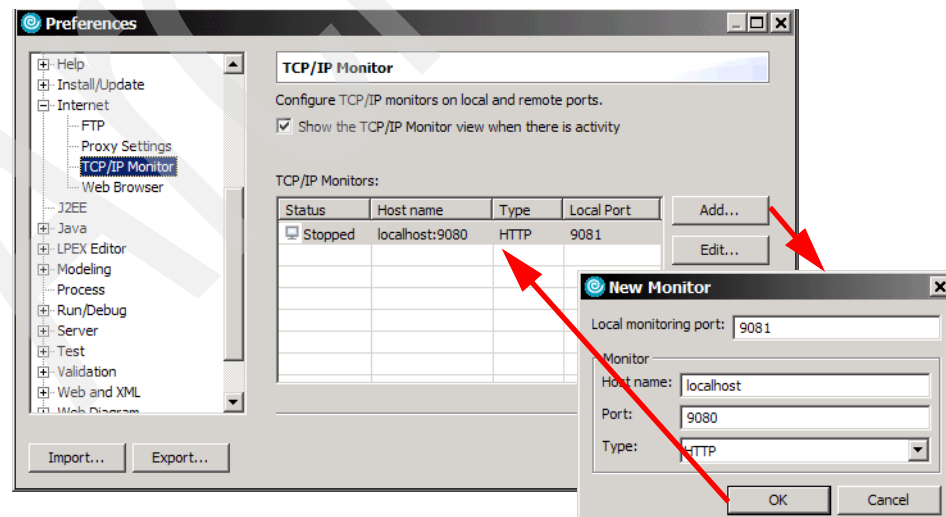


Figure 17-33 Configuring the TCP/IP Monitor

- ▶ Select *Show the TCP/IP Monitor view when there is activity*. Enabling this option opens and activates the TCP/IP Monitor view automatically when any TCP/IP traffic on one of the defined and started ports is detected.
- ▶ Define TCP/IP Monitors to manually start and stop monitoring ports. To define a new port, click *Add* and enter the appropriate data in the fields:
 - Local monitoring port—Specify the local port name. When using this port for a TCP/IP request, the TCP/IP Monitor traces the communication and routes the request to the system defined in the following TCP/IP Monitor settings.
 - Host name—Enter the host name or IP address of the Web service server.
 - Port—Enter the port number of the server.
 - Type—To monitor Web services, we recommend that you use the HTTP setting. With the HTTP type setting, the HTTP header information in the request and response can be viewed separately. The displayed message data is only XML and is easier to read.
- ▶ To enable monitoring, select the new monitor and click *Start*. All TCP/IP requests to the defined local monitoring port will now go through the TCP/IP Monitor.

TCP/IP Monitor view

All traced TCP/IP activities are shown in the TCP/IP Monitor view. The view either opens automatically when configured in the monitor preferences, or can be added to a perspective by selecting *Window → Show View → Other → Debug → TCP/IP Monitor*.

Running a Web service through the monitor

There are two ways to route the Web service traffic through the monitor:

- ▶ Open the service locator in the client project. For example, open the `WeatherJavaBeanServiceLocator` in the `WeatherJavaBeanClientWeb` project. Find the address and change the port to 9081:

```
private final java.lang.String weatherJavaBean_address =
    "http://localhost:9081/WeatherBeanWeb/services/WeatherJavaBean";
```

Do not forget to reset the port when you are done testing.

- ▶ Dynamically change the endpoint address by invoking the `setEndpoint` method of the proxy class. This can also be done from the test client JSPs where the `getEndpoint` and `setEndpoint` methods are enabled:
 - Invoke the `getEndpoint` method and copy the result.

- Invoke the `setEndpoint` method with the changed address (use copy/paste).

After you have changed the endpoint address, run the `getDayForecast` method through the monitor.

Using the TCP/IP Monitor view

The TCP/IP Monitor view consists of the three panes:

- **Requests**—Located at the top of the TCP/IP Monitor view (Figure 17-34), this pane contains the list of all monitored requests.

The request list shows the requests grouped by the host system server name and port. When selecting a request, the monitor time for request, response time, and request type are shown on the right side of this view. The response time is the elapsed time that it takes for a request to arrive at the TCP/IP Monitor until the response is forwarded from the server.

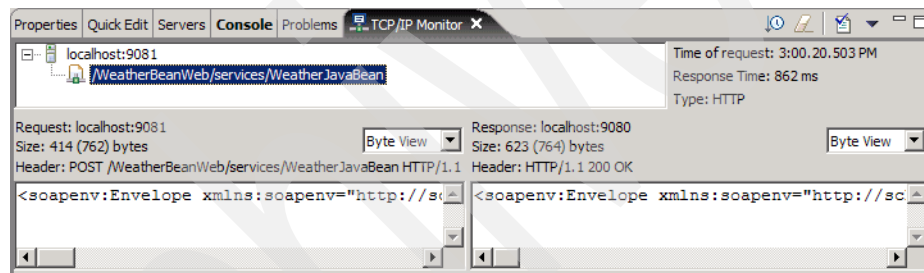


Figure 17-34 TCP/IP Monitor view

- **Request**—Located on the left of the TCP/IP Monitor view, this pane contains information about the request as forwarded to the application server. We recommend that you change the view type to *XML View*, which is only possible when the monitor configuration is defined with HTML type (Figure 17-35).
- **Response**—Located on the right side of the TCP/IP Monitor view, this pane contains information about the response as received from the application server. As with the Request pane, this view should be viewed in XML.

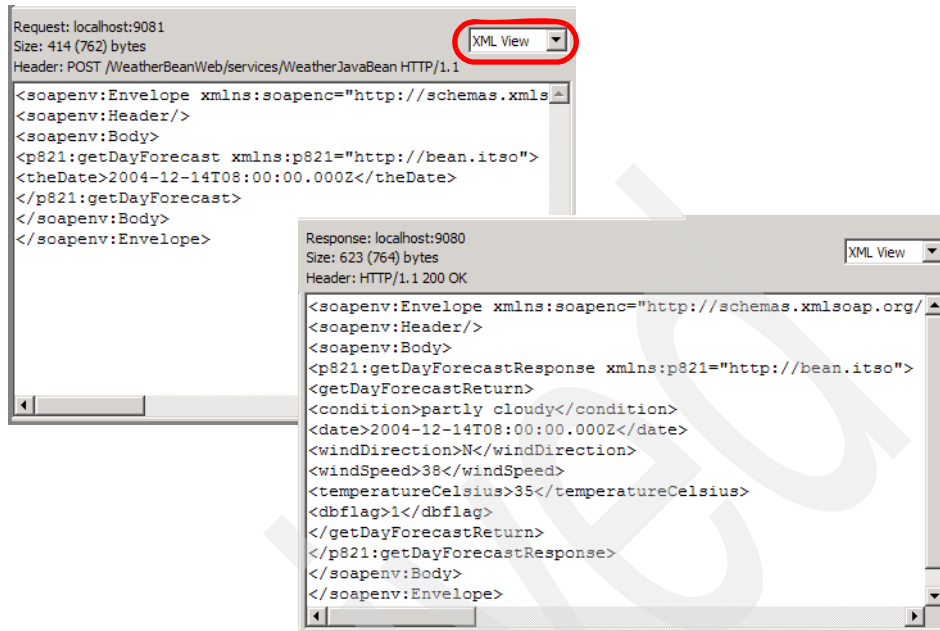


Figure 17-35 TCP/IP Monitor view: Request and response as XML

To view the HTTP header for the request and response, click the Menu icon , and enable *Show header* (Figure 17-36).

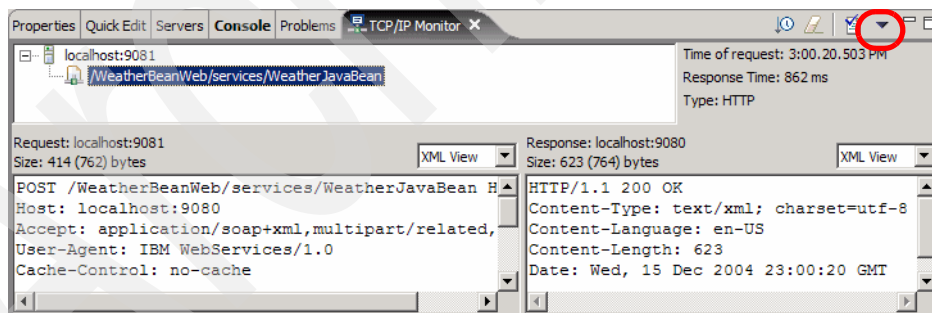


Figure 17-36 Show the HTTP header in the monitor view

Using the TCP/IP Monitor menu icons, it is also possible to sort the requests by response time, clear the list, view the WS_I validation log, suppress SOAP security parts of a message, and open the properties windows to change the configuration.

Monitoring one server

The TCP/IP Monitor can also be configured in the Servers view:

- ▶ Select the server and *Monitoring* → *Properties* (Figure 17-37).
- ▶ Click *Add* to define a monitor for one of the server ports.
- ▶ Click *Start* to start the monitor.

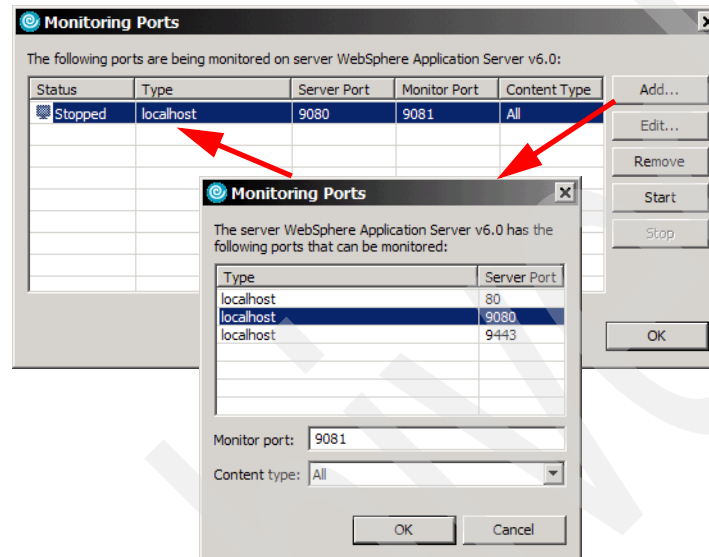


Figure 17-37 Adding a monitor to a server

WebSphere Application Server TCP/IP Monitor

WebSphere Application Server provides a similar tool called `tcpmon`. The function of this tool is the same as the TCP/IP Monitor. The `tcpmon` tool is provided as a Java rich-client application and can be used without Application Developer.

`Tcpmon` is included in the Java archive file `<WAS_HOME>\lib\webservices.jar`.

Starting tcpmon

To start `tcpmon`, switch to the directory `<WAS_HOME>/lib`, and execute:

```
<JAVA_HOME>\bin\java -cp webservices.jar  
com.ibm.ws.webservices.engine.utils.tcpmon  
[localPort remoteHostName remotePort]
```

When providing the optional command parameters, tcpmon immediately starts monitoring with this configuration. Without the parameters, only the tcpmon user interface starts, and a configuration must be provided.

For the following example, we started tcpmon with these commands (after stopping the monitor in Application Developer):

```
cd c:\<RAD60_HOME>\runtimes\base_v6\lib
java -cp webservices.jar com.ibm.ws.webservices.engine.utils.tcpmon 9081
localhost 9080
```

This starts tcpmon, as shown in Figure 17-38.

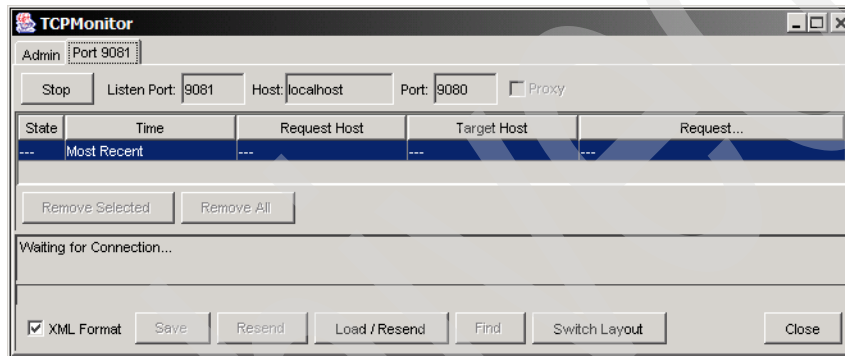


Figure 17-38 Tcpmon started with command-line options

The Admin page is used to configure tcpmon when started without any options, or to add further monitor ports. On the Admin page, the listener port, target host, and port can be added. A tcpmon configuration supports routing requests through proxy servers.

Tcpmon provides functions to save, load, and resend monitored SOAP requests. Further, for the request and response message, an XML format can be applied, which makes a SOAP message much easier to read.

Figure 17-39 shows the tcpmon window for a sample SOAP request.

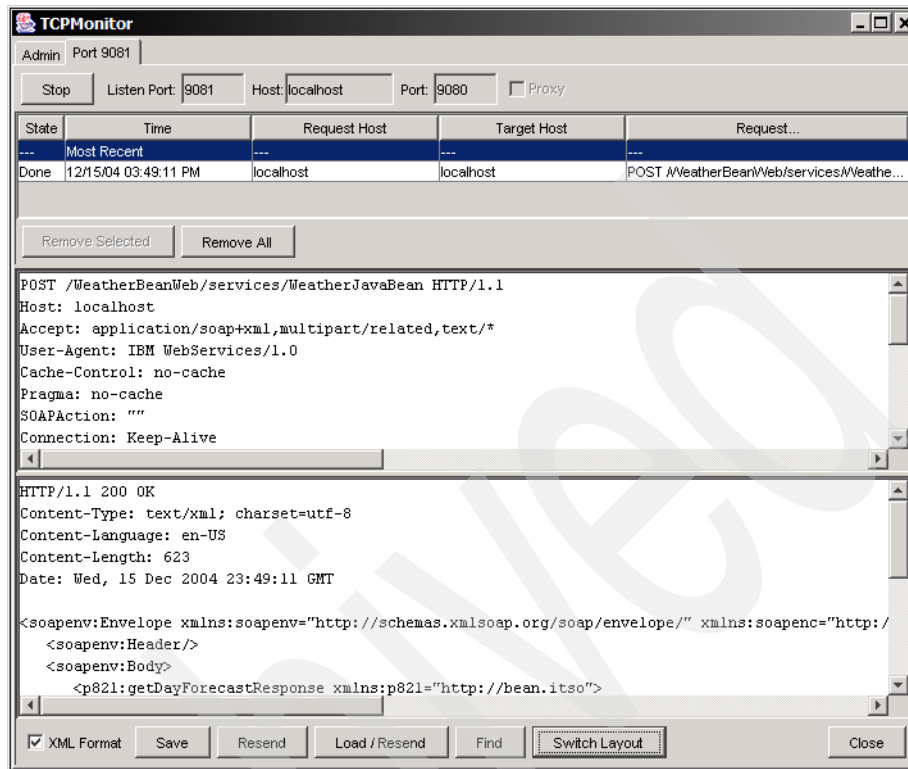


Figure 17-39 Sample SOAP request in tcpmon

You can change the data in the request and click *Resend* to execute the same Web service with different data. You can save requests into files for reexecution.

Use the *Switch Layout* button to position the request and response next to each other.

Summary

In this chapter, we explained the functions provided with WebSphere Application Server and Rational Application Developer to test and monitor Web services.

We explained the variety of different test functions provided with Application Developer. With the functions to build and run manual and automated Web services test, Application Developer provides functions that round out a whole Web services development process.

The monitoring functions are mainly focused on Web service development use, but can also be helpful in analyzing problem situations in production environments. We did not cover production runtime monitoring. To monitor a Web service production environment, we recommend that you refer to the following products:

- ▶ WebSphere Studio Application Monitor:
<http://www.ibm.com/software/awdtools/studioapplicationmonitor/>
- ▶ Tivoli product information:
<http://www.ibm.com/software/tivoli/>

More information

Further information about the test functions in Application Developer can be found in the product help system by selecting *Help* → *Help Contents* → *Testing applications*.

Additional information about the Tivoli Performance Monitor can be found in the *WebSphere Application Server Information Center*, available at:

<http://www.ibm.com/software/webservers/appserv/infocenter.html>

Deploy and run Web services in WebSphere Application Server V6.0

This chapter describes Web services deployment in the IBM WebSphere Application Server environment. We cover administration basics necessary to understand the deployment process and describe Web service Java implementation packaging and SOAP enablement, as well as Web service administration. We also discuss some useful troubleshooting techniques.

It is not the intention of this chapter to cover the WebSphere Application Server details beyond the scope necessary to understand Web services deployment. However, we provide links to additional resources. In addition, Chapter 13, “IBM products for Web services” on page 233 provides a product overview, and Appendix A, “Installation and setup” on page 649 covers the product installation.

Overview

In this section, we cover IBM WebSphere Application Server as a Web services deployment platform of choice. We do not discuss all the product details. We only deal with the aspects necessary to understand Web services deployment and administration. Web services security, monitoring, and setting up a private UDDI registry are covered in separate chapters.

WebSphere Application Server general concepts

WebSphere Application Server, Application Server for short, represents one of the basic building blocks of the enterprise e-business environment. It is a part of the IBM WebSphere product family, which we describe in Chapter 13, “IBM products for Web services” on page 233.

Administration basics

Application Server Version 6 continues the tradition of WebSphere Application Server Version 5.x of using a Web-based client to manage the application server and to store all configuration data in a file system using XML as the chosen format. In addition to the Web-based administration application, command-line tools also are available for interactive and batch processing of administrative tasks.

WebSphere Application Server topology building blocks

Before we discuss the details of Web application deployment, let us first cover some fundamental administrative concepts:

- ▶ **Managed process or server**—Denotes any instance of a JVM that can be managed in a WebSphere environment. Application servers, JMS servers (a special type of server that supports JMS communication), node agents, and deployment managers are all examples of managed processes. We discuss node agents and deployment managers in the subsections that follow.
- ▶ **Node agent**—Responsible for controlling all the servers running on a certain machine. These servers can be application servers or a JMS server, of both. In most cases, we find a single node agent on one physical system, although it is also possible that on some very high-end systems, multiple node agents can be concurrently active.
- ▶ **Cell**—A network of related node agents makes a cell. The concept of a cell is very similar to the concept of *domain*, which we know from the previous versions of WebSphere Application Server. In each cell, there is a single deployment manager. However, despite its similarity, this process is *not*

equivalent to the administrative server of previous releases; its main purpose is to allow an administrator to manage the resources in the entire cell.

- **Profile**—A set of files that describe a runtime environment. You can have more than one profile defined, and each application can be configured to run in different profiles, if any.

Figure 18-1 shows an example of a WebSphere Application Server Version 6 topology.

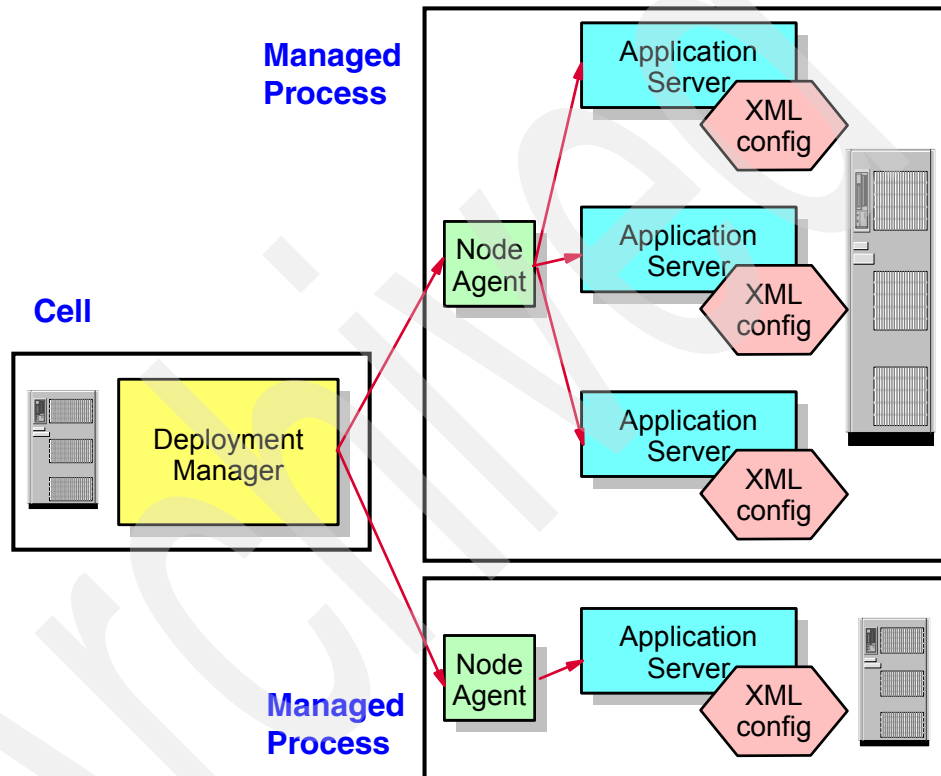


Figure 18-1 WebSphere Application Server Version 6 example topology

Each physical machine holds a separate installation of the product that is not aware of installations on other machines. The configuration files are XML files. In the base application server, the node agent code is there although it performs no role until the node is added into a cell.

WebSphere Application Server Network Deployment (Network Deployment for short) represents a single point of administration of multiple nodes. It is possible to install Network Deployment on any machine in the network to create a network

deployment manager cell. A WebSphere Application Server installation on the same machine is not a prerequisite.

After we install and start the Network Deployment instance, we can use the **addNode** tool to add a WebSphere Application Server instance (node) to the Network Deployment cell. The network deployment manager assumes responsibility for the configuration of any application server nodes that belong to the cell.

Each process keeps all the data necessary to start itself. The configuration data is in XML files, and the application data in enterprise application EAR files. In the Network Deployment environment, you can attach an administrative client to any managed process to make changes to the configuration. The deployment manager contains the master repository of XML files. Individual nodes and servers synchronize the files with the master configuration data, and all the changes applied to them directly are lost after synchronization. It is best to change configuration data only through the deployment manager.

Upgrading from Version 5

The update from Version 5 of WebSphere Application Server is performed by a wizard that guides you through the actual migration. You need to have access to the Version 5 deployment manager and to have created an empty profile on the Version 6 server. More information about the migration process can be found in the product's *Information Center*.

Administrative console

The WebSphere Application Server administrative console is a graphical, Web-based tool designed to manage the WebSphere Application Server administrative server. The administrative console builds on the Admin Web application architecture and functions that were introduced in WebSphere Application Server Version 4.0 AEs and extended in Version 5.

The administrative console Web application is installed by default during the Application Server installation. After we start the server (in the case of the base application server, the command is `startserver server1`), we can access the console using a Web browser:

```
http://<hostname>:9060/ibm/console
```

If global security is disabled, no password is required and the user ID we have to enter is used only to track and save user-specific configuration data. If, however, security is enabled, you are redirected to:

`https://<hostname>:9043/ibm/console`

Figure 18-2 shows the initial layout of the administrative console.

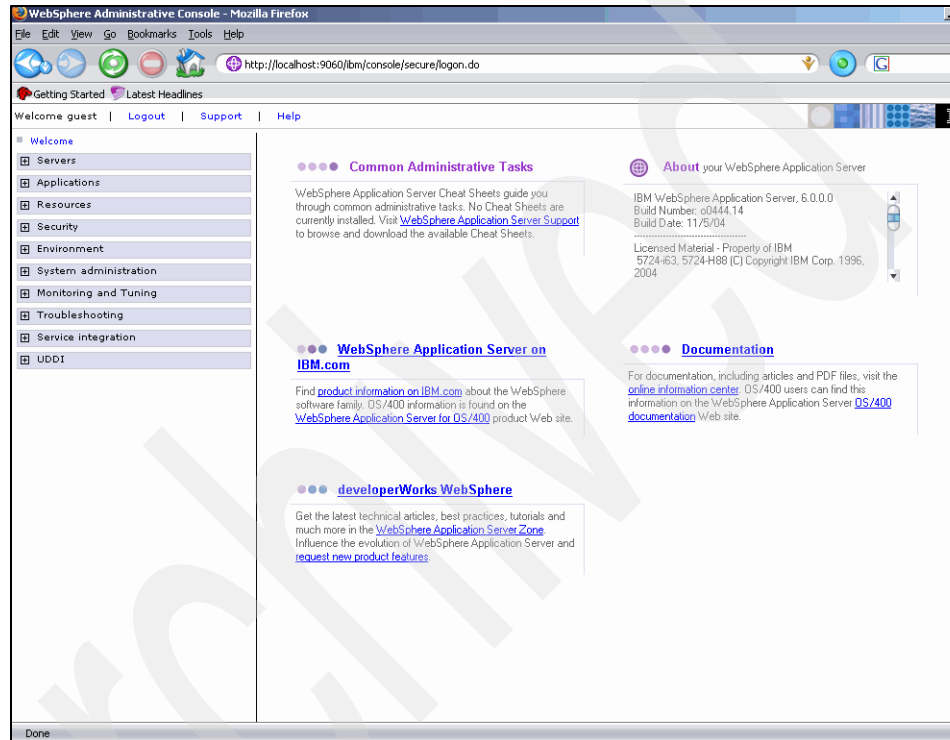


Figure 18-2 WebSphere Administrative Console

The administrative console window contains the following main areas: A task bar, a navigation tree view, and a workspace. We can resize these areas to fit our needs. Figure 18-2 shows these areas:

- ▶ **Task bar**—The task bar offers options for logging out of the console, accessing product information, and accessing support.
- ▶ **Navigation tree**—The navigation tree on the left side of the console offers links to console pages that you use to create and manage components in a WebSphere Application Server administrative cell.

- ▶ **Workspace**—The workspace on the right side of the console contains pages that you use to create and manage configuration objects such as servers and resources. Click the links in the navigation tree to view the different types of configured objects. Within the workspace, click the configured objects to view their configurations, runtime status, and options.

Enterprise application deployment

Enterprise applications (or J2EE applications) are applications that conform to the Java 2 Platform Enterprise Edition specification. They consist of EJB modules (EJB JAR files), Web modules (WAR files), connector modules (RAR files), and application client modules (JAR files). None of these modules are mandatory, and any combination of modules is allowed. Optionally, additional JAR files containing dependent classes or other components required by the application can be added to the application. An enterprise application is packaged in an enterprise archive (EAR) file.

The WebSphere Application Server provides two *assembly tools*, which replace the Application Assembly Tool (AAT) and the Assembly Toolkit of earlier Application Server versions:

- ▶ Rational Web Developer is the basis for the Rational Application Developer. It is an integrated development environment that provides development, testing, assembly, and deployment capabilities.
- ▶ WebSphere Application Server Toolkit (ASTK) is a separate product and has to be installed in addition to WebSphere Application Server. Follow the instructions available with the ASTK to install the Application Assembly Tool that comes with ASTK.

Both tools can be used independently of each other and provide for more than just the assembly of applications.

Configuring the server with a data source

The weather forecast application requires a data source with the JNDI name of `jdbc/weather`, which can point to DB2 or Cloudscape.

This data source is configured in the extended deployment descriptor of the three basic enterprise applications in Application Developer: `WeatherJavaBeanServer`, `WeatherEJBServer`, and `WeatherEJBJMSServer`.

Such a configuration can propagate to a real WebSphere server. However, in many cases, the real server might be configured differently than the built-in test environment. It is, therefore, advisable to configure the real server manually.

Configuring WebSphere variables

Access to JDBC drivers is done using a variable. To define the variable for DB2, open the administrative console, and in the Navigation bar, select *Environment* → *WebSphere Variables*. Select the `DB2_JDBC_DRIVER_PATH` variable and enter the location of the JAR files, for example, `C:\SQLLIB\java`, and click *OK*.

Defining the data source for the weather forecast database

We require a data source for the WEATHER database. In this example, we show how to configure the data source for DB2; Cloudscape would be similar.

In the administrative console, perform these steps:

- ▶ In the Navigation bar, select *Resources* → *JDBC Providers*, and then click *New*. (Note that we define at the *node* level.)
- ▶ Select *DB2* as database type, and then select *DB2 Legacy CLI-based Type 2 JDBC Driver*, and then select *XA data source*. Click *Next*.
- ▶ The panel expands, click *Apply*, and an *Additional Properties* section is activated (Figure 18-3).

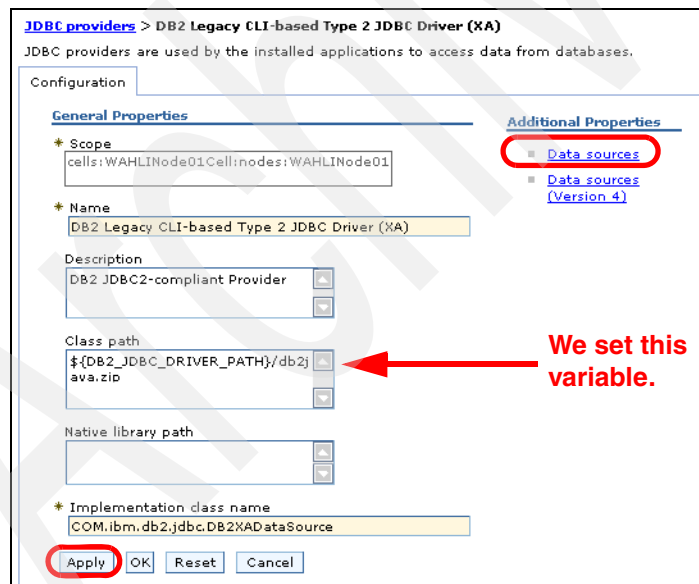


Figure 18-3 JDBC provider for DB2

- ▶ Under *Additional Properties*, select *Data sources*, and then click *New*. Enter `WEATHERDataSourceDB2` as the Name and `jdbcn/weather` as the JNDI Name. At the bottom, enter `WEATHER` as the database name. Click *Apply*.

- ▶ Under Related Items, select *J2EE Connector Architecture (J2C) authentication data entries*, and then click *New*.
- ▶ Enter DB2user as the Alias, and a valid user ID and password with access to DB2.
- ▶ In the selection line at the top, select *WeatherDataSourceDB2* to get back to the data source definition.
JDBC Providers > DB2 Legacy CLI-based Type2 2 JDBC Provider (XA) > Data Sources > **WEATHERDataSourceDB2** > J2EE Connector
- ▶ Select the new DB2user alias for the Component-managed authentication alias and the Container-managed authentication alias. Click *OK*.
- ▶ Under Additional Properties, select *Custom Properties*. Select the databaseName property, set the value to WEATHER, and click *OK*.

Save the configuration, and then restart the server.

Installing an enterprise application

In this section, we describe the steps required to use the administrative console to install an (enterprise) application.

Note: For this example, we install the JavaBean Web service enterprise applications, WeatherJavaBeanServer and WeatherJavaBeanClient.

We can export the EAR files from Application Developer and place them into the <WAS_HOME>\installableApps directory.

We begin by selecting *Applications* → *Install New Application* in the console navigation tree. The first of the Preparing for the application installation pages opens.

Preparing for the application installation

Preparing for the application installation consists of two separate pages that we describe briefly.

On the first page, we have to specify the full path name of the enterprise application EAR file (Figure 18-4).

We can also specify a stand-alone WAR or JAR file for installation. If we are installing a stand-alone WAR file, we have to specify the context root. Click *Next* to proceed to the second page.

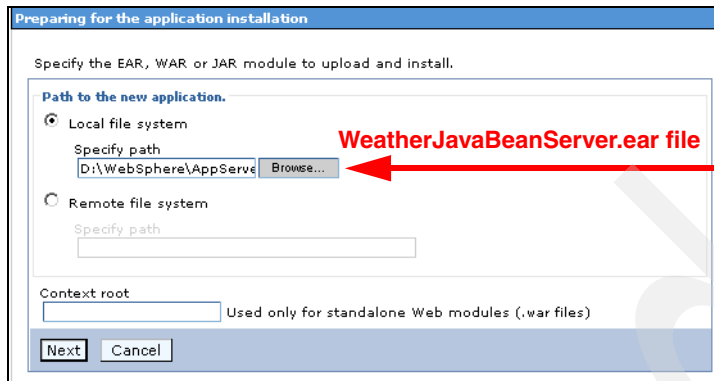


Figure 18-4 Preparing for application installation (1)

On the second page, we select whether to generate default bindings (Figure 18-5). Using the default bindings causes any incomplete bindings in the application to be filled in with default values. Existing bindings are not altered. Click *Next*.

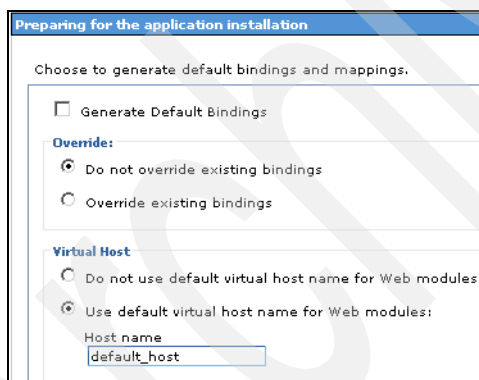


Figure 18-5 Preparing for application installation (2)

Install new application pages

The Install new application pages consist of six pages, which we describe briefly in the remainder of this section.

We now show the Install New Application pages, starting with the Select installation options page. If we decide to generate default bindings, we could proceed to the Summary page, but it is always better to through the pages.

Select installation options

In this step, we provide values for the following settings specific to WebSphere Application Server (Figure 18-6). Default values are used if a value is not specified.

The screenshot shows the 'Install New Application' dialog box. On the left is a sidebar with five steps: Step 1 (selected), Step 2, Step 3, Step 4, and Step 5. The main area is titled 'Select installation options' and contains the following settings:

- ☐ Pre-compile JSP (indicated by a red arrow and the text 'could choose')
- Directory to install application: [text box]
- ☒ Distribute application
- ☐ Use Binary Configuration
- ☐ Deploy enterprise beans
- Application name: [text box with 'WeatherJavaBeanServer']
- ☒ Create MBeans for resources
- ☐ Enable class reloading
- Reload interval in seconds: [text box]
- ☐ Deploy Web services
- Validate Input off/warn/fail: [dropdown menu with 'warn' selected]
- ☒ Process embedded configuration

At the bottom are 'Next' and 'Cancel' buttons.

Figure 18-6 Select installation options

The installation options are:

- ▶ **Pre-compile JSP**—We specify whether to precompile JSP files as a part of installation. The default is not to precompile JSP files. Also, this option should only be selected when deploying to a Version 6 server. Otherwise, the installation will be rejected.
- ▶ **Directory to install application**—We specify the directory in which the application EAR file will be installed. The default value is APP_INSTALL_ROOT/cell_name, where the APP_INSTALL_ROOT variable is the server installation root.
- ▶ **Distribute application**—We specify whether WebSphere Application Server expands or deletes application binaries in the installation destination. The default is to enable binary distribution.

- ▶ **Use Binary Configuration**—We specify whether the application server uses the binding, extensions, and deployment descriptors located with the application deployment document, the `deployment.xml` file (default), or those located in the EAR file. This option also only should be selected when deploying to a Version 6 server. Otherwise, the installation will be rejected.
- ▶ **Deploy enterprise beans**—We decide whether the **EJBDeploy** tool, which generates code required to run EJB files, runs during the application installation. If the deployed code has been generated in Rational Application Developer, this step can be skipped.
- ▶ **Application name**—We decide how to name the application. Application names must be unique within a cell and cannot contain characters that are not allowed in object names.
- ▶ **Create MBeans for Resources**—We specify whether to create MBeans for various resources (such as servlets or JSP files) within an application when the application is started. The default is to create MBeans.
- ▶ **Enable class reloading**—We specify whether to enable class reloading when application files are updated. The default is not to enable class reloading, because it results in slower program execution, because the server has to check periodically whether or not a class has changed. This normally never happens in a production environment.
- ▶ **Reload interval in seconds**—We specify the number of seconds to scan the application's file system for updated files. The default is the value of the reload interval attribute in the IBM extension (`META-INF/ibm-application-ext.xml`) file of the EAR file. This setting takes effect only if class reloading is enabled.

The reload interval specified here overrides the value specified in the IBM extensions for each Web module in the EAR file (which, in turn, overrides the reload interval specified in the IBM extensions for the application in the EAR file).
- ▶ **Deploy Web services**—If the EAR file contains Web services and has not been assembled using Application Developer or ASTK. If selected, the **wsdeploy** tool will be launched during installation. The default is to not run the **wsdeploy** tool.
- ▶ **Validate Input off/warn/fail**—We specify whether or not to examine references within the application to be validated and what to do if a reference is invalid.
- ▶ **Process embedded configuration**—If selected, the embedded configuration that is stored in the `resource.xml` and `variables.xml` files is processed. The default of this option depends on the presence of the embedded configuration. This selection will use the extended deployment information in the enterprise application deployment descriptor.

Up to four optional panels appear under specific circumstances.

Map modules to servers

In this panel, you can map the applications that are contained in the enterprise application to different servers or clusters (Figure 18-7).

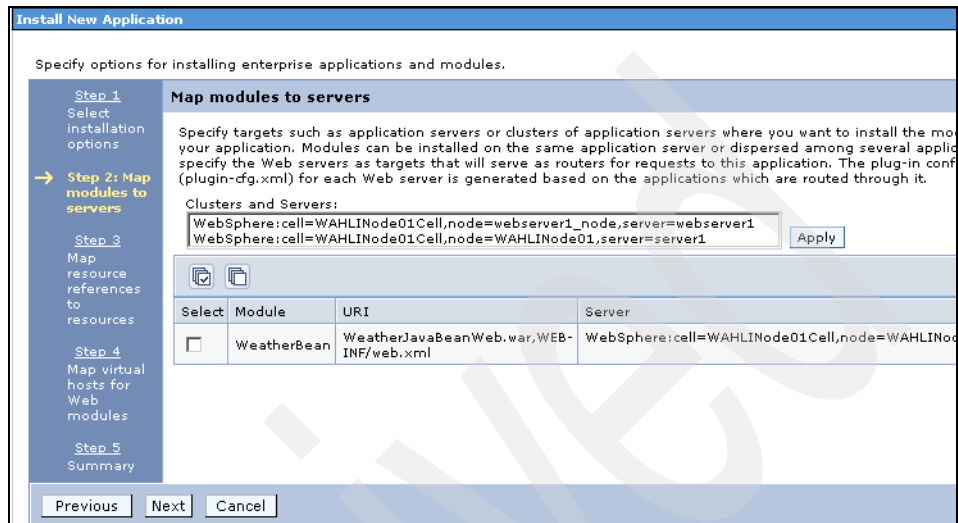


Figure 18-7 Map modules to servers

Provide options to perform EJB Deploy

In this panel, you define the options for EJB deployment. This includes parameters and options for the **RMIC** tool, database, and schema to be used for entity EJB deployment. Our application does not contain any entity EJBs. If we deploy the WeatherEJBServer application, this panel will be shown.

Provide JNDI Names for Beans

This panel appears when the application uses EJB modules. Here, we can provide JNDI names for each enterprise bean in every EJB module. Our application does not contain any EJBs. If we deploy the WeatherEJBServer application, this panel will be shown.

Map resource references to resources

This panels appears when the application contains resource references (Figure 18-8). Our application contains the WeatherDataSourceReference that points to the jdbc/weather JNDI name:

- ▶ Select the jdbc/weather JNDI name and click *Apply*.
- ▶ Select the DB2user authentication alias and click *Apply*.
- ▶ The bottom part shows the reference.

Install New Application

Messages

No check box has been selected. Select a check box.

Specify options for installing enterprise applications and modules.

Step 1

Select installation options

Step 2

Map modules to servers

Step 3: Map resource references to resources

Map resource references to resources

Each resource reference that is defined in your application must be mapped to a resource.

javax.sql.DataSource

To set multiple existing resource JNDI names:

1. Select one or more checkboxes in the table

2. Select existing resource JNDI name

3. Click Apply

Specify existing Resource JNDI name:

jdbc/weather

Apply

Specify authentication method:

none

Use default method

Select authentication data entry

DB2user

Use custom login configuration

Select application login configuration

Select...

Apply

Select

Module

EJB

URI

Reference binding

JNDI name

Login config

WeatherBean

WeatherJavaBeanWeb.war/WEB-INF/web.xml

WeatherDataSourceReference

jdbc/weather

Resource authentication per application

Figure 18-8 Map resource references to resources

Map virtual hosts for Web modules

This panel appears when the application uses Web modules (Figure 18-9). This is normally true for the applications containing Web services. Here, we select a virtual host from the list that should map to a Web module defined in the application. The port number specified in the virtual host definition is used in the URL that is used to access artifacts, such as servlets and JSP files in the Web module. Each Web module must have a virtual host to which it maps.

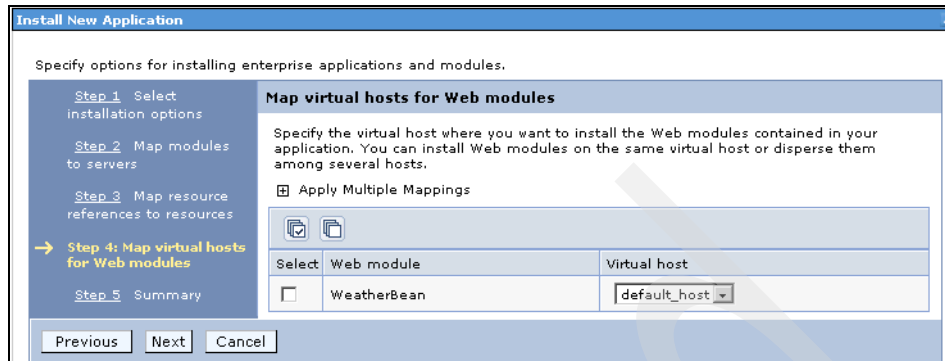


Figure 18-9 Map virtual hosts for Web modules

Summary panel

On the Summary panel, we verify that all parameters are as expected (Figure 18-10). The cell, node, and server where the application modules will be installed is not displayed on this panel, but can be verified by clicking the provided link.

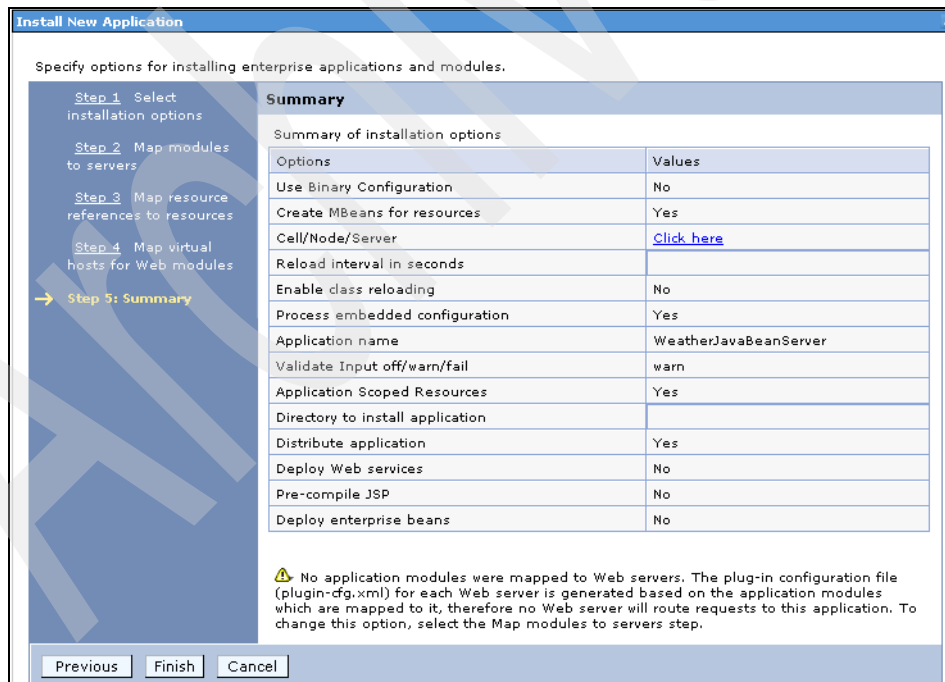


Figure 18-10 Summary

We start the deployment by clicking *Finish*. Watch the messages for a successful installation.

Save the configuration. There is no requirement to restart the server.

Regenerating the HTTP server plug-in

This step is only necessary if you are running a separate HTTP server (Web server) to serve incoming HTTP requests.

The HTTP server plug-in is an extension of an HTTP server that is forwarding incoming requests to proper application servers. Because this routing is dependent on the installed applications, the information about installed applications has to be propagated to the HTTP server plug-in.

This is done by selecting *Servers* → *Web Servers* and then selecting the HTTP server for which the plug-in configuration has to be generated.

Starting and stopping enterprise applications

After an enterprise application has been installed, it can be started and stopped from the administrative console. To do this, select *Applications* → *Enterprise Applications*. You can now select what enterprise application should be started or stopped (Figure 18-11).

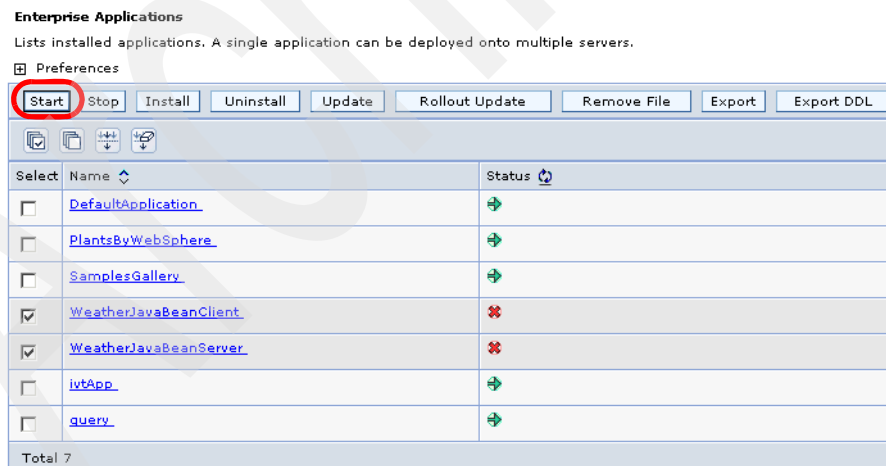


Figure 18-11 Starting enterprise applications

Web services deployment in WebSphere environment

Web services are contained in Web modules or EJB modules. When using Application Developer, the Web services are fully deployed in the modules and the containing EAR file.

These Web services are defined in the deployment descriptors of the EJB and WEB modules, for example, created as described in “Rational software development products” on page 236. While deploying these Web services into the WebSphere runtime, no special activities are required, and the EAR file can be installed like any other (non-Web service) EAR file.

An exception to this are SOAP over JMS Web services, where you have to create and configure the JMS queue and queue connection factory prior to installing the Web service.

We describe how to configure the server for SOAP over JMS in “Setting up messaging for the JMS Web service” on page 713. Although that section applies to the test environment in Application Developer, the same configuration has to be performed in a real server if you want to install the WeatherEJBJMSServer enterprise application.

Web services enabling with the SoapEarEnabler tool

In the APP_INSTALL_ROOT/bin directory, you can find a command-line tool, **soapEarEnabler.bat**, for enabling a set of SOAP services within an EAR file.

The term “enabling Web services” refers to adding HTTP routers for the Web services. Therefore, you have to use the tool when you are deploying an application that includes Web services but has no routers included.

When you use Rational Application Developer Version 6, you do not have to run this program because the Web Service wizard creates all the necessary routers for you.

Running the applications against the real server

After the enterprise applications have been installed, you can run them. For example, run the JavaBean Web service test client using this address in a browser:

```
http://<hostname>:9080/WeatherJavaBeanClientWeb/sampleWeatherJavaBeanProxy/  
TestClient.jsp
```


Running a stand-alone client against the real server

In “Stand-alone Java client” on page 289, we created a stand-alone client that accesses the Web service. We can export this application to the file system and run it against the server:

- ▶ Select the `WeatherClientStandAlone` project and click *Export*.
- ▶ Select *File system*.
- ▶ Clear the system files (`.classpath`, `.project`, `.runtime`). You can also clear all the source files (`.java`) in the packages. Select the target directory, for example:

```
\SG246461\sampcode\solution\JavaClient
```

Select *Create only selected directories*.

- ▶ Click *Finish*.

A command file named `runJavaWebSphere.bat` is provided in that directory. You might have to tailor the command file with the WebSphere installation directory:

```
set WASHOME=D:\WebSphere\AppServer60
set javabin=%WASHOME%\java\jre\bin
set javalib=%WASHOME%\java\jre\lib
set lib=%WASHOME%\lib
set channel=%WASHOME%\installedChannels
set path=%javabin%;%path%

java -Djava.ext.dirs=%LIB%;%CHANNEL% itso.client.WeatherJavaClient
pause
```

You can run this client against the test environment or against the real WebSphere Application Server. A sample output is shown here:

```
WEATHER FORECAST - STANDALONE CLIENT
=====
Getting WeatherForecast proxy ...
Using endpoint:
    http://localhost:9080/WeatherBeanWeb/services/WeatherJavaBean
Invoking getDayForecast ...
Todays weather: Weather: Wed. Dec 15, 2004 GMT, ..., 15 Celcius
Raise temperature by 5 degrees ...
Invoking setWeather ...
Invoking getDayForecast ...
Warmer weather: Weather: Wed. Dec 15, 2004 GMT, ..., 20 Celsius
End
```

Summary

In this chapter, we discussed IBM WebSphere Application Server as the basis for the deployment of enterprise applications and Web services. We looked at the general concepts and the administration facilities. We also showed how to deploy Web services to a WebSphere server.

More information

For general information about the IBM WebSphere family of products, refer to:

<http://www.software.ibm.com/websphere/>

For information about IBM WebSphere Application Server, refer to:

<http://www.ibm.com/webserver/appserv/>

The *WebSphere Application Server Information Center* is one of the most valuable and up-to-date online resources:

<http://www.ibm.com/software/webserver/appserv/infocenter.html>

The WebSphere Application Server product is also covered in detail in these publications:

- ▶ *WebSphere Application Server V6: Planning and Design*, SG24-6446
- ▶ *WebSphere Application Server V6: Security Handbook*, SG24-6316
- ▶ *WebSphere Application Server V6: System Management and Configuration Handbook*, SG24-6451
- ▶ *WebSphere Application Server V6: Performance, Scalability, and High Availability*, SG24-6392

Command-line tools, Ant, and multiprotocol binding

In this chapter, we introduce the command-line tools provided with IBM Rational Application Developer.

The command-line tools are based on open specifications for Web services, such as SOAP, WSDL, and UDD, and conform to the WS-I Organization's Basic Profile 1.0.

We also describe how to create Apache Ant build scripts for Web services operations, such as Java2WSDL and WSDL2Java.

Finally, we show an example of using a Web service with EJB binding to illustrate the multiprotocol binding support.

Command-line tools

If you prefer not to use the Web Service wizard, you can use command-line tools to create Web services using the IBM WebSphere runtime environment. Once you have created a Web service, you can then deploy it to a server, test it, and publish it as a business entity or business service.

For using command-line tools on WebSphere Application Server Version 5, refer to the IBM Redbook *WebSphere Version 5.1 Application Developer 5.1.1 Web Services Handbook*, SG24-6891.

The command-line tools provided with Application Developer V6 (in `<RAD60_HOME>/bin`) allow us to enable, deploy, and publish J2EE applications with Web services:

- ▶ **Bean2WebService**—Creates a fully deployable Web service from a Java class.
- ▶ **EJB2WebService**—Creates a fully deployable Web service from a stateless session EJB contained in an EJB module (contained in an EAR file) and optionally deploys it to the application server.
- ▶ **WSDL2WebService**—Creates a fully deployable Web service from one or more WSDL documents and optionally deploys it to the application server.
- ▶ **WSDL2Client**—Generates fully-deployable Web service clients from one or more WSDL documents and optionally deploys them to the application server.
- ▶ **UDDIPublish**—Publishes a business entity or a business service to a public or a private UDDI registry.
- ▶ **UDDIUnpublish**—Removes a business entity or a business service from a public or a private UDDI registry.

All the above commands are also supplied as a J2EE 1.3 command with the suffix 13. Documentation for the command-line tools can be found in the Application Developer online help by selecting *Developing Web services* → *Creating Web services* → *Creating Web service with command line tools*.

Command-line tools are also provided in the built-in test environment (in `<RAD60_HOME>/runtimes/base_v6/bin`), for example:

- ▶ **WSDL2Java**—Creates client artifacts from a WSDL file.
- ▶ **Java2WSDL**—Creates a WSDL file from a JavaBean.

These commands are used by the Application Developer command-line tools.

Using the command-line tools

In this section, we describe the development tools provided with Application Developer Version 6. For the command-line example, we create the directory `C:\SG246461\commandtest`.

Using the Web services-related tool, we create some of the same Web services as shown in Chapter 16, “Develop Web services with Application Developer V6.0” on page 273.

Bean2WebService

The Bean2WebService tool generates a fully deployable Web service from a Java class.

Generation steps explained

The automatic process followed by the command-line tool to generate the Web services is as follows:

- ▶ Step 1—Take the Java bean that will be available as a Web service.
- ▶ Step 2—Create the SEI Java code from the Java bean with the methods that will be exposed in the Web service.
- ▶ Step 3—Generate the WSDL from the SEI with the Java2WSDL tool.
- ▶ Step 4—Generate the Web service deployment descriptor templates from the WSDL with the WSDL2Java tool.
- ▶ Step 5—Configure the Web service deployment descriptor. Modify the `<servlet-link>` tag in the `webservices.xml` file with the value of the `<servlet-name>` tag in the `web.xml` file.
- ▶ Step 6—Assemble all the files previously generated in a WAR file:
 - Add the SEI file to the WAR file.
 - Add the WSDL to the WAR file in the `WEB-INF` directory.
 - Add the Web services and IBM binding deployment descriptors to the `WEB-INF` directory.
- ▶ Step 7—Assemble the WAR file in an enterprise application (EAR) file.

Note that automatic deployment to a server is not supported.

Sample setup

We use a special version of the weather forecast application as the sample code. The only difference is that this version does not require a database, because it will always use the `WeatherPredictor` class. In addition, all helper classes are in one package, `itso.objects`.

The application is provided in `\SG246461\sampcode\commandline\bean`. To run the command, we suggest that you use another directory, `\SG246461\commandtest\bean`. Copy all the code to the `commandtest` directory.

Example 19-1 shows the `WeatherJavaBean` class.

Example 19-1 WeatherJavaBean example for Bean2WebService

```
package itso.bean;

import itso.objects.Weather;
import itso.objects.WeatherForecast;
import java.util.Calendar;

public class WeatherJavaBean {

    public Weather getDayForecast(Calendar theDate) throws Exception {
        return new WeatherForecast().getDayForecast(theDate);
    }
    public Weather[] getForecast(Calendar startDate, int days)
                                throws Exception {
        return new WeatherForecast().getForecast(startDate, days);
    }
    public int[] getTemperatures(Calendar startDate, int days)
                                throws Exception {
        return new WeatherForecast().getTemperatures(startDate, days);
    }

    public static void main(String[] args) {
        WeatherJavaBean service = new WeatherJavaBean();
        try {
            System.out.println("Todays forecast: "
                + service.getDayForecast(Calendar.getInstance()));
            System.out.println("Tomorrows temperature: "
                +service.getForecast(null,1)[1].getTemperatureCelsius());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

To verify and prepare the service example class, open a command window, set up the environment, compile the class, and run the program using the following command sequence:

```
rem see command file compileRun.bat
cd \SG246461\commandtest\bean
set RADJAVA=C:\<RAD60_HOME>\java\bin
set PATH=%RADJAVA%;%PATH%
javac itso\bean\*.java
javac itso\objects\*.java

java itso.bean.WeatherJavaBean
```

Replace <RAD60_HOME> with the name of the directory where Application Developer is installed. The output from program is shown here:

```
Todays forecast: Weather: Wed. Dec 15, 2004 PST, rainy, wind: E at 19km/h ,
                  temperature: 10 Celsius

Tomorrows temperature: -1
```

Bean2WebService at work

To experiment with the Bean2WebService tool, we run the tool on the simplified WeatherJavaBean class:

```
rem see command file RunBean2WS.bat
set RADHOME=C:\<RAD60_HOME>\bin

set PATH=%RADHOME%;%PATH%

call setupenv.bat

call Bean2WebService.bat -verbose -cp . -project WeatherService
                        -genMain WeatherServiceClient -clientType J2SE
                        -host localhost:9080
                        itso.bean.WeatherJavaBean
```

Note: The default value for host is localhost:6080, so we change it to localhost:9080 to be able to test it under Application Developer.

The endpoint URL is formed by the following format:

```
http://HostName:PortNumber/<ContextRoot>/services/<PortName>
http://localhost:9080/WeatherService/services/WeatherService
```

Example 19-2 shows the command window output in condensed format, where ~ stands for C:\SG246461\commandtest\bean (the current directory).

Creating new project: WeatherService ...

Removed all existing classes from directories under
~\WeatherService\WEB-INF\classes\

Generating the service endpoint interface...

Generating WSDL:

WSWS3477I: Binding-specific properties are {MIMEStyle=WSDL11, use=literal,
debug=false, style=document, bindingName=WeatherJavaBean,
encodingStyle=http://schemas.xmlsoap.org/soap/encoding/, verbose=true,
wrapped=true, portTypeName=WeatherJavaBean,
servicePortName=WeatherJavaBean, intfNS=http://bean.itso,
location=http://localhost:9080/WeatherService/services/WeatherJavaBean,
soapAction=DEFAULT}
WSWS3010I: Info: Generating portType {http://bean.itso}WeatherJavaBean
WSWS3010I: Info: Generating message {http://bean.itso}getForecastRequest
WSWS3010I: Info: Generating message {http://bean.itso}getForecastResponse
WSWS3010I: Info: Generating type/element {http://bean.itso}ArrayOf_tns1_Weather
WSWS3010I: Info: Generating type/element {http://objects.client.itso}Weather
WSWS3010I: Info: Generating message {http://bean.itso}getDayForecastRequest
WSWS3010I: Info: Generating message {http://bean.itso}getDayForecastResponse
WSWS3010I: Info: Generating message {http://bean.itso}getTemperaturesRequest
WSWS3010I: Info: Generating message {http://bean.itso}getTemperaturesResponse
WSWS3010I: Info: Generating type/element {http://bean.itso}ArrayOf_xsd_int

WSWS3010I: Info: Generating binding

{http://bean.itso}WeatherJavaBeanSoapBinding

WSWS3010I: Info: Generating service {http://bean.itso}WeatherJavaBeanService

WSWS3010I: Info: Generating port WeatherJavaBean

Generating server side files:

WSWS3185I: Info: Parsing XML file:
~\WeatherService\WEB-INF\wsdl\WeatherJavaBean.wsdl
Retrieving document at 'file:/~/WeatherService/
WEB-INF/wsdl/WeatherJavaBean.wsdl'.
WSWS3282I: Info: Generating file://~/WeatherService\itso\objects\Weather.java.
WSWS3282I: Info: Generating itso\objects\Weather_Helper.java.
WSWS3282I: Info: Generating itso\objects\Weather_Ser.java.
WSWS3282I: Info: Generating itso\objects\Weather_Deser.java.
WSWS3282I: Info: Generating itso\bean\WeatherJavaBean.java.
WSWS3282I: Info: Generating bean\WeatherJavaBeanSoapBindingImpl.java.
WSWS3282I: Info: Generating file://~/WeatherService\WEB-INF\webservices.xml.
WSWS3282I: Info: Generating WEB-INF\ibm-webservices-bnd.xmi.
WSWS3282I: Info: Generating WEB-INF\ibm-webservices-ext.xmi.
WSWS3282I: Info: Generating WEB-INF\WeatherJavaBean_mapping.xml.
Configuring webservices.xml...

Added web module with context root: WeatherService

**Web Service archive "file:~/WeatherService/WeatherServiceEAR.ear"
has been successfully generated.**

Generating client side files:

```
WSWS3185I: Info: Parsing XML file:
           ~\WeatherService\WEB-INF\wsdl\WeatherJavaBean.wsdl
Retrieving document at 'file:~/WeatherService/
                      WEB-INF/wsdl/WeatherJavaBean.wsdl'.
WSWS3282I: Info: Generating
           file://~\WeatherService\client-side\itso\objects\Weather.java.
WSWS3282I: Info: Generating ..... \objects\Weather_Helper.java.
WSWS3282I: Info: Generating ..... \objects\Weather_Ser.java.
WSWS3282I: Info: Generating ..... \objects\Weather_Deser.java.
WSWS3282I: Info: Generating ..... \bean\WeatherJavaBean.java.
WSWS3282I: Info: Generating ..... \bean\WeatherJavaBeanSoapBindingStub.java.
WSWS3282I: Info: Generating ..... \bean\WeatherJavaBeanService.java.
WSWS3282I: Info: Generating ..... \bean\WeatherJavaBeanServiceLocator.java.
WSWS3282I: Info: Generating ..... \bean\WeatherJavaBeanServiceInformation.java.
```

Creating client-side build script...

Creating main class...

All done.

The main output of the Bean2WebService tool is the <ProjectName>.ear file in the root of the generated directory structure named <ProjectName>. The generated EAR file can be deployed on a server and made available to others.

Here, we describe the generated project structure:

- ▶ The EAR file contains a fully deployable Web service in a Web module.
- ▶ The WEB-INF directory holds the server-side Web project information:
 - Web deployment descriptor (web.xml)
 - Web services deployment descriptor (webservices.xml)
 - Java classes
 - WSDL file
- ▶ Because we did not specify the -server-side-only option, a client-side directory is generated with:
 - Proxy classes (in itso.bean)
 - Helper classes (in itso.client.objects)
 - WSDL file in META-INF
 - Deployment descriptor (applicationclient.xml) for J2EE clients
 - Main class (WeatherServiceClient in our case)
 - A command file to compile the client (buildclient.bat)

- A command file to run the client (`runclient.bat`)

You can execute the `buildclient.bat` file to compile the client classes.

Look into the generated files, especially the `WeatherJavaBean.wsdl` and the `WeatherJavaBean_SEI.java` (service endpoint interface) files. Notice that by default they include the public (and non-static) methods of the `WeatherJavaBean.java` class.

Command-line help and options

The `Bean2WebService` command has many useful options:

- ▶ `-methods <method1 method2... methodN>`—Enables you to define the subset of the public methods that should be exposed as Web services.
- ▶ `-server-side-only`—Do not generate client-side files.
- ▶ `-genMain <class>` and `-clientType <J2SE|Application|Servlet|EJB>`—Generate a main client class and implementation template.
- ▶ `-style <rpc|doc|wrapped>`—SOAP style: wrapped (default) is a special document style.
- ▶ `-use <literal|encoded>`—Encoding (literal by default).
- ▶ `-splitWsd1`—Generate separate interface and binding files.
- ▶ `-wsSecDir <directory>`—Directory for deployment descriptors configured for security.

For a full list of options and their use, refer to the online help in Application Developer.

Deploying a Web service to a server

After you create an EAR file using the `Bean2WebService` command-line tool, you can deploy it to a WebSphere server. You can import the EAR file into Application Developer and add it to the built-in server, or you can use the WebSphere Application Server administrative console to deploy the EAR file to a real server.

Deploying the generated EAR file in Application Developer

To import the EAR file, perform the following steps:

- ▶ Go to *File* → *Import* → *EAR file*, and click *Next*.
- ▶ Click *Browse* to select the command-line test directory and EAR file
`C:\SG246461\commandtest\bean\WeatherService\WeatherServiceEAR.ear`.
Click *Open*.

- ▶ In the Import EAR wizard, target the EAR to the WebSphere Application Server V6.0 server. Accept the default settings in the next two steps. Note the name of the Web project created with the EAR file, WeatherService.
- ▶ Click *Finish*.

To deploy the imported EAR file into the server, perform the following steps:

- ▶ In the Servers view, select the WebSphere Application Server v6.0 server and click *Add and Remove projects*.
- ▶ From the Available projects list, select the WeatherServiceEAR file that you imported. Click *Add* to add it to the list of configured projects.
- ▶ Click *Finish*.

The EAR file is now deployed. You can test it using the Web Services Explorer, as described in “Web Services Explorer” on page 339.

Deploying the generated EAR file in WebSphere

Follow the instructions in “Enterprise application deployment” on page 382 to install the EAR file into a server.

Running the generated client

The Bean2WebService command generated the WeatherServiceClient skeleton. Complete the sample code as shown in Example 19-3 (see \SG246461\commandtest\bean\client for the complete code).

Example 19-3 Generated stand-alone client

```
package itso.bean;
import itso.objects.Weather;
import java.util.Calendar;

public class WeatherServiceClient {
    public static void main(String[] args) {
        try {
            WeatherJavaBeanServiceLocator loc =
                new WeatherJavaBeanServiceLocator();
            // Call to the service locator's get<PortName>() method ...
            WeatherJavaBean port = null; // loc.get<PortName>();
            port = loc.getWeatherJavaBean();
            // Make calls to methods of the <PortName>PortType to ...
            Weather w = port.getDayForecast( Calendar.getInstance() );
            System.out.println("Weather: "+w.getCondition()+" "
                               +w.getTemperatureCelsius()+" degrees");
        } catch (java.lang.Exception e){
            e.printStackTrace();
        }
    }
}
```

```
}  
}  
}
```

To run the client against a deployed service, execute the two generated commands:

```
buildclient.bat  
runclient.bat
```

EJB2WebService

The EJB2WebService tool generates a fully deployable Web service from a stateless session EJB contained in an EAR file. Note that only Version 2.0 of the EJB architecture is supported when using the EJB2WebService13 command.

To use EJB2WebService, open a command window and enter:

```
EJB2WebService [<optional arguments>] -project <ProjectName>  
-ri <RemoteInterface> <EJB.ear>
```

The main output of this tool is a modified version of the original EAR file, which contains a fully deployable Web service, and optionally, a client-side directory is generated with the extra files for client development.

Here, we describe the automatic process followed by the command tool to generate the Web service:

- ▶ Step 1—Take the stateless session bean that will be available as a Web service.
- ▶ Step 2—Create the SEI Java code from the EJB with the methods that will be exposed in the Web service.
- ▶ Step 3—Generate the WSDL from the SEI with the Java2WSDL tool.
- ▶ Step 4—Generate the Web service deployment descriptor templates from the WSDL with the WSDL2Java tool.
- ▶ Step 5—Configure the Web service deployment descriptor. Modify the `<ejb-link>` tag in the `webservices.xml` file with the value of the `<ejb-name>` tag in the `ejb-jar.xml` file.
- ▶ Step 6—Assemble all the files previously generated in an EJB JAR:
 - Add the SEI file to the EJB JAR.
 - Add the WSDL to the EJB JAR in the META-INF directory.
 - Add the Web services and IBM binding deployment descriptors to the META-INF directory.

- ▶ Step 7—Assemble the EJB JAR in an enterprise application (EAR).
- ▶ Step 8—Create the Web service endpoint Web module WAR using the `endptEnabler` command.

We do not provide an example. You can take the `WeatherEJB.jar` file from `\SG246461\sampcode\_setup\EARbase` to explore the `EJB2WebService` command.

WSDL2WebService

The `WSDL2WebService` tool generates, in three stages, fully deployable Web services from one or more WSDL documents.

Generation stages explained

Here, we describe the automatic process followed by the command tool to generate the Web service:

- ▶ Stage 1: Step 1—Generate the Web service deployment descriptor templates and the implementation files from the WSDL using the `WSDL2Java` tool.
- ▶ Stage 2: Step 2—Complete the implementation of the bean.
- ▶ Stage 3:
 - Step 3—Configure the Web service deployment descriptor. Modify the `<servlet-link>` tag in the `webservices.xml` file with the value of the `<servlet-name>` tag in the `web.xml` file.
 - Step 4—Assemble all the generated files in a WAR:
 - Java bean as a servlet in the `web.xml` file
 - SEI class
 - WSDL in the `WEB-INF` directory
 - Web services IBM binding deployment descriptor into `META-INF`
 - Step 5—Assemble the WAR into an enterprise application (EAR).

Sample setup

To see the tool at work, we use the WSDL file generated for the `WeatherJavaBean` sample in “`Bean2WebService`” on page 397:

- ▶ Create a `wsdl` directory in the `SG246461\commandtest` directory and copy the content from `\SG246461\sampcode\commandline\wsdl`.
- ▶ Copy the `WeatherjavaBean.wsdl` file into the directory. Take the file from the previous example, or use the file copied from:
`\SG246461\sampcode\commandline\wsdl`

WSDL2WebService at work

The three stages reviewed:

- ▶ Stage 1—Run the tool with the `-createService` argument to create skeleton Java implementation templates for a Web service described by a particular WSDL document.
- ▶ Stage 2—Write the implementation code in the templates and compile it using the compile script generated in stage 1.
- ▶ Stage 3—Run the tool again with the `-createEar` argument to build a Web service-enabled archive from this implementation and deploy it to the application server.

Note that we can perform stage 1 several times to create related Web services in the same project directory. In stage 3, we can then create a separate module for each of these Web services and add them to the same EAR file.

Stage 1—Creating a Web service implementation skeleton

To create a skeleton implementation, open a command window and enter:

```
rem see command file RunWSDL2WS1.bat
cd \SG246461\commandline\wsdl
set RADHOME=C:\RAD60\bin
set PATH=%RADHOME%;%PATH%
call setupenv.bat
call WSDL2WebService.bat -verbose -createService WeatherService2
                        -type Bean -server-side-only -project . WeatherJavaBean.wsdl
```

After running the tool with the `-createService <ServiceName>` argument, a directory named `<ServiceName>` containing several subdirectories is created under the specified project directory. These subdirectories contain all the necessary Java templates and deployment descriptors that are required to build the Web service implementation or implementations. A build script called `compile.bat` (Microsoft Windows®) or `compile.sh` (Linux) is also generated to compile the server-side code.

- ▶ `WEB-INF`—Server-side files with deployment descriptor files and WSDL.
- ▶ `DefaultNamespace`—Interface (`WeatherJavaBeanService.java`) and implementation skeleton file (`WeatherJavaBeanServiceSoapBindingImpl.java`). In this example, `itso.bean` is used.
- ▶ `client-side`—Optional, identical to `Bean2WebService` command output.

Stage 2—Providing and compiling the implementation code

Provide the implementation code by editing the skeleton file:

```
<Project>\<Service>\<Namespace>\<Service>SoapBindingImpl.java
\SG246461\commandtest\wsdl\WeatherService\itso\bean\
WeatherJavaBeanSoapBindingImpl.java
```

Replace the return statements as shown in Example 19-4 (we provide the code in \SG246461\sampcode\commandline\wsdl).

Example 19-4 Skeleton implementation

```
package itso.bean;

import itso.objects.Weather;
import java.util.Calendar;

public class WeatherJavaBeanSoapBindingImpl
    implements itso.bean.WeatherJavaBean{
    public itso.objects.Weather[] getForecast(java.util.Calendar arg_0_0,
        int arg_1_0) throws java.rmi.RemoteException {
        Weather[] wa = new Weather[arg_1_0 + 1];
        Calendar date = (Calendar)arg_0_0.clone();
        for (int i=0; i <= arg_1_0; i++) {
            wa[i] = getDayForecast(arg_0_0);
            wa[i].setDate( (Calendar)date.clone() );
            wa[i].setTemperatureCelsius(i+5);
            date.add(Calendar.DAY_OF_MONTH, 1);
        }
        return wa;
    }
    public itso.objects.Weather getDayForecast(java.util.Calendar arg_0_1)
        throws java.rmi.RemoteException {
        Weather w = new Weather();
        w.setDate(arg_0_1);
        w.setCondition("sunny");
        w.setTemperatureCelsius(18);
        w.setWindDirection("S");
        w.setWindSpeed(10);
        return w;
    }
    public int[] getTemperatures(java.util.Calendar arg_0_2, int arg_1_2)
        throws java.rmi.RemoteException {
        int[] t = new int[arg_1_2 + 1];
        for (int i=0; i <= arg_1_2; i++) t[i] = i+10;
        return t;
    }
}
```

If the implementation code has dependencies, such as .jar files or directories containing .class files, edit the compile.bat script, and add the full path names of these dependencies to the USER_CLASSPATH variable.

To compile the code, run the compile.bat file. To see any compile errors, edit the command file and specify @echo on.

Stage 3—Creating a Web services-enabled archive

To create a Web services-enabled archive:

```
rem see command file RunWSDL2WS2.bat
cd \SG246461\commandline\wsdl
set RADHOME=C:\RAD60\bin
set PATH=%RADHOME%;%PATH%
call setupenv.bat
call WSDL2WebService.bat -verbose -createEar WeatherService2EAR.ear
                             -project .
```

The output from running this command is the WeatherService2EAR.ear file. The EAR file can be updated with additional services by using the `-add` option.

By specifying `-type Bean` in the stage 1 command, we generated a JavaBean implementation template. By using `-type EJB`, we would generate an EJB implementation template. Other command-line options are similar to the Bean2WebService command.

The generated EAR file can now be deployed on a server, as shown in “Deploying a Web service to a server” on page 402.

WSDL2Client

The WSDL2Client tool generates, in four stages, fully deployable Web service clients from one or more WSDL documents.

Sample setup

To see the tool at work, we use the generated WSDL file from the Bean2WebService command:

- ▶ Create a `wsdlclient` directory in the `SG246461\commandtest` directory and copy the content from `\SG246461\sampcode\commandline\wsdlclient`.
- ▶ Copy the `WeatherjavaBean.wsdl` file into the directory. Take the file from the previous example, or use the file copied from:
`\SG246461\sampcode\commandline\wsdlclient`

WSDL2Client at work

The four stages reviewed:

- ▶ Stage 1—Run the tool with the `-project` argument to create a client skeleton implementation for a Web service that is described by a particular WSDL document.
- ▶ Stage 2—Write the implementation code in the templates and compile it using the build script that was generated in stage 1.

- ▶ Stage 3—If the `-clientType J2SE` and `-genMain` options were specified in stage 1, run the client by using the run script generated in stage 1.
- ▶ Stage 4—If `-clientType application, ejb, or servlet` is specified in stage 1, run the tool again with the `-createEar` argument to build a Web service-enabled client archive (J2EE client) from this implementation.

Stage 1—Creating a skeleton client implementation

To create a client implementation skeleton, open a command window and enter:

```
rem see command file RunWSDL2Client1.bat
cd \SG246461\commandline\wsdlclient
set RADHOME=C:\RAD60\bin
set PATH=%RADHOME%;%PATH%
call setupenv.bat
call WSDL2Client.bat -verbose -project WeatherService3
                    -clientType servlet -genMain WeatherServiceServlet
                    WeatherJavaBean.wsdl
```

The output directory (`\j2se\client-side\DefaultNamespace`) contains all the necessary Java templates, including serializer and deserializer classes for complex types, and deployment descriptors that are required to build the Web service client implementations.

A build script called `buildclient.bat` is also generated to help us compile all the code.

In the `Bean2WebService` example, we generated a J2SE client. This time, we generate a `servlet` for a Web client. You can rerun the `WSDL2Client` command to generate multiple types of clients.

Stage 2—Writing and compiling the implementation code

The generated client program skeleton, `WeatherServiceServlet`, must be edited to add the Web service call (Example 19-5, modifications in bold).

Example 19-5 Implementing the J2EE servlet client from WSDL2Client

```
package itso.bean;
import javax.servlet.http.HttpServlet;
.....
import itso.objects.Weather;
import java.util.Calendar;

public class WeatherServiceServlet extends HttpServlet {

    public void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, java.io.IOException {
try {
    // Use JNDI to locate the Web Service
```

```

Context ctx = new InitialContext();
String webService = "java:comp/env/service/WeatherJavaBeanService";
WeatherJavaBeanService service =
    (WeatherJavaBeanService) ctx.lookup(webService);
// Call to the service's get<PortName>() method .....
WeatherJavaBean port = null;
port = service.getWeatherJavaBean();

// Make calls to methods of the Service Endpoint Interface (SEI) ...
Weather w = port.getDayForecast( Calendar.getInstance() );
PrintWriter out = resp.getWriter();
out.println("<HTML><HEAD></HEAD><BODY>");
out.println("<h2>Weather Java Bean Client</h2>");
out.println("<h3>Day Forecast</h3>");
out.println("Weather: "+w.getCondition()+" "
            +w.getTemperatureCelsius()+" degrees");
out.println("<h3>End</h3>");
out.println("</BODY></HTML>");
} catch (java.lang.Exception e){
    e.printStackTrace();
}
}

public void doGet(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, java.io.IOException {
    doPost(req, resp);
}
}

```

If our implementation code has dependencies, such as a .jar file or a directory containing .class files, edit the buildclient.bat script, and add the full path names of these dependencies to the USER_CLASSPATH variable.

To compile the code, run the buildclient.bat command file. To see any compile errors, edit the command file and specify @echo on.

Stage 3—Running the client application

To run a J2SE client, we have to install the enterprise application that contains the Web service. Refer to “Deploying a Web service to a server” on page 402 for instructions to install and deploy the EAR file. Note that we did not generate a J2SE client in this example.

Stage 4—Creating a J2EE client application

To create the Web service-enabled Web archive containing the servlet, run the WSDL2Client command with the -createEAR option:

```

rem see command file RunWSDL2Client2.bat
cd \SG246461\commandline\wsdlclient
set RADHOME=C:\RAD60\bin

```

```

set PATH=%RADHOME%;%PATH%
call setupenv.bat
call WSDL2Client.bat -verbose -createEar WeatherService3EAR.ear
                        -clientType servlet -main itso.bean.WeatherServiceServlet
                        .\WeatherService3

```

The output will be either a new or updated EAR file. The `-main` parameter is required for `-clientType` `application` or `servlet`. Notice that the client-side classes are moved to the `WEB-INF\classes` directory. With other client types, the classes would be moved to `META-INF\classes`.

Running the J2EE servlet

To run the servlet, you have to import the `WeatherService3EAR.ear` file. Add the application to the server and make sure that the `WeatherServiceEAR` application is also running.

To execute the servlet, select the servlet in the deployment descriptor and *Run on Server*, or enter this URL into a browser:

```

http://localhost:9080/WeatherService3EAR/servlet-client/WeatherServiceServlet
http://localhost:9080/WeatherService3/.... <=== 6.0.1

```

The output of the servlet looks like this:

```

Weather Java Bean Client
Day Forecast
Weather: cloudy -4 degrees
End

```

UDDIPublish/UDDIUnpublish

The values used to publish a business and service to a UDDI registry are taken from the example walkthrough in Chapter 23, “Implementing a private UDDI registry” on page 619.

If `$RAD60` is pointing to the installation directory, the tools can be found in the `$RAD60/bin` directory.¹

To publish a business, issue the following command from a shell:

```

$RAD60/bin/UDDIPublish.sh -business -businessName "Weather Information"
                        -publishUrl "http://127.0.0.1:9080/uddisoap/publishapi" -inquiryUrl
                        "http://127.0.0.1:9080/uddisoap/inquiryapi" -username "UNAUTHENTICATED"2

```

¹ The `$VARIABLE` notion refers to a Linux installation of Application Developer. In Windows, you should use `C:%<RAD60_HOME>%` instead.

² The user name `UNAUTHENTICATED` refers to the user the UDDI GUI using. Publishing businesses and services with this user name ensures that you can browse the registry using the UDDI GUI.

Of course, you have to replace the parameters for the `businessName`, `publishUrl`, and `inquiryUrl` options with the values you want to use.

After the business has been created, you can add any number of services to the registry using the following command:

```
$RAD60/bin/UDDIPublish.sh -service -servicenme "Weather Forecast"
-businessName "Weather Information" -wsdlLocation
"http://localhost:9080/WeatherEJBRouter/services/WeatherEJB/wsdl"
-accessPoint "http://localhost:9080/WeatherEJBRouter/services/WeatherEJB"
-publishUrl "http://127.0.0.1:9080/uddisoap/publishapi" -inquiryUrl
"http://127.0.0.1:9080/uddisoap/inquiryapi" -username "UNAUTHENTICATED"
```

Again, the `businessName`, `serviceName`, `wsdlLocation`, `accessPoint`, `publishUrl`, and `inquiryUrl` have to be tailored to your special needs.

Additional options can be added to the command. You will receive a full list of the options when you invoke the command with the `-help` option, as in:

```
$RAD60/bin/UDDIPublish.sh -help
```

There is also an `unpublish` command, `UDDIUnpublish.sh`, that can be used to remove businesses and services from a UDDI registry.

WebSphere Web services Ant tasks

In this section, we discuss the WebSphere Ant tasks, which can be used to build an automated Web services generation process. WebSphere Application Server ships two Ant tasks to create Web services. The Ant tasks use the same code base as the `WSDL2Java` and `Java2WSDL` command-line tools. Therefore, the tasks work and generate exactly the same artifacts as the command-line tools. In comparison to the command-line tools, these tasks can be used to build an Ant-based, automated, Web service generation process.

The command-line tools are not WebSphere specific, and do generate platform-independent Web services; this is also true for the Ant tasks. Due to this, the generated Web service artifacts do not comply with the Application Developer project structure.

The Web services Ant tasks are packed in the following Java archive:

```
<WAS_HOME>\lib\wswsanttasks.jar
```

This Java archive also contains WebSphere-specific Ant tasks.

Creating Web services Ant tasks

This section describes how to set up the environment, implement, and run the Web services Ant tasks. We create a sample script for a Web service build process. The purpose of the provided script is to generate the Web service WSDL file and create the Web service server and the client parts.

Preparation

Create an enterprise application named `WeatherServiceAntEAR`, with two Web applications, `WeatherServiceAnt` and `WeatherServiceAntClient`.

Import the simple implementation beans used in the `Bean2WebService` example into the project (packages `itso.bean` and `itso.objects` from `commandtest\bean`). Also, import the WSDL file under `WebContent\WEB-INF\wsdl`, and rename it `WeatherServiceAnt.wsdl`.

Creating the Ant script file

There is no wizard to create an Ant script file. To create a new Ant script, we create a simple file called `build.xml` by selecting *File* → *New* → *Other* → *Simple* → *File*:

- ▶ In the *New File* window, select the `WeatherServiceAnt` project for which to build the Web service, and enter `build.xml` as the name. The Ant editor in Application Developer only recognizes build scripts if they are named `build.xml`.

Using the project root directory for the build file makes it easy to locate the file. However, the build file can be created in any project and folder structure.

- ▶ Click *Finish* to close the wizard. After the wizard creates the file, the Ant editor opens the file.

Implementing the Ant script

To help you get started with a new Ant script, use template for an Ant build file provided by the Ant editor. To apply the template, use the editor content assist function (Ctrl-Space), and select *Buildfile template*. Figure 19-1 shows the auto-completion for the build file template.

Tip: The Ant editor supports the content assist function, which exposes all available Ant variables and tasks. This function is also available for self-defined tasks.

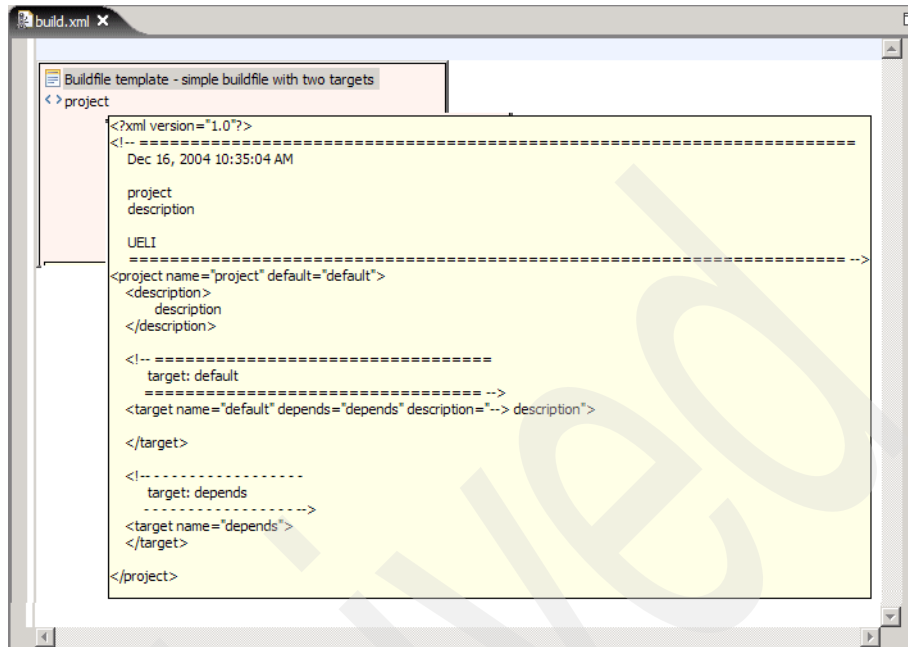


Figure 19-1 Ant build file template

The complete Ant build file is provided in \SG246461\sampcode\ant. We explain the important parts of the script in the following sections.

Properties

The properties section defines variables used later in the script (Example 19-6). You might have to change the was.install.root property for your system.

Example 19-6 Ant build script: Project and properties

```
project name="WeatherServiceAnt" default="all">
  <description>
    Using Ant to create Web service artifacts
  </description>

  <!-- START: Property definition section -->
  <property name="was.install.root" value="C:/RAD60/runtimes/base_v6"/>
  <property name="wsAntTaskJar" value="{was.install.root}/lib/wsanttasks.jar"/>
  <property name="wsWebServJar" value="{was.install.root}/lib/webservices.jar"/>
  <property name="j2eeJar" value="{was.install.root}/lib/j2ee.jar"/>
  <property name="xmlJar" value="{was.install.root}/java/jre/lib/xml.jar"/>

  <!--Define the Web service source class -->
  <property name="webServiceClassName" value="itso.bean.WeatherJavaBean" />
```

```

<!--Define the Web service SERVER project -->
<property name="wsServerProjectName" value="WeatherServiceAnt" />
<!--Define the Web service server context root -->
<property name="wsContextRoot" value="WeatherServiceAnt" />
<!--Define the classes needed for the source bean -->
<property name="classes" value=".\\WebContent\\WEB-INF\\classes;" />
<!--Define the Web service CLIENT project -->
<property name="wsClientProjectName" value="WeatherServiceAntClient" />
<!-- END: Property definition section-->

```

Tasks

The definitions for the WebSphere commands are implemented as tasks. The tasks use the JAR files defined in the properties section for the class path (Example 19-7).

Example 19-7 Ant build script: Tasks

```

<taskdef name="Java2WSDL" classpath="{wsAntTaskJar};{wsWebServJar}"
        classname="com.ibm.websphere.ant.tasks.Java2WSDL">
    </taskdef>
<taskdef name="WSDL2Java" classpath="{wsAntTaskJar};{wsWebServJar}"
        classname="com.ibm.websphere.ant.tasks.WSDL2Java">
    </taskdef>

```

For the task creation, the Ant task class name, the class path to this class, and a name for the task must be defined. The class name for the tasks can be found in the `wsanttasks.jar` file.

Targets

The targets are what Ant executes. Targets can depend on each other and, therefore, execute other targets before executing the own target.

CreateWSDL target

The `createWSDL` target generates the WSDL file from a JavaBean (Example 19-8).

The parameters for the `Java2WSDL` task are exactly the same as for the `Java2WSDL` command-line tool. For a detailed description of the parameters, refer to the *WebSphere Application Server Information Center*.

Example 19-8 Ant build script: createWSDL target

```

<target name="createWSDL"
        description="Creates a WSDL file for the given JavaBean class-Java2WSDL">
    <echo>Starting: Java2WSDL for {webServiceClassName}</echo>
    <Java2WSDL

```

```

        output=".\\WebContent\\WEB-INF\\wsdl\\${wsServerProjectName}.wsdl"
        location="http://localhost:9080/${wsContextRoot}/service/
                                ${wsServerProjectName}Service"
        classname="${webServiceClassName}"
        classpath="${classes};${wsWebServJar};${xmlJar};${j2eeJar}"
        verbose="true"
        methods="getForecast getDayForecast getTemperatures">
</Java2WSDL>
<echo>Done: Java2WSDL for ${webServiceClassName}</echo>
</target>

```

This script creates the WSDL file from the JavaBean (classname parameter) and stores it at the output parameter location. The WeatherJavaBean methods listed in the methods parameter will be exposed as operations in the WSDL file.

Note: The three JAR files in the class path might not be necessary if the JAVA_HOME environment variable points to <RAD60_HOME>/eclipse/jre.

GenerateServer target

The generateServer target generates the Web service in the server project. The script uses the WSDL2Java task (Example 19-9).

Example 19-9 Ant build script: generateServer target

```

<target name="generateServer"
    description="Create the Server artefacts for the defined WSDL file">
    <echo>Starting: WSDL2Java, create SERVER for ${webServiceClassName}</echo>
    <WSDL2Java
        url=".\\WebContent\\WEB-INF\\wsdl\\${wsServerProjectName}.wsdl"
        container="web"
        role="server"
        output=".\\WebContent"
        genjava="overwrite"
        genxml="overwrite">
    </WSDL2Java>
    <!-- move the java classes to the source directory -->
    <move overwrite="true" todir=".\\JavaSource">
        <fileset dir=".\\WebContent" includes="itso\\*" > </fileset>
    </move>
    <echo>Done: WSDL2Java, create SERVER for ${webServiceClassName}</echo>
</target>

```

This Ant task is based on the WSDL2Java tooling, and a detailed description of the parameters is available in the *WebSphere Application Server Information Center*.

As already mentioned, the Web services Ant tasks do not comply with the Application Developer project structure. Therefore, we use the move task to put the generated Java classes into the JavaSource directory.

GenerateClient target

Example 19-10 shows how we implemented the Ant task to generate the client code. As for the server part, we have to move the Java classes to the JavaSource directory. We also copy the WSDL file to the client.

Example 19-10 Ant build script: generateClient target

```
<target name="generateClient" description="Create the Client artefacts for the
    defined WSDL file-WSDL2Java">
    <echo>Starting: WSDL2Java, create CLIENT for ${webServiceClassName}</echo>
    <WSDL2Java
        url=".\\WebContent\\WEB-INF\\wsdl\\${wsServerProjectName}.wsdl"
        container="web"
        role="client"
        output="..\\${wsClientProjectName}\\WebContent"
        genjava="overwrite" genxml="overwrite">
    </WSDL2Java>
    <!-- move the java classes to the source directory -->
    <move overwrite="true" todir="..\\${wsClientProjectName}\\JavaSource">
        <fileset dir="..\\${wsClientProjectName}\\WebContent" includes="itso\\**">
        </fileset>
    </move>
    <!-- copy the WSDL to the WEB-INF directory -->
    <copy overwrite="true" todir="..\\${wsClientProjectName}\\WebContent\\
                                                WEB-INF\\wsdl">
        <fileset dir=".\\WebContent\\WEB-INF\\wsdl" includes="*.wsdl"> </fileset>
    </copy>
    <echo>Done: WSDL2Java, create CLIENT for ${webServiceClassName}</echo>
</target>
```

All target

The all target runs the three other targets in the required sequence. It is also the default target (Example 19-11).

Example 19-11 Ant build script: All target

```
<target name="all" depends="createWSDL, generateServer, generateClient"
    description="Build WSDL then server then client">
    <echo>Done: ALL for ${webServiceClassName}</echo>
</target>
```

Running the Ant script

To run the script, select the `build.xml` file and *Run* → *Ant Build*. This executes the default target (or whatever was last selected). By selecting *Run* → *Ant Build*, a page opens where the target can be selected and many other options can be set.

Example 19-12 shows the execution log for the `generateServer` target.

Example 19-12 Ant build script: Execution

```
Buildfile: E:\Workspaces\RAD60sg246461\WeatherServiceAnt\build.xml
generateServer:
    [echo] Starting: WSDL2Java, create SERVER for itso.bean.WeatherJavaBean
[WSDL2Java] WSDL2Java .\WebContent\WEB-INF\wsdl\WeatherServiceAnt.wsdl
    [move] Moving 6 files to
E:\Workspaces\RAD60sg246461\WeatherServiceAnt\JavaSource
    [echo] Done: WSDL2Java, create SERVER for itso.bean.WeatherJavaBean
BUILD SUCCESSFUL
Total time: 20 seconds
```

Important: The `CreateWSDL` target can fail if the class path becomes too long, for example, when using the default installation location for Rational Application Developer.

Conclusions about Ant

The Web services Ant tasks do not generate the necessary configuration to run the Web service in WebSphere Application Server. For example, neither the router servlet for HTTP messages, nor the MDB router project for JMS bound Web services can be created with the tasks.

Therefore, we recommend that you use the Application Developer Web Service wizards to create the initial project set (create Web service and Web service client). After that, the Web services Ant tasks can be used for any further automated Web services generation.

Multiprotocol binding

The multiprotocol Java API for JAX-RPC is an extension of the JAX-RPC programming model that includes some of the features of the Web Services Invocation Framework (WSIF), which is now deprecated in WebSphere. The aim of the multiprotocol binding is to extend the existing JAX-RPC capabilities to support non-SOAP bindings.

WebSphere Application Server Version 6 supports the RMI binding for SOAP. The reason for this support comes from direct integration of EJBs in a service-oriented architecture. Normally, you would *Web service enable* the EJB by adding a router servlet to the EJB. This servlet accepts incoming SOAP over HTTP requests, demarshalls them, and calls the corresponding EJB. The last step in this chain also involves a marshalling/demarshalling, because EJB are invoked using the RMI over IIOP protocol (when invoked remotely). This basically means that the parameters are (Java) serialized and deserialized again.

The goal of the RMI binding is to avoid the extraneous marshalling/demarshalling when invoking SOAP over HTTP and to directly contact the EJB. In general, this accounts for increased performance of the application. In addition, the client that is invoking the JAX-RPC call can participate in user transactions. This cannot be achieved with the standard means of invoking Web services.

Command-line example for multiprotocol binding

The following fragment shows the command line that can be used to create the WSDL file that uses the RMI binding for the WeatherEJB:

```
rem see genWsd1.bat command in sampcode\multiprotocol
set RADHOME=C:\<RAD60_HOME>\runtimes\base_v6
set PATH=%RADHOME%\bin;%PATH%
set CLASSPATH=.;%RADHOME%\lib\webservices.jar;%CLASSPATH%

call Java2WSDL.bat
    -output WeatherEJB-RMI-Binding.wsdl
    -bindingTypes ejb
    -style wrapped -use literal
    -servicePortName WeatherEJB
    -namespace "http://ejb.itso"
    -classpath %CLASSPATH%
    -methods setWeather,getForecast,getDayForecast,getTemperatures
    -location "wsejb:/itso.ejb.WeatherEJBHome?jndiName=ejb/WeatherEJB
              &jndiProviderURL=corbaloc:iiop:localhost:2809"
    itso.ejb.WeatherEJB
```

In this example, `localhost` refers to the server name on which the sample application is installed, and port 2809 is the default port used for communication with the JNDI services.

The Java2WSDL tool can be also used to generate more than one binding type per WSDL file, thus the name *multiprotocol binding*. In order to create more than one binding, you have to supply more than one value to the `-bindingTypes` option, separated by commas:

```
call Java2WSDL.bat
    -output WeatherEJB-RMI-Binding.wsdl
    -bindingTypes http,jms,ejb,none
```

The given example lists all possible binding types. You can get a full list of all available options by specifying the `-help` or `-helpX` options in the command. In the case of multiple bindings, you will have a slightly more complex syntax, because you have to distinguish the different locations, bindings, and so forth. Refer to the command's help for details.

Running the Java2WSDL tool

We provide an example in `\SG246461\sampcode\multiprotocol`:

- ▶ Copy the `itso` directory (`itso\bean` and `itso\objects`) and the `.bat` command files to `\SG246461\commandtest\multiprotocol`.
- ▶ Run the `genWSDL.bat` command (after tailoring) to generate the `WeatherEJB-RMI-Binding.wsdl` file.

WSDL file with EJB binding

In the case of EJB bindings only, the generated WSDL file looks like Example 19-13 (after having removed empty lines and reformatting).

Example 19-13 WSDL file with EJB binding: WeatherEJB-RMI-Binding.wsdl

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions xmlns:ejb="http://www.ibm.com/ns/2003/06/wsdl/mp/ejb"
  xmlns:generic="http://www.ibm.com/ns/2003/06/wsdl/mp"
  xmlns:impl="http://ejb.itso" xmlns:intf="http://ejb.itso"
  xmlns:tns1="http://objects.itso"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:ws-i="http://ws-i.org/profiles/basic/1.1/xsd"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://ejb.itso">
  <wsdl:types>
    <schema xmlns="http://www.w3.org/2001/XMLSchema" .....>
      <import namespace="http://objects.itso"/>
      .....
    </schema>
    <schema xmlns="http://www.w3.org/2001/XMLSchema" .....>
      <complexType name="Weather">
        <sequence>
          <element name="condition" nillable="true" type="xsd:string"/>
          <element name="date" nillable="true" type="xsd:dateTime"/>
          <element name="windDirection" nillable="true" type="xsd:string"/>
          <element name="windSpeed" type="xsd:int"/>
          <element name="temperatureCelsius" type="xsd:int"/>
        </sequence>
      </complexType>
    </schema>
```

```

</wsdl:types>

<wsdl:message name="getTemperaturesRequest">.....</wsdl:message>
<wsdl:message name="getForecastRequest">..... </wsdl:message>
<wsdl:message name="getForecastResponse">.....</wsdl:message>
<wsdl:message name="getDayForecastResponse">.....</wsdl:message>
<wsdl:message name="getTemperaturesResponse">.....</wsdl:message>
<wsdl:message name="setWeatherRequest">.....</wsdl:message>
<wsdl:message name="setWeatherResponse">.....</wsdl:message>
<wsdl:message name="getDayForecastRequest">.....</wsdl:message>

<wsdl:portType name="WeatherEJB">
  <wsdl:operation name="getDayForecast">
    <wsdl:input message="impl:getDayForecastRequest"
      name="getDayForecastRequest"/>
    <wsdl:output message="impl:getDayForecastResponse"
      name="getDayForecastResponse"/>
  </wsdl:operation>
  <wsdl:operation name="getForecast">..... </wsdl:operation>
  <wsdl:operation name="getTemperatures">..... </wsdl:operation>
  <wsdl:operation name="setWeather">.....</wsdl:operation>
</wsdl:portType>

<wsdl:binding name="WeatherEJBBinding" type="impl:WeatherEJB">
  <ejb:binding/>
  <wsdl:operation name="getDayForecast">
    <ejb:operation methodName="getDayForecast"/>
    <wsdl:input name="getDayForecastRequest"></wsdl:input>
    <wsdl:output name="getDayForecastResponse"></wsdl:output>
  </wsdl:operation>
  <wsdl:operation name="getForecast">
    <ejb:operation methodName="getForecast"/>
    <wsdl:input name="getForecastRequest"></wsdl:input>
    <wsdl:output name="getForecastResponse"></wsdl:output>
  </wsdl:operation>
  <wsdl:operation name="getTemperatures">
    <ejb:operation methodName="getTemperatures"/>
    <wsdl:input name="getTemperaturesRequest"></wsdl:input>
    <wsdl:output name="getTemperaturesResponse"></wsdl:output>
  </wsdl:operation>
  <wsdl:operation name="setWeather">
    <ejb:operation methodName="setWeather"/>
    <wsdl:input name="setWeatherRequest"></wsdl:input>
    <wsdl:output name="setWeatherResponse"></wsdl:output>
  </wsdl:operation>
</wsdl:binding>

<wsdl:service name="WeatherEJBService">
  <wsdl:port binding="impl:WeatherEJBBinding" name="WeatherEJB">

```

```

        <generic:address location="wsejb:/itso.ejb.WeatherEJBHome
            ?jndiName=ejb/WeatherEJBJS
            &jndiProviderURL=corbaloc:iiop:<hostname>:2809"/>
    </wsdl:port>
</wsdl:service>

</wsdl:definitions>

```

If you compare the WSDL file with the WSDL file with the SOAP binding, you notice the changes that we highlighted in the file. Basically, you could have done manual changes without the Java2WSDL tool.

Of particular interest is the location of the service. The given example lists all possible parameters and also exhibits the EJB binding's location syntax. You can omit the `jndiProviderURL`, but we do not recommend this because it makes clear how you are communication with the server.

Generating a client

Using this WSDL file, you can now create a test client and proxy. To this end, invoke the New Web Service wizard and proceed as with any other WSDL file. See “Web Service wizard” on page 275 for details about how to use the wizard. The generated test client and proxy can be used just in the case of SOAP clients and proxies.

Generating a client with command-line tools

You can use commands similar to these to generate a J2EE application client:

```

rem see RunWSDL2Client1.bat command
set RADHOME=C:\RAD60\bin
set PATH=%RADHOME%;%PATH%
call setupenv.bat

call WSDL2Client.bat -verbose -project WeatherMultiClient
                    -clientType application -genMain WeatherMultiJava
                    WeatherEJB-RMI-Binding.wsdl

```

The output is in the `WeatherMultiClient` subdirectory. You could continue and generate an EAR file.

Generating a client with Application Developer

In this section, we provide instructions about how to create a servlet client that uses the EJB binding. Follow these steps:

- Verify that the `WeatherEJBServer` application is running in the server.

- ▶ Create an enterprise application named `WeatherEJBMultiProtocolEAR` with a Web module named `WeatherEJBMultiProtocolWeb`.
- ▶ Add `WeatherEJBClientClasses.jar` and `WeatherBase.jar` as utility JARs (in the enterprise application deployment descriptor).
- ▶ Open the Web application deployment descriptor and define an EJB reference named `ejb/WeatherEJB` that points to the `WeatherEJB` with JNDI name `ejb/WeatherEJB`. This enables the Web project to access the EJB directly.
- ▶ Import the `WeatherEJB-RMI-Binding.wsdl` file into the `WebContents` folder.
- ▶ Select the `WeatherEJB-RMI-Binding.wsdl` file and *Web Services* → *Generate Client*. Go through the wizard to generate the client proxy classes into the same project. Make sure that the client type is *Web* and the client project is `WeatherEJBMultiProtocolWeb`.

Restriction: In Version 6.0 of Application Developer, the Web Service wizard abends after generating the proxy and does not continue. The test client JSPs are not generated. However, a client using the proxy can be created as described in the following text.

The wizard works fine in Application Developer 6.0.1, and test client JSPs are created and can be executed.

- ▶ Create an `itso.servlet` package and a servlet named `WeatherMultiServlet`. Edit the servlet with the code provided in:

```
\SG246461\multiprotocol\servlet\WeatherMultiServlet.java
```

- ▶ Add the `WeatherEJBMultiProtocolEAR` application to the server (or restart the application). Verify that the `WeatherEJBServer` application is also running in the server. Then, run the servlet:

```
http://localhost:9080/WeatherEJBMultiProtocolWeb/WeatherMultiServlet
```

- ▶ The servlet runs the Web service with EJB binding and displays one weather forecast, such as:

Weather EJB Binding Client

Day Forecast for: Wed Dec 22 00:00:00 PST 2004

Weather: cloudy 3 degrees

End

Summary

In this chapter, we described Web services tooling through commands that are useful when tasks should be automated through command files.

Command files are also advantageous to explore different Web service options and to compare the generated output.

In addition to the command-line tools, Ant tasks can be used to automate the deployment of the client and servers.

Finally, the new multiprotocol binding enables fast access to EJB Web services through RMI-IIOP, without the serialization that is required when going through the SOAP protocol.



Part 3

Advanced Web services techniques

In Part 3, we explore some advanced Web services techniques, such as interoperability, security (WS-Security implementation), the service integration bus, a private UDDI registry, and caching of Web services.

Web services interoperability tools and examples

This chapter provides information about the tools and features provided in IBM Rational Application Developer that assist in the development and compliance testing of Web services that conform to the standards developed by the WS-I Organization.

In addition, we provide step-by-step instructions for developing a number of simple Web service clients that run on different runtime engines and show that they are capable of interoperating successfully with the sample weather forecast application that was introduced in Chapter 14, “Sample application: Weather forecast” on page 247 and implemented in Chapter 16, “Develop Web services with Application Developer V6.0” on page 273.

Interoperability tools in Application Developer

In this section, we discuss the tools and features available in Rational Application Developer that provide assistance in the development of WS-I compliant Web services.

Setting the compliance level

Application Developer can be 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 20-1 shows the preference at the workspace level.

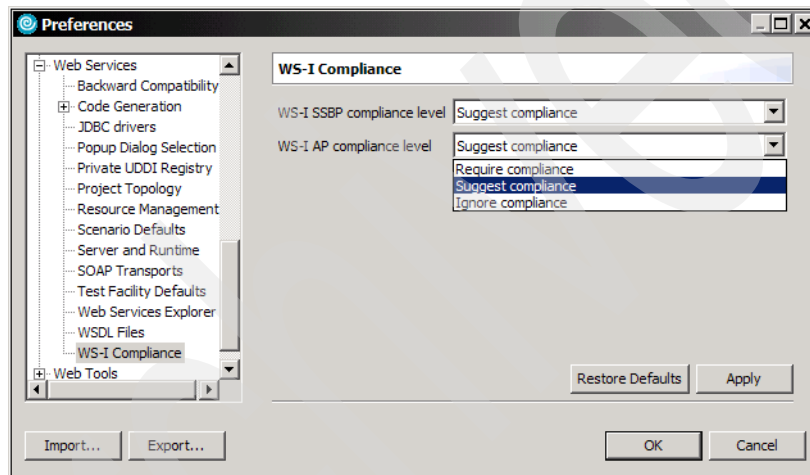


Figure 20-1 WS-I compliance levels preferences

These preferences determine the behavior of the WS-I validation tools built into Rational Application Developer:

- ▶ **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.

These settings can also be 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).

WSDL validator

WSDL files that are created, or are imported into Application Developer, can be validated against the W3C Web Services Description Language (WSDL) V1.1 specifications and against the WS-I profiles.¹ WSDL that is generated by 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 *Validate WSDL file* (context). Validation succeeds or fails, and a pop-up message window opens (Figure 20-2).

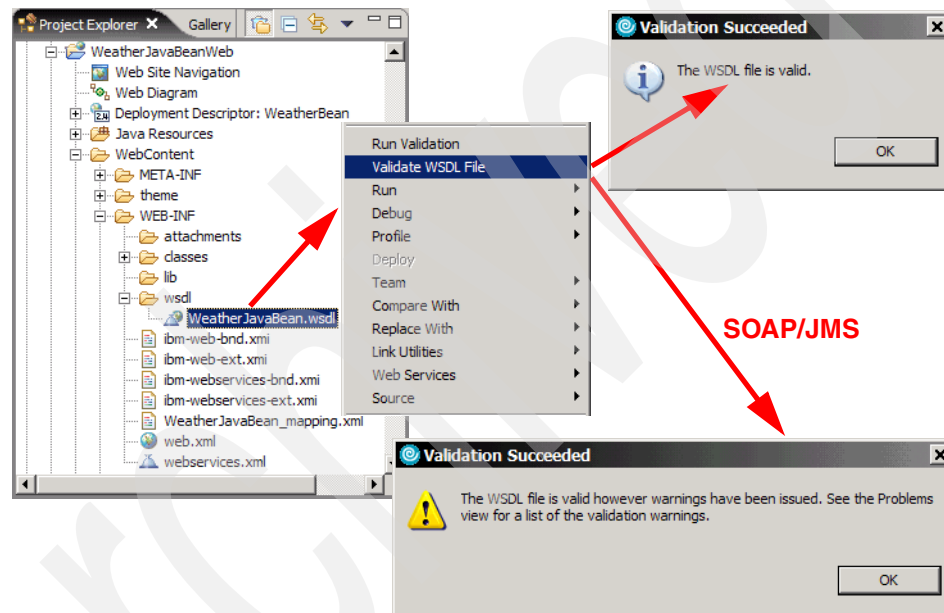


Figure 20-2 Validating a WSDL file

WS-I message validation

Similar to the WS-I validation tools (see “WS-I tools” on page 198), Application Developer allows the capture of Web service SOAP/HTTP messages using the TCP/IP Monitor tool (see “TCP/IP Monitor” on page 368). The request and response SOAP messages can then be validated against the WS-I standards.

¹ If you selected Ignore WS-I Compliance in your preferences, the WSDL will not be validated against the WS-I standards.

To validate a Web service, perform these steps:

- ▶ Set up the TCP/IP Monitor and route the Web services calls to the monitor (for example, using port 9081).
- ▶ 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, you can generate a log file by clicking the  icon (Figure 20-3).

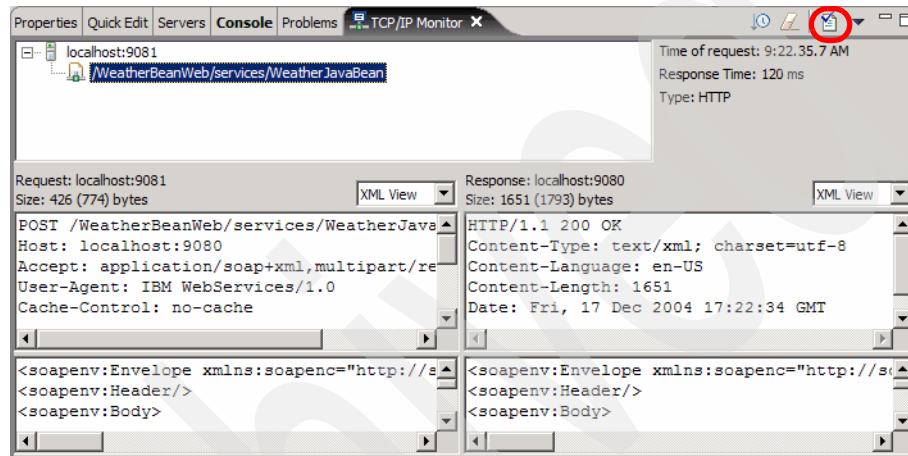


Figure 20-3 Validating SOAP/HTTP messages

- ▶ In the dialog box that opens, enter a name for the log file and specify where you want it to be stored, for example, in the WebContent folder of the project that runs the Web service. The file is validated, and a message window displays the result (Figure 20-4).

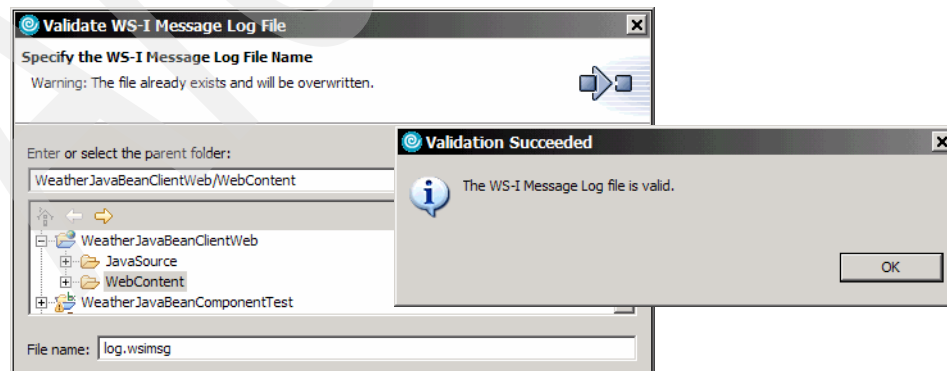


Figure 20-4 SOAP/HTTP message validation result

Interoperability examples

In this section, we take a step-by-step approach for developing a number of simple Web service clients that run on different runtime engines: Apache Axis Version 1.0 and Microsoft .NET Framework Version 1.1.

We develop and test clients that use the Web services provided by the sample weather forecast application that was introduced in Chapter 14, “Sample application: Weather forecast” on page 247, and which was implemented in Chapter 16, “Develop Web services with Application Developer V6.0” on page 273. We show that these Web service clients are able to interoperate successfully against the sample application that is running in an IBM WebSphere Application Server Version 6.0 environment.

Prerequisites

We assume that you have already followed the step-by-step instructions provided in “Creating a Web service from a JavaBean” on page 275. Alternatively, you might have downloaded the additional material from the IBM Redbooks Web site (see Appendix C, “Additional material” on page 701) and imported the WeatherJavaBeanServer sample enterprise application.

You should test that the sample application is working correctly before proceeding. Run the `TestClient.jsp` in the WeatherJavaBeanClientWeb project.

Note: Stop any TCP/IP Monitors that are currently running. We will add a monitor dynamically through the Web Service wizard.

Apache Axis V1.0 example

In this example, we implement an Apache Axis V1.0 Web client using Application Developer V6.0.

Client development

We are now ready to develop our simple Axis client application for the WeatherJavaBeanServer application:

- ▶ Select the WeatherJavaBean.wsdl file (in the WeatherJavaBeanWeb project) and *Web Services* → *Generate Client*.
- ▶ In the Web Service Client wizard, select *Test the Web service* and *Monitor the Web service* and click *Next*.

- ▶ On the Web Service Selection page, the WSDL file is preselected.
- ▶ On the Client Environment Configuration page (Figure 20-5), click *Edit*:
 - In the dialog box, select *Choose Web service runtime protocol first*. Select *Apache Axis 1.0*. Expand *Existing Servers* and select *WebSphere Application Server v6.0*. Click *OK* to return to the wizard.
 - *Web* is the only client type supported for Apache Axis 1.0 and IBM SOAP runtime engines. For the client project, enter *WeatherJavaBeanAxisWeb* and for the EAR project enter *WeatherJavaBeanAxisEAR*.

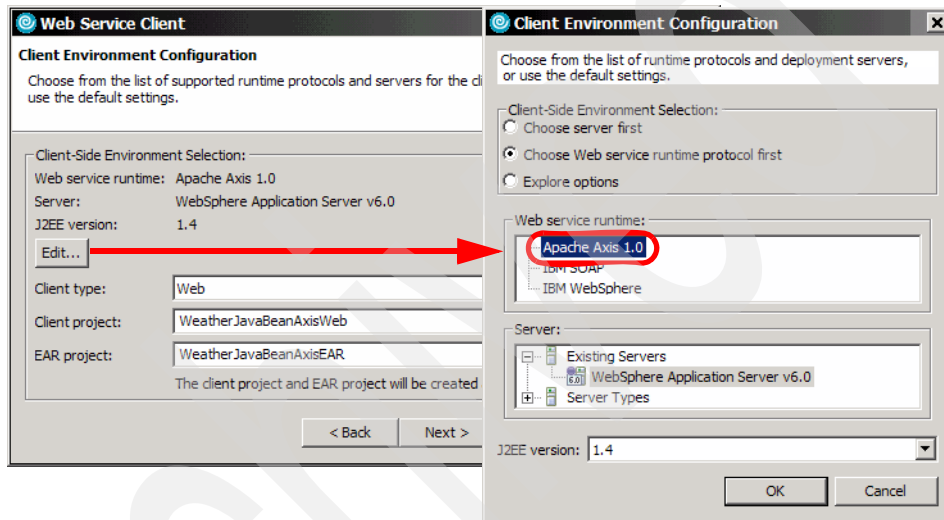


Figure 20-5 Selecting Client Environment Configuration

- ▶ Click *Next* on the proxy page.

Note: A pop-up message window stating that port 9080 cannot be monitored opens if TCP/IP Monitor is already running.

- ▶ Select *Test the generated proxy*, *Run test on server*, and click *Finish*.
- ▶ The Web service Web client is generated into *WeatherJavaBeanAxisWeb* and deployed to the server.
- ▶ A Web browser window opens and the Web test client interface is displayed. Run some of the methods (Figure 20-6).

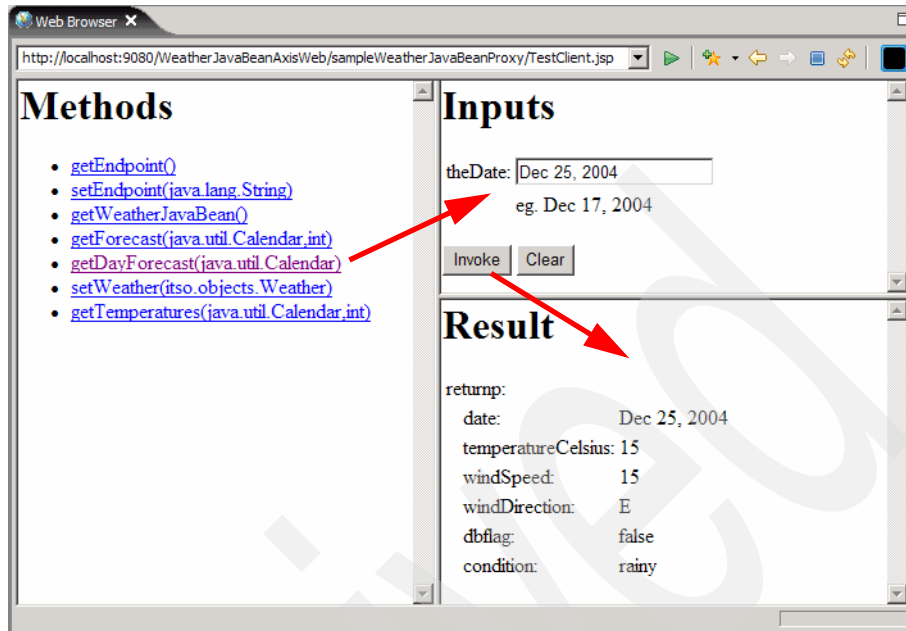


Figure 20-6 Axis test client

- ▶ Invoke the `getEndpoint` method. The result shows that port 9081 is used and Web services calls go through the TCP/IP Monitor. This has been automatically configured because we selected *Monitor the Web service*.

Comparing the Axis client to the WebSphere client

Here, we compare the generated code in `WeatherJavaBeanAxisWeb` to the code in `WeatherJavaBeanClientWeb`:

- ▶ Axis generates wrapper classes for arrays (`ArrayOf_Tns2_Weather` and `ArrayOf_Xsd_Int`).
- ▶ WebSphere generates serializers (`Weather_Ser`, `_Deser`, and `_Helper`).
- ▶ Axis generates serializer code into the `Weather` class itself.
- ▶ WebSphere generates a Web service deployment descriptor (`web.xml` and extension files `ibm-webservicesclient-bnd/ext`).

Conclusion

An Axis client can interoperate with a service running the WebSphere engine.

Microsoft .NET example

In this example, we implement a simple Microsoft .NET Windows form *fat client* application using the C# language.

The client is developed using the Borland C#Builder Personal Download Edition, which is free for non-commercial development. The final application is a Microsoft Windows .NET managed executable that makes a Web service request against the weather forecast Web service, which was developed in “Creating a Web service from a JavaBean” on page 275, and which is running in the WebSphere Application Server V6.0 test environment.

The Windows form application contains a text box that holds the Web service endpoint address, a button to initiate the call to the Web service, and a label to hold the weather forecast result or error message (Figure 20-7).

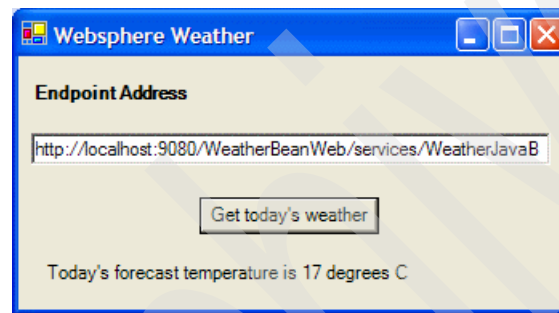


Figure 20-7 Microsoft .NET Windows form Web service client

Environment setup

Before beginning the development of this example, we have to install the base components required for .NET development. This example was developed using the software shown in Table 20-1.

Table 20-1 Microsoft .NET example environment setup

Client/consumer	Service/producer
▶ Microsoft Windows XP Professional Version 2002 SP1 ▶ Microsoft .NET Framework V1.1 ▶ Microsoft .NET Framework Software Development Kit (SDK) V1.1 ▶ Borland C#Builder	▶ Microsoft Windows 2000 5.00.2195 SP4 ▶ IBM Rational Application Developer V6.0 with WebSphere Application Server V6.0 test environment

Install Microsoft .NET Framework SDK V1.1

To run .NET applications, you must first have the Microsoft .NET Framework V1.1 installed. To develop the .NET client application, you will also require the Microsoft .NET Framework SDK V1.1 and Redistributable.

You can check to see if you already have these installed by selecting *Start* → *Control Panel* → *Add or Remove Programs* on the desktop. If you see both Microsoft .NET Framework 1.1 and Microsoft .NET Framework SDK (English) 1.1, you do not have to install the product again.

To download and install the product, follow the instructions in the article *MSDN .NET Framework Developer Center: Get the .NET Framework 1.1*, available at:

http://msdn.microsoft.com/netframework/downloads/framework1_1/#section3

Install Borland C#Builder Personal Download Edition

The Borland C#Builder Personal Download Edition is an integrated development environment (IDE) for building and deploying C# applications that run in the Microsoft .NET environment. This product is currently available free for non-commercial development only.

You can download² a copy of the product from:

http://www.borland.com/products/downloads/download_csharpbuilder.html

Install and activate the product using the instructions provided.³

Note: C#Builder has been integrated into the Delphi product, and the download page might be removed. If you cannot find the C#Builder, download the **Delphi 2005 Architect Trial** to develop the .NET client:

http://www.borland.com/downloads/download_delphi.html

Client development

We are now ready to develop our simple client application:

- ▶ Start up the Borland C#Builder application. You should see the Borland welcome page.
- ▶ Create a new project by selecting *Project* → *Add New Project*. Select a C# Application project, as shown in Figure 20-8, and click *OK*.

² Note that you have to register with a valid e-mail address to download the product.

³ The Borland C#Builder product requires Microsoft Internet Explorer 6.0 SP1, Microsoft .NET Framework V1.1, and Microsoft .NET Framework SDK V1.1 to be preinstalled on the machine.

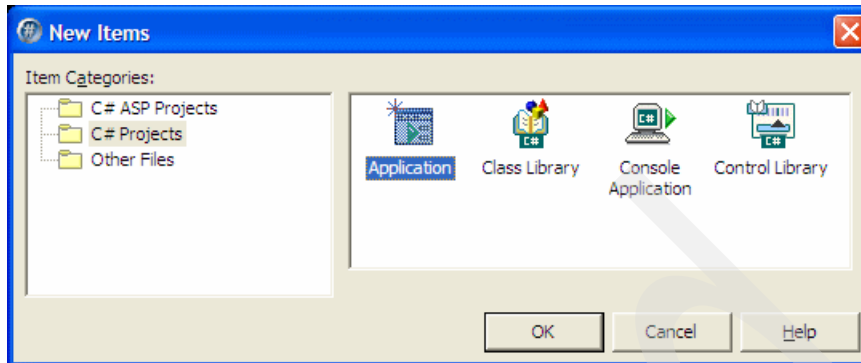


Figure 20-8 Creating a C# application project

- Enter WeatherCSApp for the application name. Use the default location. Click *OK* to create the project. The new project should be opened, and a number of files automatically created. The WinForm.cs file should be opened in the Design view (Figure 20-9).

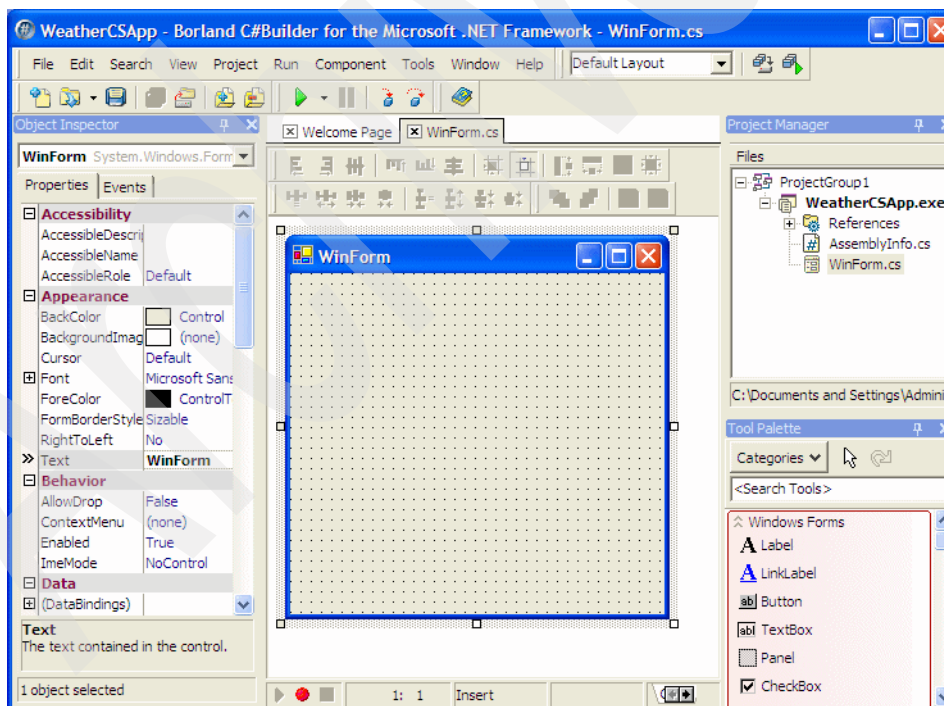


Figure 20-9 New WeatherCSApp C# project

- First, add a text box to the form by selecting the *TextBox* object in the Tool Palette (lower right of the window) and then clicking in the WinForm window at the position where you would like the text box to be located. The text box is created with a default name of `textBox1`. Move or resize it as necessary (Figure 20-10).

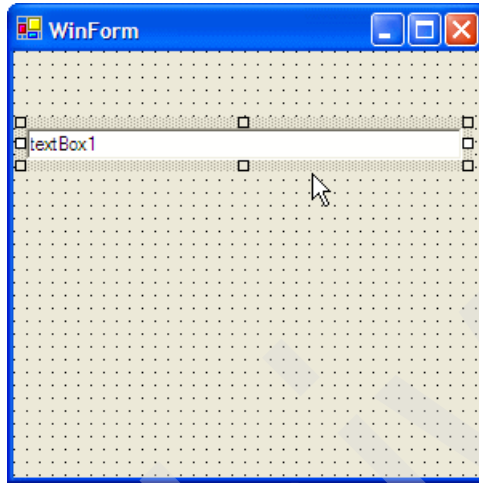


Figure 20-10 Adding a text box to the form

- In the Properties view (left side of the window), edit the Text value (not the Name value) and change it to contain the value of the weather forecast Web service, for example:
`http://<hostname|IPaddress>/WeatherBeanWeb/services/WeatherJavaBean`
Refer to the WSDL file in Application Developer, and remember to substitute `localhost` with the host name or IP address if Application Developer is running on a different machine.
- Add a button to the WinForm using the Tool Palette. This button will be used to initiate the call to the Web service. Keep the default name of `button1`.⁴ In the Properties view, change the Text value to `Get today's weather`.
- Add a large Label to the WinForm window at the bottom. This label will be used to contain the weather forecast results (or an error message). In the Properties view, change the Text value to `""` (Figure 20-11).

⁴ It is a bad coding practice to use the default names, but in this case, we are trying to keep the example as simple as possible.

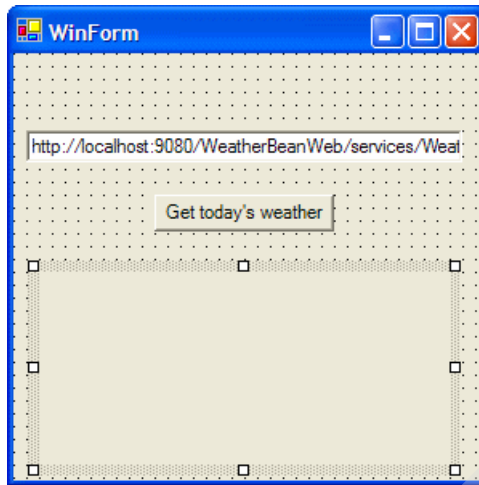


Figure 20-11 Adding the result label to the form

- ▶ Add another label above textbox1 and change its Text property to Endpoint Address and its Font property to Bold.
- ▶ Next, we add a reference to the Web service by selecting *Project* → *Add Web Reference*. The UDDI Browser opens (Figure 20-12).

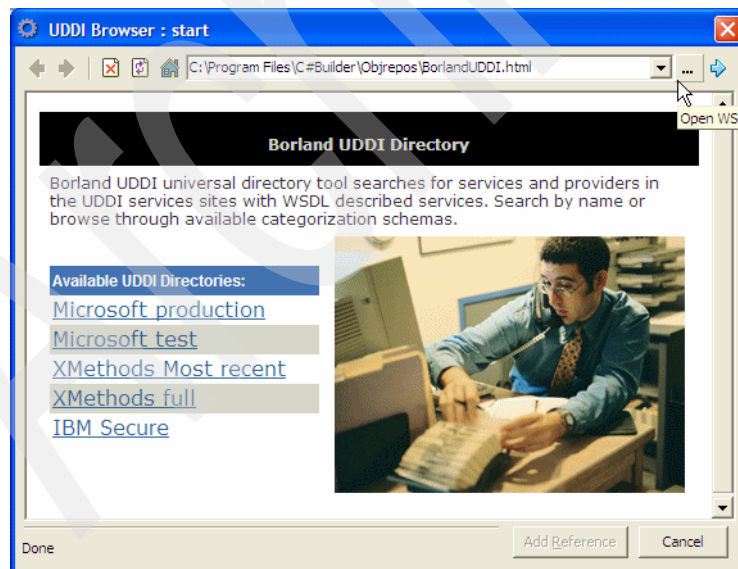


Figure 20-12 UDDI Browser

- In the UDDI Browser window, browse to where the WSDL file is located. If the WSDL file is not located on the local machine, you can browse across the network, or enter the URL of the WSDL, for example:

`http://<tcpaddress>:9080/WeatherBeanWeb/wsdl/itso/bean/WeatherJavaBean.wsdl`

In this case, the endpoint address in the WSDL file does not have to be correct because we are overriding it in our application.

- When the WSDL is displayed in the UDDI Browser window, click *Add Reference*. The Web reference is added to the project, and a new file, *Reference.cs*, is added to the project. This file contains the necessary C#.NET Web service proxy code. Figure 20-13 shows the Web reference in the Project Manager view.

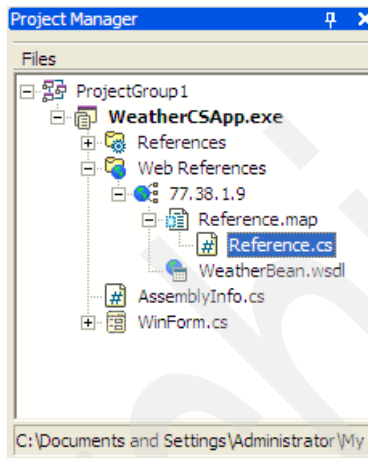


Figure 20-13 *WeatherJavaBean reference*

- Next, we have to add code to the button so that when we click it, the Web service is invoked and the results displayed in the label area. We do not want to block the user interface thread while making a distributed network call, so we create a worker thread that runs the *getDayForecast* method.

In the *WinForm.cs* design view, double-click the *button1*. This switches *WinForm.cs* to the Code view and provides a C# method stub for *button1*. Modify *Winform.cs* as shown in Figure 20-14.

```

private void button1_Click(object sender, System.EventArgs e)
{
    label1.Text = "Getting today's weather forecast...";
    label1.Refresh();
    // Now create a worker thread to make call to web service - we
    // don't want to block the user interface thread
    try {
        Thread wsThread = new Thread(new ThreadStart(this.getForecast));
        wsThread.Name = "Web Service Worker Thread";
        wsThread.Start();
    }
    catch(System.Exception se) {
        label1.Text = "Caught this exception: "+se;
        label1.Refresh();
    }
}

```

Figure 20-14 C# code for the push button click event

- We have to add a using directive (similar to a Java import statement) at the top of the file. Add the following line (shown in **bold**) near the top of WinForm.cs (Figure 20-15).

```

using System;
using System.Drawing;
...
using System.Data;
using System.Threading;

namespace WeatherCSCClient
{
    ...
}

```

Figure 20-15 C# using directive

- We now have to add the remainder of the code. Return to the bottom of WinForm.cs, and insert the code shown in **bold** in Figure 20-16 after the button1_Click method. Almost all of this code is user interface related.⁵

It should be evident that the method call `ws.getDayForecast(now)` is the code that actually makes the call to the Web service.

⁵ .NET allows you to call `System.Windows.Forms.Control` (user interface) functions only from the thread in which the control was created. To call them from our worker thread, we have to use the `Control.Invoke()` function and a delegate.


```

private void button1_Click(object sender, System.EventArgs e) {
    ....
}

/* The updateLabel method needs to be run by the user interface
 * thread, so we create a delegate, and use the Control.Invoke()
 * method to implement this
 */
private delegate void updateLabelDelegate(String s);

private void updateLabel(String text) {
    label1.Text = text;
    label1.Refresh();
}

/* This method runs in a worker thread. It calls the WeatherBeanService
 * web service, and updates label1 with the results
 */
private void getForecast() {
    String output;
    try {
        // Create webservice proxy object
        WeatherBeanService ws = new WeatherBeanService();
        ws.Url = textBox1.Text; // Set web service endpoint address
        ws.Timeout = 10000; // 10 second timeout for response
        System.DateTime now = System.DateTime.Now;
        Weather theForecast = ws.getDayForecast(now); // Call web service
        output = "Today's forecast temperature is "
            + theForecast.temperatureCelsius + " degrees C";
    }
    catch(System.Exception se) {
        output = "Caught this exception: " + se;
    }
    // Cause the updateLabel() method to be run in the user
    // interface thread
    this.Invoke(new updateLabelDelegate(this.updateLabel),
        new Object[] {output});
}
}

```

Figure 20-16 C# code for calling Web service

Client test

We are now ready to compile and test the simple client application.

- Compile and run the application by clicking the *Run* icon in the toolbar (Figure 20-17).

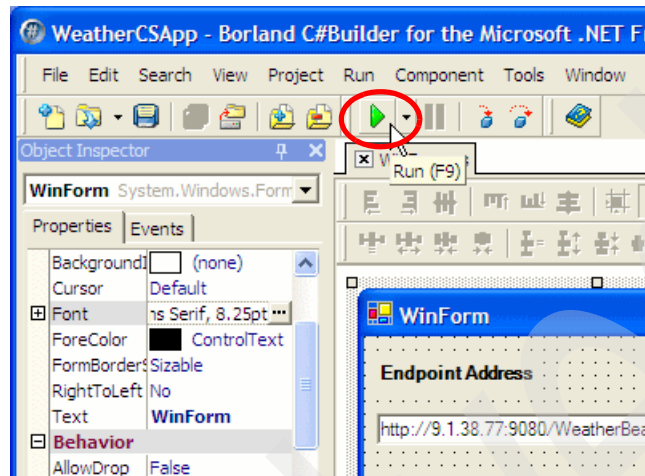


Figure 20-17 Running the C# application

- The C# project is compiled and built, the C#Builder window changes to Debug Layout, and the application window starts on top.
- Click *Get today's weather* to call the Web service and display the results, as shown in Figure 20-7 on page 434.
- We distribute the finished program in:

\SG246461\sampcode\interop\WeatherCSApp.exe

Note that if you use the distributed WeatherCSApp.exe file, you have to overwrite <tcpaddress> with localhost or the real address of the host:

http://<tcpaddress>:9080/WeatherBeanWeb/services/WeatherJavaBean

Important: When executing the application, it must be located on a *local* disk drive. Attempting to execute the application from a network drive or remote location causes a Windows SecurityException to be thrown.

Verifying WS-I compliance

At this point, we can run the application using a modified endpoint address (such as 9081) so that the Web service messages are sent through the TCP/IP Monitor and check that the messages are WS-I compliant.

Refer to “WS-I message validation” on page 429 in conjunction with the Help information for instructions.

Summary

We have explained the functions and features of Rational Application Developer that assist and promote the development of Web services that will interoperate with other platforms.

We also provided examples of developing non-IBM Web service clients and shown that they are able to interoperate with the IBM WebSphere Application Server runtime platform.

More information

More information about the subject of interoperability, including the WS-I Organization, the evolution of support for Web services in the IBM WebSphere Application Server products, and best practices for interoperating with the Microsoft .NET platform, can be found in Chapter 10, “Web services interoperability” on page 195. This chapter also contains useful online references about the subject.

Securing Web services

This chapter describes the functions provided by IBM Rational Application Developer Version 6.0 and IBM WebSphere Application Server Version 6.0 to develop secure Web services application using Web services security.

We review the support of the Web services security specifications supported by WebSphere Application Server Version 6 and describe the architecture of the security configurations in the client and server. Then, we implement authentication, integrity, confidentiality, and timestamp options on one of our Web services applications. We describe how to configure a server for security testing and test and monitor the secure application. Finally, we explore the options for deployment of secure Web services in WebSphere Application Server and how to migrate from a Version 5 server.

For a description of the basic Web services security concepts, refer to Chapter 9, “Web services security” on page 173.

Overview

To secure a Web service application, we have a number of options, which have been explained in Chapter 9, “Web services security” on page 173. In this chapter, we describe how to develop a Web services application secured by WS-Security with Rational Application Developer.

WS-Security can be configured by either Application Developer or the Application Server Toolkit (ASTK). Both functions of WS-Security configuration are equivalent; it depends on the user which tool is used. In this chapter, we configure WS-Security with Application Developer.

Who should implement Web services security?

First of all, let us define the target of this section: “Who implements Web services security and what is needed?” The situation is as follows:

- ▶ A Web application is developed by an application developer or programmer.
- ▶ An assembler assembles the application as a Web services application.
- ▶ After that, a security developer implements security features for the Web services application.

These three roles might be performed by only one person, but in this chapter, we describe the role of the security developer.

To implement WS-Security for the existing Web services application, a security developer has to configure WS-Security configuration files for both the Web services application client and service, assemble these files in Web services application EAR files, and test the application in the test environment of Rational Application Developer Version 6.0 or WebSphere Application Server Version 6.0.

In this chapter, we describe the chain of implementing WS-Security using this sequence:

- ▶ Typical scenario for WS-Security
- ▶ Features of WS-Security in Application Server V6.0
- ▶ Development of WS-Security

The compatibility of Web services security with WebSphere Application Server Version 5.x is described in “Configuration mapping: Version 5.x to Version 6.0” on page 666.

Typical scenario for WS-Security

WS-Security is a message-level security, which means that we can apply various scenarios of WS-Security according to the characteristics of each Web service application. For example, to verify who requests the service, we can add an authentication mechanism by inserting various types of security tokens. To keep the integrity or confidentiality of the message, digital signatures and encryption are typically applied.

For a typical scenario with WS-Security, let us look at a bank teller scenario. Without message-level security, the SOAP message is sent in clear text, and personal information, such as a user ID or an account number, is not protected from an attacker. When message-level security is applied in this scenario, we can protect the SOAP message with a security token, digital signature, and encryption. There are many choices of how to protect information by WS-Security. Figure 21-1 shows one typical example of applying WS-Security in the bank teller scenario.

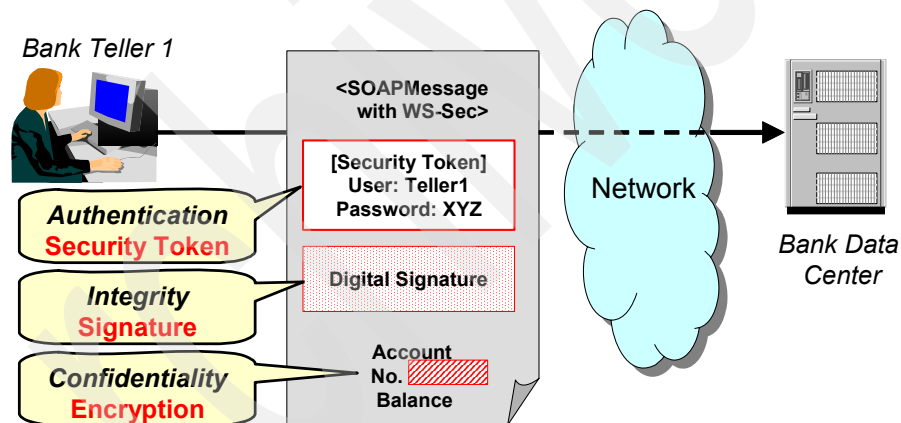


Figure 21-1 An example of a typical scenario with WS-Security

Figure 21-1 shows one of the most fundamental scenarios. However, there are not only simple scenarios, but also more complex authentication scenarios using WS-Security: Kerberos using WS-Security, WS-Trust, and WS-SecureConversation.

WebSphere Application Server Version 6.0 provides flexibility to extend the capability to support higher-level specifications:

- ▶ The WS-SecurityKerberos architecture can be implemented by using WS-Security, and its sample implementation is provided as a technical preview of WebSphere Application Server Version 6.0.

- ▶ The scenario of WS-SecureConversation and WS-Trust offers a way to establish a session context that is issued by a trusted party and is used like a temporary ticket between a client and a server.

The details of these scenarios are described in these documents:

- ▶ *Web Services Security Kerberos Binding*
<ftp://www6.software.ibm.com/software/developer/library/ws-seckerb/WS-Security-Kerberos.pdf>
- ▶ *Web Services Trust Language (WS-Trust)*
<ftp://www6.software.ibm.com/software/developer/library/ws-trust.pdf>
- ▶ *Web Services Secure Conversation Language (WS-SecureConversation)*
<ftp://www6.software.ibm.com/software/developer/library/ws-secureconversation.pdf>

Authentication

To apply an authentication mechanism, it is necessary to insert a security token into the request message of a client. WebSphere Application Server Version 6.0 has a pluggable token architecture, which enables a user to implement a custom token.

WebSphere Application Server Version 6.0 provides some types of security tokens as the default, and you implement custom tokens yourself using the token pluggability. The provided default types of tokens are a username token and a binary security token, including the X.509 certificate and LTPA token. We explain the details of each token in the sections that follow.

For user authentication, the message with a security token is processed as shown in Figure 21-2, which is an example of basic authentication. The user information is extracted and passed to a user registry for verification. If the information is valid, the result is returned to the server, and the server accepts the message as it is authenticated.

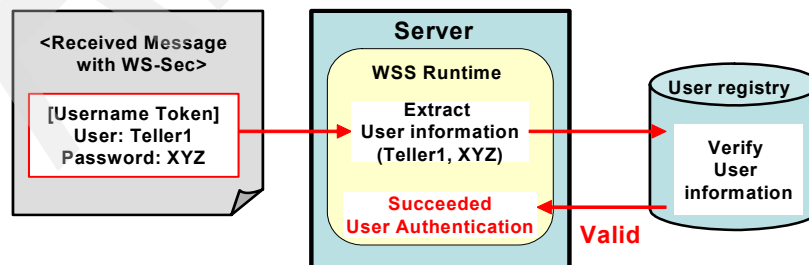


Figure 21-2 Processing of basic authentication

Username token

A username token is a token for basic authentication, which has a user name and password. The username token profile 1.0 is published by OASIS, and WS-Security in WebSphere Application Server Version 6.0 supports the profile. Example 21-1 shows an example of a username token.

Example 21-1 Example of a username token

```
<wsse:UsernameToken
  xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd">
  <wsse:Username>David</wsse:Username>
  <wsse:Password>d1vaD</wsse:Password>
</wsse:UsernameToken>
```

In the username token profile, the digest of the password is specified as a password type, but the password digest is not supported in WebSphere Application Server Version 6.0. Only the clear-text password can be inserted in a username token and should not be transferred over a non-secure network. Basic authentication should be used over the secure network, such as HTTPS or intranet, or encryption should be applied to hide the user information.

Binary security token

The two types of binary security tokens that are provided as a default are the X.509 certificate token and LTPA token:

- ▶ The LTPA token is a WebSphere-specific binary token that is authenticated by a WebSphere security mechanism.
- ▶ The X.509 certificate token is published, and the profile is supported by the WS-Security implementation in WebSphere Application Server Version 6.0.

The encoding method is specified in each binary security token. Examples of an X.509 single certificate token and an LTPA token are shown in Example 21-2 and Example 21-3.

Example 21-2 Example of an X.509 single certificate token

```
<wsse:BinarySecurityToken
  EncodingType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#Base64Binary"
  ValueType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509v3">
  MIIbzzCCATigAwIBAgIEQYDVQQG.....9FkdI=
</wsse:BinarySecurityToken>
```

```
<wsse:BinarySecurityToken
  xmlns:wsst="http://www.ibm.com/websphere/appserver/tokentype/5.0.2"
  ValueType="wsst:LTPA">
  nwHBBZwUF+m94fAuY57oQrGFyK.....nIIYMM5001kbtMWx5yaIo=
</wsse:BinarySecurityToken>
```

Custom token

A user can implement a custom token using a pluggable token architecture provided WebSphere Application Server Version 6.0. To implement a custom token, interfaces provided by WebSphere should be implemented by the user classes:

- ▶ For generating a custom token:
 - Token generator class that implements `TokenGeneratorComponent`
 - Callback handler class that implements `CallbackHandler`
- ▶ For receiving a custom token:
 - Token consumer class that implements `TokenConsumerComponent`
 - Login module class that implements `LoginModule`

See the *WebSphere Application Server Version 6.0 Information Center* and *JavaDocs* for a detailed explanation.

Identity assertion

Identity assertion is another authentication mechanism, which is applied between three parties: a client, an intermediary server, and an endpoint server:

- ▶ A request message is sent to an intermediary server with a client's security token.
- ▶ An intermediary server authenticates the client and transfers the client's request message to an endpoint server with the intermediary's security token.

There are several options of how to send a client token with a intermediary token to an endpoint server. This is provided as an extended mechanism in WebSphere Application Server Version 6.0. For details, refer to "Extensions in WebSphere Application Server V6.0" on page 460.

Integrity and confidentiality

Integrity is provided by applying a digital signature to a SOAP message. Confidentiality is applied by SOAP message encryption. In Version 6.0, multiple signatures and encryptions are supported. In addition, both signing and encryption can be applied to the same parts, such as the SOAP body.

Example 21-4 shows a SOAP message with integrity and confidentiality. This scenario is appropriate when the message includes personal information, such as a credit card number or a bank account number. In the example, the whole SOAP body is signed and encrypted.

Example 21-4 An example of applying integrity and confidentiality to SOAP body

```
<soapenv:Envelope xmlns:soapenc="....." xmlns:soapenv="....."
  xmlns:xsd="....." xmlns:xsi=".....">
  <soapenv:Header>
    <wsse:Security soapenv:mustUnderstand="1" xmlns:wsse=".....">
*----- SECURITY TOKEN FOR SIGNATURE -----*
      <wsse:BinarySecurityToken
        EncodingType="....."
        ValueType="....."
        wsu:Id="x509bst_8371113208094682629" xmlns:wsu=".....">
        MIIBzzCCATigAwIB.....+CDN9XXYmlqiYhR9FkdI=
      </wsse:BinarySecurityToken>
*-----*

*===== ENCRYPTION INFORMATION =====*
      <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 ValueType=".....">
              Vniy7MUOXBumPoH1MNbDpiIWOPA=
            </wsse:KeyIdentifier>
          </wsse:SecurityTokenReference>
        </ds:KeyInfo>
        <CipherData>
          <CipherValue>
            YQ93Q6d0khP80Vp...../8yDt9r8Lt0fNMNA8=
          </CipherValue>
        </CipherData>
        <ReferenceList>
          <DataReference
            URI="#wssecurity_encryption_id_8938640161692105181"/>
          </DataReference>
        </ReferenceList>
      </EncryptedKey>
*=====*

*:::::::::: SIGNATURE ON SOAP BODY ::::::::::::*
      <ds:Signature xmlns:ds=".....">
        <ds:SignedInfo>
          <ds:CanonicalizationMethod
            Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#">
            <ec:InclusiveNamespaces PrefixList="...." xmlns:ec="....."/>
          </ds:CanonicalizationMethod>
        </ds:SignedInfo>
      </ds:Signature>
    </wsse:Security>
  </soapenv:Header>
  <soapenv:Body>
```

```

</ds:CanonicalizationMethod>
<ds:SignatureMethod
  Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
<ds:Reference
  URI="#wssecurity_signature_id_2025289588509637461">
  <ds:Transforms>
    <ds:Transform
      Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"
      <ec:InclusiveNamespaces
        PrefixList="....." xmlns:ec="...."/>
      </ds:Transform>
    </ds:Transforms>
    <ds:DigestMethod
      Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
    <ds:DigestValue>dg1MzU0bU6SxNa2wdOQ7baBczCg=</ds:DigestValue>
  </ds:Reference>
  </ds:SignedInfo>
<ds:SignatureValue>
  1mish0L3edefo7xZ0.....oF7zkDBnMbpL6SHXSjyKfg0=
</ds:SignatureValue>
<ds:KeyInfo>
  <wsse:SecurityTokenReference>
    <wsse:Reference URI="#x509bst_8371113208094682629"
      ValueType="....."/>
  </wsse:SecurityTokenReference>
</ds:KeyInfo>
</ds:Signature>
*::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::*
</wsse:Security>
</soapenv:Header>
<soapenv:Body wsu:Id="wssecurity_signature_id_2025289588509637461"
  xmlns:wsu=".....">
*===== ENCRYPTED SOAP BODY =====*
  <EncryptedData Id="wssecurity_encryption_id_8938640161692105181"
    Type="http://www.w3.org/2001/04/xmlenc#Content" xmlns="....#">
    <EncryptionMethod
      Algorithm="http://www.w3.org/2001/04/xmlenc#tripleDES-cbc"/>
    <CipherData>
      <CipherValue>
        Ijwtyy4rSyaNNB.....xWFwDUB2t0+GqQt/y13sc5P5G74=
      </CipherValue>
    </CipherData>
  </EncryptedData>
*=====*
  </soapenv:Body>
</soapenv:Envelope>

```

It is also possible that only a part of the SOAP message is signed or encrypted, for example, the credit card number. In addition, multiple signing or encryption using multiple certificates or keys is possible. Multiple signing and encryption is appropriate in the situation illustrated in Figure 21-3.

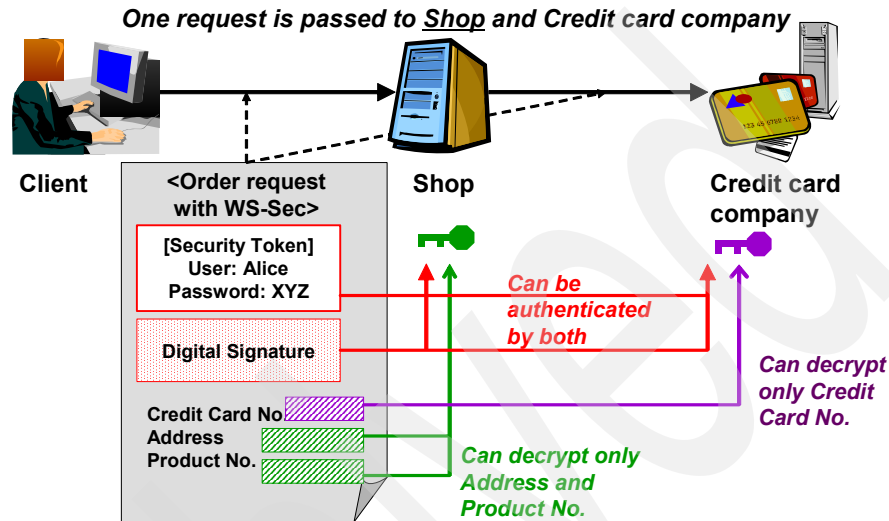


Figure 21-3 An example of applying integrity and confidentiality

In this example:

- ▶ The client sends a request to buy something through the Internet. The request message contains a credit card number, address, and product number.
- ▶ The credit card number should be revealed only to the credit card company. The client's address and the product number are only needed by the shop.
- ▶ In such case, the credit card number can be encrypted by a credit card company's key, and the client's address and the product number can be protected by encrypting for the shop.
- ▶ The three parts of the request message, the credit card number, the client's address, and the product number, are encrypted using an appropriate receiver's key.
- ▶ In addition, the client can sign the complete request message for integrity.

Using WS-Security, message confidentiality and integrity can be kept for multiple receivers within one SOAP message. This is one of the most important advantages of WS-Security over SSL, because with SSL, it is impossible to protect message security for multiple receivers.

Kerberos

Kerberos is well-known authentication mechanism that uses a third-party authority. The WS-SecurityKerberos specification describes how to use Web services security with Kerberos. It mainly consists of two sections: GSS-API Kerberos Interoperability and General Kerberos Interoperability.

The WS-SecurityKerberos implementation is not provided as part of the standard functions in WebSphere Application Server Version 6.0; however, a sample implementation is provided as a technical preview that supports GSS-API for Kerberos Interoperability.

Figure 21-4 illustrates the Kerberos scenario:

- ▶ Request ticket granting ticket (TGT) and a service ticket with Kerberos KDC.
- ▶ Request a security context token (SCT) from the security token service.
- ▶ Send the Web service request to the service with the security context token.

These three connections are realized by WS-Trust and WS-SecureConversation, which are described in “Establishing a security context” on page 455.

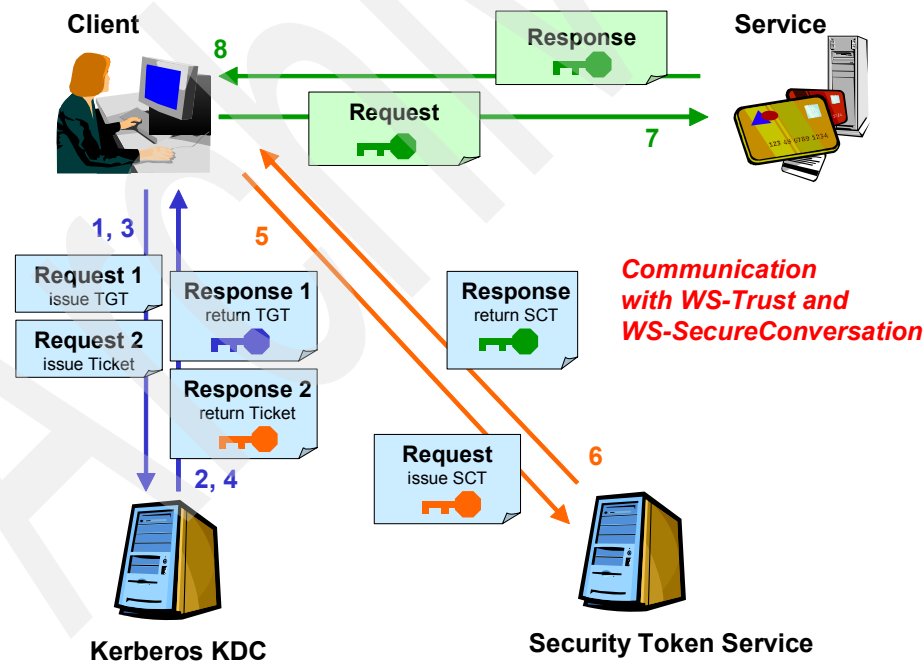


Figure 21-4 Kerberos through GSS-API scenario

For details about the implementation, refer to the documents of the WS-Security Kerberos technical preview:

- ▶ “The Kerberos Network Authentication Service (V5)”
<http://www.ietf.org/rfc/rfc1510.txt>
- ▶ *Web Services Security Kerberos Binding*
<ftp://www6.software.ibm.com/software/developer/library/ws-seckerb/WS-Security-Kerberos.pdf>

Establishing a security context

To keep integrity and confidentiality, a security context is established and shared between two parties. The security context is shared information that is available within a certain period of time. To establish and share a security context between two parties before communication, the two parties must exchange security credentials that are used to determine if they can trust the other party.

The WS-Trust specification describes how to issue and exchange security tokens, and the WS-SecureConversation specification describes how to establish a security context token between two parties. A secured communication with a security context token is realized with WS-Trust and WS-SecureConversation.

Figure 21-5 shows a typical scenario of a secure communication with a security context token between two parties.

- ▶ In this scenario, a connection from the client to a service that issues a security context is necessary to establish a security context token before sending a Web service request from a client.
- ▶ After a security context token is shared between the two parties, the security context can be used as a shared symmetric key of signing or encryption to keep integrity and confidentiality.

By using the shared security context for integrity and confidentiality, it is expected that the CPU load decreases as compared to a typical integrity and confidentiality scenario. The performance of signing and encryption using a shared symmetric key is better than using an asymmetric key, which is used in a typical scenario.

Which scenario should be applied for security is based on the Web services application characteristics.

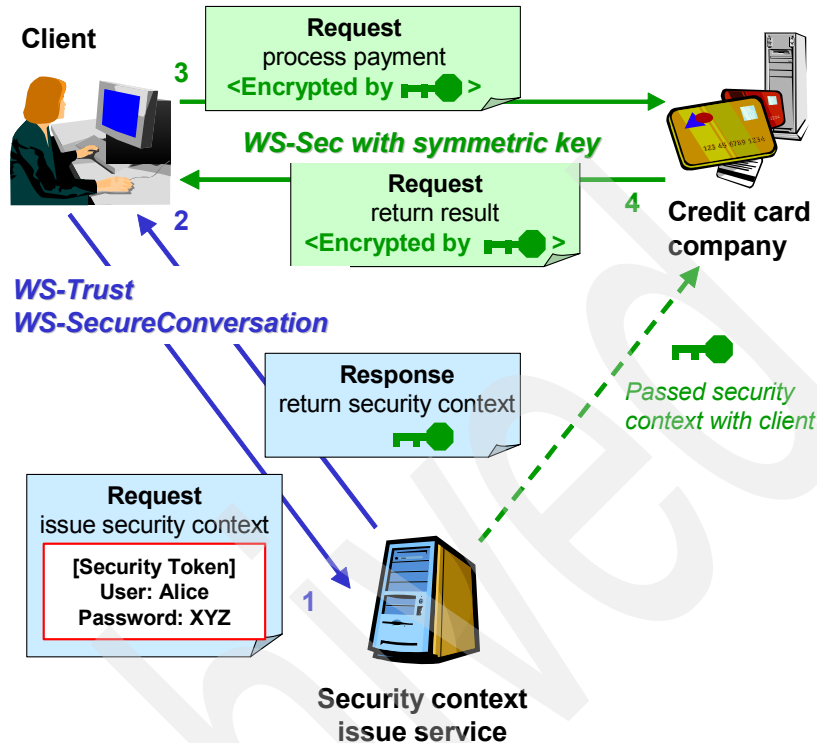


Figure 21-5 Scenario of secure communication with a security context token

The sample messages that are exchanged to establish a security token are shown in Example 21-5 and Example 21-6:

- ▶ A client request to issue a security context (Example 21-5).
- ▶ An established security context is sent to a client as a security context token (Example 21-6).

The implementation of a security context token is not provided in WebSphere Application Server Version 6.0, but an extensible architecture is provided as a capability for implementing a security context token in WebSphere Application Server V6.0.

Example 21-5 Example of requesting a security context token

```
<soapenv:Envelope xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
```



```

<soapenv:Header>
  <wsse:Security soapenv:mustUnderstand="1"
    xmlns:wsse=".....">
    ....[Integrity on SOAP Body].....
  </wsse:Security>
</soapenv:Header>
<soapenv:Body wsu:Id="wssecurity_signature_id_8828089944286341710"
  xmlns:wsu=".....">
  <wst:RequestSecurityToken
    xmlns:wsc="http://schemas.xmlsoap.org/ws/2004/01/sc"
    xmlns:wst="http://schemas.xmlsoap.org/ws/2004/01/trust">
*===== SECURITY CONTEXT TOKEN REQUEST =====*
    <wst:TokenType>
      http://schemas.xmlsoap.org/ws/2004/01/security/sc/sct
    </wst:TokenType>
    <wst:RequestType>
      http://schemas.xmlsoap.org/ws/2004/01/security/trust/Issue
    </wst:RequestType>
*=====
    <wst:Base>
      <wsse:KeyIdentifier
        ValueType="....."
        xmlns:wsse=".....">
          uuid:4598AD0E-0100-4000-E000-08840A0A0A02
        </wsse:KeyIdentifier>
      </wst:Base>
    </wst:RequestSecurityToken>
  </soapenv:Body>
</soapenv:Envelope>

```

Example 21-6 Example of issuing a security context token

```

<soapenv:Envelope xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Header>
    <wsse:Security soapenv:mustUnderstand="1"
      xmlns:wsse=".....">
      ....[Integrity on SOAP Body].....
    </wsse:Security>
  </soapenv:Header>
  <soapenv:Body wsu:Id="wssecurity_signature_id_4161873387039194833"
    xmlns:wsu=".....">
    <wst:RequestSecurityTokenResponse
      xmlns:wsc="http://schemas.xmlsoap.org/ws/2004/01/sc"
      xmlns:wsse="....."
      xmlns:wst="http://schemas.xmlsoap.org/ws/2004/01/trust"

```

```

xmlns:wsu="....">
.....
<wst:RequestedSecurityToken>
*===== SECURITY CONTEXT TOKEN =====*
  <wsc:SecurityContextToken
    xmlns:wsc="http://schemas.xmlsoap.org/ws/2004/01/sc"
    xmlns:wsu="....">
      <wsu:Identifier>
        uuid:4598B01B-0100-4000-E000-08840A0A0A02
      </wsu:Identifier>
      <wsu:Created>2004-11-17T17:14:22.235Z</wsu:Created>
      <wsu:Expires>2004-11-18T05:14:22.235Z</wsu:Expires>
    </wsc:SecurityContextToken>
*=====
  </wst:RequestedSecurityToken>
  <wst:RequestedProofToken>
    .....
  </wst:RequestedProofToken>
</wst:RequestSecurityTokenResponse>
</soapenv:Body>
</soapenv:Envelope>

```

Features of WS-Security in Application Server V6.0

The WS-Security in WebSphere Application Server Version 6.0 supports many functions based on the latest specifications. In addition, some extension functions are provided. In this section, we describe these supported functions and the architecture of the WS-Security configuration.

Supported specifications

WebSphere Application Server 6.0 is based on the implementation of WS-Security feature in the following OASIS specification and profiles.

- ▶ Web Services Security: SOAP Message Security 1.0 (March 2004):
<http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf>
- ▶ Web Services Security: UsernameToken Profile 1.0 (March 2004):
<http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0.pdf>
- ▶ Web Services Security: X.509 Certificate Token Profile V1.0 (March 2004):
<http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0.pdf>

Table 21-1 lists the detailed supported items in each specification and profile.

Table 21-1 WebSphere Application Server V6.0 supported items in each specification

Feature	Support
SOAP Message Security V1.0 specification	
Security tokens	Two default security tokens: ▶ Username token and binary security token Custom token
Token references	Five types of token reference patterns: ▶ Direct reference ▶ Key identifier ▶ Key name ▶ Embedded reference ▶ X509 issuer name and serial number
Signature features	Three signature method algorithms: ▶ HMAC-SHA1 ▶ DSA with SHA1 ▶ RSA with SHA1 One digest method algorithm: ▶ SHA1 Four canonicalization algorithms: ▶ Canonical XML with/without comments ▶ Exclusive XML Canonicalization with/without comments Six transform algorithms: ▶ Exclusive XML Canonicalization without Comments ▶ STR Dereference Transform ▶ XPath Transform ▶ Enveloped Signature ▶ XPath Filter2 ▶ Decryption Transform
Encryption features	Four data encryption algorithms: ▶ Triple DES in CBC ▶ AES 128/192/256 in CBC Five key encryption algorithms: ▶ RSA Version 1.5 ▶ Triple DES key Wrap ▶ AES 128/192/256 Key Wrap
Timestamp	Inserted within WSS security header

Feature	Support
Error handling	SOAP fault handling
Username Token Profile V1.0	
Password types	Text
Token references	Direct reference
X.509 Token Profile V1.0	
Token types	▶ X.509 Version 3: Single Certificate ▶ X.509 Version 3: X509PKIPathv1 without CRLs ▶ X.509 Version 3: PKCS7 with/without CRLs (IBM JDK supports both, Sun JDK supports only “without CRL”)
Token references	▶ Key identifier ▶ Direct reference ▶ Custom reference

Unsupported specifications

The unsupported specifications and profiles are:

- ▶ WS-SecureConversation
- ▶ WS-Trust
- ▶ SAML token profile
- ▶ Kerberos token profile
- ▶ REL Token Profile
- ▶ SwA (SOAP with Attachments) Profile
- ▶ WS-I Basic Security Profile
- ▶ JSR 105, JSR 106, JSR 183

Extensions in WebSphere Application Server V6.0

In this section, we describe the WebSphere Application Server Version 6.0 extension features for WS-Security.

Identity assertion

As described in “Authentication” on page 448, identity assertion (IDAssertion) is available in WebSphere Application Server Version 6.0. In a secure environment, such as an intranet or SSL, it is useful to only send the client (caller) identity without credentials (such as password) together with other trusted credentials (such as the intermediary server identity).

When applying IDAssertion, a requester credential is verified by an intermediate trusted party. If a caller credential is valid, the credential is removed from the request message and the request message is transferred to an endpoint server with a trusted intermediary's identity and credential. Figure 21-6 illustrates the architecture of IDAssertion.

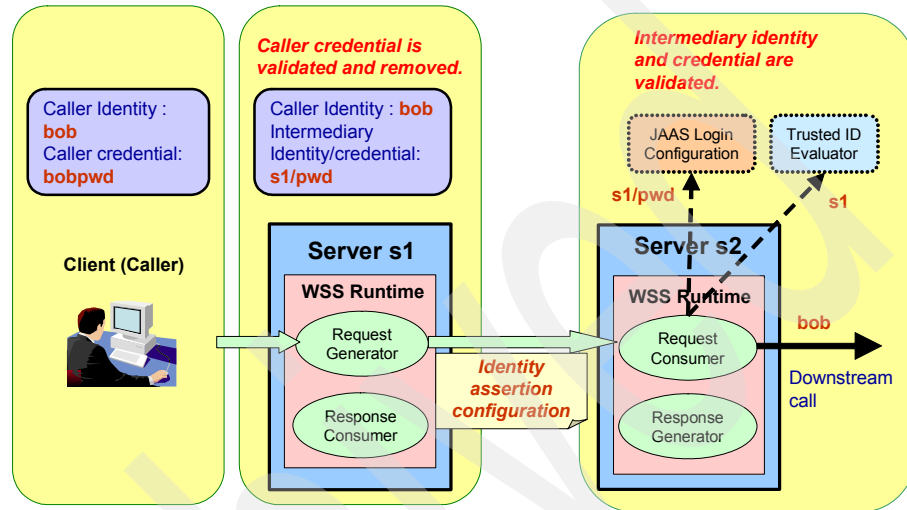


Figure 21-6 Identity assertion architecture

We support two ways of sending a caller identity to an endpoint server:

- ▶ Caller's username token without the caller's password
- ▶ Caller's X.509 certificate token

For the latter, we use the distinguished name in the certificate as a caller identity.

Trust mode

There are also three trust modes of sending the intermediate server identity and credential. We define these three methods as:

- ▶ **None**—The intermediate server does not send its own security token because it is trusted by a endpoint server already.
- ▶ **BasicAuth**—The intermediate server sends a username token with its own user name and password to the endpoint server. The endpoint server authenticates the username token and validates the trust based on the TrustedIDEvaluator implementation (which implements the TrustedIDEvaluator interface). The default implementation is provided in WebSphere Application Server Version 6.0, and a user can implement their own evaluator.

- ▶ **Signature**—The intermediate server signs the caller credential by its own X.509 certificate, and its certificate token is sent to the endpoint server. The endpoint server verifies the signature and validates the X.509 certificate token. The identity (distinguished name) from the X.509 certificate token, which is used for the signature, is validated based on the `TrustedIDEvaluator` implementation.

Pluggable token architecture

WebSphere Application Server Version 6.0 supports some types of tokens as a default: username token, X.509 certificate token, and LTPA token. Additionally, WebSphere Application Server provides an architecture for using a custom token that is defined by a user. According to this pluggable token architecture, a user can specify not only a default provided token, but also a custom token implementation.

To implement a custom token, a user has to prepare four classes:

- ▶ For generating a custom token, a custom token generator class (`com.ibm.wsspi.wssecurity.token.TokenGeneratorComponent` interface) and a callback handler class (`javax.security.auth.callback.CallbackHandler` interface) are necessary.
- ▶ For consuming a custom token, a custom token consumer class (`com.ibm.wsspi.wssecurity.token.TokenConsumerComponent` interface) and a JAAS login module (`javax.security.auth.spi.LoginModule` interface) are necessary.

`TokenGeneratorComponent` and `TokenConsumerComponent` are specific interfaces in WebSphere Application Server Version 6.0:

- ▶ A `TokenGenerator` class receives the contents of the token from the `CallbackHandler` class and generates a token object to pass to the WS-Security (WSS) runtime.
- ▶ A `TokenConsumer` class receives the token from the WSS runtime and retrieves the contents of the token to pass it to the JAAS login module. The login module validate the contents of the token and returns the result of the validation. If the token is valid, the `TokenConsumer` passes the result to the WSS runtime, and the process at the consumer continues. Figure 21-7 shows the pluggable token architecture.

You can refer to detailed information about the pluggable token architecture in the *WebSphere Application Server Version 6.0 Information Center*.



WebSphere Application Server Version 5.x can sign and encrypt only limited target elements, such as the SOAP body, timestamp, or security token. The target of the signature and encryption can be specified only by predefined keywords, so a user cannot sign or encrypt an arbitrary element in Version 5.x.

► Keywords for signing:

Chapter 21. Securing Web services 463

to	To element that is inserted by the WS-Addressing runtime
action	Action element that is inserted by the WS-Addressing runtime
relatesto	RelatesTo element that is inserted by the WS-Addressing runtime
► Keywords for encrypting:	
bodycontent	Contents of SOAP body element
username token	Username token element
digestvalue	Digest value element that is included in the signature element

To specify a custom target that cannot be defined using a keyword, you have to use an XPath expression. For example, if you want to specify the `<text>` element in Example 21-7, the required XPath expression is shown in Example 21-8.

Example 21-7 A sample SOAP message without WS-Security

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Header/>
  <soapenv:Body>
    <pingns:Ping xmlns:pingns="http://xmlsoap.org/Ping">
      <text>this is the custom target text</text>
    </pingns:Ping>
  </soapenv:Body>
</soapenv:Envelope>
```

Example 21-8 XPath expression for text element

```
/*[namespace-uri()='http://schemas.xmlsoap.org/soap/envelope/' and
  local-name()='Envelope']
/*[namespace-uri()='http://schemas.xmlsoap.org/soap/envelope/' and
  local-name()='Body']
/*[namespace-uri()='http://xmlsoap.org/Ping' and local-name()='Ping']
/*[namespace-uri()=' ' and local-name()='text']
```

LTPA token

WebSphere Application Server Version 6.0 defines the LTPA token, which is a specific binary security token. Example 21-3 on page 450 shows an example of an LTPA token. The LTPA token implementation is provided as a default, and the

token is verified by the WebSphere architecture if you specify the token for authentication.

Timestamp extension

A timestamp can be attached in the target element of signature and encryption to put a lifetime on the signed and encrypted element. If the timestamp is specified when an element is signed, a timestamp is attached to the target element, and the target element with the timestamp is signed. Because this is an extension of WebSphere Application Server Version 6.0, other vendor implementations might not be able to consume the message generated with the additional timestamp inserted in the message.

Nonce extension

Similar to the timestamp, a nonce (a randomly generated value) can be inserted to the signing or encrypting target element. This is to reduce the chance of a reply attack. However, this is an extension of Web Sphere Application Server Version 6.0; other vendors might not be able to consume messages with nonce inserted in the target element.

Distributed nonce cache

The distributed nonce cache makes it possible to replicate nonce data among servers in a WebSphere Application Server cluster. If nonce elements are in a SOAP header, all nonce values are cached by the server in the cluster. If the distributed nonce cache is enabled, the cached nonce values are copied to other servers in the same cluster. Then, if the message with the same nonce value is sent to (one of) other servers, the message is rejected. A received nonce cache value is cached and replicated in a push manner among other servers in the cluster with the same replication domain. The replication is an out-of-process call and, in some cases, is a remote call; therefore, there is latency when the content of the cache in the cluster is updated.

Figure 21-8 shows a schematic of the nonce cache:

- ▶ A SOAP client sends a message with nonce abc to a server wssec01.
- ▶ The server caches the value and pushes it to the other server wssec02.
- ▶ If the client sends the message with nonce abc to a server wssec02 after a certain time frame, the message is rejected, and a `SoapSecurityException` is thrown by the server wssec02.
- ▶ If the client sends the message with another nonce value of xyz, the message is accepted, and the value is cached by the server wssec02.

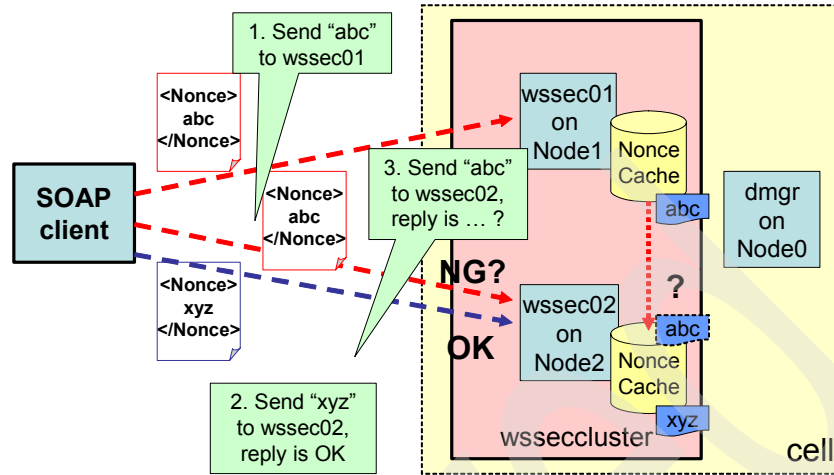


Figure 21-8 Distributed nonce cache

Certificate caching

To improve performance, WebSphere Application Server Version 6.0 supports caching of received certificates in the local cache.

Figure 21-9 shows the mechanism of certificate caching.

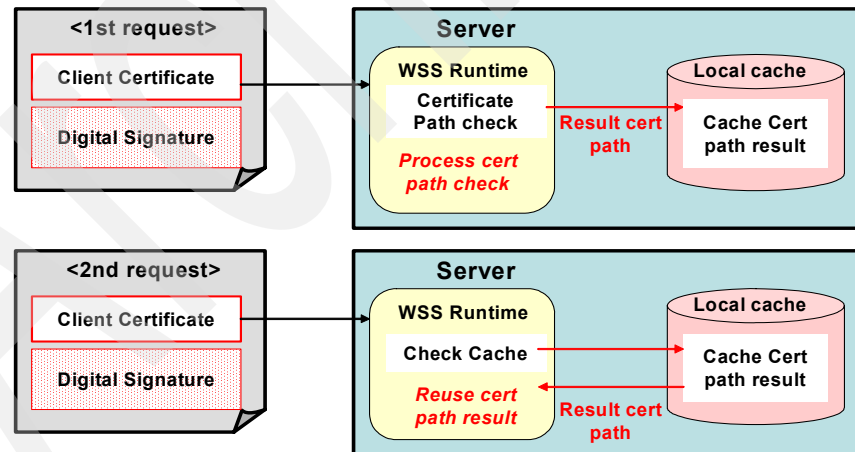


Figure 21-9 Certificate caching

In this example of certificate caching:

- A client sends a signed request message to the server.

- ▶ The server verifies the client certificate and caches the results in the local cache.
- ▶ In the next request from the same client, the cached verification is used instead of verifying the certificate again.

By caching and reusing the verification certificate, the server does not have to verify the same certificate many times. Verification of certificates is costly, so the cache can help performance of the server, especially in the case where clients send requests many times. The certificates are cached in the local cache and are available during the time the server is running. When the server is restarted, the cache is removed.

Architecture and deployment model

In this section, we describe the WS-Security implementation architecture and the underlying configuration.

High-level architecture

WebSphere Application Server Version 6.0 uses an extension of the JSR 109 deployment descriptor and binding deployment model to implement WS-Security. The WS-Security runtime is implemented as handlers. These handlers are registered to different parts of a Web service. Figure 21-10 shows the handlers and their responsibilities:

1. At the **client** side, the **request security handler** is used to generate the required security headers in the SOAP message before the SOAP message is sent to the server. These security constraints are defined in the client configuration deployment descriptor.
2. At the **server** side, the **request security handler** is invoked to verify that the SOAP message complies with all the security constraints specified by the server deployment descriptors prior to dispatching the message to the Web service implementation.
3. At the **server** side, the **response security handler** is invoked to generate the SOAP security header information before the message is sent back to the client. The nature of that information is also specified in the server deployment descriptors.
4. At the **client** side, the **response security handler** is called to manage the SOAP message security information to verify that the message complies with the client deployment descriptor specifications before the message is passed to the client implementation.

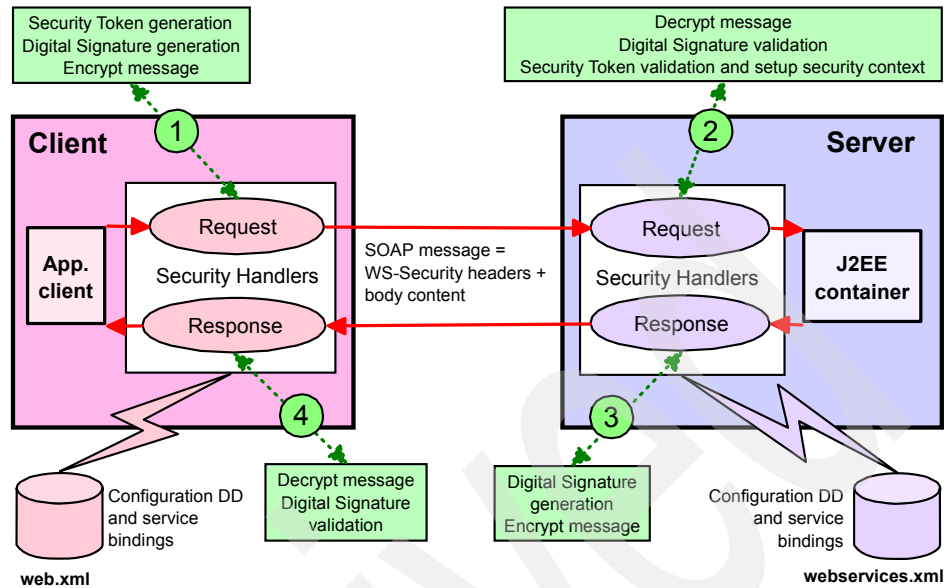


Figure 21-10 Web services message security architecture

When the requirements of the security constraints are not satisfied for a SOAP message, a SOAP fault response, as defined in the SOAP specification, is sent to the requester. This behavior is valid when no security constraints are applied to the server answer.

If any security information is required in the server response, the client receives the fault fired by its own response handler. This handler generates a new SOAP fault with the lack of security information received by the server response. Although the original fault usually accompanies the message, we have lost the security constraints. Figure 21-11 shows this inappropriate behavior.

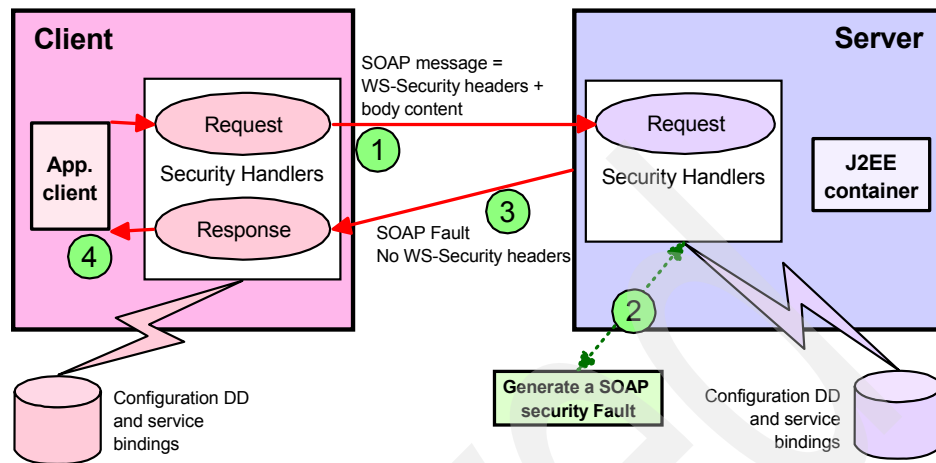


Figure 21-11 SOAP message flow with a SOAP fault error in the server

The steps are:

1. The client request handler generates the security information, filling the header of the SOAP message.
2. The server request handler validates the security information provided by the client. If this information does not match the requirements, a security SOAP fault is generated by the server security handler.
3. The generated security SOAP fault message is sent to the client. This message is sent without passing through the server response handler. Therefore, there is no security header in the SOAP message sent to the client.
4. When the client receives SOAP fault message, the response handler does not process the message, because there is no constraint for SOAP faults. Then, the SOAP fault is passed to the client application.

Configuration structure

The WS-Security constraints are specified in the IBM extension of the Web services deployment descriptors and bindings. The WS-Security deployment descriptor and binding are based on a Web service port. Each Web service port can have its own unique WS-Security constraints defined. The WS-Security requirements can be defined outside of the application business logic and the separation of roles; the application developer can just focus on the business logic, and the security expert can (later) specify the security requirements.

There are two sets of security handler configurations on the client side and two sets on the server side.

Client side

On the client side:

- ▶ **Request generator**—Defines the constraints of the request security handler on the outgoing SOAP request message, such as generating a SOAP message request with WS-Security (signature, encryption, and attach security tokens).
- ▶ **Response consumer**—Defines the constraints of the response security handler on the incoming SOAP response message, such as making sure that required integrity parts are signed (and verifying the signature) and that the required confidential parts are encrypted (and performing decryption), and validates security tokens.

Server side

On the server side:

- ▶ **Request consumer**—Defines the constraints of the request security handler on the incoming SOAP request message, such as making sure that required integrity parts are signed (and verifying the signature) and that the required confidential parts are encrypted (and performing decryption), and validates the security tokens and sets up the WebSphere security context with the appropriate identity.
- ▶ **Response generator**—Defines the constraints of the response security handler on the outgoing SOAP response message, such as generating a SOAP message request with WS-Security (signature, encryption, and attach security tokens).

The WS-Security requirements defined in the request generator must match the request consumer, and the response generator must match the response consumer. Otherwise, the request or response will be rejected, because the WS-Security constraints cannot be met at the request consumer and response consumer. WS-Security requires security policy negotiation or exchange between the client and server. Based on the exchanged security policy, these four security configurations should be defined.

WS-Security configuration files

The WS-Security constraints are defined in the IBM extension of the J2EE Web services deployment descriptor. There are four configuration files, which are the application-level deployment descriptor extensions for a client and a server, and binding files for a client and a server (Figure 21-12).

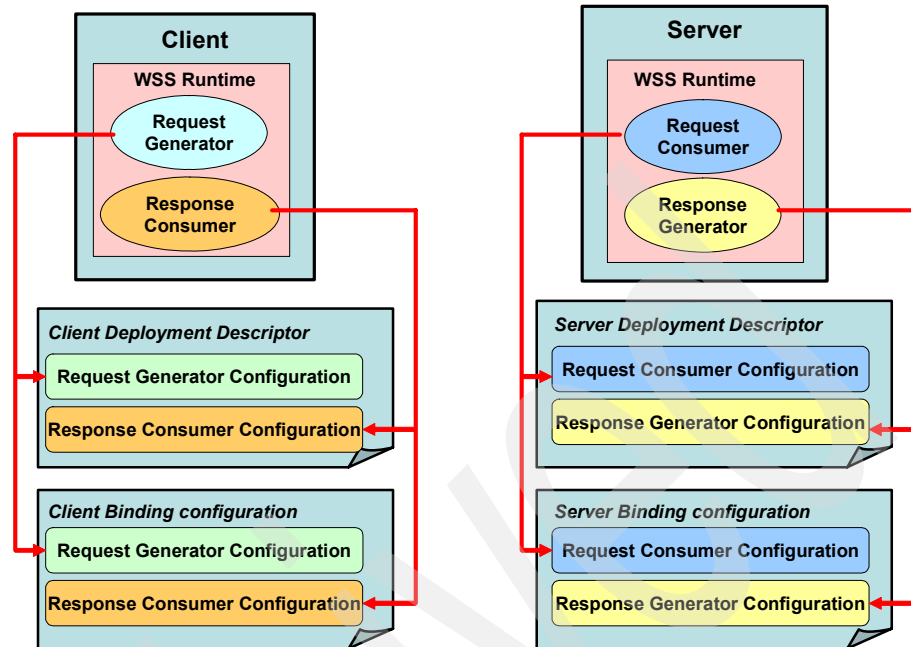


Figure 21-12 Structure of WS-Security configuration files

The configuration files are:

- ▶ **Client deployment descriptor extension** file—Includes request generator and response consumer constraints:
`ibm-webservicesclient-ext.xmi`
- ▶ **Client binding** configuration file—Includes how to apply request generator and response consumer constraints:
`ibm-webservicesclient-bnd.xmi`
- ▶ **Server deployment descriptor extension** file—Includes request consumer and response generator constraints:
`ibm-webservices-ext.xmi`
- ▶ **Server binding** configuration file—Includes how to apply request consumer and response generator constraints:
`ibm-webservices-bnd.xmi`

The deployment descriptor extension specifies *what* security constraints are required, for example, signing the body and encrypting the username token. The binding files specifies *how* to apply the required security constraints defined in the deployment descriptor extension, for example, which keys are used for signing or encryption and which security token is inserted.

These deployment descriptor extension and binding files define the application-level security constraints and they belong to each application.

Platform-level configuration

Besides the application-level constraints, WebSphere Application Server Version 6.0 has a cell-level (only for Network Deployment) and server-level WS-Security configuration, which is called a *platform-level configuration*:

- ▶ These configurations are global for all applications and include some configurations only for V5.x applications and some only for V6.x applications.
- ▶ In the platform configuration, general properties and additional properties can be specified, and the default binding is included. The default binding can be used as an application-level binding configuration so that applications do not have to define the binding in the application.
- ▶ There is only one set of default bindings that can be shared by multiple applications. This is only available for Application Server V6.0 applications.

Figure 21-13 shows how to be use the default binding configuration. Application EAR1 has its own binding file in the application. However, applications EAR2 and EAR3 do not have binding files, so the default binding is used for these two applications.

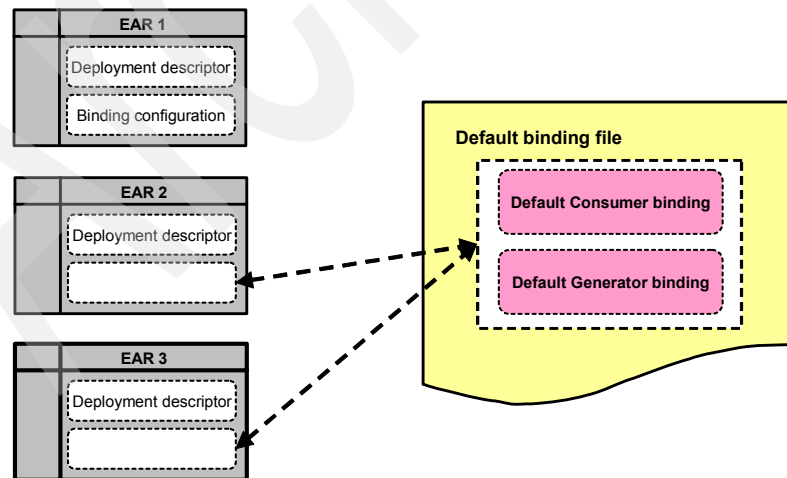


Figure 21-13 Default binding configuration

Therefore, binding configuration files can be specified at three levels: application, server, and cell. Each binding configuration overrides the next higher one. For any deployed application, the nearest configuration binding is applied (Figure 21-14).

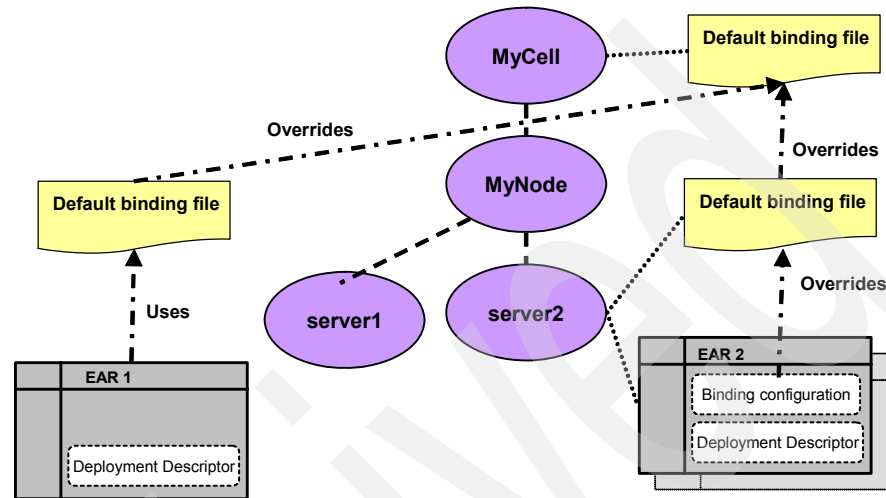


Figure 21-14 Overriding binding configurations

The visibility scope of the binding depends on where the file is located. If the binding is defined in an application, its visibility is scoped to that particular application. If it is located at the server level, the visibility scope is all applications deployed on that server. If it is located at the cell level, the visibility scope is all applications deployed on all servers of the cell.

Editing configuration files

The WS-Security configuration files can be edited by tools. Rational Application Developer V6.0 and the Application Server Toolkit should be used to edit the application deployment descriptor extension and binding files.

The WebSphere Application Server V6.0 administrative console can be used to edit the application binding that is deployed on the server and the platform configurations.

Figure 21-15 illustrates the configuration architecture of WebSphere Application Server Version 6.0.

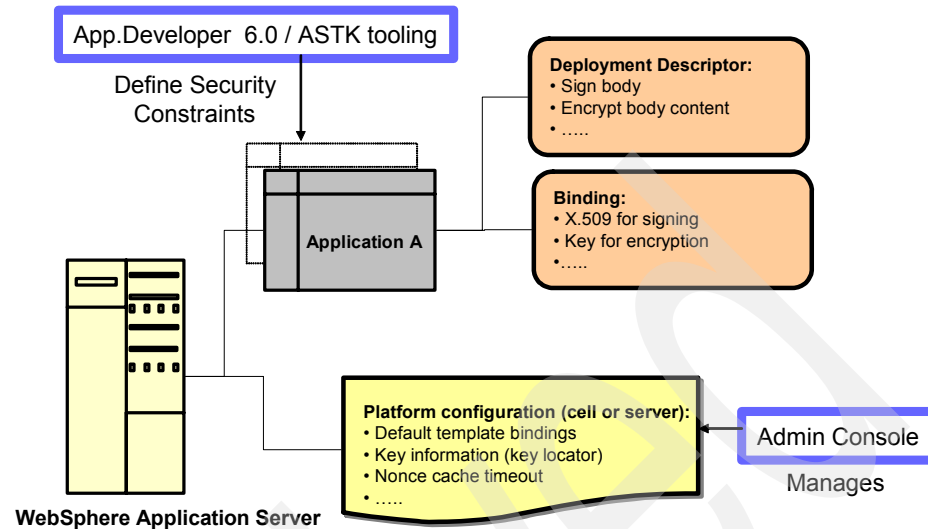


Figure 21-15 Configuration architecture of WS-Security

Development of WS-Security

To secure a Web service application by WS-Security, it is necessary to define WS-Security configurations using Rational Application Developer V6.0 or the Application Server Toolkit. If the appropriate WS-Security configurations exist in the application EAR file, WS-Security runtime is invoked when the SOAP message is outgoing or incoming and the SOAP messages are secured.

To explain how to define the WS-Security configuration, we use the JavaBean Web service application created in Chapter 16, “Develop Web services with Application Developer V6.0” on page 273.

To follow this example, the JavaBean weather forecast Web service must be installed in the workspace. Either you followed the instructions in “Creating a Web service from a JavaBean” on page 275, or you can import the application from \SG246461\sampcode\zSolution\EARfiles. These two enterprise applications are used:

- ▶ WeatherJavaBeanServer, with WeatherJavaBeanWeb and WeatherBase (utility module)
- ▶ WeatherJavaBeanClient, with WeatherJavaBeanClientWeb

How to define WS-Security configuration

This section provides information about how to configure WS-Security to apply message-level security to the Web service. Many steps are necessary to configure WS-Security, so an outline of the configuration steps is given here. Then, we describe how to configure each security mechanism: authentication, integrity, and confidentiality.

Outline of how to secure Web Services

The outline of securing Web services describes what should be done first and where can define WS-Security configurations.

You can use the GUI editor for specifying WS-Security configurations for both the client and server. As you can see in Figure 21-12 on page 471, there are four WS-Security configuration files, and each of them contains both generator and consumer configurations. Application Developer V6.0 provides two GUI editors for WS-Security configuration: one for the client and one for the server. Each editor handles both the generator and the consumer sections for both the extension and the binding information.

Here is the outline of the steps used to configure WS-Security:

- ▶ Configure WS-Security for sending the request message (client)—Edit the `web.xml` file for the request generator with the Deployment Descriptor Editor.
- ▶ Configure WS-Security for receiving the request message (server)—Edit the `webservices.xml` file for the request consumer with Web Services Editor.
- ▶ Configure WS-Security for sending the response message (server)—Edit the `webservices.xml` file for the response generator with the Web Services Editor.
- ▶ Configure WS-Security for receiving the response message (client)—Edit the `web.xml` file for the response consumer with the Deployment Descriptor Editor.

Note: The client-side security information is accessed through the Deployment Descriptor Editor of the client module:

- ▶ For a Web client (servlet, JSP, JavaBean), we edit the `web.xml` file.
- ▶ For an EJB client (a session bean accessing a Web service), we edit the `ejb-jar.xml` file.
- ▶ For a J2EE application client, we edit the `application-client.xml` file.

In each editor, there are two tabs for WS-Security configuration:

- ▶ **WS Extension**—*What* security measures to apply
- ▶ **WS Binding**—*How* to apply the security measures

Editing the client configuration

For the client configuration, you access the GUI for the WS-Security configuration from the Deployment Descriptor Editor.

To open the GUI, expand the client project (WeatherJavaBeanClientWeb) in the Project Explorer and open (double-click) the deployment descriptor. You can also expand WebContent → WEB-INF and open the web.xml file directly.

When the Deployment Descriptor Editor opens, select the WS Extension or WS Binding page. The WS Extension page is for editing the client's deployment descriptor extension file, so you can specify what security is required. The WS Binding page is for editing the client's binding file, so you can specify how to apply the required security. Figure 21-16 shows the WS Extension page, and Figure 21-17 shows the WS Binding page in the Deployment Descriptor Editor.

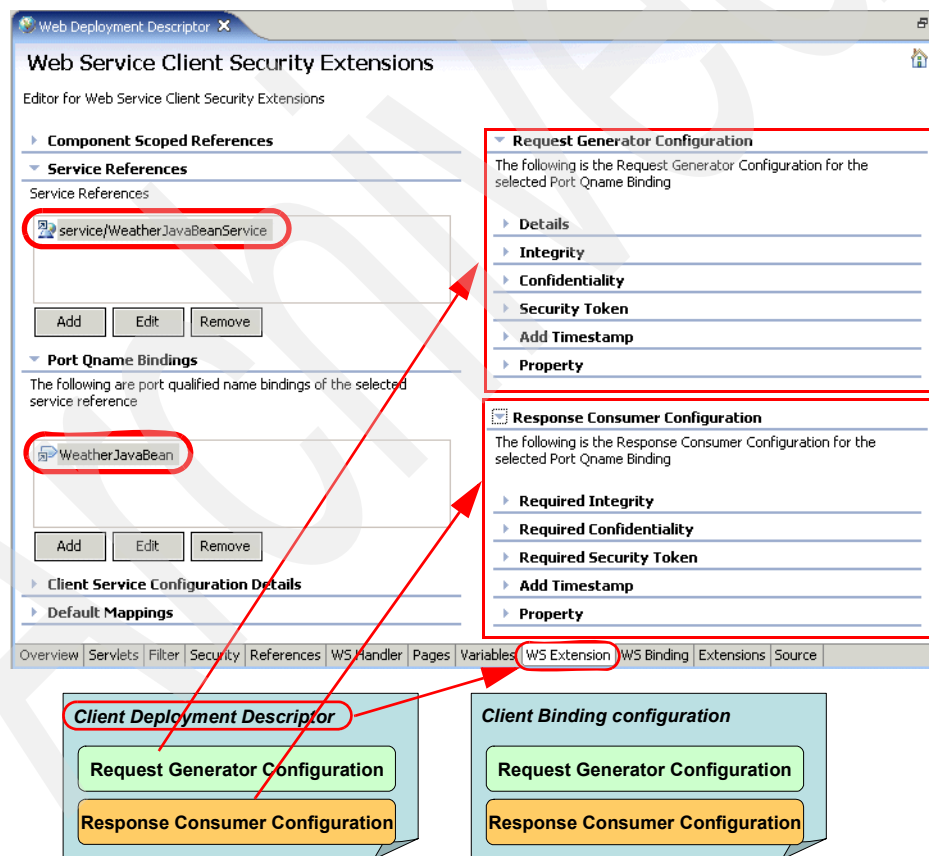


Figure 21-16 WS Extension page in the Deployment Descriptor Editor

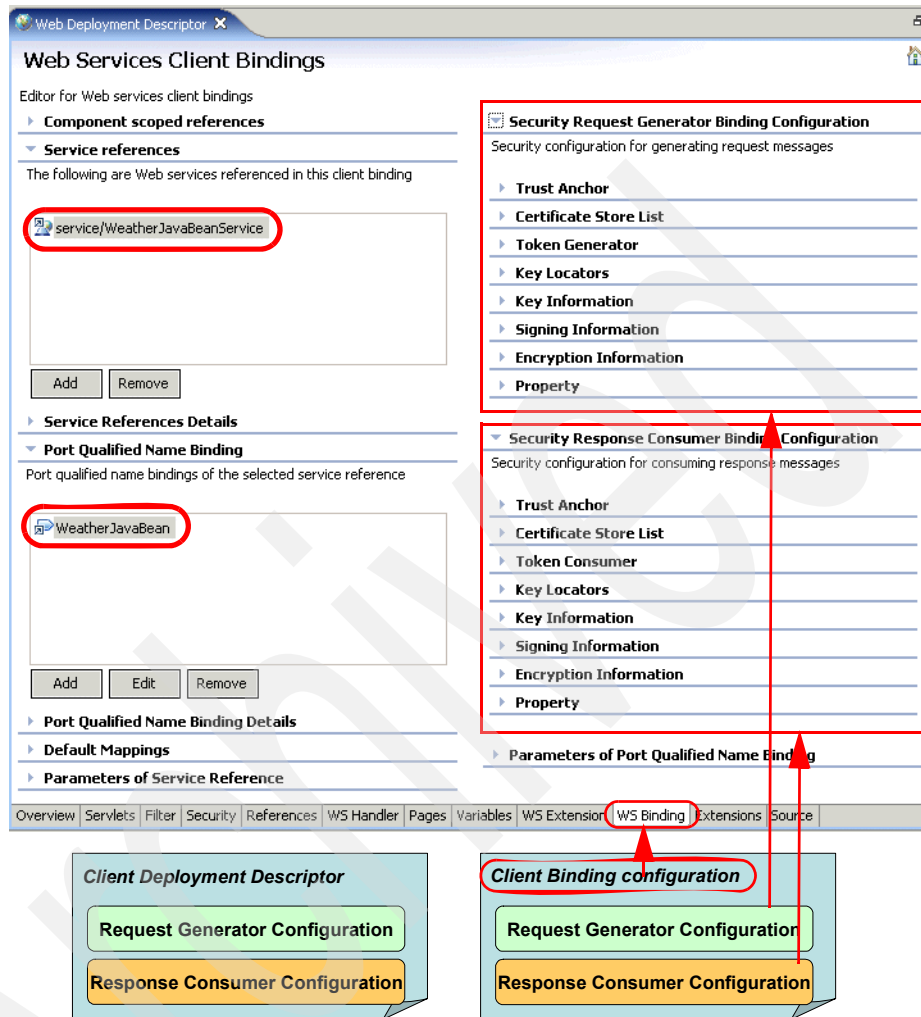


Figure 21-17 WS Binding page in the Deployment Descriptor Editor

Editing the server configuration

For the server configuration, you access the GUI for the WS-Security configuration using the Web Services Editor.

To open the GUI, expand the server project (WeatherJavaBeanWeb) in the Project Explorer and open (double-click) the webservices.xml file under WebContent → WEB-INF.

When the Deployment Descriptor Editor opens, select the Extensions or Binding Configurations page. The Extensions page is for editing the server's extension file, so you can specify what security is required. The Binding Configurations page is for editing the server's binding file, so you can specify how to apply the required security. Figure 21-18 shows the Extensions page, and Figure 21-19 shows the Binding Configurations page in the Web Services Editor.

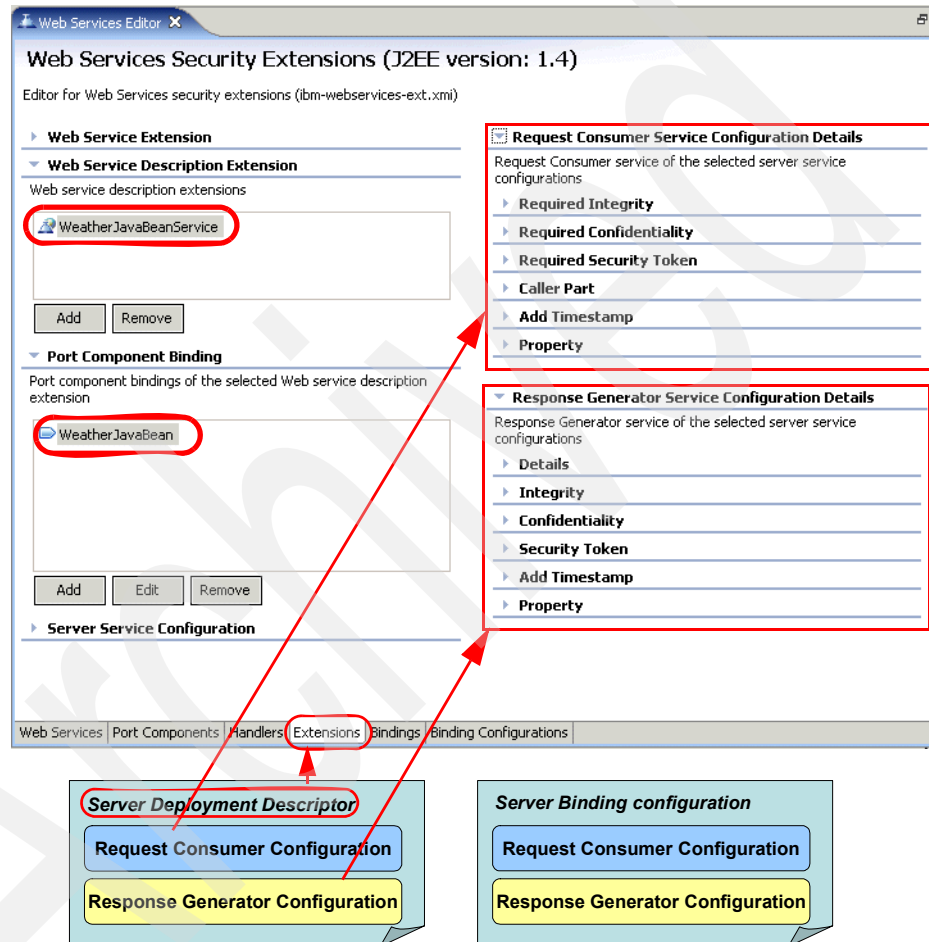


Figure 21-18 Extensions page in the Web Services Editor

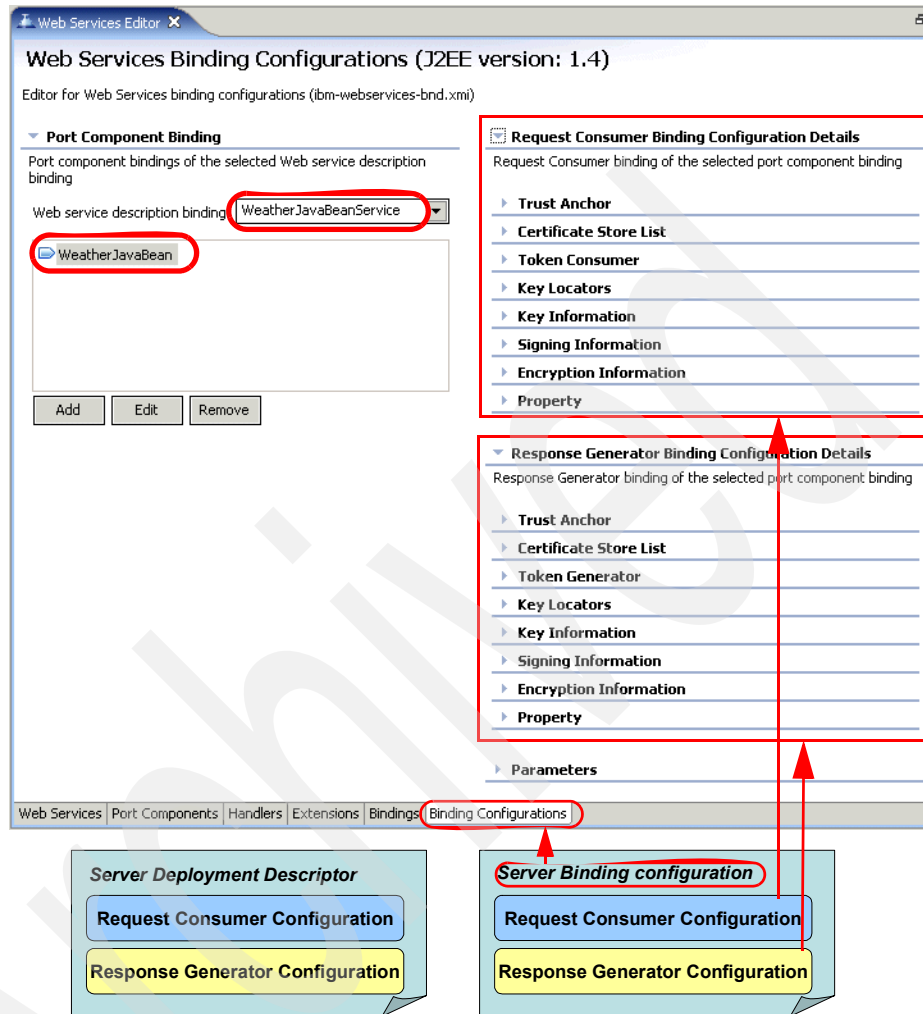


Figure 21-19 Bindings Configurations page in the Web Services Editor

Web services configuration editors

All WS-Security configurations at the application level can be edited using these two editors. There are many steps to specify WS-Security configurations from scratch, so let us first outline the steps.

WS-Security on the request message

To configure WS-Security on the request message (client send):

- Open the Deployment Descriptor Editor and select the *WS Extension* page.

- ▶ Select the service reference (under Service References) and the port name (under Port QName Bindings) on the left side of the page. The WS-Security configuration can be specified by port for each Web service.
- ▶ Specify items under *Request Generator Configuration* (security token, integrity, confidentiality, timestamp).
- ▶ On the *WS Binding* page, specify items under *Security Request Generator Binding Configuration* (matching what you defined on the WS Extension page).

To configure WS-Security on the request message (server receive):

- ▶ Open the `webservices.xml` file in the service project with the Web Services Editor and select the *Extensions* page.
- ▶ Select the matching Web service extension and port name under *Port Component Binding* on the left side of the page.
- ▶ Specify items under *Request Consumer Service Configuration Details*. These configuration items should match the client's configuration.
- ▶ On the *Binding Configurations* page, specify items under *Request Consumer Binding Configuration Details* (matching what you defined on the WS Extensions page).

WS-Security on the response message

To configure WS-Security on the response message, repeat this process by starting on the server side (Response Generator Binding Configuration Details) and matching the definition on the client side (Response Consumer Configuration).

Generating sample key stores

Before starting the WS-Security configuration, you have to prepare the client and server key stores to sign or encrypt the message. This section shows how to create a sample key store for testing WS-Security. If you want to apply WS-Security to a real application, you should prepare the appropriate certificate and keys. Do not use this sample key store for a real application.

WebSphere Application Server and Rational Application Developer provide sample key stores under:

```
<WAS60_HOME>\etc\ws-security\samples
<RAD60_HOME>\runtimes\base_v6\etc\ws-security\samples
```

These key stores can be used for testing application security, but should never be used in a production application.

Generating key stores

In this section, two key stores that contain the following keys are created by using the Java keytool:

Client key

Algorithm	RSA
Distinguished Name	CN=Client, OU=IBM, C=US
Key size	1024
Storepass	client
Storetype	JKS
Keypass	client_rsa
Alias	client_rsa
Key store file	client.jks
Certificate file	client_rsa.cer

Server key

Algorithm	RSA
Distinguished Name	CN=Server, OU=IBM, C=US
Key size	1024
Storepass	server
Storetype	JKS
Keypass	server_rsa
Alias	server_rsa
Key store file	server.jks
Certificate file	server_rsa.cer

Using the Java keytool

To generate sample key stores:

- ▶ Go to the directory where you want to generate the key stores. Here, we generate key stores in \SG246461\sampcode\mykeystore.
- ▶ Run the keytool in a DOS command prompt as follows. The keytool is located in <RAD60_HOME>\runtimes\base_v6\java\jre\bin.

```
set PATH=<RAD60_HOME>\runtimes\base_v6\java\jre\bin;%PATH%
```

- Generate a client RSA key:

```
keytool -genkey -keyalg RSA -keystore client.jks -storetype jks  
-storepass client -alias client_rsa -keypass client_rsa -dname  
"CN=Client, OU=IBM, C=US" -keysize 1024 -validity 1460
```

- Generate a server RSA key:

```
keytool -genkey -keyalg RSA -keystore server.jks -storetype jks  
-storepass server -alias server_rsa -keypass server_rsa -dname  
"CN=Server, OU=IBM, C=US" -keysize 1024 -validity 1460
```

- Export a client RSA certificate:

```
keytool -export -keystore client.jks -storetype jks -storepass client
-alias client_rsa -file client_rsa.cer
```

- Import a client RSA certificate to a server key store:

```
keytool -import -noprompt -keystore server.jks -storetype jks -storepass
server -alias client_rsa -file client_rsa.cer
```

- Export a server RSA certificate:

```
keytool -export -keystore server.jks -storetype jks -storepass server
-alias server_rsa -file server_rsa.cer
```

- Import a server RSA certificate to a client key store:

```
keytool -import -noprompt -keystore client.jks -storetype jks -storepass
client -alias server_rsa -file server_rsa.cer
```

- You should now have two key store files and two certificate files:

Client key store file	client.jks
Server key store file	server.jks
Client certificate file	client_rsa.cer
Server certificate file	server_rsa.cer

- To verify the information of the keys in these key stores, run the following commands:

```
keytool -list -keystore client.jks -storepass client -v
keytool -list -keystore server.jks -storepass server -v
```

The created key store information is shown in Example 21-9 and Example 21-10. The client key store has both a self-signed certificate and a server certificate that can be trusted by a client. The server key store also has both certificates, and the client certificate can be trusted by a server.

Example 21-9 Client key information

```
Keystore type: jks
Keystore provider: IBMJCE
```

Your keystore contains 2 entries

```
Alias name: client_rsa
Creation date: Dec 20, 2004
Entry type: keyEntry
Certificate chain length: 1
Certificate[1]:
Owner: CN=Client, OU=IBM, C=US
Issuer: CN=Client, OU=IBM, C=US
Serial number: 42dbe06f
Valid from: 7/18/05 10:01 AM until: 7/17/09 10:01 AM
```

Certificate fingerprints:

MD5: 09:F3:EF:57:04:6B:35:E5:6F:73:6C:BE:E4:A9:AD:97

SHA1: 34:74:F7:3C:E8:B3:8F:8B:16:FC:20:A1:38:EC:F8:96:36:AF:8C:8E

Alias name: server_rsa

Creation date: Dec 20, 2004

Entry type: trustedCertEntry

Owner: CN=Server, OU=IBM, C=US

Issuer: CN=Server, OU=IBM, C=US

Serial number: 42dbe072

Valid from: 7/18/05 10:01 AM until: 7/17/09 10:01 AM

Certificate fingerprints:

MD5: 72:96:D3:84:39:3F:15:DC:F7:A4:66:E8:5F:2B:2A:87

SHA1: 66:23:2D:5E:57:E3:F7:C7:F8:7F:C8:70:6D:82:AD:81:8E:69:73:96

Note: If you specify client self-signed certificate information in a token consumer (refer to “Configuring the server for security token” on page 489), specify as issuerName "CN=Client, OU=IBM, C=US" and as issuerSerial "42dbe06f" in case of this sample key store. If you generate a client key store by yourself, you should confirm these values by running the keytool.

Example 21-10 Server key information

Keystore type: jks

Keystore provider: IBMJCE

Your keystore contains 2 entries

Alias name: client_rsa

Creation date: Dec 20, 2004

Entry type: trustedCertEntry

Owner: CN=Client, OU=IBM, C=US

Issuer: CN=Client, OU=IBM, C=US

Serial number: 42dbe06f

Valid from: 7/18/05 10:01 AM until: 7/17/09 10:01 AM

Certificate fingerprints:

MD5: 09:F3:EF:57:04:6B:35:E5:6F:73:6C:BE:E4:A9:AD:97

SHA1: 34:74:F7:3C:E8:B3:8F:8B:16:FC:20:A1:38:EC:F8:96:36:AF:8C:8E

Alias name: server_rsa

Creation date: Dec 20, 2004

Entry type: keyEntry

Certificate chain length: 1

Certificate[1]:

Owner: CN=Server, OU=IBM, C=US
Issuer: CN=Server, OU=IBM, C=US
Serial number: 42dbe072
Valid from: 7/18/05 10:01 AM until: 7/17/09 10:01 AM
Certificate fingerprints:
MD5: 72:96:D3:84:39:3F:15:DC:F7:A4:66:E8:5F:2B:2A:87
SHA1: 66:23:2D:5E:57:E3:F7:C7:F8:7F:C8:70:6D:82:AD:81:8E:69:73:96

Note: If you specify server self-signed certificate information in a token consumer (refer to “Configuring the server for security token” on page 489), specify it as issuerName "CN=Server, OU=IBM, C=US" and as issuerSerial "42dbe072" in case of this sample key store. If you generate a server key store by yourself, you should confirm these values by running the keytool.

In our sample WS-Security configuration, we use the generated sample key stores that are located in \SG246461\sampcode\mykeystore. You can copy our key store files from \SG246461\sampcode\mykeystore\sample. Note that keystores expire after a certain time period; however, we set the period to four years.

Important: When using the keytool, only a self-signed certificate can be created. If you want to create a certificate path, you have to use another tool. More detailed information about creating certificates and keys is provided in *WebSphere Application Server V6: Security Handbook*, SG24-6316.

Authentication

To add an authentication mechanism in the client's request message, insert a security token in the message. To configure an authentication, you have to perform the following steps:

- ▶ Add a security token for the request generator configuration.
- ▶ Add a corresponding token generator for the specified security token in the Security Request Generator Binding Configuration.
- ▶ If you use identity assertion, specify security token or signature configuration according to a trust mode.
- ▶ Add the required security token in the Request Consumer Service Configuration Details.
- ▶ Add a corresponding token consumer for the specified required security token in the Request Consumer Binding Configuration Details.
- ▶ If you use identity assertion, specify the required security token or signature verification configuration according to a trust mode. Also, add a caller part in the Request Consumer Service Configuration Details.

In the following sections, we describe the detailed steps of specifying the security token and identity assertion.

Important: WebSphere Application Server V6.0 provides several types of security tokens for authentication, such as a username token and X509 certificate token. In this section, all possible choices are explained, but you can specify basic authentication by following the screen captures.

How to specify a security token

For basic authentication, a username token is inserted in the request message. You can select other types of security tokens for client authentication.

Configuring the client for a security token

To configure a security token to a request message sent by a client, open the `WeatherJavaBeanClientWeb` deployment descriptor and go to the **WS Extension** page under **Request Generator Configuration**:

- Expand *Security Token* and click *Add*. Enter a name, for example, *basicauth*.

Select a Token type from the drop-down list. The available choices are:

Username Username token with a user name and password

X509 certificate token Binary security token of X.509 certificate

X509 certificates in a PKIPath Binary security token of an ordered list of X.509 certificates packaged in a PKIPath

A list of X509 certificates and CRLs in a PKCS#7 Binary security token of a list of X.509 certificates and (optionally) CRLs packaged in a PKCS#7 wrapper

LTPAToken Binary security token of a Lightweight Third Party Authentication (LTPA) token

Custom token Custom-defined token

If you want basic authentication (Figure 21-20), select *Username* as the Token type. When you select a Token type, the Local name is filled in automatically. For Username and three types of X509 certificates, the URI is not necessary (leave it empty). If you select a custom token, you have to enter the URI and the Local name of the custom token manually.

- Click *OK*, and a security token is created. Save the configuration.

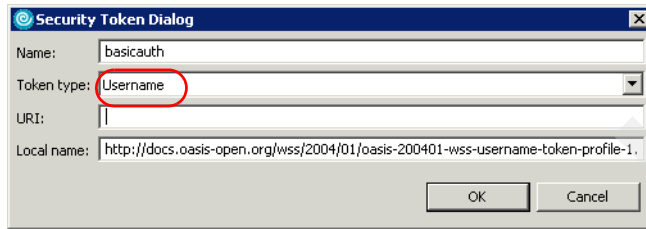


Figure 21-20 Security Token Dialog for specifying basic authentication

After specifying the security token, a corresponding token generator should be specified in the binding configuration. Go to the **WS Binding** page under **Request Generator Configuration**:

- To specify a token generator for a list of X.509 certificates and CRLs in a PKCS#7, expand *Certificate Store List* and click *Add* (for basic authentication, you do not specify this).

Enter any name. Add a CRL Path pointing to the CRL file. These specified CRLs are packaged in a PKCS#7 wrapper. Click *OK*, and a collection certificate store is created.

- Expand *Token Generator* and click *Add*.

- Input a Token generator name, for example, basicauthToken (Figure 21-21).
- Select a token generator class or input your custom token generator class name manually. You have to select a corresponding token generator class for the specified security token type. For example, if you select *Username* as the token type, you have to select the *UsernameTokenGenerator* class as the token generator class. The provided token generators are as follows:

Username—com.ibm.wsspi.wssecurity.token.UsernameTokenGenerator

X509—com.ibm.wsspi.wssecurity.token.X509TokenGenerator

LTPA—com.ibm.wsspi.wssecurity.token.LTPATokenGenerator

We select the *UsernameTokenGenerator* in Figure 21-21.

- For the security token, expand the drop-down list and select the security token defined on the WS Extension page.
- Select *Use value type*. Select the value type from the drop-down list that matches your selection, in our case, *Username Token*. This selection fills the local name and callback handler. If you specify a token generator for a custom token, select *Custom Token* as the value type and enter the URI

and Local name manually. In Figure 21-21, we select *Username Token* for basic authentication.

- Select the callback handler class or input your custom callback handler class name manually (for a custom token). Some provided callback handler classes can be selected from the list. The provided default callback handler classes are as follows:

NonPromptCallbackHandler	Enter user ID and password manually.
GUIPromptCallbackHandler	Request user ID and password by displaying a GUI prompt dialog box. This is useful for a J2EE application client.
X509CallbackHandler	Get an X.509 certificate for a key store file.
PkiPathCallbackHandler	Create an X.509 certificate and binary data without CRL using PKIPath encoding.
PKCS7CallbackHandler	Create an X.509 certificate and binary data with or without CRL using PKCS#7 encoding.
LTPATokenCallbackHandler	Get user credentials for the LTPA token from a user registry.
StdinPromptCallbackHandler	Prompt user for user ID and password on the command line.

In Figure 21-21, we select the `NonPromptCallbackHandler` for basic authentication.

- If the token generator is the `NonPromptCallbackHandler` or `LTPATokenCallbackHandler`, enter the user ID and password of the client.
- If the selected callback handler requires a key store (for example, the `X509CallbackHandler`, `PkiPathCallbackHandler`, and `PKCS7CallbackHandler`), select *Use key store* and specify the key store-related information. You have to specify the Key store storepass (password to access the key store), Key store path, and Key store type from the list. The available key store types are:

JKS	Used if you are not using Java Cryptography Extensions (JCE) and if your key store file uses the Java Keystore (JKS) format.
JCEKS	Used if you are using Java Cryptography Extensions.
PKCS11	Used if your key store file uses the PKCS#11 file format.
PKCS12	Used if your key store file uses the PKCS#12 file format.

- If you specified a key store, add the key. You can specify which key is used. Enter the alias of the key, the key pass (password to get the key from key store file), and the key name.

- If you have to specify the properties of the callback handler or token generator, add a Call back handler property or a Property.
- If you specify a token generator for a list of X.509 certificates and CRLs in a PKCS#7, select *Use certificate path settings* and select the *Certificate path reference*. If you select *Certificate path reference*, you can select the Certificate store from the drop down-list. Select one Collection certificate store name from the list, which is packed in PKCS#7.

Token Generator dialog

Token generator name: basicauthToken

Token generator class: com.ibm.wsspi.wsssecurity.token.UsernameTokenGenerator

Security token: basicauth

☒ Use value type

Value type: Username Token

Local name: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#UsernameToken

URI:

Call back handler: com.ibm.wsspi.wsssecurity.auth.callback.NonPromptCallbackHandler

User ID: David

Password: divaD

☐ Use key store

Key store storepass:

Key store path:

Key store type:

Key:

Alias	Key pass	Key name

Add Remove

Call back handler property:

Name	Value

Add Remove

Property:

Name	Value

Add Remove

☐ Use certificate path settings

☐ Certificate path reference

Trust anchor reference:

Certificate store reference:

☒ Trust any certificate

Figure 21-21 Token Generator dialog for specifying basic authentication

- Click *OK*, and a token generator is created. Save the configuration.

Configuring the server for security token

To configure how to receive a security token sent by client, open the `webservices.xml` file in the WeatherJavaBeanWeb project. Go to the **Extensions** page under **Request Consumer Service Configuration Details**:

- Select the port component, expand *Required Security Token* and click *Add*.
 - Enter a name of this required security token, for example, `reqtoken` (Figure 21-22).
 - Select a Token type from the drop-down list, matching the client specification. If a client's security token is the Username Token, you should select *Username* as the Token type. The URI and Local name are filled in automatically except when using a custom token.
 - Select the Usage type from the drop-down list. The available choices are *Required* or *Optional*. If you select *Required*, a SOAP fault is thrown if a required security token is not included in a client's request message. If you select *Optional*, the process of consuming a request message continues without throwing a SOAP fault.
 - Click *OK* and save the configuration.

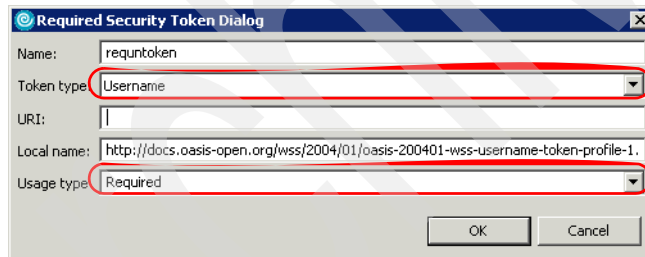


Figure 21-22 Required Security Token Dialog for specifying basic authentication

- Expand *Caller Part* under Request Consumer Service Configuration Details and click *Add*:
 - Enter a name of for the caller part, for example, `basicAuth` (Figure 21-23).
 - If a client's security token is used for signing or encryption, select the name of an Integrity or Confidentiality part from the drop-down list. The security token that is used for the selected integrity or confidentiality part is regarded as the client's security token. In our case, the security token for basic authentication is not used for signing or encryption.
 - If the client's security token is not used for signing or encryption, select a Token type of the security token. If the security token is Username Token,

select *Username* as the type. If you select Custom Token as the type, you have to specify a custom URI and Local name of the token.

- Click *OK*, and a caller is created. Save the configuration.

The image shows a 'Caller Part Dialog' window. The 'Token type' dropdown menu is highlighted with a red circle and contains the text 'Username'. The 'Name' field is 'basicAuth'. The 'Integrity or Confidentiality part' dropdown is empty. The 'URI' field is empty. The 'Local name' field contains the URL 'http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-user'. There is an unchecked checkbox for 'Use IDAssertion'. Below it, the 'Trust method name' dropdown is set to 'None'. The 'Integrity or Confidentiality part' dropdown is empty. The 'URI' field is empty. The 'Local name' field is empty. There are two sections for properties: 'Trust method property' and 'Property', each with a table with 'Name' and 'Value' columns and 'Add'/'Remove' buttons. At the bottom right are 'OK' and 'Cancel' buttons.

Figure 21-23 Caller Part Dialog for specifying basic authentication

After specifying the required security token, a corresponding token consumer should be specified in the binding configuration. Go to the **Binding Configurations** page under **Request Consumer Binding Configuration Details**:

- ▶ If you want to specify a token consumer for the three types of X.509 certificate tokens, expand *Trust Anchor* and click *Add* (not for basic authentication):
 - Enter a name for the trust anchor.
 - Specify the key store information (key store storepass, key store path, and key store type). If you trust any certificate, you do not have to specify this.
 - Click *OK*, and a trust anchor is created.
- ▶ If the token consumer is for an X.509 certificate token, and you want to specify an intermediate trusted certificate store or CRL, add a Collection

Certificate Store under the Certificate Store List (for basic authentication, you do not have to specify this).

- Enter a name for the collection certificate store.
- Add the X.509 Certificate Path for the trusted X.509 certificate file.
- Add the CRL Path for a CRL file.
- Click *OK*, and a collection certificate store is created.

► Expand *Token Consumer* and click *Add*.

- Enter a Token consumer name, for example, `con_untoken` (Figure 21-24).
- Select a Token consumer class or input your custom Token consumer class name manually. The Token consumer class matches the security token type. For a Username Token, you select the `UsernameTokenConsumer` class. The provided token consumers are:

<code>UsernameTokenConsumer</code>	For a username token
<code>X509TokenConsumer</code>	For X.509 certificate token, X.509 certificates in PKIPath and a list of X509 certificates and CRLs in a PKCS#7
<code>LTPATokenConsumer</code>	For an LTPA token
<code>IDAssertionUsernameTokenConsumer</code>	For a username token with the user name only (used for identity assertion)

In our case, we specify `UsernameTokenConsumer`.

- Expand the drop-down list for Security token and select the token specified on the extension page (`requntoken`).
- Select *Use value type* and select the value type from the list (*Username Token* in our case). For a custom token, you have to enter the URI and Local name manually.
- Select *Use jaas.config* to validate a security token in a client's request message. Specify the name of the JAAS configuration. The default JAAS configurations are specified in WebSphere Application Server, and you can add your custom JAAS configuration name to invoke your custom JAAS login module implementation. Refer to “Adding a custom JAAS configuration” on page 551 for information about how to add your custom JAAS configuration name. The predefined JAAS configuration names are:

<code>system.wssecurity.UsernameToken</code>	Validates a username token with the user name and password.
<code>system.wssecurity.X509BST</code>	Validates an X.509 certificate token.
<code>system.wssecurity.PkiPath</code>	Validates a token of X509 certificates in a PKIPath.

<code>system.wssecurity.PKCS7</code>	Validates a token of a list of X509 certificates and CRLs in a PKCS#7.
<code>system.wssecurity.IDAssertionUsernameToken</code>	Validates a username token with the user name only.

However, as for an LTPA token, it is not necessary to configure a JAAS configuration name for the username token.

- If you use a self-signed certificate that has no trust anchor and intermediate certificate, and you cannot trust any certificate, you have to specify your self-signed certificate information in two `jaas.config` properties as follows:

- Name: `com.ibm.wsspi.wssecurity.token.x509.issuerName`
Value: Issuer name of your self-signed certificate
- Name: `com.ibm.wsspi.wssecurity.token.x509.issuerSerial`
Value: Serial number of your self-signed certificate

You can see your certificate issuer name and serial number using the `keytool`. Refer to “Generating sample key stores” on page 480.

- If you specify a token consumer for identity assertion, you can specify the trusted ID evaluator. It evaluates the trust of the endpoint identity sent by identity assertion. Select *Use trusted ID evaluator* and enter the Trusted ID evaluator class name manually. One default implementation of trusted ID evaluator is provided:

```
com.ibm.wsspi.wssecurity.id.TrustedIDEvaluatorImpl
```

This class validates the identity by comparing name and value defined in the trusted ID evaluator property. To use this class, you have to add a trusted ID evaluator property (the name must start with `trustedId`) whose value evaluates as a trusted ID. For example:

```
Name: trustedId_1
Value: alice
```

Instead of specifying a class name of a trusted ID evaluator, you can specify a reference to a trusted ID evaluator defined in a default configuration. The predefined trusted ID evaluator name is `SampleTrustedIDEvaluator`. You can modify the default configuration using the administrative console in WebSphere Application Server. Refer to “Modifying the default binding configuration” on page 548. In our case, we do not use this.

- If you specify a token consumer for the three types of X.509 certificates, select *Use certificate path settings*. If you trust any certificate, select *Trust any certificate*. If you want to specify a trust anchor and a trusted certificate store, select *Certificate path reference* and select the Trust

anchor and Certificate store references from the drop-down lists. For an X.509 certificate in a PKIPath and a list of X509 certificates and CRLs in a PKCS#7, only the trust anchor reference is required. We do not specify this option because of basic authentication.

- Click *OK*, and a token consumer is created. Save the configuration.

The screenshot shows the 'Token Consumer dialog' window. The following fields are highlighted with red circles:

- Token consumer name: `con_unitcon`
- Token consumer class: `com.ibm.wsspi.wsssecurity.token.UsernameTokenConsumer`
- Security token: `requntoken`
- Value type: `Username Token`
- Local name: `http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#UsernameToken`
- URI: (empty)
- jaas.config name: `system.wssecurity.UsernameToken`

Other visible settings include:

- ☒ Use value type
- ☒ Use jaas.config
- ☐ Use trusted ID evaluator
- ☐ Use trusted ID evaluator reference
- ☐ Use certificate path settings
- ☐ Certificate path reference
 - Trust anchor reference: (empty)
 - Certificate store reference: (empty)
- ☒ Trust any certificate

At the bottom are 'Add', 'Remove', 'OK', and 'Cancel' buttons.

Figure 21-24 Token Consumer dialog for specifying basic authentication

The WS-Security configuration for basic authentication is finished.

Testing the basic authentication

To test this basic authentication configuration, refer to “Testing on WebSphere Application Server” on page 531.

How to specify identity assertion

WebSphere Application Server provides several types of authentication, such as basic authentication and identity assertion. In this section, we describe how to specify identity assertion, which is a WS-Security configuration between an intermediary server and an endpoint server.

As described in “Identity assertion” on page 460, WebSphere Application Server Version 6.0 supports three types of trust modes. Before configuring identity assertion, you have to decide which trust mode to use.

Configuring the client for identity assertion

To configure identity assertion, open the deployment descriptor of the WeatherJavaBeanClientWeb project (Requester Generator Configuration):

- Specify a security token on the WS Extension page and a corresponding token generator on the WS Binding page. Refer to “Configuring the client for a security token” on page 485.

If you specify a username token as the client’s token, you should specify a username token generator with a user name only (without a password) and add properties for identity assertion (see Table B-25 on page 689, in the section “Binding: Token generator callback handler”). Therefore, you do not have to specify a client password in the Token Generator dialog.

- Follow these steps according to the trust mode (see “Trust mode” on page 461):

None—No further configuration is necessary.

BasicAuth—Specify the security token of an intermediary username token and a corresponding username token generator. The intermediary’s username token should have a user name and password in the token generator.

Signature—Specify signing of a requester token by an X.509 certificate. Refer to “Configuring the client to specify an integrity part” on page 500 to specify integrity.

- Click *OK* and save the configuration.

Configuring the server for identity assertion

To configure how to process identity assertion, open the `webservices.xml` file in the `WeatherJavaBeanWeb` project:

- ▶ On the Extensions page, select the port component, and expand *Required Security Token*. You have to specify a security token and a matching token consumer (Binding Configurations page). Refer to “Configuring the server for security token” on page 489.

If you define the client token as a username token, there is no password. Therefore, you should specify the `IDASsertionUsernameTokenConsumer` as the token consumer and `system.wssecurity.IDASsertionUsernameToken` as the JAAS configuration name.

- ▶ Specify the configuration according to the trust mode:

None—No further configuration of the security token or signing is necessary.

BasicAuth—Specify the username token (the intermediary sends the user ID and password) and specify `UsernameTokenConsumer` as the token consumer and `system.wssecurity.UsernameToken` as the JAAS configuration name.

Signature—Specify a signature verification configuration for a signature on a client’s token. Refer to “Configuring the server to specify an integrity part” on page 509.

Figure 21-25 shows a Required Integrity Dialog for specifying integrity of the client’s security token.

The image shows a 'Required Integrity Dialog' window. At the top, the title bar says 'Required Integrity Dialog'. Below it, there are two input fields: 'Required Integrity Name' with the value 'trustmode_reqint' and 'Usage type' with a dropdown menu set to 'Required'. The main section is titled 'Message Parts' and contains a table with two columns: 'Message parts dialect' and 'Message parts keyword'. The first row of the table has the values 'http://www.ibm.com/websphere/webservices...' and 'securitytoken', which are highlighted with a red oval. Below the table are 'Add' and 'Remove' buttons. There are similar sections for 'Nonce' and 'Timestamp', each with their own tables and 'Add'/'Remove' buttons. At the bottom right are 'OK' and 'Cancel' buttons.

Message parts dialect	Message parts keyword
http://www.ibm.com/websphere/webservices...	securitytoken

Nonce dialect	Nonce keyword

Timestamp dialect	Timestamp keyword	Timestamp exp

Figure 21-25 Required Integrity Dialog for identity assertion (TrustMode=Signature)

- Expand the *Caller Part* under Request Consumer Service Configuration Details and click *Add* (Figure 21-26).

Caller Part Dialog

Name: IDAssertion_trustmode_signature

Integrity or Confidentiality part: []

Token type: Username

URI: []

Local name: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-user

☒ Use IDAssertion

Trust method name: Signature

Integrity or Confidentiality part: trustmode_reqint

URI: []

Local name: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509

Trust method property:

Name	Value

Add Remove

Property:

Name	Value

Add Remove

OK Cancel

Figure 21-26 Caller Part Dialog for identity assertion (TrustMode=Signature)

- Enter a name of for the caller part (IDAssertion_trustmode_signature). The caller part specifies which token is used and the trust mode of identity assertion.
- If an intermediary's security token is used for signing or encryption, select the name of an Integrity or Confidentiality part from the drop-down list. The security token that is used for the selected integrity or confidentiality part is regarded as the intermediary's security token.
- If the intermediary's security token is not used for signing or encryption, select the Token type of the security token. If the security token is Username Token, select *Username* as the type. If you select Custom Token as the type, you have to specify a custom URI and Local name of the token.
- Select *Use IDAssertion* and select a trust mode from the drop-down list:
 - None**—No further configuration is necessary.
 - BasicAuth**—The local name is filled in automatically.

Signature—The local name of X.509 certificate token is filled in automatically. Select one of the Integrity or Confidentiality parts from the drop-down list (select a required integrity or confidentiality part that signs or encrypts the client's security token).

- Click *OK* and save the configuration.

Testing identity assertion

To test this authentication configuration, refer to “Testing on WebSphere Application Server” on page 531.

Integrity

To provide integrity on the client's request message, add a digital signature to the request message. WebSphere Application Server Version 6.0 supports many options for signatures, for instance, the signature method, digest method, and patterns of key information. Here, we provide an outline of how to configure a digital signature:

- ▶ Add an integrity part in the Requester Generator Configuration.
- ▶ Add the information required for signing. The required items are key information and key locators; the other items are optional.
- ▶ Add corresponding signing information for the specified integrity part in the Security Request Generator Binding Configuration.
- ▶ Add the required integrity part under Request Consumer Service Configuration Details.
- ▶ Add the information for verifying the signature. The required items are key information and key locators; the other items are optional.
- ▶ Add corresponding signing information for the specified required integrity part in the Request Consumer Binding Configuration Details.

In this section, we describe the detailed steps of specifying an integrity part and signature binding information.

Important: WebSphere Application Server V6.0 provides many choices for signing a message. In this section, all possible choices are explained, but you can specify one of the most typical types of signatures by following the screen captures. The most typical choice is *Signing the SOAP Body by X.509 certificate*, which is a direct reference.

How to specify the integrity part

As described in “Supported specifications” on page 458, WebSphere Application Server Version 6.0 supports five token reference patterns. The token references are inserted to show which security token is used for signing or encryption within a KeyInfo element. The supported token reference patterns are as follows:

- **Direct reference**—Provides a reference to a security token using URIs. An example of a direct reference is shown here:

```
<wsse:SecurityTokenReference wsu:Id="...">
  <wsse:Reference URI="..." ValueType="..." />
</wsse:SecurityTokenReference>
```

- **Key identifier**—Specifies a security token using a keyword that is used to uniquely identify a security token. The exact value type and generation algorithm are defined in the token-specific profiles. An example of a key identifier reference is shown here:

```
<wsse:SecurityTokenReference>
  <wsse:KeyIdentifier wsu:Id="..." ValueType="..." EncodingType="...">
    ...
  </wsse:KeyIdentifier>
</wsse:SecurityTokenReference>
```

- **Key name**—Specifies a security token by a key name that is used to extract a key from the key locator. An example of a key name reference is shown here:

```
<ds:KeyInfo wsu:Id="..." xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
  <ds:KeyName>CN=Client, OU=IBM, C=US</ds:KeyName>
</ds:KeyInfo>
```

- **Embedded reference**—A security token exists outside of the KeyInfo element in case of a direct reference, but a security token is embedded in the KeyInfo element in this case. An example of an embedded reference is shown here:

```
<wsse:SecurityTokenReference>
  <wsse:Embedded wsu:Id="...">
    ...
  </wsse:Embedded>
</wsse:SecurityTokenReference>
```

- **X509 issuer name and serial number**—Specifies an X.509 certificate token by an issuer and serial number of the certificate. The reference is used only in case of an X.509 certificate token. An example of an X509 issuer name and serial number is shown here:

```
<wsse:SecurityTokenReference>
  <ds:X509Data>
    <ds:X509IssuerSerial>
      <ds:X509IssuerName>CN=Client, OU=IBM< C=US</ds:X509IssuerName>
      <ds:X509SerialNumber>4199d0ec</X509SerialNumber>
    </ds:X509IssuerSerial>
  </ds:X509Data>
</wsse:SecurityTokenReference>
```

```
</ds:X509IssuerSerial>  
</ds:X509Data>  
</wsse:SecurityTokenReference>
```

You have to decide which type of token reference is used for the integrity part. This selection decides whether a security token is inserted into the message. It is possible to insert a security token in case of direct reference, key identifier, and embedded reference. This information is necessary when the binding is configured.

Configuring the client to specify an integrity part

To configure integrity for a request message sent by a client, open the `WeatherJavaBeanClientWeb` deployment descriptor and go to the **WS Extension** page under **Request Generator Configuration**:

- ▶ Expand *Integrity* and click *Add*:
 - Enter a name identifying the part, for example, `int_body` (Figure 21-27).
 - Select the order in which the signature is generated. Multiple integrity parts can be specified, and you have to specify the order of generating the signature. In our case, we select 1.

Note: The WS-Security runtime of Version 6.0 supports multiple signature and encryption parts in one SOAP message. For multiple signature and encryption parts, you have to specify the processing order. For example, if you want to sign and encrypt the SOAP body, you should specify 1 in the Integrity Dialog and 2 in the Confidentiality Dialog.

- ▶ Click *Add* for the Message Parts, and one integrity part is created. The default created part is for signing the SOAP body. If you want to sign the SOAP body only, you have nothing more to do. For another integrity part, you have to specify two items here:
 - In the Message parts dialect section, you can select either WebSphere keywords or XPath:
 - <http://www.ibm.com/websphere/webservices/wssecurity/dialect-was> uses WebSphere keywords.
 - <http://www.w3.org/TR/1999/REC-xpath-19991116> uses an XPath expression.

- In the Message parts keyword section, you can select from predefined keywords for which message parts are signed. The keywords are:
 - **body**—SOAP body element is signed.
 - **timestamp**—All timestamp elements are signed.
 - **securitytoken**—All security tokens are signed.
 - **dsigkey**—KeyInfo elements of the signature are signed.
 - **enckey**—KeyInfo elements of the encryption are signed.
 - **messageid**—MessageID element of WS-Addressing is signed.
 - **to**—To element of WS-Addressing is signed.
 - **action**—Action element of WS-Addressing is signed.
 - **relatesto**—RelatesTo element of WS-Addressing is signed.
- If the XPath expression dialect is selected, you should input the XPath expression to specify the integrity part. For example, to specify the SOAP body by an XPath expression, you should enter the expression shown in Example 21-8 on page 464.
- Add a Nonce or Timestamp, or both, if you require WebSphere Application Server Version 6.0 extensions, as described in “Nonce extension” on page 465 and “Timestamp extension” on page 465.
 For both, you select the dialog and keyword in the same way as for the message parts. For the timestamp, you can specify an expiration date. Refer to “Adding a security timestamp” on page 528 for details about using timestamps.
- You can add multiple message parts, a nonce, and timestamp in one dialog. All message parts that are specified in one Integrity dialog are signed by the same signing information, which is defined on the WS Binding page.
- Click *OK*, and an integrity part is created.
- ▶ If you need multiple integrity parts, you can add them.
- ▶ Save the configuration.

Figure 21-27 shows an Integrity Dialog for specifying a signature on the SOAP body.

Integrity Dialog

Integrity Name: ht_body

Order: 1

Message parts dialect	Message parts keyword
http://www.ibm.com/websphere/webservices...	body

Add Remove

Nonce

Nonce dialect	Nonce keyword

Add Remove

Timestamp

Timestamp dialect	Timestamp keyword	Timestamp exp

Add Remove

OK Cancel

Figure 21-27 Integrity Dialog

After specifying an integrity part, corresponding information must be specified on the **WS Binding** page under **Security Request Generator Binding Configuration**. You have to know if a token reference type and a security token are inserted or not.

- To specify a token generator for a list of X.509 certificates and CRLs in a PKCS#7, expand *Certificate Store List* and click *Add* (in our case, you do not specify this).

Enter any name. Add a CRL Path pointing to the CRL file. These specified CRLs are packaged in a PKCS#7 wrapper. Click *OK*, and a collection certificate store is created.

- If you want to insert a security token into the message for signing, expand *Token Generator* and click *Add*.

For information about how to specify a token generator, refer to “Configuring the client for a security token” on page 485. Note that to specify a token generator used for signing, a security token is not specified on the WS Extension page:

- Enter a Token generator name, for example, gen_dsigtgen.

- For the Token generator class, select the X509TokenGenerator.
- Do not select a Security token.
- Select *Use value type*, and then select *X509 certificate token* and the X509CallbackHandler.
- Select *Use key store*. For key-related information, refer to “Generating sample key stores” on page 480. In our case, we enter client as the storepass, C:\SG246461\sampcode\mykeystore\client.jks as the key store path, and JKS as key store type.
- Click *Add* under Key and enter client_rsa as the alias, client_rsa as the key pass, and CN=Client, OU=IBM, C=US as the key name.
- Click *OK*, and a token generator is created.

Figure 21-28 shows the Token Generator dialog for specifying a signature by an X.509 certificate.

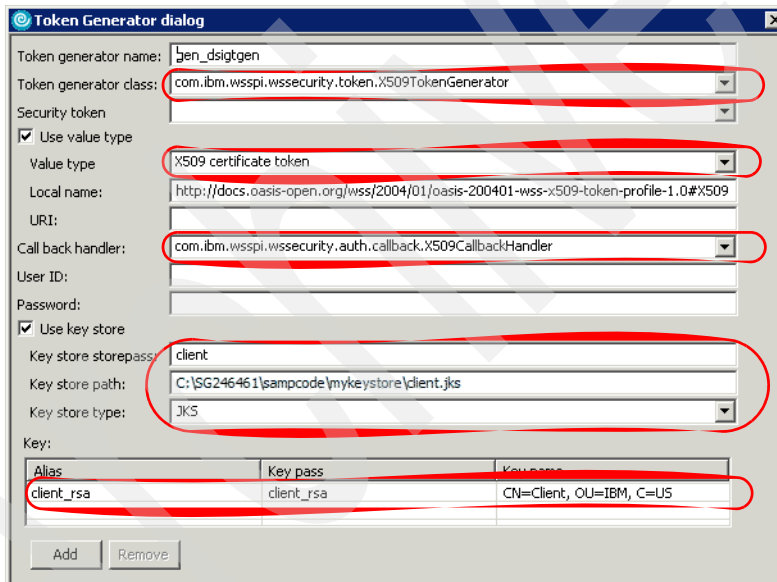


Figure 21-28 Token Generator dialog for signing by an X.509 certificate

- Expand *Key Locators* and click *Add*. Specify how to retrieve a key for signing:
 - Enter a Key locator name, for example, gen_dsigtlocator (Figure 21-29).
 - Select or enter a Key locator class name. The class to retrieve a key implements the com.ibm.wsspi.wssecurity.keyinfo.KeyLocator interface. Three implementations of KeyLocator are provided in WebSphere Application Server Version 6.0, and you can implement your own class if necessary.

The provided implementations are:

KeyStoreKeyLocator	Locates and obtains the key from the specified key store file.
SignerCertKeyLocator	Uses the public key from the certificate of the signer of the request message. This implementation is used by the response generator.
X509TokenKeyLocator	Uses the X.509 security token from the requester message for signature validation and encryption. This implementation is used by the request consumer and the response consumer.

In our case, we select the KeyStoreKeyLocator.

- If the selected key locator class requires the key store file, you can specify the key store and key by selecting *Use key store*. Enter the same information as in the previous dialog.
- Click *OK*, and a key locator is created.

Figure 21-29 shows the Key Locator dialog for specifying a signature by an X.509 certificate.

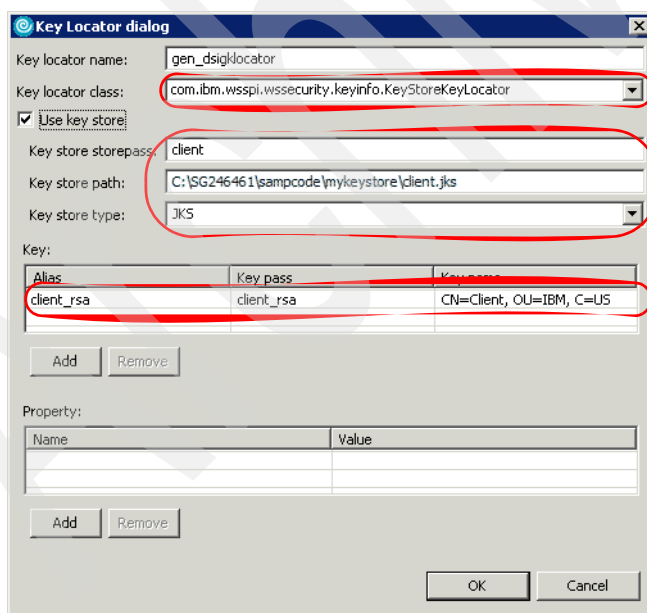


Figure 21-29 Key Locator dialog for signing by an X.509 certificate

- Expand *Key Information* and click *Add*. Specify which type of security token reference is used:

- Enter a name, for example, `gen_dsigkeyinfo` (Figure 21-30).

- Select a Key information type from these choices:

STRREF	Direct reference (our choice)
EMB	Embedded reference
KEYID	Key identifier reference
KEYNAME	Key name reference
X509ISSUER	X.509 issuer name and serial number reference

The key information class name is filled in when a type is selected.

- Select *Use key locator* and select a Key locator from the list. Key locators that have been defined are listed. Select `gen_dsiglocator`. Also select one of the predefined keys (`CN=Client, OU=IBM, C=US`).

- If a security token is inserted into the message, and a token generator is specified already, specify which token generator is used. Select *Use token* and select one token generator name from the list. The selected token generator is invoked to generate the token that is referenced from this key information. In our case, we select `gen_dsigtgen`.

- Click *OK*, and the key information is created.

Figure 21-30 shows a Key Information dialog for specifying a signature by an X.509 certificate.

Key Information dialog

Key information name: `gen_dsigkeyinfo`

Key information type: `STRREF`

Key information class: `com.ibm.ws.webservices.wssecurity.keyinfo.STRReferenceContentGenerab`

☒ Use key locator

Key locator: `gen_dsiglocator`

Key name: `CN=Client, OU=IBM, C=US`

☒ Use token

Token: `gen_dsigtgen`

Property:

Name	Value

Add Remove OK Cancel

Figure 21-30 Key Information dialog for signing by an X.509 certificate

- ▶ Expand *Signing Information* and click *Add*. You have to specify how to sign:
 - Enter a name, for example, `sign_body` (Figure 21-31).
 - Select a Canonicalization method algorithm. The supported algorithms are:
 - `http://www.w3.org/2001/10/xml-exc-c14n#`: Canonical XML algorithm without comments (our choice)
 - `http://www.w3.org/2001/10/xml-exc-c14n#WithComments`: Canonical XML algorithm with comments
 - `http://www.w3.org/TR/2001/REC-xml-c14n-20010315`: Exclusive XML canonicalization algorithm without comments
 - `http://www.w3.org/TR/2001/REC-xml-c14n-20010315#WithComments`: Exclusive XML canonicalization algorithm with comments
 - Select a Signature method algorithm from the list. The supported algorithms are:
 - `http://www.w3.org/2000/09/xmldsig#rsa-sha1`: RSA with SHA1 (our choice)
 - `http://www.w3.org/2000/09/xmldsig#dsa-sha1`: DSA with SHA1
 - `http://www.w3.org/2000/09/xmldsig#hmac-sha1`: HMAC-SHA1
 - Enter any Key information name, for example, `sign_kinfo`.
 - Select a Key information element from the list. Predefined key information is listed. Select `gen_dsigkeyinfo`.
 - If you specify signature information for integrity on the KeyInfo element of the signature or encryption, meaning that `dsigkey` or `enckey` is specified as an integrity part, you can specify how to sign the KeyInfo element. Select *Use key information signature* and select from the following choices:

<code>keyinfo</code>	Specifies the whole KeyInfo element to be signed.
<code>keyinfochildelements</code>	Specifies the child elements of KeyInfo element to be signed.
- In our case, we do not have to specify this.
- Click *OK*, and the signing information is created.

Figure 21-31 shows a Signing Information dialog for specifying a signature by an X.509 certificate.

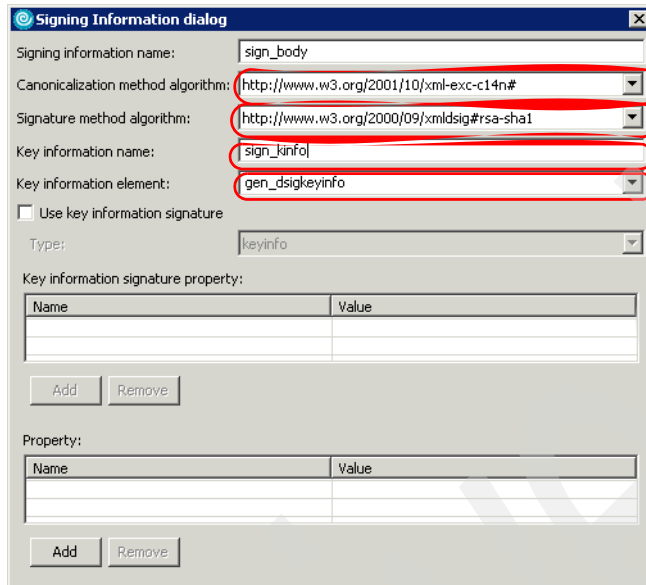


Figure 21-31 Signing Information dialog for signing by an X.509 certificate token

- Expand *Part References* and click *Add*. This specifies an integrity part that applies to this signature information:
 - Enter a Part reference name, for example, `sign_part` (Figure 21-32).
 - Select an Integrity part from the list of parts defined on the WS Extensions page. In our case, we select `int_body`.
 - Select a Digest method algorithm from the list. The supported digest method algorithm is `http://www.w3.org/2000/09/xmldsig#sha1`.
 - Click *OK*, and a part reference is created. You can specify multiple part references that apply to the same signature information.

Figure 21-32 shows a Part Reference dialog for specifying a signature by an X.509 certificate.

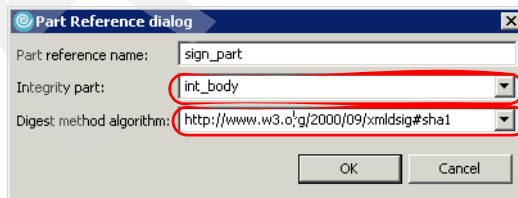


Figure 21-32 Part Reference dialog for signing by an X.509 certificate

- After adding part references, adding a transform becomes enabled. Expand *Transforms* and click *Add* and complete the dialog:

- Enter a name, for example, sign_trans (Figure 21-33).
- Select a transform algorithm from the list. You can specify multiple transforms for one part reference. The supported transform algorithms are as follows:

- <http://www.w3.org/2001/10/xml-exc-c14n#>: Exclusive canonicalization algorithm (our choice).
- <http://www.w3.org/TR/1999/REC-xpath-19991116>: XPath transform algorithm. To use this transform, you have to specify the property name and value by adding a transform property. For example, specify the following information as the property:

Name: com.ibm.wsspi.wssecurity.dsig.XPathExpression
Value: XPath expression

- <http://www.w3.org/2002/06/xmldsig-filter2>: XML-Signature XPath Filter Version 2.0. To use this transform, you have to specify a set of the transform properties:

Name: com.ibm.wsspi.wssecurity.dsig.XPath2Expression_*
Value: XPath expression for XPath filter 2

Name: com.ibm.wsspi.wssecurity.dsig.XPath2Filter_*
Value: intersect | subtract | union
intersect: intersect the Node specified by XPath in Property1
subtract: subtract the Node specified by XPath in Property1
union: union the Node specified by XPath in Property1

Name: com.ibm.wsspi.wssecurity.dsig.XPath2Order_*
Value: Number of processing order of each XPath

This example shows how the part specified by Property 1 is intersected for signature:

```
com.ibm.wsspi.wssecurity.dsig.XPath2Expression_1=[XPath expression]
com.ibm.wsspi.wssecurity.dsig.XPath2Filter_1=intersect
com.ibm.wsspi.wssecurity.dsig.XPath2Order_1=1
```

- <http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#STR-Transform>: This STR-Transform is specified in Section 8.3 in the OASIS WS-Security specification. It is used to sign a KeyInfo element or security tokens. When a message part of integrity is dsigkey or enckey, this transform should be specified.
- <http://www.w3.org/2002/07/decrypt#XML>: Decryption transform (<http://www.w3.org/TR/xmlenc-decrypt>).

- <http://www.w3.org/2000/09/xmlsig#enveloped-signature>: Enveloped signature.
- Click *OK*, and a transform is created. Save the configuration.

Figure 21-33 shows a Transform dialog for specifying a signature by an X.509 certificate.

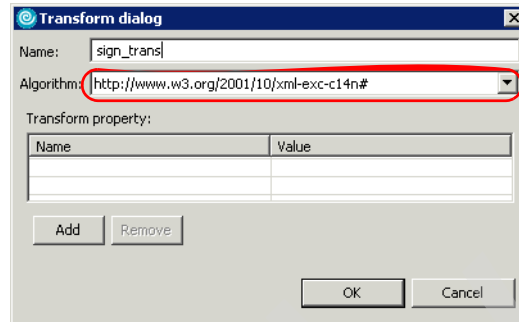


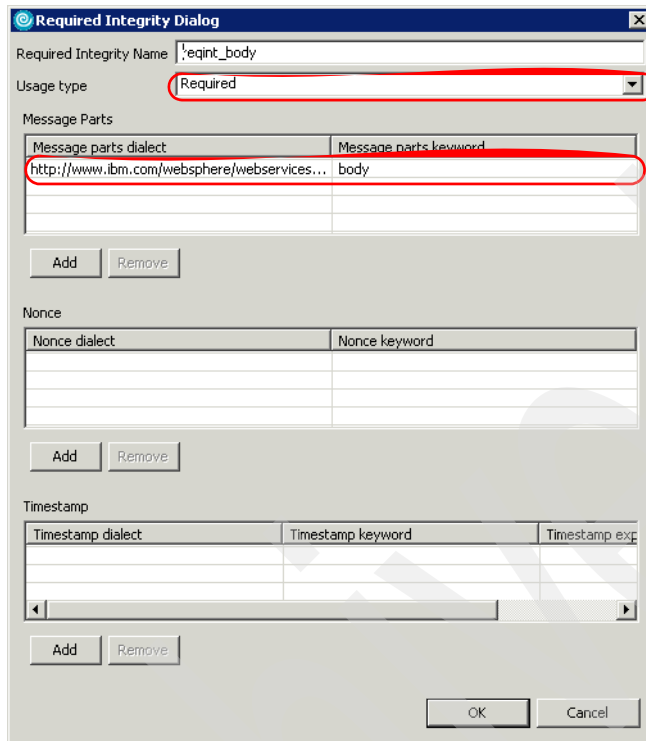
Figure 21-33 Transform dialog for signing by an X.509 certificate

Configuring the server to specify an integrity part

To configure how to verify integrity, open the `webservices.xml` file in the WeatherJavaBeanWeb project with the Web Services Editor. To receive the digitally signed request message from a client, a server should configure signature verification on the **Extension** page under **Request Consumer Service Configuration Details** as follows.

- ▶ Expand *Required Integrity* and click *Add*:
 - Enter a name denoting the part, for example, `reqint_body` (Figure 21-34).
 - Select the Usage type, *Required* or *Optional*. If the usage type is *Required*, a request message without integrity throws a SOAP fault. If the usage type is *Optional*, and a request message does not have the integrity part, the message is received. We select *Required*.
 - Click *Add* under Message Parts to specify an integrity required part. We select `http://....../dialect-was` and `body`.
 - Nonce and timestamp extensions can be specified in the same way as for the client. Refer to “Configuring the client to specify an integrity part” on page 500. In our case, we do not specify them.
 - Click *OK*, and a required integrity part is created.

Figure 21-34 shows a Required Integrity Dialog for specifying that integrity is required on the SOAP body.



The image shows a 'Required Integrity Dialog' window. It has a title bar with a close button. The 'Required Integrity Name' field contains 'reqint_body'. The 'Usage type' dropdown is set to 'Required'. Below this is a 'Message Parts' section with a table. The first row of the table has 'Message parts dialect' as 'http://www.ibm.com/websphere/webservices...' and 'Message parts keyword' as 'body'. There are 'Add' and 'Remove' buttons below the table. Below the 'Message Parts' section is a 'Nonce' section with a similar table and 'Add'/'Remove' buttons. At the bottom is a 'Timestamp' section with a table and 'Add'/'Remove' buttons. The 'OK' and 'Cancel' buttons are at the bottom right.

Message parts dialect	Message parts keyword
http://www.ibm.com/websphere/webservices...	body

Nonce dialect	Nonce keyword

Timestamp dialect	Timestamp keyword	Timestamp exp

Figure 21-34 Required Integrity Dialog

We have to provide corresponding information on the **Binding Configurations** page under **Request Consumer Binding Configuration Details**. You have to know if a required token reference type and a security token are required or not. These settings should match the client configuration:

- ▶ If you want to specify a token consumer for the three types of X.509 certificate tokens, specify a Trust Anchor. Refer to “Configuring the server for security token” on page 489. In our case, we do not specify this.
- ▶ If the token consumer for the X.509 certificate token is necessary, and you want to specify an intermediate trusted certificate store, specify a Collection Certificate Store under Certificate Store List. Refer to “Configuring the server for security token” on page 489. In our case, we do not specify this.
- ▶ If the security token is inserted in the request message, add a Token Consumer (expand *Token Consumers* and click *Add*):
 - To specify a token consumer, refer to “Configuring the server for security token” on page 489. To specify a token consumer for the token used for signing, a required security token is not specified in the deployment descriptor.

Figure 21-35 shows a Token Consumer dialog for signature verification by X.509 certificates.

Token consumer name: con_dsigtcon

Token consumer class: com.ibm.wsspi.wsssecurity.token.X509TokenConsumer

Security token:

☒ Use value type

Value type: X509 certificate token

Local name: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509

URI:

☒ Use jaas.config

jaas.config name: system.wsssecurity.X509BST

jaas.config property:

Name	Value

Add Remove

☐ Use trusted ID evaluator

Trusted ID evaluator class:

Trusted ID evaluator property:

Name	Value

Add Remove

☐ Use trusted ID evaluator reference

Trusted ID evaluator reference:

Property:

Name	Value

Add Remove

☒ Use certificate path settings

☒ Certificate path reference

Trust anchor reference:

Certificate store reference:

☒ Trust any certificate

Figure 21-35 Token Consumer dialog for signature verification (X.509 certificate)

- Enter a name, for example, con_dsigtcon.
- Select the Token consumer class X509TokenConsumer.
- Do not select a security token.
- Select *Use value type* and select *X509 certificate token*.

- Select *Use jaas.config* and enter `system.wssecurity.X509BST` as the name.
- Select *Use certificate path settings* and select *Trust any certificate*.

Note: When you specify a token consumer for signature verification and you use a self-signed certificate, you should not specify `jaas.config` properties that specify trusted certificate information. If you specify them, only specified issuer certificates can be verified. Therefore, the token consumer cannot accept other issuer certificates. You should specify *Trust any certificate* in the Token Consumer dialog for signature verification in case you use a self-signed certificate.

- Expand *Key Locators* and click *Add* to specify how to retrieve a key for a signature verification:
 - To specify key-related information, refer to “Configuring the client to specify an integrity part” on page 500.
 - Enter a name, for example, `con_dsigklocator`, and select the `X509TokenKeyLocator` class.

Figure 21-36 shows a Key Locator dialog for signature verification by X.509 certificates.

Key Locator dialog

Key locator name:

Key locator class:

☐ Use key store

Key store storepass:

Key store path:

Key store type:

Key:

Alias	Key pass	Key name

Property:

Name	Value

Figure 21-36 Key Locator dialog for signature verification (X.509 certificate)

- Expand *Key Information* and click *Add* to specify which type of security token reference is required. The type should match the client configuration:
 - To specify key information, refer to “Configuring the client to specify an integrity part” on page 500.
 - Enter a name, for example, `con_dsigkeyinfo`, select `STRREF` as the type, select *Use key locator* and `con_dsiglocator`, and select *Use token* and `con_dsigtcon`. On the consumer side, we do not have to specify the key name.

Figure 21-37 shows a Key Information dialog for signature verification by X.509 certificates.

The screenshot shows the 'Key Information dialog' window. It contains the following fields and controls:

- Key information name:** `con_dsigkeyinfo`
- Key information type:** `STRREF` (selected from a dropdown menu)
- Key information class:** `com.ibm.ws.webservices.wssecurity.keyinfo.STRReferenceContentConsum...`
- Use key locator:** ☒
- Key locator:** `con_dsiglocator` (selected from a dropdown menu)
- Key name:** (empty dropdown menu)
- Use token:** ☒
- Token:** `con_dsigtcon` (selected from a dropdown menu)
- Property:** A table with two columns: **Name** and **Value**. The table is currently empty.
- Buttons:** **Add**, **Remove**, **OK**, and **Cancel**.

Figure 21-37 Key Information dialog for signature verification (X.509 certificate)

- Expand *Signing Information* and click *Add* to define how to verify a required integrity part:
 - To specify the signing information, refer to “Configuring the client to specify an integrity part” on page 500. For the consumer, multiple signing key information can be specified.
 - Enter a name, for example, `sign_body` (Figure 21-38).
 - Select `http://www.w3.org/2001/10/xml-exc-c14n#` as the Canonicalization method algorithm and `http://www.w3.org/2000/09/xml-dsig#rsa-sha1` as the Signature method algorithm.
 - For the Signing key information, click *Add*, enter `sign_kinfo` as the Key information name, and select `con_dsigkeyinfo` as the Key information element.

The **Signing Information dialog** is shown with the following fields and values:

- Signing information name:** sign_body
- Canonicalization method algorithm:** http://www.w3.org/2001/10/xml-exc-c14n#
- Signature method algorithm:** http://www.w3.org/2000/09/xmldsig#rsa-sha1
- Signing key information:**

Key information name	Key information element
sign_kinfo	con_dsigkeyinfo
- ☐ Use key information signature
- Type:** keyinfo
- Key information signature property:**

Name	Value
- Property:**

Name	Value

Buttons: Add, Remove, OK, Cancel.

Figure 21-38 Signing Information dialog for signature verification (X.509 certificate)

- Expand *Part Reference* and click *Add* and complete the dialog:
 - To specify a part reference, refer to “Configuring the client to specify an integrity part” on page 500. For the consumer, you should select the name of a required integrity part from the list instead of a part name.
 - Enter a name, for example, sign_part, select reqint_body from the list and http://www.w3.org/2000/09/xmldsig#sha1 as the algorithm (Figure 21-39).

The **Part Reference dialog** is shown with the following fields and values:

- Part reference name:** sign_part
- Required integrity part:** reqint_body
- Digest method algorithm:** http://www.w3.org/2000/09/xmldsig#sha1

Buttons: OK, Cancel.

Figure 21-39 Part Reference dialog for signature verification (X.509 certificate)

- Expand *Transform* and click *Add* and complete the dialog:
 - To specify Transform information, refer to “Configuring the client to specify an integrity part” on page 500.
 - Enter a name, for example, `sign_trans`, and select `http://www.w3.org/2001/10/xml-exc-c14n#` as the algorithm (Figure 21-40).

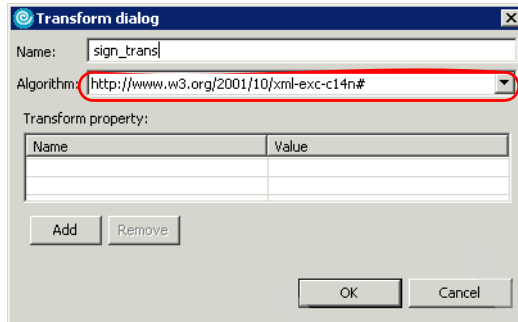


Figure 21-40 Transform dialog for signature verification (X.509 certificate)

- Save the configuration.

Testing integrity

To test the integrity configuration, refer to “Testing on WebSphere Application Server” on page 531.

Integrity on the response message

If it is necessary to add integrity on the response message from the server, you have to configure the Response Generator in the server configuration and the Response Consumer in the client configuration:

- For the server configuration, you can configure the Response Generator Service Configuration Details in the Extensions page and the Response Generator Binding Configuration Details in the Binding Configurations page by referring to “Configuring the client to specify an integrity part” on page 500.
- For the client configuration, you can configure the Response Consumer Configuration in the WS Extension page and the Security Response Consumer Binding Configuration in the WS Binding page by referring to “Configuring the server to specify an integrity part” on page 509.

Confidentiality

To provide confidentiality on the client's request message, we encrypt part of the request message. WebSphere Application Server Version 6.0 supports many options for encryption, for instance, the data encryption method, key encryption method, and patterns of key information. Here, we provide an outline of how to configure encryption:

- ▶ Add a confidentiality part in the Requester Generator Configuration.
- ▶ Add the information needed for encryption. The required items are key information and key locators. The other items are optional.
- ▶ Add the corresponding encryption information for the specified confidentiality part in the Security Request Generator Binding Configuration.
- ▶ Add a required confidentiality part in the Request Consumer Service Configuration Details.
- ▶ Add the information for decrypting the encryption. The required items are key information and key locators. The other items are optional.
- ▶ Add the corresponding encryption information for the specified required integrity part in the Request Consumer Binding Configuration Details.

In this section, we describe the detailed steps of specifying a confidentiality part and key binding information.

How to specify a confidentiality part

You have to decide which type of token reference is used for the confidentiality part in the same way as for the integrity configuration. To configure a confidentiality part, decide which type of token reference is used and whether a security token is inserted or not. This information is necessary when the binding is configured.

Note: WebSphere Application Server V6.0 provides many choices for encrypting a message. In this section, all possible choices are explained, but you can specify one of the most typical types of encryption by following the screen captures. The most typical encryption is *Encrypting the SOAP Body content by X.509 certificate* with a key identifier.

Configuring the client to specify a confidentiality part

To configure confidentiality for a request message sent by a client, open the WeatherJavaBeanClientWeb deployment descriptor and go to the **WS Extension** page under **Request Generator Configuration**:

- ▶ Expand *Confidentiality* and click *Add*:
 - Enter a name, for example, conf_body (Figure 21-41).

- Select the order in which the encryption is generated. Multiple confidentiality parts can be specified, and you have to specify the order of generating the encryption. In our case, we select 1.

Note: The WS-Security runtime of Version 6.0 supports multiple signature and encryption parts in one SOAP message. For multiple signature and encryption parts, you have to specify the processing order. **For example, if you want to sign and encrypt the SOAP body, you should specify 1 in the Integrity dialog and 2 in the Confidentiality dialog.**

- Click *Add* for Message Parts, and one confidentiality part is created. The created part is an encryption of the SOAP body content. To specify other confidentiality parts, specify the Message parts dialect and Message parts keyword. Refer to the definition of message parts dialect described in “Configuring the client to specify an integrity part” on page 500. The message parts keywords for specifying a confidentiality part are different from an integrity part. The keywords for a confidentiality part are:

bodycontent	Content of SOAP body element
usertoken	Username token element
digestvalue	Digest value element from a signature element

We select `http://...../dialect-was` and `bodycontent` as the keyword.

- Add a Nonce or Timestamp, or both, if you require WebSphere Application Server Version 6.0 extensions, as described in “Nonce extension” on page 465 and “Timestamp extension” on page 465.

For both, you select the dialog and keyword in the same way as for the message parts. For the timestamp, you can specify an expiration date. Refer to “Adding a security timestamp” on page 528 for details about using timestamps.

- You can add multiple message parts, a nonce, and timestamp in one dialog. All message parts that are specified in one Confidentiality dialog are encrypted by the same encryption information, which is defined on the WS Binding page.
- Click *OK*, and a confidentiality part is created.
- ▶ If you need multiple confidentiality parts, you can add more.
- ▶ Save the configuration.

Figure 21-41 shows a Confidentiality Dialog for specifying the encryption of the SOAP body content.

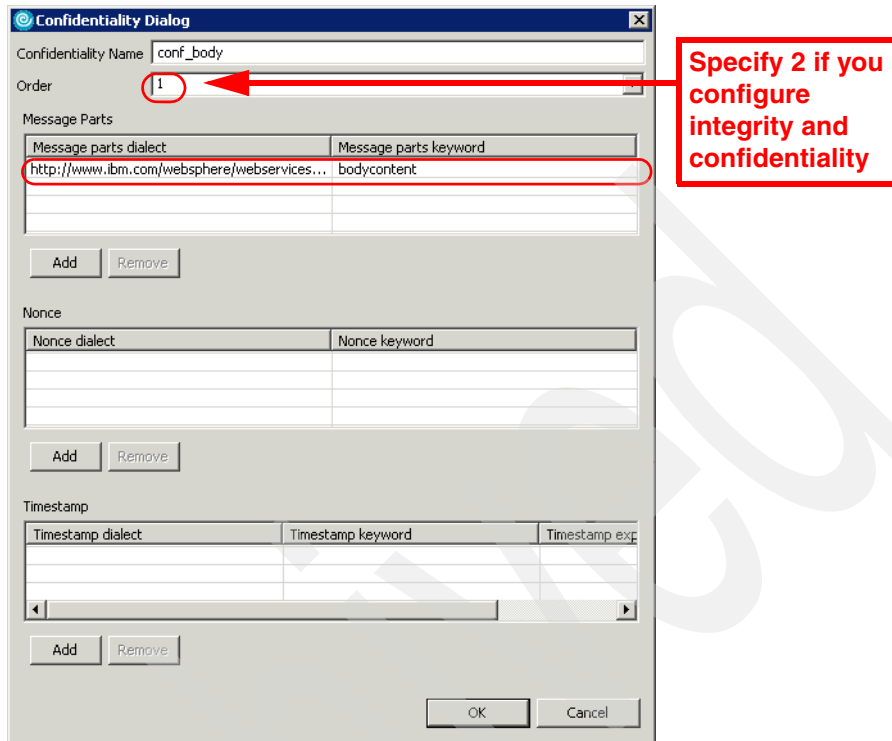


Figure 21-41 Confidentiality Dialog for body content encryption

After specifying a confidentiality part, the corresponding information must be specified in the **WS Binding** page under **Security Request Generator Binding Configuration**. You have to know if a token reference type and a security token are inserted or not.

- ▶ If you want to insert a security token into the message, add a Token Generator. To specify a token generator, refer to “Configuring the client for a security token” on page 485. To specify a token generator used for encryption, a Security Token is not specified in the extensions. Therefore, it is not necessary to specify a Security Token in the Token Generator dialog. In our case, a security token for encryption is not inserted, so we do not have to specify this.
- ▶ Expand *Key Locators* and click *Add* to specify how to retrieve a key for encryption:
 - To specify a key locator, refer to “Configuring the client to specify an integrity part” on page 500.
 - Enter a name, for example, `gen_encklocator`.

- Select `KeyStoreKeyLocator` as the class name. The class to retrieve a key implements the `com.ibm.wsspi.wssecurity.keyinfo.KeyLocator` interface.
- Select *Use key store* and specify a client key store and server public key. We specify `client`, `C:\SG246461\sampcode\mykeystore\client.jks` and JKS.
- Click *Add* under *Key* to define the key. Enter `server_rsa` as the alias and `CN=Server, OU=IBM, C=US` as the key name. Key pass should be empty, because a client does not know the key password of a server key in the client key store.

Figure 21-42 shows a Key Locator dialog for specifying the encryption of the SOAP body content.

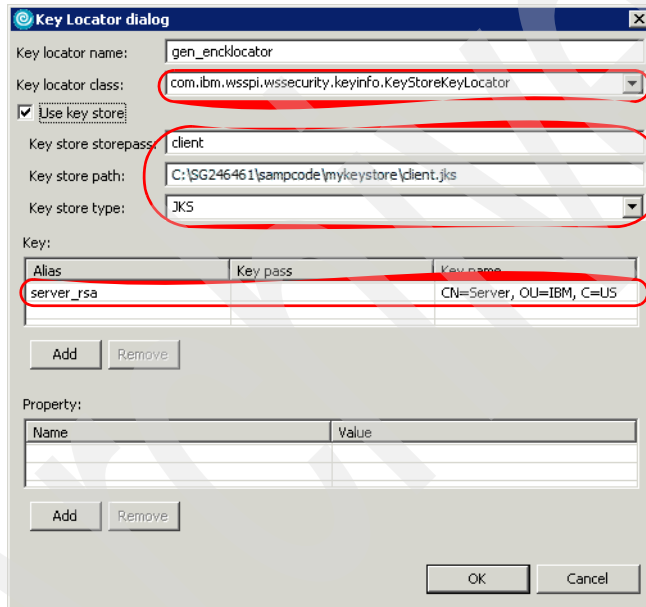


Figure 21-42 Key Locator dialog for body content encryption

- Expand *Key Information* and click *Add* to specify which type of key reference is used:
 - To specify key information, refer to “Configuring the client to specify an integrity part” on page 500.
 - Enter a name, for example, `gen_enckeyinfo`, and select a type (KEYID) from Key information type list (Figure 21-43).
 - Select *Use key locator*, and then select `gen_encklocator` and `CN=Server, OU=IBM, C=US`.

Figure 21-43 shows a Key Information dialog for specifying the encryption of the SOAP body content.

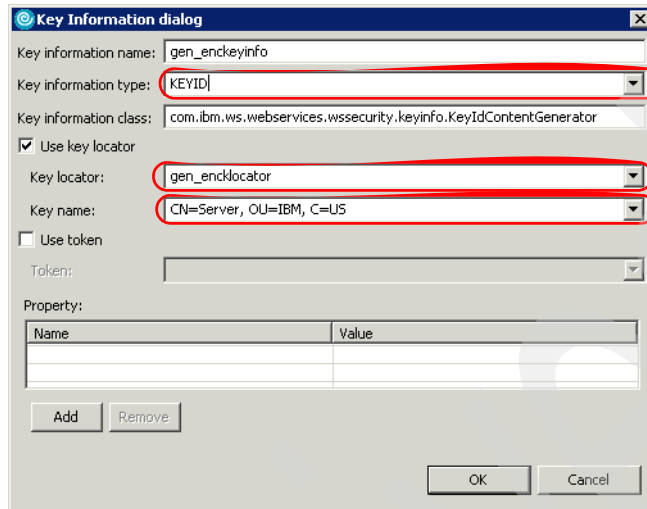


Figure 21-43 Key Information dialog for body content encryption

- Expand *Encryption Information* and click *Add* and specify how to encrypt:
 - Enter a name, for example, enc_body (Figure 21-44).
 - Select a Data encryption method algorithm from the list. The supported algorithms are:

http://www.w3.org/2001/04/xmlenc#tripledes-cbc	Triple DES in CBC (our choice)
http://www.w3.org/2001/04/xmlenc#aes128-cbc	AES 128 in CBC
http://www.w3.org/2001/04/xmlenc#aes192-cbc	AES 192 in CBC
http://www.w3.org/2001/04/xmlenc#aes256-cbc	AES 256 in CBC
 - Select a Key encryption method algorithm from the list. The supported algorithms are:

http://www.w3.org/2001/04/xmlenc#rsa-1_5	RSA Version 1.5 (our choice)
http://www.w3.org/2001/04/xmlenc#kw-tripledes	Triple DES Key Wrap
http://www.w3.org/2001/04/xmlenc#kw-aes128	AES 128 Key Wrap
http://www.w3.org/2001/04/xmlenc#kw-aes192	AES 192 Key Wrap
http://www.w3.org/2001/04/xmlenc#kw-aes256	AES 256 Key Wrap
- If no algorithm selected, the encryption key is not encrypted.
- Enter any Key information name; it specifies the key information reference. We specify enc_keyinfo.

- Select a Key information element from the list of the key information that was defined. The selected key information is used for encryption. In our case, we select `gen_enckeyinfo` from the list.
- Select a Confidentiality part from the list of confidentiality parts that were defined in the extensions. In our case, we select `conf_body` from the list.
- Click *OK*, and the encryption information is created. Save the configuration.

Figure 21-44 shows an Encryption Information dialog for specifying the encryption of the SOAP body content.

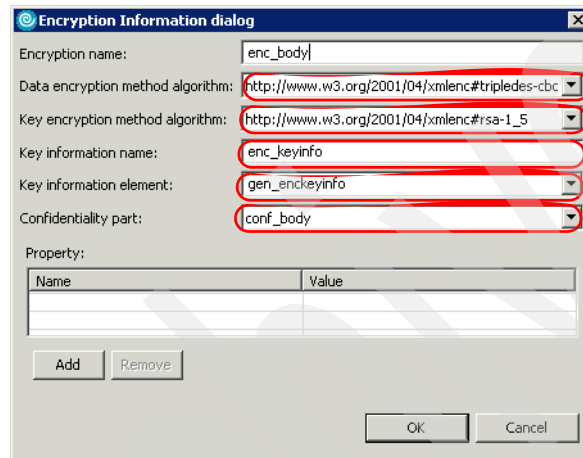


Figure 21-44 Encryption Information dialog for body content encryption

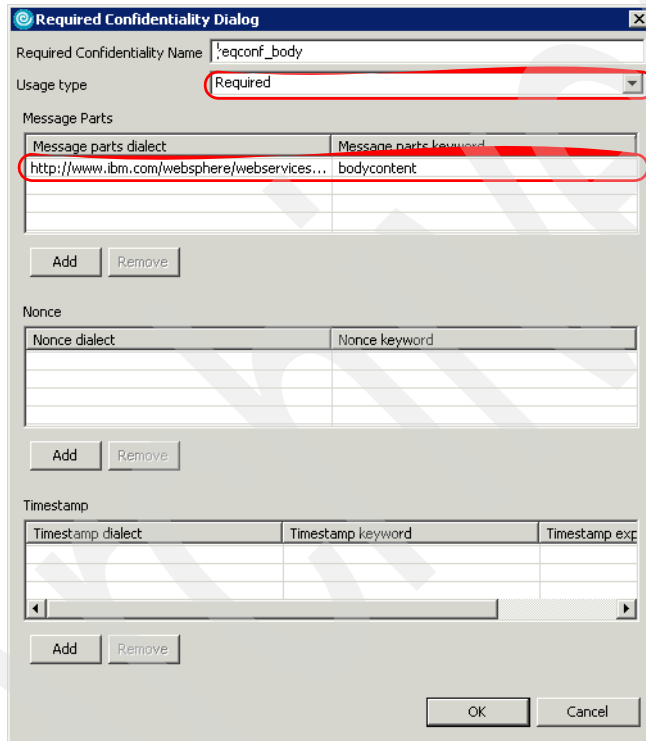
Configuring the server to specify a confidentiality part

To configure how to decrypt the message, open the `webservices.xml` file in `WeatherJavaBeanWeb` with the `Web Services Editor`. To receive an encrypted message from a client, the server should configure how to decrypt the message on the **Extension** page under **Request Consumer Service Configuration Details** as follows:

- Expand *Required Confidentiality* and click *Add*.
 - Enter a name, for example, `reqconf_body` (Figure 21-45).
 - Select the Usage type, either *Required* or *Optional*. If the usage type is *Required*, an unencrypted request message throws a SOAP fault. If the usage type is *Optional*, an unencrypted message is received. Select *Required*.

- Click *Add* for Message Parts and select `http://.../dialect-was` and `bodycontent` as the keyword.
- Nonce and timestamp extensions can be specified as on the generator side. To specify message parts, the nonce, and the timestamp, refer to “Configuring the client to specify a confidentiality part” on page 516. In our case, we do not specify them.
- Click *OK*, and a required confidentiality part is created.

Figure 21-45 shows a Required Confidentiality Dialog for decryption of the SOAP body content.



The image shows a 'Required Confidentiality Dialog' window. The 'Required Confidentiality Name' field contains 'reqconf_body'. The 'Usage type' dropdown is set to 'Required'. Under the 'Message Parts' section, there is a table with two columns: 'Message parts dialect' and 'Message parts keyword'. The first row contains 'http://www.ibm.com/websphere/webservices...' and 'bodycontent'. Below this table are 'Add' and 'Remove' buttons. The 'Nonce' section has a similar table with 'Nonce dialect' and 'Nonce keyword' columns, and 'Add' and 'Remove' buttons. The 'Timestamp' section has a table with 'Timestamp dialect', 'Timestamp keyword', and 'Timestamp exp' columns, and 'Add' and 'Remove' buttons. At the bottom right are 'OK' and 'Cancel' buttons.

Figure 21-45 Required Confidentiality Dialog for body content decryption

After specifying the required confidentiality part, the corresponding information must be specified on the **Binding Configurations** page under **Request Consumer Binding Configuration Details**. You have to know the required token reference type and if a security token is required to be inserted or not. These settings should match the client configuration:

- ▶ If a security token is inserted in the request message, expand *Token Consumer* and click *Add* and complete the dialog:
 - Enter a name, for example, `con_enctcon` (Figure 21-46). To specify a token consumer, refer to “Configuring the server for security token” on page 489.
 - Select `com.ibm.wsspi.wssecurity.token.X509TokenConsumer` as the class.
 - Do not specify a security token. To specify a token consumer for the token used for encryption, a required security token is not specified in the extension, so it is not necessary to specify a security token.
 - Select *Use value type* and select *X509 certificate token*.
 - Select *Use jaas.config* and enter `system.wssecurity.X509BST` as the name. In addition, specify two `jaas.config` properties:
 - `com.ibm.wsspi.wssecurity.token.x509.issuerName:`
`CN=Server, OU=IBM, C=US`
 - `com.ibm.wsspi.wssecurity.token.x509.issuerSerial:` **42dbe072**
This is the value allocated when the key is generated (refer to “Generating sample key stores” on page 480 and Example 21-10 on page 483). If you generate your own keystore, the value will be different and must be entered correctly: It is the serial of the **server_rsa** alias.

Note: Even if a security token for encryption is not inserted in a request message, the token consumer should be specified at the consumer side. When you specify a token consumer for encryption and you use a self-signed certificate, you must specify `jaas.config` properties that refer to a trusted certificate.

Figure 21-46 shows a Token Consumer dialog for decryption of the SOAP body content.

Token Consumer dialog

Token consumer name:

Token consumer class:

Security token:

☒ Use value type

Value type:

Local name:

URI:

☒ Use jaas.config

jaas.config name:

jaas.config property:

Name	Value
com.ibm.wsspi.wsssecurity.token.x509.issuerName	CN=Server, OU=IBM, C=US
com.ibm.wsspi.wsssecurity.token.x509.issuerSerial	42dbe072

Add **Remove**

☐ Use trusted ID evaluator

Trusted ID evaluator class:

Trusted ID evaluator property:

Name	Value

Add **Remove**

☐ Use trusted ID evaluator reference

Trusted ID evaluator reference:

Property:

Name	Value

Add **Remove**

☐ Use certificate path settings

☐ Certificate path reference

Trust anchor reference:

Certificate store reference:

☒ Trust any certificate

OK **Cancel**

Value of the serial number of the server_rsa alias.

Figure 21-46 Token Consumer dialog for body content decryption

- Expand **Key Locators** and click **Add** to specify how to retrieve a key for decryption:
 - To specify key-related information, refer to “Configuring the client to specify an integrity part” on page 500.

- Enter a name, for example, con_encklocator (Figure 21-47).
- Select com.ibm.wsspi.wssecurity.keyinfo.KeyStoreKeyLocator as the class.
- Select *Use key store* and specify the server key store and server private key (refer to “Generating sample key stores” on page 480). In our case, we specify server, C:\SG246461\sampcode\mykeystore\server.jks and JKS.
- Click *Add* under Key and enter server_rsa as the alias, server_rsa as the key pass, and CN=Server, OU=IBM, C=US as the key name.

Figure 21-47 shows a Key Locator dialog for the decryption of the SOAP body content.

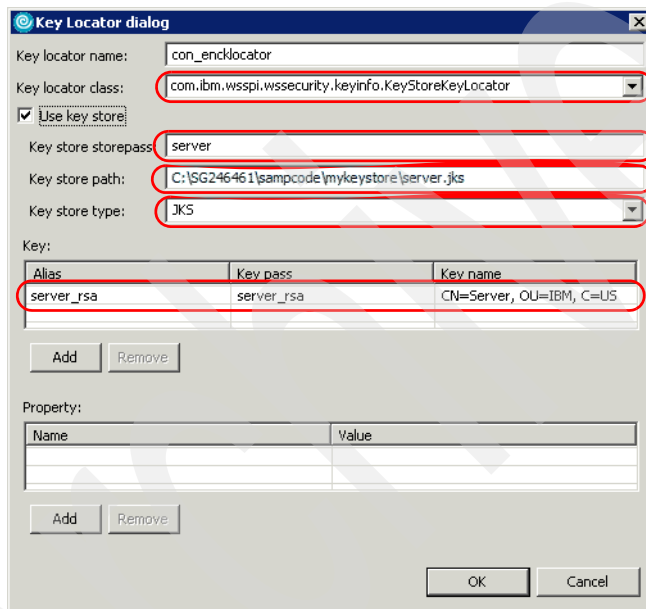


Figure 21-47 Key Locator dialog for body content decryption

- Expand *Key Information* and click *Add* to specify which type of security token reference is required:
 - To specify key information, refer to “Configuring the client to specify an integrity part” on page 500.
 - Enter a name, for example, con_enckeyinfo (Figure 21-48). The type should match the client configuration, select KEYID.
 - Select *Use key locator*, and then select con_encklocator and CN=Server, OU=IBM, C=US from the pull-down.
 - Select *Use token* and select con_enctcon.

Figure 21-48 shows a Key Information dialog for the decryption of SOAP body content.

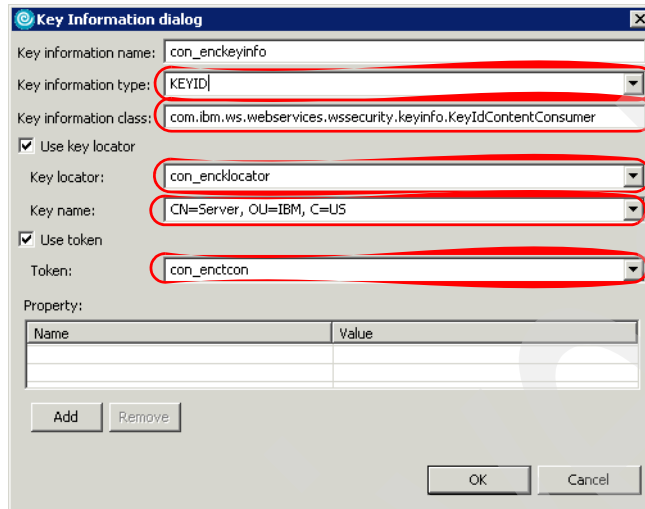


Figure 21-48 Key Information dialog for body content decryption

- Expand *Encryption Information* and click *Add* to define how to decrypt a required confidentiality part:
 - To specify encryption information, refer to “Configuring the client to specify a confidentiality part” on page 516.
 - Enter a name, for example, enc_body (Figure 21-49).
 - Select <http://www.w3.org/2001/04/xmlenc#tripledes-cbc> for the Data encryption method algorithm and http://www.w3.org/2001/04/xmlenc#rsa-1_5 for the Key encryption method algorithm.
 - Enter enc_kinfo as the Key information name and select con_enckeyinfo as the Key information element.
 - Select reqconf_body as the RequiredConfidentiality part.
- Save the configuration.

Figure 21-49 shows an Encryption Information dialog for the required confidentiality part.

Encryption name:

Data encryption method algorithm:

Key encryption method algorithm:

Encryption key information

Key information name:	Key information element:
enc_kinfo	con_enckeyinfo

Add Remove

RequiredConfidentiality part:

Property:

Name	Value

Add Remove

OK Cancel

Figure 21-49 Encryption Information dialog for body content decryption

Testing confidentiality

To test the confidentiality configuration, refer to “Testing on WebSphere Application Server” on page 531.

Confidentiality on the response message

If it is necessary to add confidentiality on the response message from the server, you have to configure the Response Generator in the server configuration and the Response Consumer in the client configuration:

- ▶ For the server configuration, you can configure the Response Generator Service Configuration Details on the Extensions page and the Response Generator Binding Configuration Details on the Binding Configurations page (refer to “Configuring the client to specify a confidentiality part” on page 516).
- ▶ For the client configuration, you can configure the Response Consumer Configuration on the WS Extension page and the Security Response Consumer Binding Configuration on the WS Binding page (refer to “Configuring the server to specify a confidentiality part” on page 521).

Adding a security timestamp

WebSphere Application Server Version 6.0 supports the insertion of a timestamp element under the security element in the SOAP header. It is defined in the OASIS WS-Security specification. Example 21-11 shows an example of a timestamp element.

Example 21-11 Example of timestamp element

```
<wsu:Timestamp
  xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd">
  <wsu:Created>2004-11-17T07:04:08.925Z</wsu:Created>
</wsu:Timestamp>
```

How to specify a security timestamp

To add a timestamp element in the client request message, you configure the Web services deployment descriptor.

Note: For this example, we assume that confidentiality is configured; that is, we continue from the confidentiality example.

Configuring the client to specify a security timestamp

To configure the security timestamp, open the deployment descriptor of the WeatherJavaBeanClientWeb project and go to the **WS Extension** page under **Request Generator Configuration**. The processes to specify the timestamp under Request Generator Configuration is the following:

- ▶ Expand *Add Timestamp* and select *Use Add Timestamp* (Figure 21-50).
- ▶ If desired, expand *Expires* and select *Use Expires*. Specify the expiration date, for example, 5 seconds.
- ▶ You can specify properties if necessary. The defined properties names and values are as follows:
 - `com.ibm.wsspi.wssecurity.timestamp.SOAPHeaderElement:`
1 or true
Sets the mustUnderstand flag to the timestamp element.
 - `com.ibm.wsspi.wssecurity.timestamp.dialect:`
`http://www.ibm.com/websphere/webservices/wssecurity/dialect-was`
Indicates the timestamp element insertion point is specified in another property.

- `com.ibm.wsspi.wssecurity.timestamp.keyword:`
SecurityFirst, SecurityLast, SOAPHeaderFirst, SOAPHeaderLast
Indicates where the timestamp element is inserted:
 - SecurityFirst: Insert as the first child element of the security header
 - SecurityLast: Insert as the last child element of the security header
 - SOAPHeaderFirst: Insert as the first child element of the SOAP header
 - SOAPHeaderLast: Insert as the last child element of the SOAP header

- Save the configuration.

Figure 21-50 shows a view of configuring the timestamp element for the client.

Figure 21-50 Add Timestamp page for client

Note: The property values might disappear after you move to the next value.

Configuring the server to specify a security timestamp

To configure how to receive the security timestamp, open the `webservices.xml` file in WeatherJavaBeanWeb with the Web Services Editor. On the **Extension** page under **Request Consumer Service Configuration Details**, do the following to specify how to receive timestamp elements:

- Expand *Add Timestamp* and select *Use Add Timestamp*. On the consumer side, selecting *Use Expires* is not necessary.
- You can specify some properties if necessary. The defined properties' names and values are as follows. The properties should match the client's configuration, except the property for setting the `mustUnderstand` flag. Refer "Configuring the client to specify a security timestamp" on page 528
- Save the configuration.

Figure 21-51 shows a view of configuring the timestamp element for the server.

Name	Value
com.ibm.wsspi.wssecurity.timestamp.dialect	com.ibm.wsspi.wssecurity.timestamp.dialect
com.ibm.wsspi.wssecurity.timestamp.keyword	com.ibm.wsspi.wssecurity.timestamp.keyword

Figure 21-51 Add Timestamp page for server

Testing the timestamp

To test the timestamp configuration, refer to “Testing on WebSphere Application Server” on page 531.

Timestamp in the response message

If it is necessary to add a timestamp to the response message from the server, you have to configure the Response Generator in the server configuration and the Response Consumer in the client configuration:

- ▶ For the server configuration, you can configure the Response Generator Service Configuration Details on the Extensions page and the Response Generator Binding Configuration Details on the Binding Configurations page (refer to “Configuring the client to specify a security timestamp” on page 528).
- ▶ For the client configuration, you can configure the Response Consumer Configuration on the WS Extension page and the Security Response Consumer Binding Configuration on the WS Binding page (refer to “Configuring the server to specify a security timestamp” on page 529).

Testing on WebSphere Application Server

To test the application with the WS-Security configuration, we have to change the server configuration. We secure the server where the Web service and client run.

Enabling security on the server

To enable the WS-Security configuration, WebSphere Application Server must run with global security enabled. If global security is off, a security token for authentication cannot be verified in the server. You change the server configuration using the administrative console of the server.

Configuring the server using the administrative console

To configure global security, follow these steps:

- ▶ Open the administrative console (select the server and *Run administrative console*). Log in to the administrative console.
- ▶ Expand *Security* → *Global security*. Under User registries (top right), select *Local OS*.
- ▶ Enter a user ID and password under General Properties. The user should be part of the Administrators group on Windows systems. Refer to the *WebSphere Application Server Information Center* for more details. Click *OK*.
- ▶ Back in Global security, select *Enable global security*, and then clear *Enforce Java 2 security*.
- ▶ If you want to authenticate a user by LTPA token, select *Lightweight Third Party Authentication (LTPA)* from the Active authentication mechanism list. Otherwise, select *Simple WebSphere Authentication Mechanism (SWAM)*.
- ▶ Select *Local OS* from the Active user registry list.
- ▶ Click *OK* and save the configuration.
- ▶ Log out from the administrative console.

Important: This is a setting when you use a Local OS user registry for user authentication. If you have to use another user registry, you should select and configure *Custom* or *LDAP* for the user registry. Also, select the corresponding registry from the Active user registry drop-down list. If the appropriate user registry is not specified, the security token verification fails, and a user cannot be authenticated.

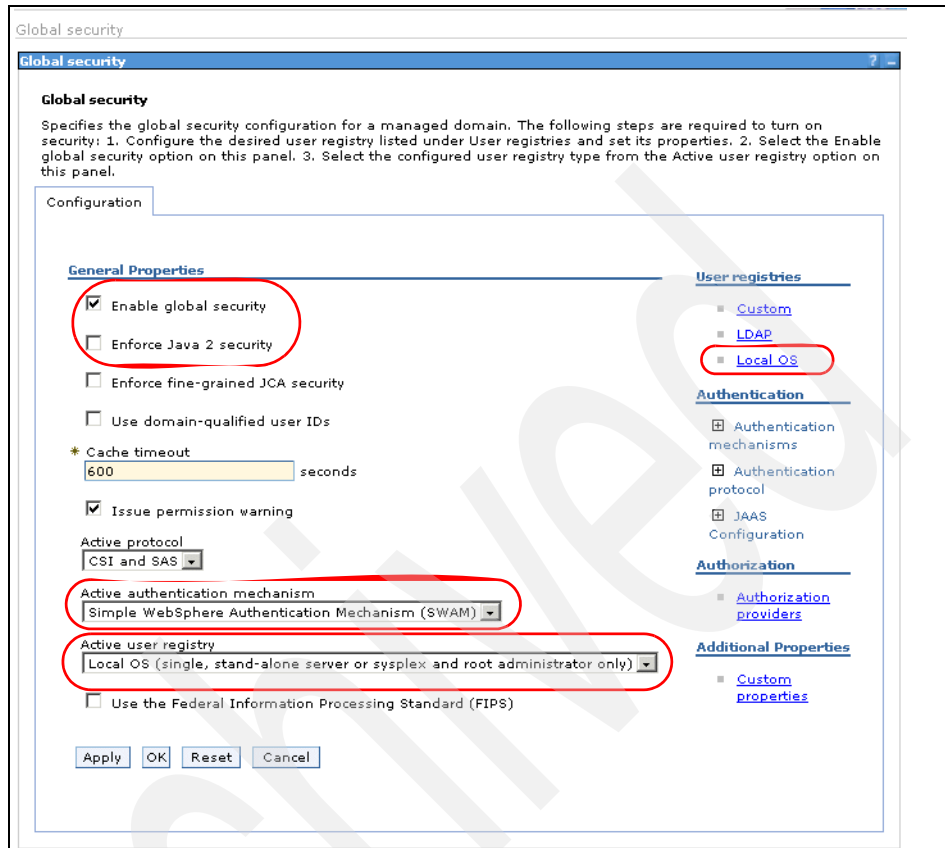


Figure 21-52 Global security configuration in administrative console

Configuring the server in Rational Application Developer

To test WS-Security in the WebSphere test environment in Rational Application Developer, complete the following steps:

- ▶ Open the server in the Server view (Figure 21-53).
- ▶ Expand *Security* (right side) and select *Enable security*.
- ▶ Enter the same user ID and password specified in administrative console.
- ▶ Save and close the configuration.

After configuring the server, restart the server. If security is enabled, you are prompted for a user ID and password when opening the administrative console. Use the user ID and password that you configured. This is an easy process to verify that security is enabled.

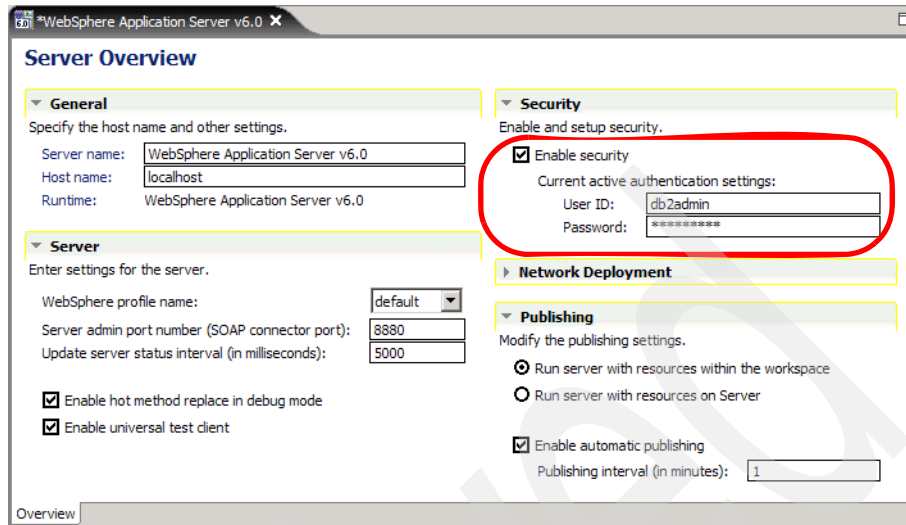


Figure 21-53 Server configuration for global security

In Version 6.0.1, a new selection for RMI or SOAP is available for the connection to the server. Select **SOAP** when using security (Figure 21-54).

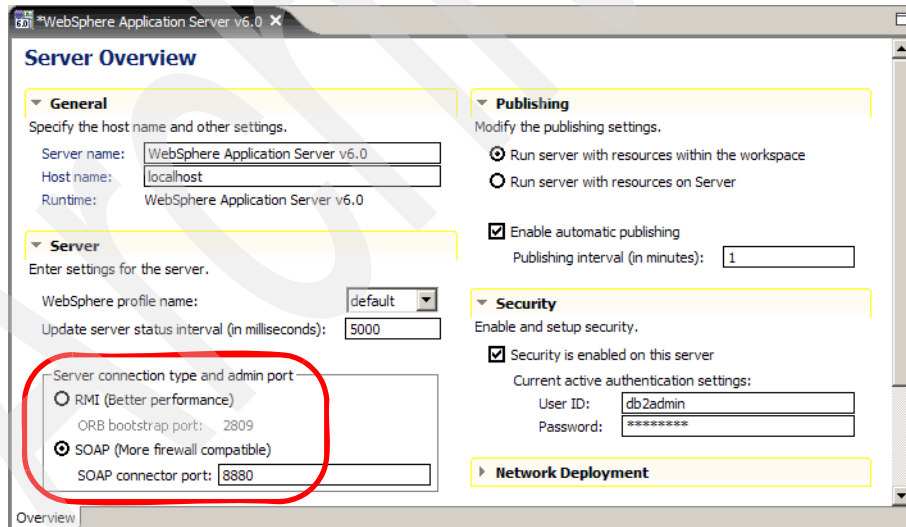


Figure 21-54 Server configuration for global security (Version 6.0.1)

Accessing the console when running with security

When you run the server with security enabled, the console output is not visible. You have to install a different file transfer application to retrieve the console. Run these commands:

```
C:
cd \<RAD60_HOME>\runtimes\base_v6\bin
call wsadmin.bat -profile redeployFileTransfer.jacl -lang jacl
-c "fileTransferAuthenticationOn
<cellname> <nodename> server1" -user <userid> -password <password>
```

Restart the server, and the console should show up.

Enabling the TCP/IP Monitor

If you want to see if WS-Security is applied to SOAP messages, you have to capture the SOAP message using the TCP/IP Monitor. See “TCP/IP Monitor” on page 368. Make sure that the TCP/IP Monitor is started on port 9081.

Tip: If you run the Web service to port 9081 and you get a connection error, stop the TCP/IP Monitor and restart it by selecting *Windows → Preferences → Internet → TCP/IP Monitor*. The server might report that it is running, but connection fails.

To force the client application through the TCP/IP Monitor, change the port in the proxy locator class. This is easier than changing the endpoint dynamically for every test.

Open the `WeatherJavaBean.wsdl` file in the `WeatherJavaBeanClientWeb` project (under `WebContent/WEB-INF/wsdl`) and change the SOAP address (at the bottom of the file):

```
<wsdlsoap:address location="http://localhost:9081/WeatherBeanWeb/.../...">
```

Save and close the file. This changes the `WeatherJavaBeanServiceLocator` class:

```
http://localhost:9081/WeatherBeanWeb/services/WeatherJavaBean
```

Note: In Version 6.0.1, changing the WSDL file does not change the service locator. You have to update the `WeatherJavaBeanServiceLocator` class.

Testing the application with WS-Security

After configuring the server, starting the TCP/IP Monitor, and preparing the client application, you can test the Web service security.

To expedite processing, we suggest that you only deploy the two required projects to the server: `WeatherJavaBeanServer` and `WeatherJavaBeanClient`. Select the server and *Add and remove projects* to remove other applications.

To run the tests, perform these steps:

- ▶ Before each test, restart the server and client projects by selecting the server and *Restart Project* → *WeatherJavaBeanServer* (and for the client, *Restart Project* → *WeatherJavaBeanClient*).
- ▶ Select the `TestClient.jsp` in the `WeatherJavaBeanClientWeb` project and *Run on Server*.
- ▶ Invoke the `getDayForecast` service, and both the request and response SOAP messages with WS-Security are captured by the TCP/IP Monitor.
- ▶ In the TCP/IP Monitor view, you can see the request message sent by a client in the left pane and the response message sent by the server in the right pane. Select *XML View* to format the message and to confirm the SOAP messages with security.

Note: The SOAP messages shown in this section only show one security option configured: authentication, integrity, or confidentiality. Messages will be different with multiple options configured.

Basic authentication

Example 21-12 shows the SOAP message with basic authentication sent by the client. The response message is not affected by authentication.

Example 21-12 Request message with basic authentication

```
<soapenv:Envelope xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Header>
    <wsse:Security xmlns:wsse="..." soapenv:mustUnderstand="1">
      <wsse:UsernameToken>
        <wsse:Username>XXXXXXX</wsse:Username>
        <wsse:Password Type="...#PasswordText">XXXXXXX</wsse:Password>
      </wsse:UsernameToken>
    </wsse:Security>
  </soapenv:Header>
  <soapenv:Body>
    <p821:getDayForecast xmlns:p821="http://bean.itso">
      <theDate>2004-11-16T15:00:00.000Z</theDate>
    </p821:getDayForecast>
  </soapenv:Body>
```

</soapenv:Envelope>

WebSphere Application Server Version 6.0 does not support password-digest of the username token, so the password is included in clear text in the message. We recommend that you use basic authentication over HTTPS, or also apply encryption.

Identity assertion

Example 21-13 shows the SOAP message with identity assertion sent by the client. This shows an example in which the client security token is the username token without the password and the trust mode is the signature.

Example 21-13 Request message with identity assertion

```
<soapenv:Envelope xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Header>
    <wsse:Security soapenv:mustUnderstand="1" xmlns:wsse="...">
      <wsse:BinarySecurityToken
        EncodingType="...#Base64Binary"
        ValueType="...#X509" wsu:Id="x509bst_5662298943609733191"
        xmlns:wsu="...">
        MIIBzzCCATigAwIBAgIEQZnQ7DANBgqhkiG9w0BAQQFADAsMQswCQYDVQQGEwJVUzEMM
        AoGA1UECxMDSUJNMQ8wDQYDVQQDEwZDbG11bnQwHhcNMDQxMTE2MTAwNTMyWhcNMDUwMj
        EOMTAwNTMyWjAsMQswCQYDVQQGEwJVUzEMMAoGA1UECxMDSUJNMQ8wDQYDVQQDEwZDbG1
        1bnQwgZ8wDQYJKoZIhvcNAQEBBQADgYOAMIGJAoGBALrQpIVkTwPc72jTzST3d+fKTYLO
        .....
      </wsse:BinarySecurityToken>
      <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
        <ds:SignedInfo>
          <ds:CanonicalizationMethod
            Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#">
            <ec:InclusiveNamespaces PrefixList="..."
              xmlns:ec="http://www.w3.org/2001/10/xml-exc-c14n#" />
          </ds:CanonicalizationMethod>
          <ds:SignatureMethod
            Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1" />
          <ds:Reference
            URI="#wssecurity_signature_id_8174905720277009053">
            <ds:Transforms>
              <ds:Transform
                Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#">
                <ec:InclusiveNamespaces PrefixList="..."
                  xmlns:ec="http://www.w3.org/2001/10/xml-exc-c14n#" />
              </ds:Transform>
            </ds:Transforms>
          </ds:Reference>
        </ds:SignedInfo>
      </ds:Signature>
    </wsse:Security>
  </soapenv:Header>
  <soapenv:Body>
    ...
  </soapenv:Body>
</soapenv:Envelope>
```



```

        <ds:DigestMethod
          Algorithm="http://www.w3.org/2000/09/xmlsig#sha1"/>
        <ds:DigestValue>e7HiBTGaEkxzXUXgH99YpcCTuK0=</ds:DigestValue>
      </ds:Reference>
    </ds:SignedInfo>
    <ds:SignatureValue>
      tWAdVxueRZ7Z/9EJhdgA0iBYsRKGZqwnT6aQCzG7fP+5HMOjfgYW2uY5sTOY6/W3Ce3+u
      JJdz1JCLVMH5RS2EYRyY6wW/y/6aYD0SKYUOfVUr1ZXm1VyrKu2XUcN03XzL4URxXlpeI
      7LmHx2N61YTnyi1tmqQ4yt072hUKUjPeE=
    </ds:SignatureValue>
    <ds:KeyInfo>
      <wsse:SecurityTokenReference>
        <wsse:Reference URI="#x509bst_5662298943609733191"
          ValueType="....#X509"/>
      </wsse:SecurityTokenReference>
    </ds:KeyInfo>
  </ds:Signature>
  <wsse:UsernameToken
    wsu:Id="wssecurity_signature_id_8174905720277009053" xmlns:wsu="...">
    <wsse:Username>XXXXXXX</wsse:Username>
  </wsse:UsernameToken>
</wsse:Security>
</soapenv:Header>
<soapenv:Body>
  <p821:getDayForecast xmlns:p821="http://bean.itso">
    <theDate>2004-11-22T15:00:00.000Z</theDate>
  </p821:getDayForecast>
</soapenv:Body>
</soapenv:Envelope>

```

Integrity

Example 21-14 shows the SOAP message with integrity sent by the client.

Example 21-14 SOAP message with integrity

```

<soapenv:Envelope xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Header>
    <wsse:Security xmlns:wsse="...." soapenv:mustUnderstand="1">
      <wsse:BinarySecurityToken .... EncodingType="....#Base64Binary"
        ValueType="....#X509" wsu:Id="x509bst_968757157168641698">
        MIIBzzCCATigAwIBAgIEQZnQ7DANBgkqhkiG9w0BAQQFADAsMQswCQYDVQQGEwJVUzEMM
        AoGA1UECxMDSUJNMQ8wDQYDVQQDEwZDbG11bnQwHhcNMDQxMTE2MTAwNTMyWhcNMDUwMj
        EOMTAwNTMyWjAsMQswCQYDVQQGEwJVUzEMMAoGA1UECxMDSUJNMQ8wDQYDVQQDEwZDbG1
        1bnQwgZ8wD.....=
      </wsse:BinarySecurityToken>
    </wsse:Security>
  </soapenv:Header>
  <soapenv:Body>
    <p821:getDayForecast xmlns:p821="http://bean.itso">
      <theDate>2004-11-22T15:00:00.000Z</theDate>
    </p821:getDayForecast>
  </soapenv:Body>
</soapenv:Envelope>

```

```

<ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
  <ds:SignedInfo>
    <ds:CanonicalizationMethod
      Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"
      <ec:InclusiveNamespaces PrefixList="..."
        xmlns:ec="http://www.w3.org/2001/10/xml-exc-c14n#"/>
    </ds:CanonicalizationMethod>
    <ds:SignatureMethod
      Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
    <ds:Reference URI="#wssecurity_signature_id_304715513213671414">
      <ds:Transforms>
        <ds:Transform
          Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"
          <ec:InclusiveNamespaces PrefixList="..."
            xmlns:ec="http://www.w3.org/2001/10/xml-exc-c14n#"/>
        </ds:Transform>
      </ds:Transforms>
      <ds:DigestMethod
        Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
      <ds:DigestValue>1fs/xSwe6i3oA9T9uWCICVoXVUY=</ds:DigestValue>
    </ds:Reference>
  </ds:SignedInfo>
  <ds:SignatureValue>
    muhi9oHKhDp3YkET6WGz/o6PqqLHrAck5MCYRM6fg+71Kmx5hL8fVWhMA4JSFuL5VFyca
    jircx7Mr9NACAT4BwjPxh38iUIpxE+WX56b8stx0EDo1Aq0fwf/wbNO/EZEWZJ9GS3m2
    LZXM2CtnR+PyYy7ri0iYARm/E2+MbPyY=
  </ds:SignatureValue>
  <ds:KeyInfo>
    <wsse:SecurityTokenReference>
      <wsse:Reference URI="#x509bst_968757157168641698"
        ValueType="...#X509"/>
    </wsse:SecurityTokenReference>
  </ds:KeyInfo>
</ds:Signature>
</wsse:Security>
</soapenv:Header>
<soapenv:Body ..... wsu:Id="wssecurity_signature_id_304715513213671414">
  <p821:getDayForecast xmlns:p821="http://bean.itso">
    <theDate>2004-11-18T15:00:00.000Z</theDate>
  </p821:getDayForecast>
</soapenv:Body>
</soapenv:Envelope>

```

Confidentiality

Example 21-15 shows the SOAP message with confidentiality sent by the client.

```

<soapenv:Envelope xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:soapenv="..." xmlns:xsd="..." xmlns:xsi="...">
  <soapenv:Header>
    <wsse:Security xmlns:wsse="..." soapenv:mustUnderstand="1">
      <EncryptedKey xmlns="http://www.w3.org/2001/04/xmlenc#">
        <EncryptionMethod Algorithm="http://.../xmlenc#rsa-1_5"/>
        <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
          <wsse:SecurityTokenReference>
            <wsse:KeyIdentifier
              ValueType="...#X509v3SubjectKeyIdentifier">
                Vniy7MU0XBumPoH1MNBdpiIWOPA=</wsse:KeyIdentifier>
            </wsse:SecurityTokenReference>
          </ds:KeyInfo>
          <CipherData>
            <CipherValue>
              BY8p8LB8LAdUovGKkfS9D6Xw1ZZnJDRBHgnor04hyGTgt9falsugn0snMoMUn80vnUOMJ
              03fwvazRvBL10yDAPxri706AJD1VtCagmXlwJrOKpzeyvZmq6ADd9KCTiAbr5116dMPzp
              F8xAXxMDXBQgyUiUFtArBiSGniZ23SRD8=
            </CipherValue>
          </CipherData>
          <ReferenceList>
            <DataReference URI="#wssecurity_encryption_id_5123762619732234012"/>
          </ReferenceList>
        </EncryptedKey>
      </wsse:Security>
    </soapenv:Header>
    <soapenv:Body>
      <EncryptedData xmlns="http://www.w3.org/2001/04/xmlenc#"
        Id="#wssecurity_encryption_id_5123762619732234012"
        Type="http://www.w3.org/2001/04/xmlenc#Content">
        <EncryptionMethod
          Algorithm="http://www.w3.org/2001/04/xmlenc#tripledes-cbc"/>
        <CipherData>
          <CipherValue>
            dsN0grBiIPRAaIrrL1SYqlB5yMeccCJTWM0+hxySDoGC17TJviOhQRSiHJz9sFtS3+0Lu
            CXeYKKxvPLaEVXS29deW0fei5zhS8uDERjgftRDDxsLYNomqMmFhrfCz3iYe1Beo0SDDe
            z4Lqx0ejP/RHi3ccIFE8SG6SYvVyIZzVo=
          </CipherValue>
        </CipherData>
      </EncryptedData>
    </soapenv:Body>
  </soapenv:Envelope>

```

Timestamp

Example 21-16 shows the SOAP message with the timestamp sent by the client.

Example 21-16 SOAP message with timestamp

```
<soapenv:Envelope xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:soapenv="..." xmlns:xsd="..." xmlns:xsi="...">
  <soapenv:Header>
    <wsse:Security xmlns:wsse="..." soapenv:mustUnderstand="1">
      <wsu:Timestamp xmlns:wsu="..." soapenv:mustUnderstand="1">
        <wsu:Created>2004-12-22T03:48:14.570Z</wsu:Created>
        <wsu:Expires>2004-12-22T03:48:19.590Z</wsu:Expires>
      </wsu:Timestamp>
    </wsse:Security>
  </soapenv:Header>
  ....
</soapenv:Envelope>
```

To test if the expiration date works, set the expiration to 0.1 seconds. Then, rerun the test to see if you receive this SOAP fault:

```
exception: com.ibm.wsspi.wssecurity.S SoapSecurityException: WSEC5323E: The
message was expired: creation date of timestamp [Tue Dec 21 20:04:14 PST
2004], expiration date of timestamp [Tue Dec 21 20:04:14 PST 2004], and
machines current date [{2}].
```

Debugging and tracing

You can include a Web service security trace in the server configuration using the administrative console:

- ▶ Open the administrative console and log in.
- ▶ Expand *Troubleshooting*, select *Logs and Trace*, and select the server.
- ▶ To change a trace file, select *Diagnostic Trace* under General Properties. Then, modify the trace file name and click *OK*.
- ▶ To add traced components, select *Change Log Detail Levels* under General Properties. Wait for the dialogs to fill in.
- ▶ In the view of the [All Components] tree, you can select components to be traced. To trace WS-Security components, expand the parent component (for example, `com.ibm.ws.*`), and then select the following components and *all* from the pop-up menu:
 - `com.ibm.ws.webservices.*`
 - `com.ibm.wsspi.wssecurity.*`
 - `com.ibm.xml.soapsec.*`

This creates entries in the form: `*=info: com.ibm.ws.webservices.*=all`

- ▶ Click *OK* and save the configuration.

After restarting the server, the WS-Security components are traced in the specified trace file when you invoke a Web service application with WS-Security.

Typical errors

Here, we provide information about typical errors. When you receive errors with WS-Security, it helps to determine the source of the problem. The error information is written to the trace log, SystemOut.log, FFDC files, and an exception message. You can find log files in this directory by default:

```
<WAS_HOME>\profiles\<profilename>\logs\<server name> | ffdc  
<RAD60_HOME\runtimes\base_v6\profiles\default\logs\server1 | ffdc
```

Misconfiguration

If the WS-Security configuration is invalid, the application with the configuration is not started. If the application cannot be started when the server (containing the application) is started, it is possible that the WS-Security configuration is invalid. Check the WS-Security configuration again if you cannot start the application.

Expired timestamp

A timestamp with an expiration date can be included in the SOAP message, for example, in the username token or under the security header. When the SOAP message with an expiration date is rejected, you get the message in SystemOut.log, as shown in Example 21-17.

Example 21-17 System out example of timestamp expiration

```
[12/21/04 20:04:14:410 PST] 0000004f enterprise I TRAS0014I: The following  
exception was logged WebServicesFault  
  faultCode: {http://docs.oasis-open.org/wss/2004/01/  
    oasis-200401-wss-wssecurity-utility-1.0.xsd}MessageExpired  
  faultString: com.ibm.wsspi.wssecurity.S SoapSecurityException: WSEC5323E:  
    The message was expired: creation date of timestamp [Tue Dec 21  
    20:04:14 PST 2004], expiration date of timestamp [Tue Dec 21  
    20:04:14 PST 2004], and machines current date [{2}].  
  faultActor: null  
  faultDetail:  
  
com.ibm.wsspi.wssecurity.S SoapSecurityException: WSEC5323E: The message was  
expired: creation date of timestamp [Tue Dec 21 20:04:14 PST 2004], expiration  
date of timestamp [Tue Dec 21 20:04:14 PST 2004], and machines current date  
[{2}].  
  at com.ibm.ws.webservices.engine.WebServicesFault.makeUserFault  
    ( WebServicesFault.java:211)
```

Invalid key store

If the configured key store path is invalid, the runtime cannot get the key for integrity or confidentiality. The FFDC file gets the message shown in Example 21-18. In this case, the key store path is invalid, but other possible key

store-related errors are that the key store storepass, keypass, or key store type is not correct. In those cases, a similar exception message is written to the FFDC file and the trace log.

Example 21-18 FFDC example of invalid key store path

```
Exception = java.io.FileNotFoundException
Source = com.ibm.xml.soapsec.util.ConfigUtil.createKeyStore
probeid = 194
Stack Dump =
  java.io.FileNotFoundException: C:\SG246461\xxxxcode\mykeystores\client.jks
  (The system cannot find the path specified)
    at java.io.FileInputStream.open(Native Method)
    at java.io.FileInputStream.<init>(FileInputStream.java:129)
  .....

```

Repeated nonce

If the message has a nonce value, and the value is same as the one that the server received before, you get a message in the FFDC file (Example 21-19).

Example 21-19 FFDC example of repeated nonce exception

```
Exception = com.ibm.wsspi.wssecurity.SoapSecurityException
Source = com.ibm.ws.webservices.wssecurity.core.WSSConsumer.invoke
probeid = 464
Stack Dump = com.ibm.wsspi.wssecurity.SoapSecurityException:
  WSEC5321E: Repeated nonce (randomly generated value).
  .....

```

Repeated nonce in the cluster

WebSphere Application Server 6.0 supports a distributed nonce cache in a cluster environment. If a server wss01 and a server wss02 exist in the same cluster, and the distributed nonce cache is enabled, the same nonce cannot be received by both servers. The configuration of the distributed nonce cache is described in “Configuring the distributed nonce cache in a cluster environment” on page 552.

When a SOAP message with a nonce value is sent to wss01, and a message with the same nonce value is sent to wss02, a SOAP fault message is returned. In this case, you can see the following errors in the trace log (Example 21-20) and FFDC file (Example 21-21) in the server wss0.

Example 21-20 Trace log example of nonce cache exception

```
.....
The following exception was logged
  com.ibm.wsspi.wssecurity.SoapSecurityException:

```

```
WSEC5321E: Repeated nonce (randomly generated value).  
at  
com.ibm.wsspi.wssecurity.S SoapSecurityException.format(S SoapSecurityException  
.java:68)  
at  
com.ibm.ws.webservices.wssecurity.util.NonceUtil.checkNonce(NonceUtil.java:  
375)  
.....
```

Example 21-21 FFDC file example of nonce cache exception

```
Exception = com.ibm.wsspi.wssecurity.S SoapSecurityException  
Source = com.ibm.ws.webservices.wssecurity.core.WSSConsumer.invoke  
probeid = 464  
Stack Dump = com.ibm.wsspi.wssecurity.S SoapSecurityException:  
WSEC5321E: Repeated nonce (randomly generated value).  
.....
```

Generating WS-Security sample configurations

When running the Web Service wizard in Application Developer, you can generate WS-Security definitions for both integrity (XML digital signature) and confidentiality (XML encryption):

- ▶ Server-side: Figure 16-5 on page 280 shows where security can be specified for the service project.
- ▶ Client-side: Figure 16-7 on page 281 shows where security can be specified for the client project.

The selections are:

- ▶ No Security
- ▶ XML Digital Signature
- ▶ XML Encryption
- ▶ XML Digital Signature and XML Encryption

Important: The generated security definitions use the WebSphere Application Server sample key stores. These are good for testing, but should not be used in a real application.

Running the Web Service wizard with security

Here, we provide brief instructions for running the wizard with security:

- ▶ Remove global security from the server (clear global security in the administrative console and in the server settings). Restart the server.
- ▶ Run the Web Service wizard starting with the WeatherJavaBean. Follow the instructions in “Creating a Web service from a JavaBean” on page 275:
 - For the security configuration, select *XML Digital Signature and XML Encryption* for both the server and the client.
 - Skip generating the test client JSPs (they do not change).
- ▶ Analyze the generated deployment information:
 - Open the deployment descriptor of the WeatherJavaBeanClientWeb project and study the WS Extension and WS Binding pages.
 - Open the webservices.xml file in the WeatherJavaBeanWeb project and study the Extension and Binding Configurations pages.

Note that integrity and confidentiality are configured for both the request and the response messages.

The configuration looks somewhat different than the manual configuration described in “Development of WS-Security” on page 474.

Note: The Web Service wizard does not use some of the new options of WS-Security that are supported in Version 6. It might be useful to learn about WS-Security if you are not familiar with security configurations.

- ▶ You can run the TestClient.jsp and monitor the request and response messages. Both are signed and encrypted.

Scenario to generate and modify security definitions

It is feasible to run the Web Service wizard with security to generate basic security definitions in the deployment descriptors and then update the information:

- ▶ Change integrity and confidentiality to use self-generated key stores and keys. The WebSphere sample key stores should not be used for a real application.
- ▶ Test that integrity and confidentiality work after the modifications.
- ▶ Add authentication.
- ▶ Optionally, add a timestamp.

Modifying generated definitions to use own key stores

Here, we provide short instructions about how to modify the generated definitions to use the self-generated key stores. Open both the client deployment descriptor (`web.xml` in `WeatherJavaBeanClientWeb`) and the server deployment descriptor (`webservices.xml` in `WeatherJavaBeanWeb`).

Client request (generator)

On the **WS Binding** page, expand **Security Request Generator Binding Configuration** and make the changes outlined below.

Token Generator

- ▶ Key store path: `C:/SG246461/sampcode/mykeystore/client.jks`
- ▶ Key: `client_rsa, client_rsa, CN=Client, OU=IBM, C=US`

Key Locator: sig_klocator1

- ▶ Key store path: `C:/SG246461/sampcode/mykeystore/client.jks`
- ▶ Key: `client_rsa, client_rsa, CN=Client, OU=IBM, C=US`

Key Locator: gen_klocator

- ▶ Key store storepass: `client`
- ▶ Key store path: `C:/SG246461/sampcode/mykeystore/client.jks`
- ▶ Key store type: `JKS`
- ▶ Key: `server_rsa, server_rsa, CN=Server, OU=IBM, C=US`

Key Information: gen_signkeyinfo

- ▶ Key name: `CN=Client, OU=IBM, C=US`

Key Information: gen_enckeyinfo

- ▶ Key name: `CN=Server, OU=IBM, C=US`
- ▶ Clear *Use token* (this might reappear if you open it again)

There are no changes for Signing Information, Part References, Transforms, and Encryption Information.

Server request (consumer)

On the **Binding Configurations** page, expand **Request Consumer Binding Configuration Details** and make the changes outlined below.

Trust anchor

- ▶ Key store path: `C:/SG246461/sampcode/mykeystore/server.jks`

Certificate store list

- ▶ Key store path: `C:/SG246461/sampcode/mykeystore/server_rsa.cer`

Key Locator: con_klocator

- ▶ Key store storepass: server
- ▶ Key store path: C:/SG246461/sampcode/mykeystore/server.jks
- ▶ Key store type: JKS
- ▶ Key: server_rsa, server_rsa, CN=Server, OU=IBM, C=US

There are no changes for Token Consumer (both), Key Locator (sig_klocator1), Key Information (both), Signing Information, Part References, Transforms, and Encryption Information.

Server response (generator)

On the **Binding Configurations** page, expand **Response Generator Binding Configuration Details** and make the changes outlined below.

Token Generator

- ▶ Key store path: C:/SG246461/sampcode/mykeystore/server.jks
- ▶ Key: server_rsa, server_rsa, CN=Server, OU=IBM, C=US

Key Locator: sig_klocator2

- ▶ Key store path: C:/SG246461/sampcode/mykeystore/server.jks
- ▶ Key: server_rsa, server_rsa, CN=Server, OU=IBM, C=US

Key Locator: gen_klocator

- ▶ Key store storepass: server
- ▶ Key store path: C:/SG246461/sampcode/mykeystore/server.jks
- ▶ Key store type: JKS
- ▶ Key: client_rsa, client_rsa, CN=Client, OU=IBM, C=US

Key Information: gen_signkeyinfo

- ▶ Key name: CN=Server, OU=IBM, C=US
- ▶ Select *Use token* and select *gen_signtgen* (this might disappear if you open it again)

Key Information: gen_enckeyinfo

- ▶ Key name: CN=Client, OU=IBM, C=US
- ▶ Clear *Use token* (this might reappear if you open it again)

There are no changes for Signing Information, Part References, Transforms, and Encryption Information.

Client response (consumer)

On the **WS Binding** page, expand **Security Response Consumer Binding Configuration** and make the changes outlined below.

Trust anchor

- ▶ Key store path: C:/SG246461/sampcode/mykeystore/client.jks

Certificate store list

- ▶ Key store path: C:/SG246461/sampcode/mykeystore/client_rsa.cer

Key Locator: con_klocator

- ▶ Key store storepass: client
- ▶ Key store path: C:/SG246461/sampcode/mykeystore/client.jks
- ▶ Key store type: JKS
- ▶ Key: client_rsa, client_rsa, CN=Client, OU=IBM, C=US

There are no changes for Token Consumer (both), Key Locator (sig_klocator2), Key Information (both), Signing Information, Transforms, and Encryption Information.

Testing the modifications

Restart the WeatherJavaBeanServer and WeatherJavaBeanClient applications in the server. Then, run the TestClient.jsp in the WeatherJavaBeanClientWeb project. The digital signature and encryption are applied to both the request and response messages.

Adding authentication and timestamp

After modifying the configuration to use your own key stores, you could add basic authentication and an optional timestamp.

Configuring WS-Security on a real Application Server

In this section, we describe how to configure WS-Security on a production WebSphere Application Server.

Modifying binding configurations

WebSphere Application Server has a default binding configuration, as described in “Configuration structure” on page 469. You can specify the default binding as the application binding. The WebSphere Application Server 6.0 administrative console can be used to modify the default binding and the application binding to refer to the configuration in the default binding.

Modifying the default binding configuration

To modify the default binding configuration, perform the following steps:

- ▶ Open the administrative console of the server. Expand *Servers* → *Application servers* and select the server.
- ▶ Select *Web services: Default bindings for Web services security* under *Security* (right side).

Figure 21-55 shows the default generator bindings and default consumer bindings.

Application servers

Application servers > server1 > Web services: Default bindings for Web services security

Specifies a default binding for Web services security. If a Web services archive does not have its own binding, WebSphere Application Server uses the default binding. Otherwise, the application server uses the binding in the archive.

Configuration

General Properties

Nonce cache timeout: 600 seconds

Nonce maximum age: 300 seconds

Nonce clock skew: 0 seconds

☐ Distribute nonce caching

Apply OK Reset Cancel

Default generator bindings

- Signing information
- Encryption information
 - Key information
- Token generators
 - Properties

Default consumer bindings

- Signing information
- Encryption information
 - Key information
- Token consumers
 - Properties

Additional Properties

- Key locators
- Collection certificate store
- Trust anchors
- Trusted ID evaluators
- Algorithm mappings
- Login mappings
- Properties

Figure 21-55 Default binding configuration

You can see the provided default binding configurations under each component, such as the default signing information for the message generator. You can add and modify each component. The sections of each component are similar to the dialog boxes shown in the editors of Application Developer, although some sections are different.

Here is the list of the differences between the administrative console and the editors of Rational Application Developer:

► **Signing information**

No part references. The administrative console can modify only the binding configurations, so the part that is linked to a deployment descriptor cannot be modified.

► **Encryption information**

No part references. The administrative console can modify only the binding configurations, so the part that is linked to a deployment descriptor cannot be modified.

► **Key information**

- Added encoding method. This specifies an encoding algorithm used to generate a key identifier. It is available when the key information type is the key identifier.
- Added calculation method. This specifies a calculation algorithm used to generate a key identifier. It is available when the key information type is the key identifier.
- Added value type. Enter the namespace URI and local name of the value type of the keyinfo element when you have to specify these.
- Get keys to support entering a key name reference. To refer to a key locator from the key information, select one from the key locator reference list and click *Get keys*. Names of the keys that are defined in the selected key locator are listed in the key name reference, and you can select one of them.
- No key name at the consumer. This is an optional element at the consumer side.

► **Token generator**

- No part references. The administrative console can modify only binding configurations, so the part that is linked to a deployment descriptor cannot be modified.
- Added *Add nonce*. This is available for the username token generator. When it is selected, a nonce is included in the generated username token.
- Added *Add timestamp*. This is available for the username token generator. When it is selected, a timestamp is included in the generated username token.

- Added Identity assertion options under the callback handler: Selecting *identity assertion* declares to send the username of the originator for a username token generator. Selecting *RunAs identity* declares to send the RunAs identity in a username token.

► **Token consumer**

- No part references. The administrative console can modify only binding configurations, so the part that is linked to a deployment descriptor cannot be modified.
- Added *Verify nonce*. This is available for the username token consumer. When it is selected, the nonce is verified in the username token.
- Added *Verify timestamp*. This is available for the username token consumer. When it is selected, the timestamp is verified in the username token.

► **Trusted ID evaluators**

You can define the trusted ID evaluator class in the default binding. The defined trusted ID evaluator class can be referred from an application binding configuration.

► **Algorithm mappings**

You can add custom algorithms for the signature, message digest, data encryption, and key encryption.

► **Login mappings**

This is a component for WebSphere Application Server Version 5.x.

After modifying the default bindings, save the configuration and restart the server. You can refer to the default bindings from the application bindings.

The provided default binding specifies a signature using an X.509 certificate token referred by a security token reference and encryption using an X.509 certificate token referred by a key identifier. Key-related information is also included, using keys that are in the WebSphere Application Server sample key stores.

Important: The sample key store in WebSphere Application Server is specified by key locators in the default binding. It can be used for testing WS-Security, but you should use our own key store for your real application and change the key locator configuration in the default binding.

Modifying the application binding configuration

The binding configurations of the installed applications can be modified in the administrative console. To open the application binding configuration, follow the steps for a client binding [for a server binding in brackets]:

- ▶ Open the administrative console of the server. Expand *Applications* → *Enterprise Applications*. Select the `WeatherJavaBeanClient` [WeatherJavaBeanServer].
- ▶ Select *Web modules* under Related items and `WeatherJavaBeanClientWeb.war` [WeatherJavaBeanWeb.war].
- ▶ Select *Web services: Client security bindings* [Web services: Server security bindings] under Additional Properties.
- ▶ If the client has a WS-Security configuration, you can edit the Request generator (sender) binding [Response consumer (receiver) binding] and the Response consumer (receiver) binding [Request generator (sender) binding]. Open a binding that you want to modify by clicking *Edit*.
- ▶ In the application binding panel, only the components that you are able to modify appear. Select the information that you want to change.

You can use a default binding for the application. To use a default binding, select *Use defaults* under General properties. Click *OK* and save the configuration. Each binding is replaced by a default configuration as follows:

- ▶ Request generator (sender) binding → Default generator bindings
- ▶ Response consumer (receiver) binding → Default consumer bindings
- ▶ Response consumer (receiver) binding → Default consumer bindings
- ▶ Request generator (sender) binding → Default generator bindings

Adding a custom JAAS configuration

As mentioned in “Configuring the server for security token” on page 489, you can add your custom JAAS configuration name to consume your custom token. If you add a new JAAS configuration, follow these steps:

- ▶ Open the administrative console of the server. Expand *Security* → *Global security*.
- ▶ Expand *JAAS Configuration* under Authentication (right side). Select *System logins*.
- ▶ Click *New* and enter an alias name that is used as a part of the JAAS configuration name in the WS-Security configuration. For example, when you specify `custom`, the JAAS configuration name is `system.custom` (Figure 21-56). Click *Apply*.

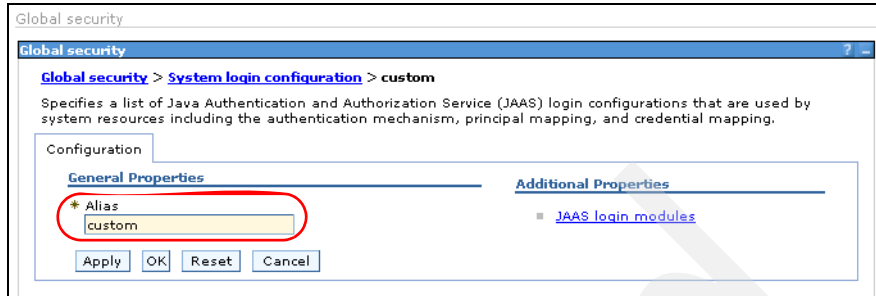


Figure 21-56 System login configuration

- ▶ Select *JAAS login modules* under Additional Properties (it is now enabled).
- ▶ Click *New* and enter the Module class name of your custom JAAS login module class name (Figure 21-57).
- ▶ Click *OK* and save the configuration.

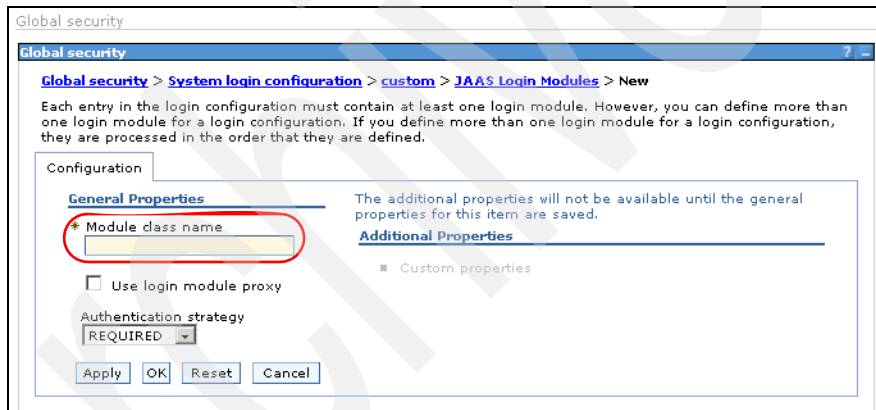


Figure 21-57 JAAS Login Modules

Configuring the distributed nonce cache in a cluster environment

As described in “Distributed nonce cache” on page 465, WebSphere Application Server Version 6.0 supports a distributed nonce cache to check message uniqueness and avoid reply attacks. It should run on the WebSphere Application Server cluster using a distributed map. Follow these steps to configure the cache:

- ▶ Make sure that you have created an appropriate *replication domain* setting when you formed the cluster.

- ▶ Open the default bindings of each server in the administrative console. Select *Distribute nonce caching* under General properties (Figure 21-58 left).
- ▶ Open *Environment* → *Replication domains* in the administrative console, and then select one replication domain.
- ▶ Make sure that the replication domain is properly secured. The nonce cache is crucial to the integrity of the nonce checking. If the nonce cache is compromised, the results of the nonce validation cannot be trusted. We select 3DES as the encryption type and click *Regenerate encryption key*.
- ▶ Select *Entire Domain* under Number of replicas (Figure 21-58 right).
- ▶ Click *OK* and save the configuration.

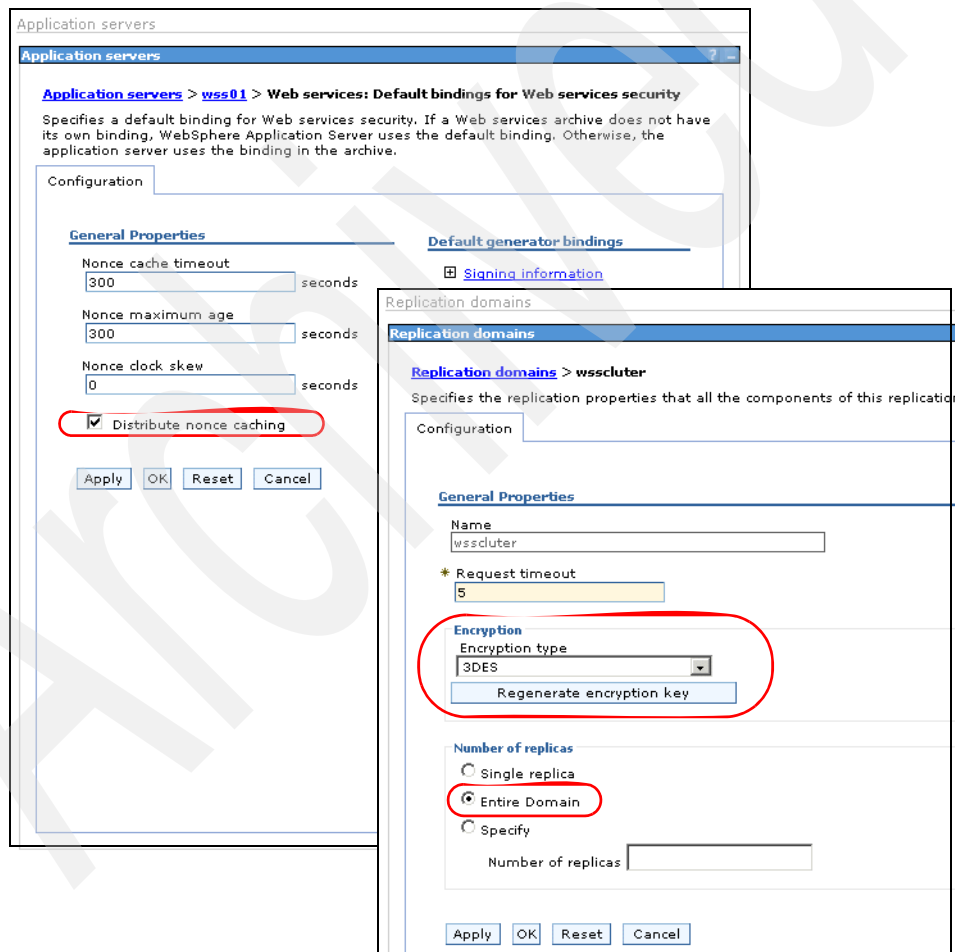


Figure 21-58 Distribute nonce caching and Entire Domain selection

- Open *Application servers* and select the server. Select *Dynamic Cache Service* under Container Services. Select *Enable cache replication* under Consistency settings (Figure 21-59). Click *OK* and save the configuration. This should be selected for all servers in the cluster.
- Restart the cluster.

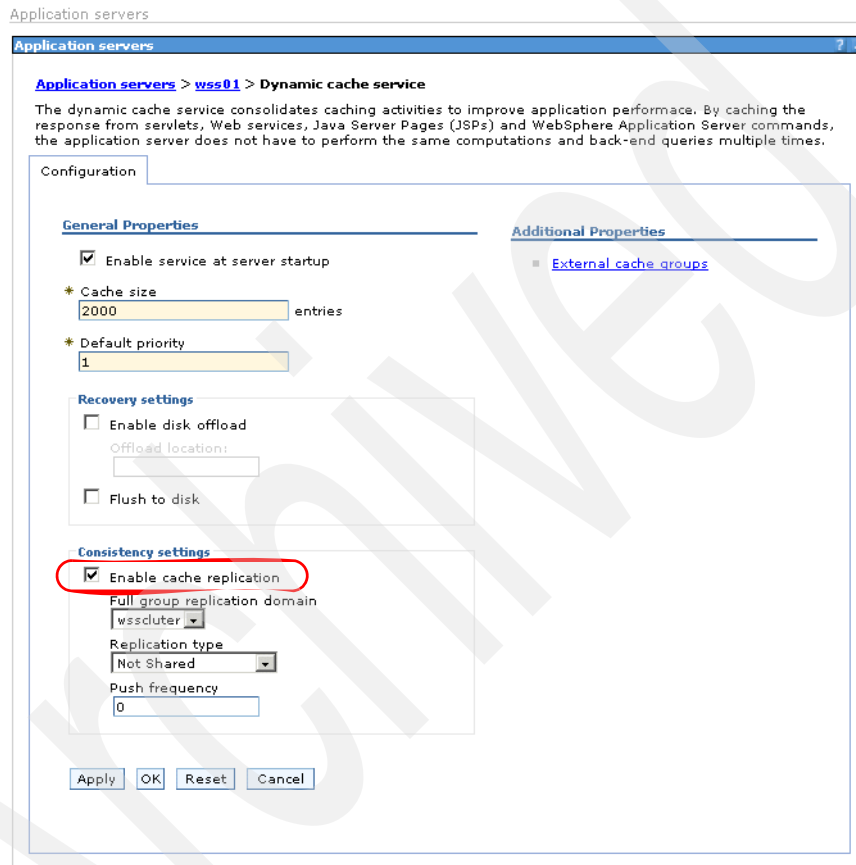


Figure 21-59 Enable cache replication selection in dynamic cache service

Configuring certificate caching

As described in “Certificate caching” on page 466, WebSphere Application Server Version 6.0 can cache received certificates to reuse in the next request to help improve performance. Follow these steps to configure the cache:

- Open the default bindings in the administrative console (open *Application servers*, select the server, and then select *Web services: Default bindings for Web services security*). Select *Properties* under Additional Properties.

- ▶ Click *New* and create the following property to enable or disable certificate caching (click *Apply* when finished):
 - `com.ibm.ws.wssecurity.config.token.certificate.useCache`
 - Value: true or false
- ▶ If you want to specify an interval of cache timeout, add the following property and click *Apply*:
 - `com.ibm.ws.wssecurity.config.token.certificate.cacheTimeout`
 - Value: timeout interval in seconds
- ▶ Save the configuration and restart the server.

Figure 21-60 shows a certificate caching configuration.

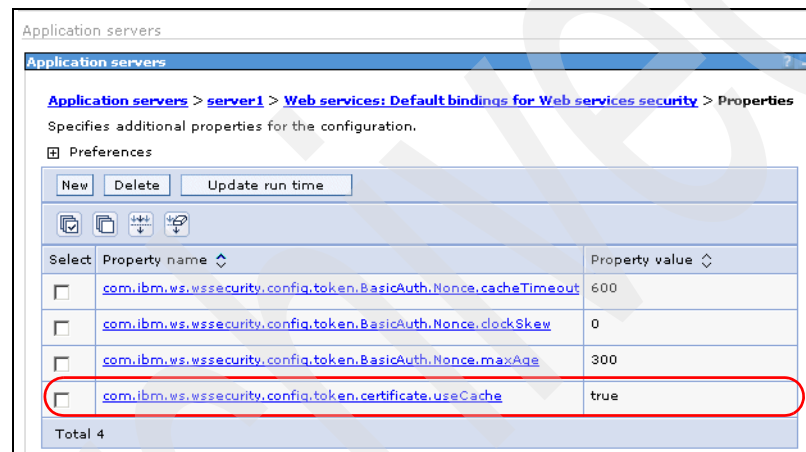


Figure 21-60 Certificate caching configuration

Compatibility and migration with Version 5.x

WebSphere Application Server Version 6.0 provides limited support for an application with Version 5.x WS-Security. This means that an application with Version 5.x WS-Security can run on WebSphere Application Server Version 6.0, but the WS-Security runtime for Version 5.x is invoked for the application. Therefore, it does not use the new features of Version 6.0.

In the following sections, we describe how Version 5.x applications are supported and how to migrate to Version 6.0.

WS-Security runtimes in WebSphere Application Server V6.0

WebSphere Application Server V6.0 has two types of the WS-Security runtime: one is for Version 5.x, the other is for Version 6.0 (Figure 21-61). Because there are two runtimes, the server can receive a request from both a J2EE 1.3 client with Version 5.x and a J2EE 1.4 client with Version 6.0 WS-Security.

However, a J2EE 1.3 client cannot connect to the Version 6.0 runtime and cannot use the new features of Version 6.0 WS-Security. If you want to use Version 6.0 functions in your secure Version 5.x application, you have to migrate from Version 5.x to Version 6.0, which is described in the next section.

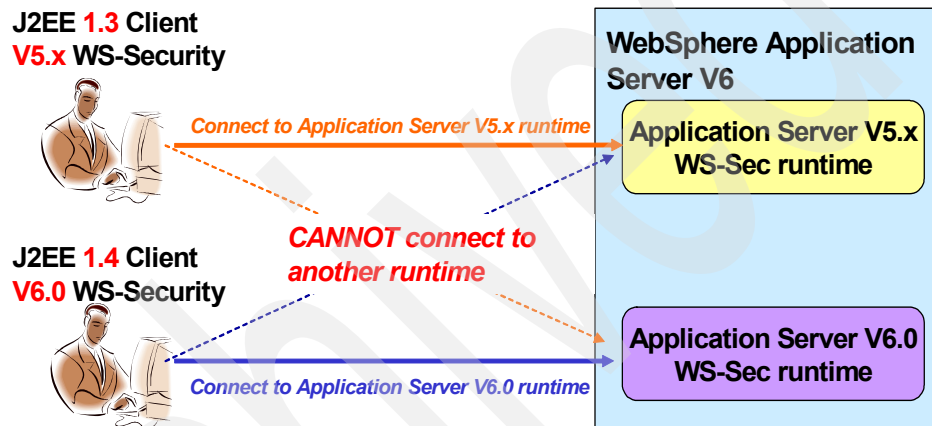


Figure 21-61 WS-Security runtimes in WebSphere Application Server V6.0

Migrating from Version 5.x

WebSphere Application Server Version 5.x supports securing applications by WS-Security, but it supports limited functions of WS-Security. Many new functions are supported by WebSphere Application Server Version 6.0, so configuration formats have changed from Version 5.x to Version 6.0.

If you have a J2EE 1.3 enterprise application with Version 5.x WS-Security, you can install it in Version 6.0, and Version 5.x WS-Security functions are enabled. The Version 5.x EAR file with WS-Security is based on J2EE 1.3, but the Version 6.0 application EAR file with WS-Security should be based on J2EE 1.4. Therefore, if you want to use the Version 6.0 WS-Security functions, you have to update the J2EE 1.3 EAR file to J2EE 1.4. Two migration steps are necessary:

- Update the J2EE version of the application from J2EE 1.3 to J2EE 1.4.
- Reconfigure the WS-Security configuration for Version 6.0.

How to migrate an application with Version 5.x WS-Security

To migrate a Version 5.x Web services application to Version 6.0, you can use the J2EE Migration Wizard. This wizard cannot migrate WS-Security configurations to Version 6, you have to reconfigure WS-Security in the Version 6.0 configuration format by referring to the Version 5.x configuration.

Here, we describe the recommended process to migrate a Version 5.x application with WS-Security to Version 6.0:

- ▶ Import the client and server EAR files into Rational Application Developer. For this description, assume that a `SampleClientWeb` project and a `SampleServerEJB` project are created.
- ▶ Import the EAR files again, but overwrite the project names that are generated, for example, `SampleClient2Web` and `SampleServer2EJB`.
- ▶ Open the J2EE perspective and the Project Explorer. Select the `SampleClient2Web` project and *Migrate* → *J2EE Migration Wizard* from the context menu. Read the information and click *OK* if you agree on the information. The `SampleClient2Web` project is migrated. Migrate the EJB project as well.
- ▶ To confirm that the Version 5.x application was migrated, open the *Properties* of the projects. Select *J2EE* to verify that the J2EE level is 1.4. You can now edit the WS-Security configurations using the appropriate editors:
 - Open the `web.xml` file of the `SampleClient2Web` project in the Deployment Descriptor Editor.
 - Open the `webservices.xml` file of the `SampleServer2EJB` project in the Web Services Editor.
- ▶ Reconfigure the WS-Security configuration in the WS Extension and WS Binding tabs (client) or Extensions and Binding Configurations tabs (server) by referring to the Version 5.x configuration of the original projects.

To refer to the Version 5.x configuration, open the `webservicesclient.xml` of the `SampleClientWeb` project in the Web Services Client Editor and the `webservices.xml` file of the `SampleServerEJB` project in the Web Services Editor. You can find the Version 5.x configurations in the Security Extensions and Port Binding tabs (client) or Security Extensions and Binding Configurations tabs (server).
- ▶ After finishing the configuration for Version 6.0, save the configuration files.

To reconfigure Version 5.x WS-Security in the Version 6.0 configuration, refer to “Configuration mapping: Version 5.x to Version 6.0” on page 666.

Summary

In this chapter, we described Web services security functions and how to configure Web services security in WebSphere Application Server Version 6.0. The WS-Security runtime in WebSphere Application Server Version 6.0 has an extensible architecture to support WS-Security-related specifications. The typical WS-Security configuration for authentication, integrity, and confidentiality is provided in this chapter, but you can configure more complex security scenarios.

The WS-Security Version 6.0 runtime has many added functions. The configuration of WS-Security is different for the two versions. To help you migrate Version 5.x to Version 6.0, we provided information about how to migrate the WebSphere Application Server Version 5.x Web services application and configuration mappings between Version 5.x and Version 6.0.

More information

The WS-Security runtime of Version 6.0 has many predefined properties. “List of predefined properties” on page 689 provides a list of these predefined properties.

Before starting the WS-Security configuration, it is good to make a list of configuration items first. For this purpose, “Forms for WS-Security configuration” on page 692 provides blank forms for the WS-Security configuration.

The best source for updated information is the *WebSphere Application Server Information Center*, available at:

<http://www.ibm.com/software/webservers/appserv/infocenter.html>

Web services and the service integration bus

This chapter describes Web services in the context of the service integration bus (simply called *bus* in this chapter), which is a key technology in IBM WebSphere Application Server Version 6.0.

Web services in the bus can be used as an extra layer between service requestors and service providers, allowing control over the flow, routing, and transformation of messages through mediations and JAX-RPC handlers.

The bus provides a flexible way to expose and call services located in an intranet from the Internet (and vice versa), while also providing mechanisms for protocol switching and security.

In addition, the Web services gateway, a core part of the bus, can be used to define proxy services that dynamically retarget service requests at runtime.

WebSphere Application Server Version 6.0 also provides tools to enable the migration of a Version 5.x gateway into the bus.

Overview

The use of Web services with the service integration bus is an evolution of the Web services gateway provided in WebSphere Application Server Version 5.x. Whereas the gateway was a stand-alone application, the bus is more tightly integrated into the Application Server, enabling users to take advantage of WebSphere Application Server administration and scalability options, and also build on top of the asynchronous messaging features provided by WebSphere Application Server.

The bus enables users to specify a level of indirection between service requestors and providers by exposing existing services at new destinations. It also provides options for managing these services through *mediations*, which can access and manipulate incoming and outgoing message content, or even route the message to a different service. Support for JAX-RPC handlers is also included in the bus, as is Web services gateway functionality.

Figure 22-1 gives a good illustration of a basic bus configuration and how it can be used to enable Web services clients in an intranet to access an Internet-based Web service. Clients would use the bus-generated WSDL to access the service, and the specified *mediations* could be used for message logging or transformation purposes. We discuss more complex bus configurations throughout the rest of this chapter.

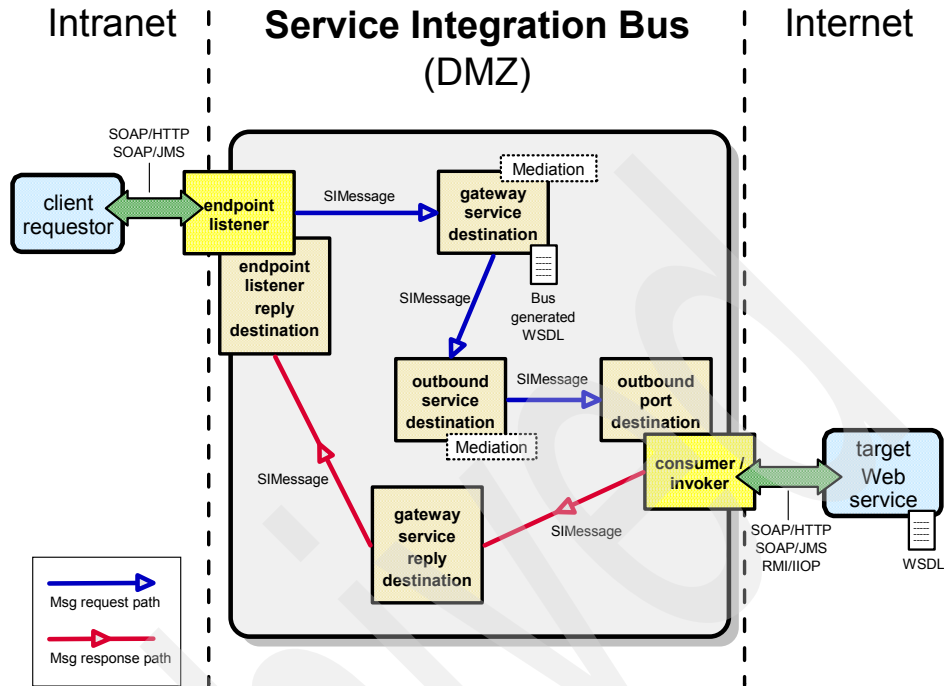


Figure 22-1 Exposing Web services through the bus

Web services in the service integration bus are made up of the following components:

- ▶ **Bus**—The entity on top of which inbound and outbound services and gateway resources can be defined.
- ▶ **Endpoint listener**—Entry points to the bus for Web services clients. Endpoint listeners control whether clients connect over SOAP/HTTP or SOAP/JMS. They are associated with inbound services and gateway resources.
- ▶ **Inbound service**—Destinations within the bus exposed as Web services.
- ▶ **Outbound service**—Destinations within the bus that represent external Web services.
- ▶ **Gateway instance**—Enable a user to create gateway and proxy services.
- ▶ **Gateway service**—Expose external Web services as bus-managed Web services.
- ▶ **Proxy service**—Expose external Web services as bus-managed Web services, but with the added feature of allowing runtime control over the target service endpoint that is called.

- ▶ **Mediation**—A stateless session EJB attached to a service destination that can apply processing to messages that pass through it, for example, logging or message transformation.
- ▶ **JAX-RPC handler**—A J2EE standard for intercepting and manipulating Web services messages.
- ▶ **JAX-RPC handler list**—Used to manage JAX-RPC handlers by determining the order in which they are executed. These lists can be associated with bus services.
- ▶ **UDDI reference**—Configuration information for UDDI registries to which the bus is able to connect.

Motivation for using the bus

In this section, we describe the reasons an enterprise might want to use the Web services features of the service integration bus.

Securely externalizing existing applications

Businesses can use the bus to expose existing applications as Web services, for use by any Web service-enabled tool, regardless of the implementation details. This enables applications or Web services deployed on a server deep inside an enterprise to be made available as Web services on the Internet to customers, suppliers, and business partners. Security options mean that this access can be tightly controlled.

Return on investment

Any number of business partners can reuse an existing process that you make available as a Web service using the bus. This gives great scope for the reuse of existing assets.

Protocol transformation

The bus provides support for exposing an existing service implemented in one protocol (for example, SOAP/JMS), as something entirely different from clients (for example, SOAP/HTTP). This function is invaluable for ensuring smooth interoperability between businesses that may implement varying Web services protocols in their business applications.

Messaging benefits

The fact that the bus is built on top of the Java Messaging Service (JMS) delivered in WebSphere Application Server means that it is able to expose messaging artifacts, such as queues and topics, as Web services. It also provides advanced options for asynchronous communication, prioritized message delivery, and message persistence.

Standards-based integration

The bus provides support for the major Web services standards, giving businesses confidence that they can use it to build flexible and interoperable solutions. Among others, the bus provides support for WS-I Basic Profile 1.1, JAX-RPC (JSR-101) 1.1, UDDI V3, WS-I Security Profile, and WS-Transaction.

Support for advanced administration

Tight integration with the WebSphere administrative model means that businesses can set up complex topologies for the bus, such as clusters, to ensure high availability for their Web services.

Installation

After installing WebSphere Application Server, users have to complete the installation of the four core components described in the following sections before they are able to use Web services with the bus. All commands should be run from a command prompt in the `install_root/bin` directory, where *install_root* is the root directory where you have installed WebSphere Application Server (for example, `/opt/WebSphere/AppServer`).

Note: You require WebSphere Application Server Network Deployment if you want to use any Web services gateway functionality. Administrative commands are available in both the stand-alone and deployment manager profiles for this product, while GUI panels are only found in the deployment manager profile.

Installing the SDO repository

The SDO repository is used by the bus for storing and serving WSDL definitions that describe configured Web services. To install the repository, perform these steps:

- ▶ Run the following command:

```
wsadmin.ext -f installSdoRepository.jacl -createDb
```

Where *ext* is the file extension—`bat` for a Microsoft Windows system and `sh` for a UNIX® system.
- ▶ Look for the message that confirms that the SDO repository installation completed successfully.

The `-createDb` flag instructs the script to install IBM Cloudscape as the underlying database for the SDO repository. Running the script without this flag

will just install the SDO application and leave the user free to configure another database type.

Important: If you are using the SDO repository in a WebSphere Application Server Network Deployment environment, you should always run the install script *without* the `-createDb` flag, and configure your own database (because Cloudscape is only supported in a stand-alone environment). See the *WebSphere Application Server Information Center* for more information about this process.

If changes have to be made to the SDO installation, users should first run the script to uninstall the SDO repository before reinstalling. To run this script, use either of the following commands:

- ▶ To remove the SDO application only, run:
`wsadmin.ext -f uninstallSdoRepository.jacl`
- ▶ If the `-createDb` flag was used during installation, run the following to remove both the SDO application and the database configuration:

```
wsadmin.ext -f uninstallSdoRepository.jacl -removeDb
```

This removes the application and JDBC configuration information. To remove the actual database itself, you will have to delete the database from the `install_root/profiles/your_profile/databases/SdoRepDb` directory.

To uninstall the SDO repository in a Network Deployment environment, refer to the *WebSphere Application Server Information Center*.

Installing the resource adapter

The resource adapter is required to enable outbound services to be correctly configured within the bus. It should be installed before the core application and endpoint listeners. To install the resource adapter:

- ▶ Run the following command (the `clusterName` is optional and should only be specified in a clustered environment):

```
wsadmin.ext -f install_root/util/sibwsInstall.jacl INSTALL_RA  
-installRoot install_root_using_forward_slashes -nodeName node_name  
[-clusterName cluster_name]
```

For example:

```
./wsadmin.sh -f /opt/WebSphere/AppServer/util/sibwsInstall.jacl INSTALL_RA  
-installRoot /opt/WebSphere/AppServer -nodeName myNode01
```

- ▶ Look for the message that confirms that the SIB_RA installation completed successfully.

Installing the core application

The core application is required for making bus-generated WSDL available. To install the core application:

- Run the following command:

```
wsadmin.ext -f install_root/util/sibwsInstall.jacl INSTALL
            -installRoot install_root_using_forward_slashes -serverName server_name
            -nodeName node_name
```

You can specify installation on either a server and node or a cluster. For example:

```
./wsadmin.sh -f /opt/WebSphere/AppServer/util/sibwsInstall.jacl INSTALL
-installRoot /opt/WebSphere/AppServer -serverName server1 -nodeName
myNode01
```

If you are installing on a cluster, you should run the following command instead:

```
./wsadmin.sh -f /opt/WebSphere/AppServer/util/sibwsInstall.jacl INSTALL
-installRoot /opt/WebSphere/AppServer -clusterName myCluster01
```

- Look for the message that confirms that the installation completed successfully and that the application has been started.

Installing the endpoint listener applications

The endpoint listener applications are required when users want to expose services using the bus to Web services clients wanting to connect using SOAP/HTTP or SOAP/JMS.

Important: If you want to make services available over SOAP/JMS, you have to install the SOAP/JMS endpoint listener applications. Before doing this, however, you *must* configure the necessary JMS resources as specified in the *WebSphere Application Server Information Center*; otherwise, the applications will not start.

- To install the **HTTP** endpoint listeners, run the following command:

```
wsadmin.ext -f install_root/util/sibwsInstall.jacl INSTALL_HTTP
            -installRoot install_root_using_forward_slashes -serverName server_name
            -nodeName node_name
```

For example:

```
./wsadmin.sh -f /opt/WebSphere/AppServer/util/sibwsInstall.jacl
INSTALL_HTTP -installRoot /opt/WebSphere/AppServer -serverName server1
-nodename myNode01
```

Look for the message that confirms that the installation completed successfully and that both applications have been started.

- To install the **JMS** endpoint listeners, run the following command:

```
wsadmin.ext -f install_root/util/sibwsInstall.jacl INSTALL_JMS  
-installRoot install_root_using_forward_slashes -serverName server_name  
-nodeName node_name
```

For example:

```
./wsadmin.sh -f /opt/WebSphere/AppServer/util/sibwsInstall.jacl INSTALL_JMS  
-installRoot /opt/WebSphere/AppServer -serverName server1 -nodeName  
myNode01
```

Look for the message that confirms that the installation completed successfully and that both applications have been started.

Using the bus

This section describes how to perform key tasks to manipulate Web services within the bus.

Performing administrative tasks

Web services in the service integration bus are fully integrated into the WebSphere Application Server administration model. As such, resources can be administered in two primary ways: GUI and command line.

Using the GUI

Using the WebSphere administrative console, most resources can be found under the Service integration tab on the left side of the console (see Figure 22-2). Some resources are located elsewhere however. Where this is so, the following sections describe where they can be found. To access the console following a default installation, make sure that your server is started and point a browser to:

```
http://hostname:9060/admin/
```



Figure 22-2 Options for configuring Web services in the bus

Using the command line

The WebSphere Application Server scripting model (accessed through the `wsadmin` program) provides support for a number of `AdminTask` and `AdminConfig` commands that can be used to administer Web services in the bus. This provides excellent scope for scripting commonly used Web services tasks in the bus.

Table 22-1 provides an overview of commands you can use.

Table 22-1 *Wsadmin commands for administering Web services in the bus*

Task description	Command
Service integration buses	
Create a bus	<code>createSIBus</code>
Add a member to a bus	<code>addSIBusMember</code>
Remove a member from a bus	<code>removeSIBusMember</code>
Delete a named bus, and everything on it	<code>deleteSIBus</code>
Endpoint listeners	
Create an endpoint listener	<code>createSIBWSEndpointListener</code>
Connect an endpoint listener to a bus	<code>connectSIBWSEndpointListener</code>
Disconnect an endpoint listener from a bus	<code>disconnectSIBWSEndpointListener</code>
Delete an endpoint listener	<code>deleteSIBWSEndpointListener</code>
Inbound services	
Create an inbound service	<code>createSIBWSInboundService</code>
Add an inbound port to an inbound service	<code>addSIBWSInboundPort</code>

Task description	Command
Refresh the WSDL definition for an inbound service	refreshSIBWSInboundServiceWSDL
Publish an inbound service to a UDDI registry	publishSIBWSInboundService
Unpublish an inbound service from a UDDI registry	unpublishSIBWSInboundService
Remove an inbound port	removeSIBWSInboundPort
Delete an inbound service	deleteSIBWSInboundService
Outbound services	
Create an outbound service	createSIBWSOutboundService
Add an outbound port to an outbound service	addSIBWSOutboundPort
Refresh the WSDL definition for an outbound service	refreshSIBWSOutboundServiceWSDL
Set the default outbound port for an outbound service	setDefaultSIBWSOutboundPort
Delete an outbound service	deleteSIBWSOutboundService
Remove an outbound port	removeSIBWSOutboundPort
Web services gateway instances	
Create, modify, delete a gateway instance	WSGWInstance*
Web services gateway services	
Create a gateway service	createWSGWGatewayService
Add an additional target service to a gateway service	addWSGWTargetService
Remove an existing target service from a gateway service	removeWSGWTargetService
Delete a gateway service	deleteWSGWGatewayService
Web services gateway proxy services	
Create a proxy service	createWSGWProxyService
Delete a proxy service	deleteWSGWProxyService

Task description	Command
Mediations	
Create a mediation	createSIBMediation
Mediate a destination	mediateSIBDestination
Unmediate a destination	unmediateSIBDestination
Delete a mediation	deleteSIBMediation
JAX-RPC handlers	
Create, modify, delete JAX-RPC handlers	JAXRPCHandler*
Create, modify, delete JAX-RPC handler lists	JAXRPCHandlerList*
UDDI references	
Create, modify, delete UDDI references	UDDIReference*
* Denotes an AdminConfig command	

Buses

The bus object is fundamental to most other configuration tasks; it is not possible to begin creating or configuring any services or gateway resources until a bus object has been created. The bus is a group of one or more interconnected servers or clusters that have been added as members of the bus. Applications connect to a bus at one of the messaging engines associated with its bus members and can then access resources defined on the bus.

Creating a bus

To configure a new bus using the GUI:

- ▶ From the WebSphere administrative console, select *Service integration* → *Buses*, and then click *New*.
- ▶ Enter a unique name for the bus. This is the only required field; other fields can be filled in to configure security and inter-engine communications. Additional properties will not be available for use until the bus has been saved.
- ▶ Click *OK* and save the changes (by clicking the *save* link at the top of the page). The bus you have just created should now be available for selection.
- ▶ Select the bus you have just created and note all the additional properties that are now available (Figure 22-3).

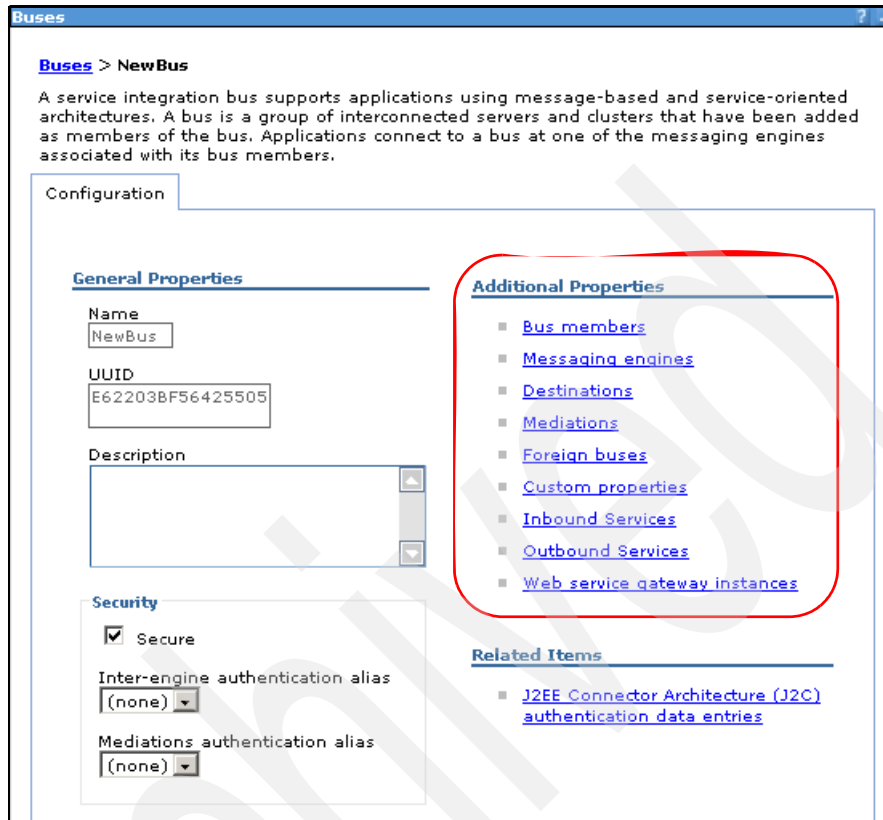


Figure 22-3 GUI configuration for a new bus

Before this new bus is usable, it must have a server or cluster associated with it (remember a bus is a collection of interconnected servers or clusters). To do this:

- ▶ From the Additional Properties list, select *Bus members*, and then click *Add*.
- ▶ From the wizard, select the server or cluster you want to add to the bus, and then click *Next*.
- ▶ Review the summary of your selections, and if satisfied, click *Finish* and then save your changes.
- ▶ Finally, restart your server. This is required to start the messaging engine, which is automatically created for you when you add a server or cluster to a bus.

Endpoint listeners

Endpoint listeners are required if you want to expose destinations on the bus to clients connecting over SOAP/HTTP and SOAP/JMS. They are the entry points to the bus for these protocols, carrying requests between Web services clients and buses, and are used by both inbound services and gateway services. An endpoint listener acts as the ultimate receiver of an incoming SOAP message to the bus and passes on the data and context of that message.

For each protocol, two endpoint listeners are supplied with WebSphere Application Server to give users the flexibility (if they want) to configure separate endpoint listeners for requests from internal users and those from external listeners. By making different services available on each listener, an added level of security can be provided to a bus configuration.

Creating an endpoint listener

To define a new endpoint listener using the GUI:

- ▶ From the WebSphere administrative console, select *Servers* → *Application servers* and click the name of the server for which you want to define an endpoint listener. (For clusters, you would select *Servers* → *Clusters*).
- ▶ From the Additional Properties list, select *Endpoint Listeners*, and then click *New*.
- ▶ Fill in the General Properties for your endpoint listener using values from Table 22-2, depending on which type of listener you are configuring. The URL root is the context root of the endpoint enterprise application, and the WSDL URL root defines the Web address root for the bus-generated WSDL of inbound services defined on the endpoint listener.

Table 22-2 Values to use when creating endpoint listeners

Listener type	Name	URL root	WSDL URL root
SOAP/HTTP 1	SOAPHTTPChannel1	http://hostname:port_number/wsgwsoaphttp1	http://hostname:port_number/sibws/wsdl
SOAP/HTTP 2	SOAPHTTPChannel2	http://hostname:port_number/wsgwsoaphttp2	http://hostname:port_number/sibws/wsdl
SOAP/JMS 1	SOAPJMShannel1	jms:/queue?destination=jms/SOAPJMSQueue1&connectionFactory=jms/SOAPJMSFactory1	http://hostname:port_number/sibws/wsdl
SOAP/JMS 2	SOAPJMShannel2	jms:/queue?destination=jms/SOAPJMSQueue2&connectionFactory=jms/SOAPJMSFactory2	http://hostname:port_number/sibws/wsdl

- After you have filled in all the values for General Properties, click *Apply*. This saves the General Properties and makes the Additional Properties available (Figure 22-4).

Application servers

Application servers > server1 > Endpoint Listeners > SOAPJMSChannel1

An endpoint listener receives requests from service requester applications within a specific application server or cluster.

Configuration

General Properties

* Name
SOAPJMSChannel1

Description

* URL root
jms:/queue?destination=jms

* WSDL serving HTTP URL root
http://localhost:9080/sibws/

Apply OK Reset Cancel

Additional Properties

■ [Connection Properties](#)

Figure 22-4 Properties for an endpoint listener

- From the Additional Properties list, click *Connection properties* and then *New*.
- A drop-down list of available buses appears to which you can connect the endpoint listener. Select the bus you want to use, click *OK*, and then click *save* to save your changes.

This completes the configuration of your endpoint listener by connecting it to a bus and making it available for use with inbound services.

Note: The URL root values supplied for the SOAP/JMS endpoint listeners in Table 22-2 assume that you have used the defaults supplied in the endpoint listener applications. If you have changed these, you also have to modify the values you supply when creating the endpoint listener.

Inbound services

Inbound services can be used to take a service destination defined within a bus and expose it as a Web service, accessible over different bindings (for example, SOAP/HTTP or SOAP/JMS). This is useful if we want to take an existing destination within a bus (for example, an outbound service destination that points to another Web service) and make it available at a new bus-generated endpoint.

Each available binding type for an inbound service is represented by an associated inbound port, and each of these ports is associated with a binding-specific endpoint listener. Figure 22-5 shows an overview of a typical inbound service configuration.

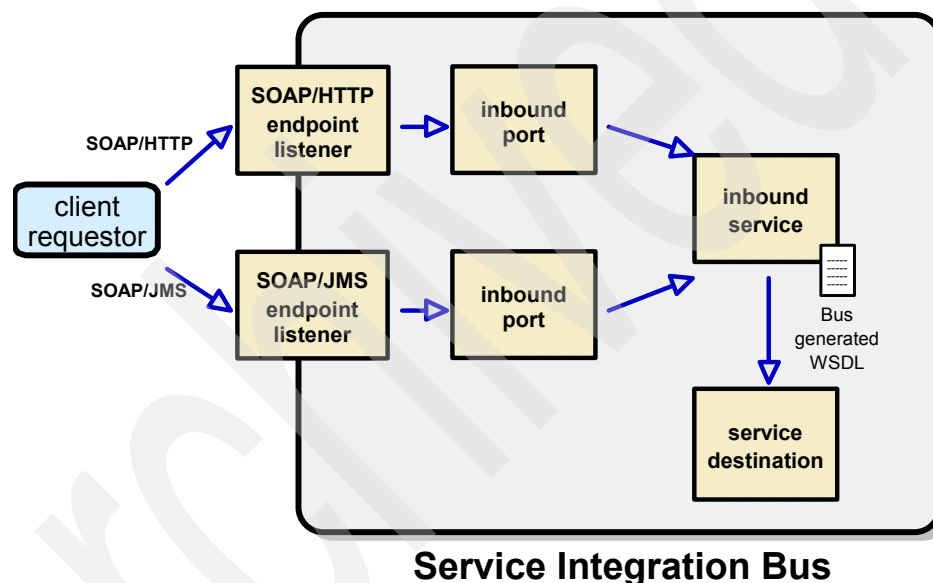


Figure 22-5 A typical inbound service configuration

Creating an inbound service

To create a new inbound service using the GUI:

- ▶ From the WebSphere administrative console, select *Service integration* → *Buses* and click the name of the bus for which you want to define an inbound service.
- ▶ From the Additional Properties list, click *Inbound Services*, and then *New*.
- ▶ In the wizard that opens, select the *Service destination name* of the destination you want to expose as an inbound service from the drop-down list.

- ▶ Next, select whether the template WSDL for this destination is to be supplied from a UDDI registry or through a URL.

Note: The template WSDL should be supplied by you, and should specify the portType schema to be used for the service supplied at the destination, and also provide details of any bindings it might have. If the destination represents another Web service, you would typically use that target service WSDL here.

- ▶ Depending on what you selected as the *Template WSDL location type*, you should now populate the *Template WSDL location* field with either the URL or the UDDI service key of the template WSDL.
- ▶ If you provide the WSDL from a UDDI registry, select the registry from the *Template WSDL UDDI registry* list (this list is populated from UDDI References that you have defined).
- ▶ Click *Next* and select the service from the template WSDL that you want to expose (if only one exists, this is populated by default).
- ▶ Click *Next*. Specify a name for your inbound service and then select which endpoint listeners you want to expose the service over. For each one you select, an inbound port will be created.
- ▶ Click *Next* and specify details if you want your inbound service to be published to a UDDI registry.
- ▶ Click *Finish*, and then click *save* to save your new inbound service.

This completes the configuration of your inbound service and makes it available for use by Web services clients over the protocols you have configured (determined by which endpoint listeners you have associated with it).

Accessing an inbound service

To access inbound services with Web services clients, you can reference the bus-generated WSDL file for the service at:

`http://hostname:port_number/sibws/wsdl/bus_name/inbound_service_name`

This WSDL enables you to determine the bus-generated endpoint for the service. Alternatively, you might choose to publish this WSDL to a ZIP file or a UDDI registry. See “Publishing WSDL for services” on page 578 for more information about this.

It is also possible to access an inbound service “directly.” See “Accessing bus services directly using a JAX-RPC client” on page 607 for more information about this.

Reloading inbound service template WSDL

When creating an inbound service, the template WSDL that you specify is loaded into a local repository. If this WSDL changes after defining your inbound service, you will have to refresh it in the local repository as well. This can be achieved by completing the following steps in the GUI:

- ▶ From the WebSphere administrative console, select *Service integration* → *Buses* and click the name of the bus containing the inbound service you want to access.
- ▶ From the Additional Properties list, click *Inbound Services* and then click the name of the service for which you want to update the WSDL.
- ▶ Click *Reload template WSDL* followed by *save* to save your changes.

Outbound services

Outbound services can be used to make a WSDL-described Web service external to the bus available as a destination on the bus, accessible to any requestors that also have access to that bus. This is useful if we want to enable non-Web services clients to access Web services. They can access the resulting outbound service destination and use this to exchange messages with the target service.

The outbound service connects to a target Web service through one or more outbound ports over whichever transport is defined in the target service WSDL (for example, SOAP/HTTP, SOAP/JMS, or RMI/IIOP). Each binding type is represented by a different outbound port (Figure 22-6).

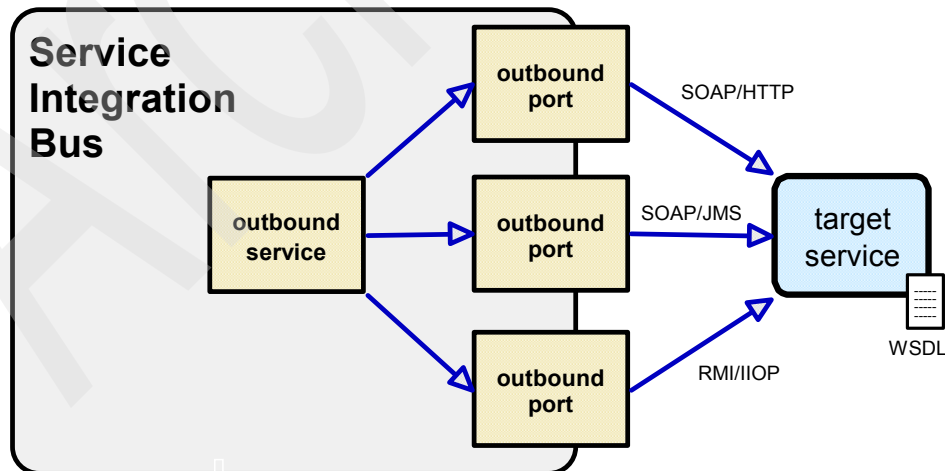


Figure 22-6 A typical outbound service configuration

Creating an outbound service

To create a new outbound service using the GUI:

- ▶ From the WebSphere administrative console, select *Service integration* → *Buses* and click the name of the bus for which you want to define an outbound service.
- ▶ From the Additional Properties list, click *Outbound Services* and then *New*.
- ▶ In the wizard that opens, select whether the WSDL describing the target Web service is to be supplied from a UDDI registry or through a URL.
- ▶ Depending on what you selected as the *WSDL location type*, you should now populate the *WSDL location* field with either the URL or the UDDI service key of the WSDL.
- ▶ If you provide the WSDL from a UDDI registry, you should select the registry from the WSDL UDDI registry list (this list is populated from the UDDI references that you have defined).
- ▶ Click *Next* and select the service from the WSDL that you want to make available to the bus (if only one exists, this is populated by default).
- ▶ Click *Next* and select the ports from the WSDL that you want to make available to the bus. You can select more than one; each will eventually be translated into an outbound port.
- ▶ Click *Next* and choose whether you want to rename the supplied values for the Outbound service name and destination names. If you have more than one port, you can set the *Default port* to be accessed when a requestor connects to the service. We cover the Port selection mediation in “Mediations” on page 590.
- ▶ Click *Next* and specify to which bus you want to assign the port destination (usually the same as you are creating the outbound service on).
- ▶ Click *Finish* and then *save* to save your new outbound service.

This completes the configuration of your outbound service and makes it available for use by the service requestors with access to the bus on which it is defined.

See “Accessing bus services directly using a JAX-RPC client” on page 607 for more details about accessing outbound service destinations.

Reloading outbound service WSDL

When creating an outbound service, the target service WSDL that you specify is loaded into a local repository. If this WSDL changes after defining your outbound service, you will have to refresh it in the local repository as well.

This can be achieved by completing the following steps in the GUI:

- ▶ From the WebSphere administrative console, select *Service integration* → *Buses* and click the name of the bus containing the outbound service you want to access.
- ▶ From the Additional Properties list, click *Outbound Services* and then the name of the service for which you want to update the WSDL.
- ▶ Click *Reload WSDL* followed by *save* to save your changes.

UDDI references

A UDDI reference is a pointer within the service integration bus to a UDDI registry, either a private or a public one (for more information about UDDI registries, see Chapter 7, “Introduction to UDDI” on page 123). The UDDI reference describes all the parameters necessary to connect to a registry and enable the bus to interact with it. This is useful if you are using UDDI to manage your Web services, because it enables you to:

- ▶ Publish inbound services and gateway services to the UDDI registry.
- ▶ Configure outbound services and gateway services to access target service WSDL through the UDDI registry.

Creating a UDDI reference

To create a new UDDI reference using the GUI, first ensure that you have created an authentication alias for the authorized name that you want to associate with the new UDDI reference.

To create an authentication alias:

- ▶ From the WebSphere administrative console, select *Service integration* → *Buses* and click the name of the bus for which you want to define a UDDI reference.
- ▶ Under the Related Items list, click *J2EE Connector Architecture (J2C) authentication data entries* and then *New*.
- ▶ Under General Properties, in the Alias field, select a unique identifier for the authentication alias.
- ▶ Next, in the User ID field, enter the user ID for your UDDI registry authorized name. In the Password field, enter the password for your UDDI registry authorized name.
- ▶ Finally, click *OK* and the *save* link to save your changes.

Now that you have a valid authentication alias, you can create the UDDI reference itself:

- ▶ From the WebSphere administrative console, select *Service integration* → *Web services* → *UDDI References* and then click *New*.
- ▶ Under General Properties, enter a unique Name for the reference followed by the *Inquiry URL*. This is the Web address that provides access to the UDDI registry for the SOAP inquiry API.
- ▶ Optionally, under Publish URL, enter the Web address that provides access to the UDDI registry for the SOAP publish API.
- ▶ Under Authentication Alias, enter the name of the authentication alias you have just created.
- ▶ Finally, click *OK* and then *save* to save your changes.

This completes the configuration of your UDDI reference and makes it available for use when configuring inbound, outbound, and gateway services.

Tip: Each UDDI reference has only one authorized name associated with it. Therefore, if you have to access two Web services in the same registry, and the services are owned by different authorized names, you will have to create two UDDI references.

Publishing WSDL for services

When creating inbound services and gateway services, the service integration bus generates WSDL files describing the newly created service. This WSDL can be accessed through a URL, or alternatively, can be published to a ZIP file or UDDI registry for use by clients of the service. These publishing options are useful for broadening the availability of the service.

This section describes how to publish the WSDL. All of the following examples use an inbound service although they equally apply to using gateway services.

Publishing WSDL to a ZIP file

Exporting WSDL to a ZIP file is a convenient way of sharing the details of the bus services when the URL to the WSDL is not available.

To publish bus-generated WSDL to a ZIP file using the GUI:

- ▶ From the WebSphere administrative console, select *Service integration* → *Buses* and click the name of the bus containing the inbound service you want to access.

- ▶ From the Additional Properties list, click *Inbound Services* and then the name of the service for which you want to publish the WSDL.
- ▶ From the Additional Properties list, click *Publish WSDL files to ZIP file*.
- ▶ The ZIP file will automatically be in the format `inbound_service_name.zip`. Click this file to save it to your local file system.
- ▶ When you examine the ZIP file, you will find all of the available bus-generated WSDL for the service:
 - `bus_name.inbound_service_namePortTypes.wsdl` contains the port type definition for the inbound service.
 - `bus_name.inbound_service_nameService.wsdl` contains the service and port elements for the inbound service.
 - `bus_name.inbound_service_nameBindings.wsdl` contains the binding elements that correspond to the ports for the inbound service.
- ▶ In addition, an extra WSDL file is included:
`bus_name.inbound_service_nameNonBound.wsdl`.

This file contains a view of the inbound service with no ports defined and can be useful for showing an abstract definition of the service.

Publishing WSDL to a UDDI registry

Publishing bus-generated WSDL to a UDDI registry gives you greater flexibility in making your Web services available to clients and making it dynamically discoverable. For more information about UDDI technologies, refer to “Introduction to UDDI” on page 123.

To publish bus-generated WSDL to a UDDI registry using the GUI:

- ▶ First ensure that you have created a reference to the UDDI registry to which you want to publish the service (as described in “Creating a UDDI reference” on page 577).
- ▶ Then, from the WebSphere administrative console, select *Service integration* → *Buses* and click the name of the bus containing the inbound service you want to access.
- ▶ From the Additional Properties list, click *Inbound Services* and then the name of the service for which you want to publish the WSDL.
- ▶ From the Additional Properties list, click *UDDI Publication*.
- ▶ Click *New* to set up the publication details of how the WSDL will be published.
- ▶ Specify a unique Name for the publication property.
- ▶ From the UDDI reference list, select the UDDI registry to which you want to publish.

- ▶ Specify the *business key* of the business to which you want to publish the service (you should query the UDDI registry to find this).
- ▶ Click *OK* and then *save* to save the publication reference.
- ▶ Finally, select the check box next to the name of the publication reference, followed by the *publish to UDDI* button. A message appears confirming that the service was successfully published, and the Published service key field on your publication reference is updated with the UDDI service key information.

Your service is now published to the UDDI registry and available for use.

Web services gateway

Web services gateway functionality enables users to take an existing Web service and expose it as a new service that appears to be provided by the gateway. Gateway functionality is supplied in the Network Deployment release of WebSphere Application Server. It builds on top of the functions provided by inbound and outbound services, essentially linking them together. By using the gateway, it is possible for a Web services client to access an external Web service hosted by the gateway (see Figure 22-1 on page 561).

As in WebSphere Application Server Version 5, the gateway can act as a single point of control for incoming Web services requests. It can be used to perform protocol transformation between messages (for example, to expose a SOAP/JMS Web service over SOAP/HTTP) and map multiple target services to one gateway service. It also has the ability to create proxy services and administer JAX-RPC handlers for services it manages.

Some of the benefits of using the gateway are:

- ▶ A gateway service is located at a different endpoint from the target service, making it possible to relocate the target service without disrupting the end user experience.
- ▶ The gateway provides a common starting point for all Web services you provide. Users need not know whether they are provided directly by you, or externally.
- ▶ You can have more than one target service for each gateway service.

Gateway GUI options can be found under *Service integration* → *Buses* → *YourBusName* → *Web service gateway instances* (you have to create a gateway instance in order to create gateway and proxy services—see “Creating a gateway instance” on page 581). This is illustrated in Figure 22-7.

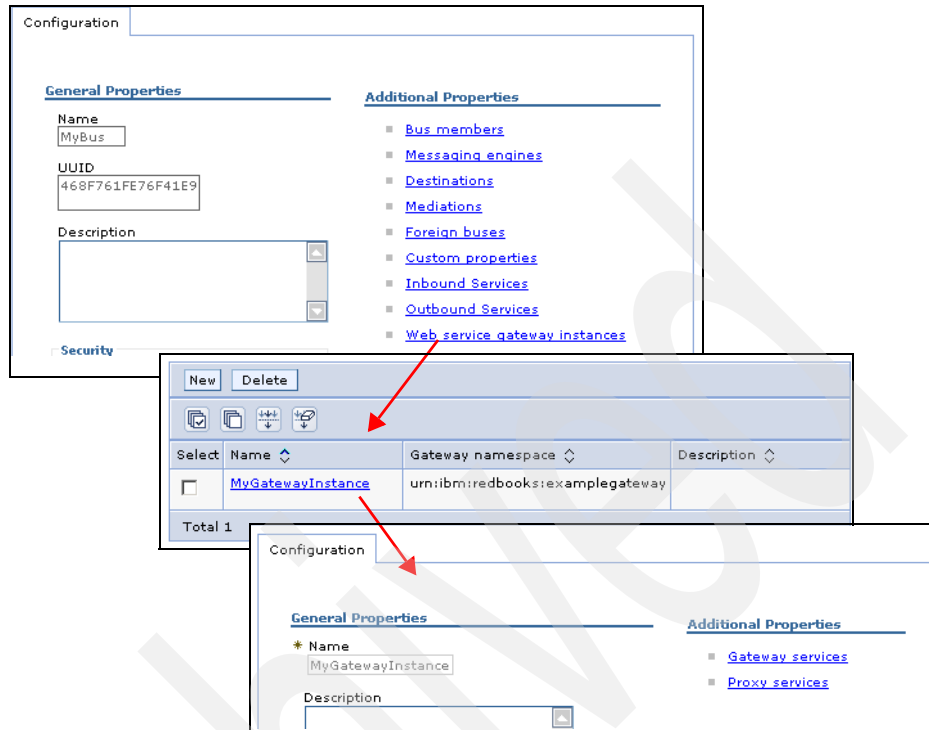


Figure 22-7 Accessing the Web services gateway through the GUI

Gateway instances

A gateway instance is the base on which you create gateway and proxy services; these services cannot be created until an instance exists. Within a service integration bus, you can create multiple gateway instances in order to partition these services into logical groups to allow simpler management.

Creating a gateway instance

To create a gateway instance using the GUI:

- ▶ From the WebSphere administrative console, select *Service integration* → *Buses* and click the name of the bus for which you want to define a gateway instance.
- ▶ From the Additional Properties list, click *Web service gateway instances* and then *New*.
- ▶ In the General Properties that appear, enter a unique Name for the instance.

- ▶ Optionally, type a description, and then enter a Gateway namespace. This is the namespace that will be used in all gateway-generated WSDL. It is good practice to use a namespace that you are happy with from the start, because changing it later will require you to redeploy any associated gateway services.

Note: There is no fixed syntax for a gateway namespace, but a useful guideline is to ensure that it is globally unique and follows guidelines for URNs. See:

<http://www.ietf.org/rfc/rfc2141.txt>

- ▶ Next, enter the location for the Default proxy WSDL URL. A gateway proxy service has no configured target services and, therefore, no WSDL. A generic (default proxy) WSDL is used instead to configure basic parameters for the invocation call. The supplied template WSDL is found at:

`http://hostname:port_number/sibws/proxywsdl/ProxyServiceTemplate.wsdl`

- ▶ Click *OK* and then *save* to save your new gateway instance.

This completes the configuration of your gateway instance, enabling you to make use of it to create gateway and proxy services.

Gateway services

Gateway services can be used to take an existing service destination in the bus or a WSDL-described Web service external to the bus and expose it as a new Web service, accessible over different bindings (for example, SOAP/HTTP or SOAP/JMS). The gateway service is useful for making services available at a new endpoint, enabling you to relocate the underlying target service (if needed) without changing the details of the gateway service—this remains visible to the client at the same endpoint.

Creating a gateway service

To create a gateway service using the GUI:

- ▶ First, ensure that you have already created a gateway instance.
- ▶ From the WebSphere administrative console, select *Service integration* → *Buses* and click the name of the bus in which you want to define the gateway service.
- ▶ From the Additional Properties list, click *Web service gateway instances* and the name of the instance in which you want to define the gateway service.
- ▶ From the Additional Properties list, click *Gateway services* and then *New*.

- ▶ Select either a *WSDL-defined Web service provider* (to expose a WSDL-described Web service external to the bus) or a *Service destination* (to expose an existing service destination).

For a WSDL-defined Web service provider

This creates a configuration similar to that shown in Figure 22-1 on page 561:

- ▶ Select *WSDL-defined Web service provider* and then click *Next*.
- ▶ Enter a unique name for your gateway service and optionally supply *Gateway destination names* (defaults based on the service name are provided if they are left blank).
- ▶ Optionally, associate mediations with the service (see “Message manipulation” on page 588 for more information about mediations), and then click *Next*.
- ▶ For steps 2-6 in the wizard, follow the instructions for creating outbound services (see “Creating an outbound service” on page 576).
- ▶ For step 7, select the endpoint listeners over which you want to expose the Web service, and then click *Next*.
- ▶ Click *Next* and specify details if you want your gateway service to be published to a UDDI registry.
- ▶ Click *Finish* and then *save* to save your new gateway service.

For a service destination

For a service destination, perform these steps:

- ▶ Select *Service destination* and then click *Next*.
- ▶ Enter a unique name for your gateway service and specify the bus on which you want to create the service. Optionally, supply *Gateway destination names* (defaults based on the service name are provided if they are left blank).
- ▶ Select the Target destination name of the destination you want to expose as a gateway service and then click *Next*.
- ▶ For steps 2-5 in the wizard, follow the instructions for creating inbound services (see “Creating an inbound service” on page 573).
- ▶ Click *Finish* and then *save* to save your new gateway service.

This completes the configuration of your gateway service and makes it available for use by Web services clients over the protocols you have configured (determined by which endpoint listeners you have associated with it).

Tip: As with inbound services, the gateway-generated WSDL (and endpoint details) for a new gateway service can be accessed at the following URL:

`http://hostname:port_number/sibws/wsdl/bus_name/gateway_service_name`

For more information about publishing this WSDL to a ZIP file or UDDI registry, see “Publishing WSDL for services” on page 578.

Reloading the gateway service WSDL

When creating a gateway service, the target service WSDL that you specify is loaded into a local repository. If this WSDL changes after defining your outbound service, you will have to refresh it in the local repository as well. This can be achieved by completing the following steps in the GUI:

- ▶ From the WebSphere administrative console, select *Service integration* → *Buses* and click the name of the bus containing the gateway service you want to access.
- ▶ From the Additional Properties list, click *Web service gateway instances* and the name of the instance the gateway service is in.
- ▶ Click the gateway service you want to change and then click *Inbound web service enablement* followed by the *Reload template WSDL* button.
- ▶ In the trail of links at the top of the page, click your gateway service name.
- ▶ Click *Target services* and then your service name. Click *Outbound web service enablement* and then *Reload WSDL*.
- ▶ Finally, click the *save* link to save your changes.

Proxy services

As in WebSphere Application Server Version 5.x, the gateway can be configured to act as a proxy for your target Web service. In this mode, a gateway proxy service is created to represent the target Web service.

The proxy service is exposed to clients in the usual way (over whichever endpoint listeners you choose), but it does not parse the incoming SOAP message. Instead, it has an associated JAX-RPC handler or mediation that determines to which endpoint address to direct the incoming request (Figure 22-8).

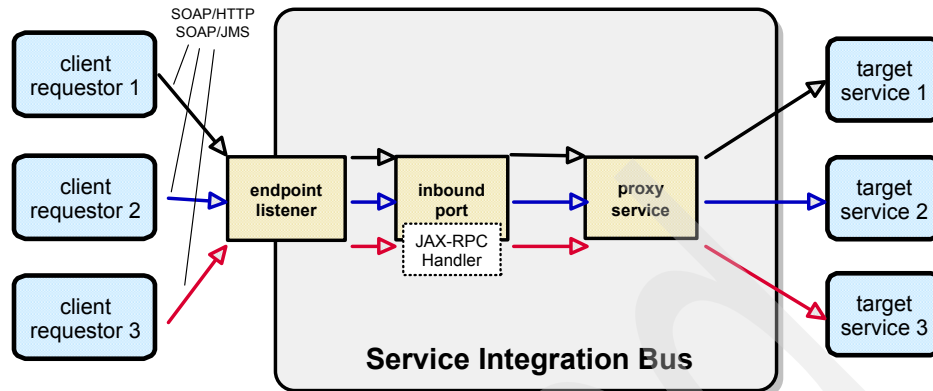


Figure 22-8 A typical gateway proxy service configuration

This mode can be useful if you have a large number of target Web services sharing the same JAX-RPC handler or mediation lists, because it saves you from having to create individual gateway services for each one.

Instead, one proxy service can be created for all of them with the same handler or mediation list (useful, for example, for logging all incoming requests). The service requestor can even pass a token on the request URL for the proxy service to enable the associated handler to identify the ultimate target service URL.

Creating a proxy service

To create a proxy service, you should first ensure that you have written an appropriate JAX-RPC handler to handle the incoming message and associated this with a JAX-RPC handler list in the bus (see “JAX-RPC handlers” on page 588 for instructions about how to do this).

The handler should include some code to set the `transport.url` in the SOAP message and can optionally include code to read the actual target service URL if this is specified as a property on the endpoint URL. Figure 22-9 shows a very simple example.

```

import com.ibm.wsspi.webservices.rpc.handler.GenericHandler;
import javax.xml.rpc.handler.MessageContext;

public class ProxyServiceHandler extends GenericHandler
{
    public ProxyServiceHandler() {
        System.out.println("ProxyServiceHandler - Inside constructor");
    }

    public boolean handleRequest(MessageContext msgContext) {
        System.out.println("ProxyServiceHandler - Inside handleRequest");
        String targetUrl = "http://MyActualTargetService_Endpoint";
        msgContext.setProperty("transport.url", targetUrl);
        return true;
    }
}

```

Figure 22-9 Example JAX-RPC handler for use with a proxy service

To then create your proxy service using the GUI:

- ▶ From the WebSphere administrative console, select *Service integration* → *Buses* and click the name of the bus in which you want to define a proxy service.
- ▶ From the Additional Properties list, click *Web service gateway instances* and the name of the instance in which you want to define the proxy service.
- ▶ From the Additional Properties list, click *Proxy services* and then *New*.
- ▶ Enter a unique name for your proxy service, select a *Proxy port point*, and optionally specify request and response *destination names* (if none are specified defaults are created based on the service name).
- ▶ Next, optionally specify request and response mediations to associate with the service.
- ▶ If you want to specify a different piece of proxy WSDL than that defined in your gateway instance, you can do this by putting the URL into the *Proxy WSDL override URL* field.
- ▶ Click *Next* and select the endpoint listeners over which you want to make your proxy service available.
- ▶ Click *Finish* and then *save* to save your new proxy service.
- ▶ To associate your proxy service JAX-RPC handler, click the name of your service, and then from the Additional Properties list, click *Inbound web service enablement*.
- ▶ Then from the next Additional Properties list, click *Inbound Ports* and then the name of the inbound port for your proxy service.

Important: Proxy service handlers should always be associated with the proxy service inbound port, because the endpoint URL needs to be set before the gateway service is called.

- ▶ From the JAX-RPC handler list, select the name of the list containing your proxy handler.
- ▶ Click *OK* and then *save* to save your new configuration.

This completes the configuration of your proxy service and makes it available for use by Web services clients connecting over the protocols you have configured (determined by which endpoint listeners you have associated with it).

Accessing a proxy service

Because a proxy service merely passes the incoming SOAP request onto the final target service, it should be possible to develop clients against the target service WSDL in the normal way and just override the endpoint URL to use that of the proxy service (for instance, JAX-RPC dynamic invocation interface clients). The endpoint URL you use should also include details of whether the request is one-way or request-response by specifying the `operationMode` parameter.

SOAP/HTTP

The format of a proxy service endpoint URL for SOAP/HTTP is as follows:

```
http://hostname:port_number/endpointlistener_url_root/soaphttpengine/bus_name/proxy_service_name/proxy_service_inboundport_name?operationMode=type_of_operation
```

For example:

```
http://localhost:9080/wsgwsoaphttp1/soaphttpengine/MyBus/MyProxyService/SOAPHHTTPChannel1InboundPort?operationMode=requestresponse
```

SOAP/JMS

The `operationMode` parameter is not required, because the operation style is implied by the presence or absence of a `ReplyToQueue` in the JMS message. The format of a proxy service endpoint URL over SOAP/JMS is, therefore, as follows:

```
jms:/queue?destination=jms/endpoint_listener_queue_name&connectionFactory=endpoint_listener_qcf_name|targetService=bus_name/proxy_service_name/proxy_service_inboundport_name
```

For example:

```
jms:/queue?destination=jms/SOAPJMSQueue1&connectionFactory=jms/SOAPJMSFactory1|targetService=MyBus/MyProxyService/SOAPJMSChannel1InboundPort
```

Message manipulation

Messages can be manipulated through JAX-RPC handlers or through mediations.

JAX-RPC handlers

The service integration bus supports the use of JAX-RPC handlers at ports within the bus (typically inbound and outbound services ports) to perform message manipulation tasks (for example, logging incoming or outgoing requests).

For more information about JAX-RPC handlers, refer to “Using Web service handlers” on page 309. Handlers within the bus are chained together in *JAX-RPC handler lists*. These lists contain one or more handlers and can increase the power and flexibility of your Web service by providing a mechanism for chaining multiple handlers together.

Tip: You have to create a JAX-RPC handler list even if you have only one handler, because only lists can be associated with ports in the bus.

Making JAX-RPC handlers available for use

For any handler that you want to use with the bus, you should make the class file you have created available for use by the runtime. To do this in WebSphere Application Server, you should copy the class and package folders for the handler into either the `install_root/classes` or `install_root/lib/app` directory. If you have a number of handlers, you can also consider bundling them in a JAR file and putting this in one of the directories instead. You have to restart the application server to make them available.

Creating JAX-RPC handlers

To create a new JAX-RPC handler using the GUI:

- ▶ From the WebSphere administrative console, select *Service integration* → *Web services* → *JAX-RPC Handlers* and then click *New*.
- ▶ In the General Properties list, enter a unique Name for the handler, an optional Description, and also the underlying Class name.
- ▶ Click *Apply*. This makes the Additional Properties for your handler available for use. These are optional and can be used to configure extra parameters on your handler:
 - SOAP Roles—A handler can have multiple roles. They define the SOAP roles in which the handler acts.

- JAX-RPC header—A handler can have multiple headers. They define the SOAP headers that should be processed by the handler, based on the Namespace URI and Local part.
- Custom properties—These are name/value pairs, where the name is a property key and the value is a string value that can be used to set internal system configuration properties.
- ▶ When you have completed the extra configuration, click the *save* link to save your new handler.

This completes the configuration of your handler and makes it available for association with a JAX-RPC handler list.

Creating JAX-RPC handler lists

To create a new JAX-RPC handler list using the GUI:

- ▶ From the WebSphere administrative console, select *Service integration* → *Web services* → *JAX-RPC Handler Lists* and then click *New*.
- ▶ In the General Properties list, enter a unique Name for the handler list and an optional Description.
- ▶ Using the *Add*, *Remove*, *Up*, and *Down* buttons determine which handlers you want to add to the list and the order in which they should be executed.
- ▶ Click *OK* and then *save* to save the handler list.

This completes the configuration of your handler list and makes it available for association with inbound, outbound, and gateway services.

Associating JAX-RPC handler lists with services

To add a JAX-RPC list to a service using the GUI:

- ▶ From the WebSphere administrative console, select *Service integration* → *Buses* and click the name of the bus that contains the service with which you want to associate the list.
- ▶ Click through to the service and port you want to associate the list with (for example, *Inbound Services* → *your_service_name* → *Inbound Ports* → *your_port_name*).
- ▶ Under General Properties, there is a JAX-RPC handler list. This will contain all of your defined lists. Select one to associate with the port.
- ▶ Click *OK* and then *save* to save the handler list association.

This completes the association of your handler list with a service. Handlers in that list will now be executed when the service is called.

Mediations

Similar to JAX-RPC handlers, mediations can be used to access message content within the service integration bus and perform various actions on that message. They should, however, be seen as distinct from handlers and be considered for use when you require more powerful message processing options than those provided by handlers. Unlike handlers, mediations can:

- ▶ Transform a message from one format to another
- ▶ Access and augment message content to add or remove data
- ▶ Route messages to one or more target destinations

A mediation can be associated with any destination in the bus to create a *mediated destination*. With Web services configurations, however, the most frequent places that mediations are applied are to either a gateway service (where *request* and *response* mediations can be configured) or an outbound service (where a *port selection mediation* can be defined).

The administration of mediations are controlled by mediation handler lists, which can contain one or more mediations and determine the order in which mediations are executed.

Writing a mediation

A Web services mediation is a stateless session EJB that implements the `com.ibm.websphere.sib.mediation.handler.MediationHandler` interface and contains some specific deployment descriptor configuration details.

To write a basic mediation for Web services using Application Developer requires the following five steps.

Step 1: Create an enterprise application containing an EJB module

- ▶ Switch to the J2EE perspective. From the menu, select *Window → Open Perspective → Other → J2EE*.
- ▶ Select *File → New → Project*.
- ▶ Expand the J2EE folder, and select *Enterprise Application Project*. Click *Next*.
- ▶ Enter a name for the project and target the project to WebSphere Application Server V6.0 (using the *Show Advanced* button). Click *Next*.
- ▶ In the *EAR Module Projects* window, click *New Module* and clear the *Application Client project*, the *Web project*, and the *Connector project* to leave just the *EJB project*. Click *Finish* (Figure 22-10).

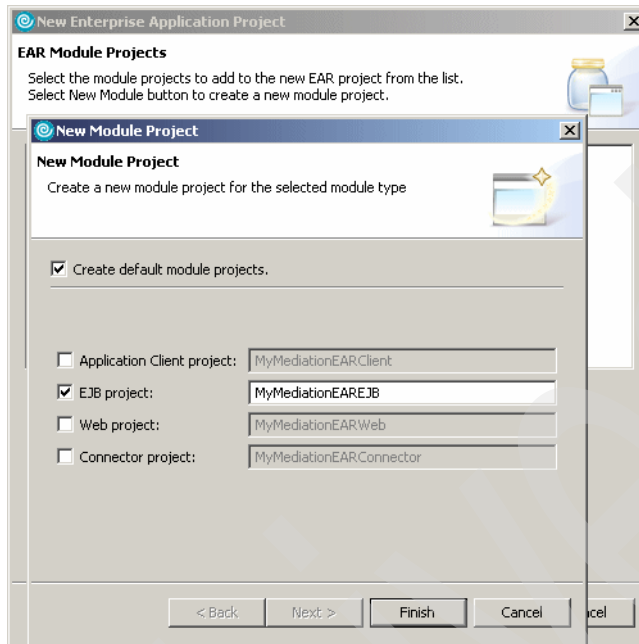


Figure 22-10 Creating an enterprise application containing an EJB module

- Click *Finish* to create the enterprise application.

Step 2: Create a mediation handler class

- Select *File* → *New* → *Other*. Expand the *Java* folder, select *Class*, and click *Next*.
- Specify the Source Folder (usually `EJB_module_name/ejbModule`) and the Name of your class.
- Add an interface by clicking *Add* next to the *Interfaces* box. In the *Choose Interfaces* box that appears, find the mediation handler interface:
`com.ibm.websphere.sib.mediation.handler.MediationHandler`
- Click *OK* and then make sure that the *Inherited abstract methods* check box is selected (Figure 22-11).

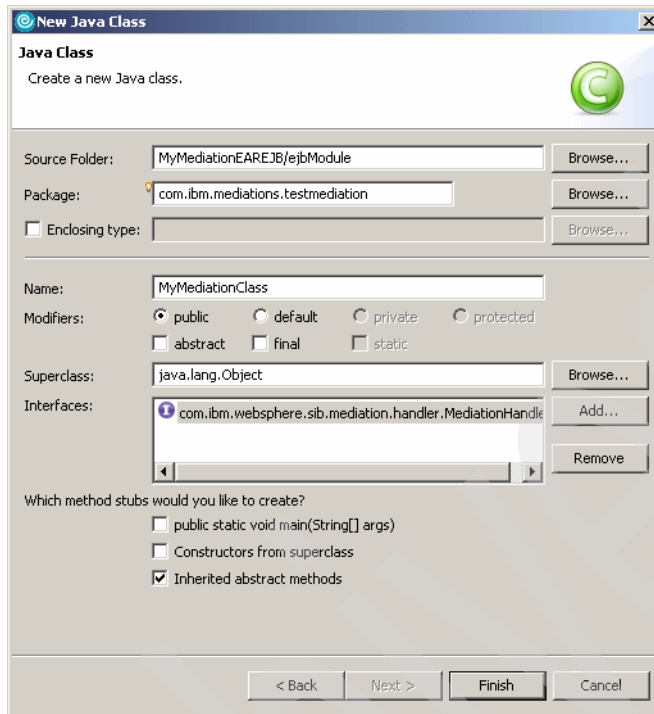


Figure 22-11 Creating a new mediation handler class

- ▶ Click *Finish* and your class become available for editing.

Step 3: Add mediation handler code

- ▶ Edit the public boolean `handle(MessageContext arg0)` method to include the code for your mediation (see “Routing mediations and the port selection mediation” on page 594 for some example mediation code).

Step 4: Update the EJB deployment descriptor with the handler list

- ▶ In the *Project Explorer* window, locate the EJB project you created earlier and open the deployment descriptor.
- ▶ On the Mediation Handlers tab, click *Add* to define your mediation handler.
- ▶ Enter a Name for your mediation handler configuration. Note this name, because this is your mediation handler list name and will be needed at deployment time.
- ▶ In the Handler class field, click *Browse* and select your handler class.
- ▶ Click *Finish* to go back to the Mediation Handlers tab.
- ▶ Save your modifications.

Step 5: Export your mediation application

- ▶ Select *File* → *Export* and select *EAR file*. Click *Next*.
- ▶ Select your enterprise application (created in step 1) from the EAR project menu.
- ▶ Select the Destination to which you want to export the project and then click *Finish*.

You have successfully created a mediation handler ready for deployment.

Deploying a mediation

To make a mediation available for use within the service integration bus, perform these steps:

- ▶ Install the application EAR file containing the mediation to the application server on which you intend to use it.
- ▶ Restart the application server to pick up the new JNDI entries for the mediation application.
- ▶ Then, using the WebSphere administrative console, select *Service integration* → *Buses* and click the name of the bus for the mediation.
- ▶ From the Additional Properties list, click *Mediations* and then *New*.
- ▶ In the Name field, enter a unique name for your mediation (this is an administrative name; it does not have to be the same name as the class).
- ▶ In the Handler list name field, enter the name you gave to the mediation handler configuration when you updated the EJB deployment descriptor for your handler (in step 4).
- ▶ Click *OK* and then *save* to save your new mediation.

This makes your mediation available for use within the service integration bus. To use it, you apply it to a Web service (either a gateway service for ordinary mediations or an outbound service for a port selection mediation):

- ▶ From the WebSphere administrative console, select *Service integration* → *Buses* and click the bus that contains the service you want to mediate.
- ▶ Navigate to the service (for example, *Web service gateway instances* → *gateway_instance_name* → *Gateway services* → *gateway_service_name*).
- ▶ Depending on whether you want to apply your mediation to the request message or the response message (or both), select your mediation from the Request mediation or Response mediation list.
- ▶ For each type deployed, you should also select the *mediation bus member* in which the mediation will operate (usually, the same as the service).
- ▶ Click *OK* and then *save* to save your new service configuration.

This completes the association of your mediation with a service. The deployed mediations will now be executed when the service is called.

Routing mediations and the port selection mediation

As discussed earlier, mediations can be configured to perform routing activities on messages they receive, such as routing the message to a different destination.

This is particularly useful when working with outbound Web services in the service integration bus. These services can have multiple outbound ports defined, and it might be necessary to make a decision at runtime as to which one to route to. This decision can be made by routing code in a mediation that is associated with the outbound service as a *port selection mediation*. Example 22-1 shows sample code for such a mediation.

Example 22-1 Mediation code example for routing

```
import java.util.List;
import javax.xml.rpc.handler.MessageContext;
import com.ibm.websphere.sib.SIDestinationAddress;
import com.ibm.websphere.sib.SIDestinationAddressFactory;
import com.ibm.websphere.sib.SIMessage;
import com.ibm.websphere.sib.mediation.handler.MediationHandler;
import com.ibm.websphere.sib.mediation.handler.MessageContextException;
import com.ibm.websphere.sib.mediation.messagecontext.SIMessageContext;

public class PortSelectionMediation implements MediationHandler {

    public boolean handle(MessageContext ctx) throws MessageContextException {
        SIMessageContext siCtx = (SIMessageContext) ctx;
        SIMessage msg = siCtx.getSIMessage();
        List frp = msg.getForwardRoutingPath();
        try {
            SIDestinationAddress destination =
                SIDestinationAddressFactory
                    .getInstance()
                    .createSIDestinationAddress(
                        "RoutingDestination", //The name of the target destination
                        false);
            frp.add(0, destination);
        } catch (Exception e) {
            return false;
        }
        msg.setForwardRoutingPath(frp);
        return true;
    }
}
```

This mediation can be associated with an outbound service using the GUI:

- ▶ From the WebSphere administrative console, select *Service integration* → *Buses* and click the name of the bus of the service in which you want to mediate.
- ▶ Navigate to the outbound service (*Outbound Services* → *outbound_service_name*).
- ▶ Select your mediation from the Port selection mediation list.
- ▶ Select the *Bus member* that the mediation will operate in (usually the same as the service).
- ▶ Click *OK* and then *save* to save your new service configuration.

This completes the association of your routing mediation with the outbound service. The mediation will now be executed when the service is called.

Security

The service integration bus provides facilities for secure communication between service requestors and the bus (inbound to the bus), and between the bus and any target Web services (outbound from the bus). Security in the bus can be applied at a number of different levels, each of which we cover in this section:

- ▶ Web services security (WS-Security) in the bus
- ▶ HTTP endpoint listener authentication
- ▶ Operation-level authorization
- ▶ Using HTTPS with the bus
- ▶ Proxy server authentication

For a more general overview of security, see Chapter 9, “Web services security” on page 173.

Web services security (WS-Security) in the bus

We can configure the service integration bus for secure transmission of SOAP messages using tokens, keys, signatures, and encryption in accordance with the Web Services Security (WS-Security) specification.

To configure a bus-deployed Web service to use WS-Security requires changes to the inbound or outbound ports associated with your service to use the following types of WS-Security resources:

- ▶ WS-Security configuration—Specifies the level of security required (for example, whether the SOAP message body should be signed).

- ▶ **WS-Security binding**—Specifies information required at runtime to implement the security configuration (for example, details of keys).

A single WS-Security binding or configuration resource can be applied to many Web services. However, the security requirements for an inbound service (which effectively acts as a target Web service for a client requestor) are different from those for an outbound service (which effectively acts as a client to a target service).

Therefore, each WS-Security resource type is further divided into subtypes. When you create a new configuration resource, you specify whether the configuration applies to inbound services or outbound services. When you create a new binding resource, you specify a subtype from the following list:

- ▶ **Request consumer**—Used on requests from a client to an inbound service.
- ▶ **Request generator**—Used when generating requests from an outbound service to a target Web service.
- ▶ **Response consumer**—Used on responses from a target Web service to an outbound service.
- ▶ **Response generator**—Used when generating responses from an inbound service to a client.

Figure 22-12 illustrates these configuration options.

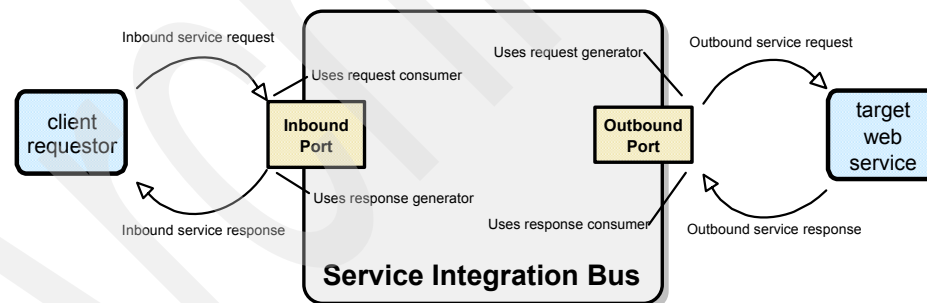


Figure 22-12 WS-Security resources used in the service integration bus

Creating a WS-Security configuration

To create this configuration using the GUI:

- ▶ From the WebSphere administrative console, select *Service integration* → *Web services* → *WS-Security configurations* and click *New*.
- ▶ Using the wizard, specify whether the configuration will be used on an *inbound service* or an *outbound service*, and then click *Next*.
- ▶ Specify a unique Name for your configuration and click *Next*.

- ▶ Click *Finish* and then *save* to save the new configuration.

You have now created a WS-Security configuration resource that can be further edited to set security levels for messages passing in and out of the service to which it is applied (for example, the required integrity, confidentiality, security tokens, and timestamps). For more information about these configuration tasks, see “How to define WS-Security configuration” on page 475.

Creating a WS-Security binding

To create a binding using the GUI:

- ▶ From the WebSphere administrative console, select *Service integration* → *Web services* → *WS-Security bindings* and click *New*.
- ▶ Using the wizard, specify whether the binding will be for a *Request consumer*, *Request generator*, *Response consumer*, or *Response generator* and then click *Next*.
- ▶ Specify a unique **Name** for your binding and click *Next*.
- ▶ Click *Finish* and then *save* to save the new binding.

You have now created a WS-Security binding resource that can be further edited to specify precise security information for messages passing in and out of the service to which it is applied (for example, information about the location of keys, encryption settings, and signing parameters). For more information about these configuration tasks, see “How to define WS-Security configuration” on page 475.

Important: When implementing WS-Security in the bus, you have to ensure that you obtain WS-Security information (such as binding information and key stores) from the owning parties of the client (when securing inbound services) and the target Web service (when securing outbound services).

Applying WS-Security resources to bus services

After you have created and configured your WS-Security resources, you can then apply them to services within the service integration bus. To do this using the GUI:

- ▶ From the WebSphere administrative console, select *Service integration* → *Buses* and then the name of the bus containing the service to which you want to apply security settings.
- ▶ Click the type of service you want to update (*inbound* or *outbound*), and then the corresponding port link (*Inbound ports* or *Outbound ports*), followed by the name of the port.

- ▶ Select the WS-Security resources you want to apply from the drop-down lists for *Security request binding*, *Security response binding*, and *Security configuration*.
- ▶ Click *OK* and then *save* to save your changes.

Your service is now configured to use the WS-Security resources that you have defined.

HTTP endpoint listener authentication

Endpoint listener authentication is a simple way of taking advantage of built-in WebSphere Application Server security settings to apply basic security to the service integration bus HTTP endpoint listeners. Using this technique, you can apply security roles and constraints to the routing servlet within an endpoint listener application so that only authenticated requestors in the correct role are able to access inbound services deployed to that endpoint listener.

Note: Endpoint listener authentication cannot be applied to the JMS endpoint listener applications.

Enabling default endpoint listener authentication

This ensures that only authenticated users are able to access inbound services deployed to the HTTP endpoint listener:

- ▶ Ensure that you have installed the HTTP endpoint listener application to which you want to apply security (see “Installing the endpoint listener applications” on page 565).
- ▶ Ensure that you have turned on global security for the application server.
- ▶ From the WebSphere administrative console, select *Applications* → *Enterprise Applications* and then click the name of the endpoint listener you want to modify (either *sibwshttp1.my_node.my_server* or *sibwshttp2.my_node.my_server*).
- ▶ From the Additional Properties, click *Map security roles to users/groups*.
- ▶ For the AuthenticatedUsers role, select *All authenticated?*
- ▶ Click *OK* and then *save* to save your new configuration.

Changing endpoint listener default security settings

The default HTTP endpoint listener application authentication settings are set to allow access to all authenticated users, but this can be changed so as to restrict access to a specific subset of users (by defining a new role in the application).

Note: If you want to change the default HTTP endpoint listener authentication settings, you must do so before you install the endpoint listener.

To do this using Rational Application Developer:

- ▶ Import the endpoint listener application you want to modify. Select *File* → *Import* → *EAR file* → *Next*, and then select the application (sibwshttpchannel1.ear or sibwshttpchannel2.ear from the install_root/installableApps directory).
- ▶ Click *Finish*.
- ▶ In the Project Explorer, find the Web project for the endpoint listener, and open the deployment descriptor.
- ▶ Go to the Security tab, and click *Add* to add your new role.
- ▶ Click the *AuthenticatedUsers* role and then *Remove* to remove it (Figure 22-13).

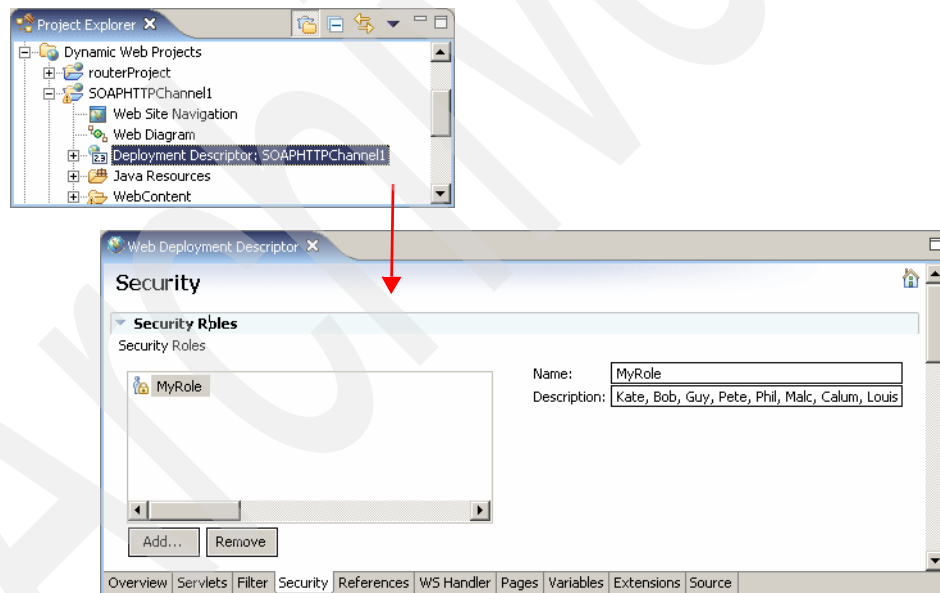


Figure 22-13 Changing the deployment descriptor for an HTTP endpoint listener

- ▶ Save the deployment descriptor and then export the application (using *File* → *Export* → *EAR file*), saving the file with a different file name than the original.
- ▶ Install the new modified endpoint listener application and then map users to the new role you have created by completing the steps described in “Enabling default endpoint listener authentication” on page 598.

Disabling endpoint listener authentication

To disable HTTP endpoint listener authentication on a server with global security enabled, follow the steps in “Changing endpoint listener default security settings” on page 598, only this time, creating a new role called `AllUsers`. You should then map this role to the special group `Everyone`.

Operation-level authorization

It is also possible to apply security to individual methods in a Web service exposed through the bus. To perform this operation-level authorization, you create a session EJB with methods matching the inbound or outbound Web service operations. These EJB methods perform no operation and are just entities for applying security. You then apply existing WebSphere Application Server authentication mechanisms to the enterprise bean. Before any Web service operation is invoked, a call is made to the EJB method. If authorization is granted, the Web service is invoked.

Your inbound or outbound Web service can have this style of security enabled by wrapping it in the `sibwsauthbean.ear`. This application is provided in the application server `installableApps` directory and is used to contain any applications to which you want to apply operation-level authorization. It is modified to set the roles and assign them to methods before being deployed to the application server. You then assign users to the roles to manage authentication.

To do this:

- ▶ First, you have to create an EAR file representing your inbound or outbound service:
 - Open a command prompt and change to the `install_root/util` directory.
 - Run the following command:

```
sibwsAuthGen.ext bus_wsd1_location service_name
```

For example:

```
sibwsAuthGen http://localhost:9080/sibws/wsd1/MyBus/MyInboundService1  
MyInboundService1
```

- ▶ Next, take a copy of the `sibwsauthbean.ear` and wrap your newly created application in this (see the *WebSphere Application Server Information Center* for instructions for how to do this).
- ▶ Install your modified `sibwsauthbean.ear` file into your application server.
- ▶ Modify your inbound or outbound service (depending on which you are enabling for authorization) to have the *Enable operation level security* check box selected.

Using HTTPS with the bus

The bus is able to send and receive messages using the SOAP/HTTPS protocol (that is, services using `https://` URLs) provided the correct Java and WebSphere Application Server security properties have been set. This is useful for exposing services in a secure fashion. To set this up:

- ▶ Edit the `install_root/java/jre/lib/security/java.security` properties file so that it includes entries for both the Sun security provider and the IBM security provider (the Sun security provider must come before the IBM provider):

```
security.provider.1=sun.security.provider.Sun
security.provider.2=com.ibm.jsse.IBMJSSEProvider
```

- ▶ Use the WebSphere administrative console to set up the following system properties (from *Application servers* → *server_name* → *Process Definition* → *Java Virtual Machine* → *Custom properties*).

Table 22-3 Properties for enabling SOAP/HTTPS in the bus

Property	Value	Description
<code>javax.net.ssl.trustStore</code>	<code>your_truststore_root_directory/TestSSL/key.jks</code>	Set truststore location
<code>javax.net.ssl.trustStore Password</code>	<code>your_truststore_password</code>	Set truststore password
<code>java.protocol.handler.pkgs</code>	<code>com.ibm.net.ssl.internal.www.protocol</code>	Use IBM reference implementation

Proxy server authentication

The service integration bus can be configured to access authenticating proxy servers when exchanging messages with target Web services or retrieving WSDL files. In this instance, the bus passes user ID and password information in HTTP headers. This is useful, because many businesses choose to locate their data and services behind proxy servers.

Enabling proxy server authentication for outbound services

To do this on a per-service basis:

- ▶ Create an authentication alias containing the user ID and password required by the proxy server (see “Creating a UDDI reference” on page 577).
- ▶ In the WebSphere administrative console, navigate to the outbound port of the service you want to modify (for example, *Service Integration* → *Buses* → *bus_name* → *Outbound Services* → *service_name* → *Outbound Ports* → *port_name*).

- Specify the proxy host name, port, and authentication alias you created and then save your modifications.
- Restart the application server.

Retrieving WSDL files through a proxy server

When creating inbound or outbound services in the bus, the server might have to pass messages through a proxy server to access WSDL documents. This requires some additional setup:

- Use the WebSphere administrative console to set up the following properties (from *Application servers* → *server_name* → *Process Definition* → *Java Virtual Machine* → *Custom properties*) and then restart the server.

Table 22-4 Properties for retrieving WSDL documents through a proxy server

Property	Value	Description
http.proxySet	true	Set this to tell the application server that it is required to work with an authenticating proxy.
http.proxyHost	hostname	Host name of the authenticating proxy.
http.proxyPort	number	The port through which the authenticating proxy is accessed, for example, 8080.
http.nonProxyHosts	hostname	List the internal machines for which authentication is not required for routing through the proxy. Separate each machine name in the list with a vertical bar (" ").

Note: when creating inbound or outbound services to retrieve WSDL files stored behind an authenticating proxy, you will have to use the command-line tools.

Gateway migration from Application Server V5.x

If you have a Web services gateway configuration from WebSphere Application Server Version 5.x that you want to use in Version 6.0, you have two options for migration. Either you can migrate the entire configuration using the supplied migration utility in Version 6.0, or you can configure your Version 6.0 application server to manage the Version 5.x node. We discuss both of these options in detail in this section.

Using the Version 6.0 gateway migration utility

In WebSphere Application Server Network Deployment Version 5.x, the Web services gateway was a stand-alone application with its own user interface, while in Version 6.0, it has been fully integrated into the service integration bus. To aid migration between the two, the 6.0 release comes equipped with a Web services gateway migration tool (`migratewsgw`) that takes a Version 5.x configuration XML file and imports it into Version 6.0, creating all the necessary configuration.

Before using `migratewsgw`

There are a number of prerequisites that must be met before using the migration tool:

- ▶ Ensure that your Version 6.0 target server is a WebSphere Application Server Network Deployment release. Gateway functionality is only supported in this release.
- ▶ Make sure that you have completed the setup steps for your Version 6.0 server to enable use of Web services with the service integration bus (see “Installation” on page 563)
- ▶ If you plan to migrate a Version 5.x configuration containing filters, install the coexistence mediation application into your Version 6.0 application server. This application (`wsgw.ear`) can be found in the `install_root/installableApps` directory of your Version 6.0 server. After the installation, the application appears in the list of installed applications under the name `sibwscoexist`.
- ▶ Ensure that you have created a bus as described in “Creating a bus” on page 569.
- ▶ Check that all the WSDL documents that were used to define the target services on the Version 5.x server are available at their given locations. If the WSDL location is a UDDI reference, check that the referenced UDDI registry is available.
- ▶ Use the Version 5.x gateway user interface to back up the gateway configuration from the Version 5.x application server as a *private configuration*.

Using `migratewsgw`

After you have completed the prerequisite steps detailed above, you are ready to use the migration tool:

- ▶ Open a command prompt and change to the `install_root/util` directory.

- Run the following command (Table 22-5 provides detailed option descriptions:

```
migratewsgw.ext -C=cell_name -S=server_name [-H=administration_hostname]
[-A=administration_port] [-T=filter_request_queue_time-to-live(ms)]
[-Q=shared_filter_request_queue_name] -G=v5_gateway_configuration_file_name
-B=bus_name -N=node_name [-U=gateway_instance_name] [-P=object_prefix]
```

Where *ext* is the file extension, *bat* for a Windows system and *sh* for a UNIX system, and square brackets ([]) indicate an optional parameter.

- If the command runs successfully, it should return without any errors.

Table 22-5 Detailed description of options for the migratewsgw command

Option	Description
-C	The name of the cell in the Version 6.0 application server to which you want to migrate.
-S	The name of the server in the Version 6.0 application server to which you want to migrate.
-H	Specifies the host name to connect to for running administration commands in Version 6.0 (default is "localhost" if not specified).
-A	Specifies the port to connect to for running administration commands in Version 6.0 (default is 8880 if not specified).
-T	When migrating a configuration containing filters, this option (an integer in milliseconds) specifies the amount of time the filter request queue should keep a message before discarding it. If it is not specified, it defaults to 0, meaning that the message is kept indefinitely.
-Q	When migrating a configuration containing filters, this option enables you to specify the name of a shared queue for processing response messages. If it is not specified, a separate queue is created for each gateway service using filters (it will have the same name as the service with StorageQueue appended).
-G	The full path to the Version 5.x gateway configuration file.
-B	The name of the service integration bus in the Version 6.0 application server to which you want to migrate.
-N	The name of the node in the Version 6.0 application server to which you want to migrate.
-U	Defines the name of the gateway instance that is created within the bus. If not specified, the name of the bus is used as the name for the gateway instance as well.
-P	A string used to specify the gateway instance namespace URI for the migrated gateway. The same string is also used to prefix the names of objects created by the migration process (for example, destinations and ports). If not specified, the default value is used (urn:ibmwsgw).

Note: A gateway configuration is migrated into a gateway instance within a service integration bus. More than one gateway can be migrated into the same bus, but in that case, the gateway instance names and namespace URIs must be different.

After using migratewsgw

Following a successful migration, the gateway configuration you migrated should now be available in the Version 6.0 application server. The following rules apply to artifacts that are being migrated:

- ▶ Gateway services, target services, UDDI references, JAX-RPC handlers, and handler lists are migrated directly. Therefore, gateway service WSDL should now be available at:

`http://hostname:port_number/sibws/wsdl/bus_name/gateway_service_name`

- ▶ Gateway services/channel combinations (Version 5.x) are replaced by specific inbound port and endpoint listener pairs (Version 6.0), because the functionality of a channel is now shared between an endpoint listener and an inbound port.
- ▶ Any use of the Apache SOAP channel in Version 5.x is migrated to a SOAP/HTTP endpoint listener and inbound port, because support for the Apache SOAP channel does not exist in Version 6.0
- ▶ Existing filters are not migrated, but are supported in Version 6.0 through the coexistence application.
- ▶ A JAX-RPC handler list is created for every gateway service/channel and gateway service/target service/port combination. These are not shared even if they contain the same handlers in the same order.

Note: WS-Security settings are not migrated using the migration tool, so these have to be recreated using the GUI in Version 6.0. See “Web services security (WS-Security) in the bus” on page 595 for more information about this.

Gateway coexistence with Version 5.x

Another way to continue using a Version 5.x gateway configuration in Version 6.0 is to use the Version 6.0 deployment manager to manage the Version 5.x cell.

Configuration

This assumes that you have a Version 5.x Network Deployment server on one machine with a federated node containing the gateway configuration on a separate machine.

To enable the coexistence setup, do the following:

- ▶ Back up your Version 5.x gateway configuration using the save and restore functionality.
- ▶ Install WebSphere Application Server Network Deployment Version 6 on the same machine as the Version 5.x Network Deployment server:
 - Create a deployment manager profile.
 - During the creation of your deployment manager profile:
 - When you choose cell and node names, you must choose names that match your V5.x deployment manager installation settings.
 - When you choose ports, you must keep your port assignments the same as your V5.x install. If the port did not exist in your V5.1 install, accept the default value.
- ▶ Ensure that the Version 5.x node and deployment manager server and the Version 6.0 deployment manager server are not running.
- ▶ Open a command prompt and change to the `install_root/bin` directory.
- ▶ Run the following command to save the applications and configuration data from your Version 5.x installation to a backup directory:

```
WASPreUpgrade.ext path_to_v5_backup_file  
                  path_to_v5_DeploymentManager_directory
```

(Where *ext* is the file extension—`bat` for a Windows and `sh` for UNIX).

- ▶ Then, run the following command to restore the Version 5.x configuration to the Version 6.0 deployment manager:

```
WASPostUpgrade.ext path_to_v5_backup_file -profileName Dmgr01  
                                         -includeApps true
```

Note: Dmgr01 is the default profile name for the Version 6.0 deployment manager profile.

- ▶ Start the Version 5.x node and the Version 6.0 deployment manager server.
- ▶ Using the backup file you saved earlier, restore your gateway configuration to the Version 5.x node.

You should now have successfully migrated your Version 5.x node (including the gateway application, configuration and associated applications) to be managed by the Version 6.0 deployment manager.

Restrictions

Note that If you choose to use a mixture of Version 5 and Version 6 gateways, the following restrictions apply to coexistence:

- ▶ The Version 5.x Web services gateway application is not supported on Version 6.0 or later application servers.
- ▶ The service integration bus endpoint listener applications are not supported on Version 5 application servers.
- ▶ The service integration bus administrative commands are only valid when run against WebSphere Application Server Version 6 application servers.

Advanced bus configuration

This section covers advanced configuration topics using the service integration bus.

Accessing bus services directly using a JAX-RPC client

In addition to accessing bus-provided services (specifically inbound and gateway services) over traditional Web services protocols such as SOAP/HTTP and SOAP/JMS, it is also possible to configure JAX-RPC clients to connect “directly” to service destinations by sending and receiving Service Data Object (SDO) messages to the destination on the service integration bus.

One of the main advantages of doing this is that users should see performance benefits in service requests and responses, due to the fact that the client and server do not perform any SOAP encoding or decoding actions (although note the restrictions at the end of this section).

If you decide to pursue this approach, the client you develop should be a container-managed JAX-RPC client so as to facilitate the sending of the SDO message (see “Client concepts” on page 106 for more information about container-managed clients).

Bus namespace and endpoint details

To access a service destination directly, the client should set the following two values in the client application deployment descriptor, or specify them dynamically at runtime:

- ▶ Binding namespace—Should be set to indicate that the client will use the service integration bus directly:

`http://www.ibm.com/ns/2004/02/wsd1/mp/sib`

- **Endpoint address**—Should be set to the destination in the service integration bus to which the client wants to connect. The format of messages that the client uses can optionally be set as well, using the following URL format:

```
sib:/[destination/path]?my_property1=my_value1&my_property2=my_value2
```

At the beginning of the URL, either `destination` or `path` must be used. When using `destination`, you should specify a specific destination name, while `path` refers to the forward routing path for the request and requires a comma-separated list of destination names. For example, both of the following URLs are valid:

```
sib:/destination?destinationName=myBus:myDestination
sib:/path?path=myBus:myDestination1,myBus:myDestination2
```

Note: When specifying a destination name in the sib URL, you should ensure that it is fully qualified. That means specifying the name of this bus it resides on first, followed by a colon (:), and then the destination name.

The sib URL can be further extended with various properties, as described in Table 22-6.

Table 22-6 Detailed description of properties for use with the sib URL

Property Name	Description
replyDestinationName	The name of the destination to be used for the reply. This should be a destination not used by other resources.
inboundService	The name of the inbound service that the requester application will use. You can omit this value if the destination is an inbound service destination connected to an outbound service configuration, because in that case, the client will attach to the outbound service through the inbound service destination.
timeout	The time the requester waits for a response. The default value is 60 seconds. A zero value indicates an unlimited wait.
reliability	The reliability of the request message (for example, assured).
replyReliability	The reliability of the reply message (for example, assured).
timeToLive	The amount of time (in milliseconds) before the request times out in the messaging engine. A zero value indicates that the request never times out.
replyTimeToLive	The amount of time (in milliseconds) before the reply times out in the messaging engine. A zero value indicates that the reply never times out.

Property Name	Description
priority	The priority of the request message.
replyPriority	The priority of the reply message.
user	The user ID required to access the request destination.
password	The password required to access the request destination.
user properties	Any user defined property, named with a user . prefix (for example, user.myProperty1).

Determining service destination names

Use the GUI to determine the destination name of the service to which you want to connect:

- ▶ From the WebSphere administrative console, select *Service integration* → *Buses* and click the bus that contains the service to which you want to connect.
- ▶ Navigate to the service.
- ▶ Click the service and then look for the Service destination name in General Properties.

Using a DII JAX-RPC client

To use this functionality in a dynamic invocation interface (DII) JAX-RPC client, ensure that you specify the sib URL format and namespace. For example:

```
myCallObject.setTargetEndpointAddress("sib:/destination
?destinationName=MyBus:MyDestination");
call.setOperationName
( new QName("http://www.ibm.com/ns/2004/02/wsdl/mp/sib",
"MyOperationCall") );
```

Using a stubbed JAX-RPC client

If you choose not to alter the deployment descriptors for a stubbed client, it is possible to programatically change it to directly call a service destination. For example:

```
((javax.xml.rpc.Stub)stub)._setProperty(
    com.ibm.websphere.webservices.Constants.PROTOCOL_NAMESPACE,
    "http://www.ibm.com/ns/2004/02/wsdl/mp/sib");
((javax.xml.rpc.Stub)stub)._setProperty(
    javax.xml.rpc.Stub.ENDPOINT_ADDRESS_PROPERTY,
    "sib:/destination?destinationName=MyBus:MyDestination");
```

Restrictions

While connecting directly to a bus-provided service is likely to provide better performance, there are certain restrictions in this approach to consider:

- ▶ JAX-RPC handler configurations for bus-provided services are not applied, because the message is not being processed over the SOAP.
- ▶ There is a limit to the datatypes that can be exchanged between the client and server using this approach. Only those that have defined mappings in the JAX-RPC specification are supported. If the service interface has any elements that map to a SOAPElement, they cannot be used.

Using the bus with the weather forecast application

In this section, we provide a practical example of using the service integration bus with the weather forecast sample application (for an introduction to the application, see Chapter 14, “Sample application: Weather forecast” on page 247).

In this example, we use the Web services features of the bus to expose our weather application as a gateway-provided Web service. This will yield the following benefits:

- ▶ Make use of the protocol transformation features of the bus to expose a SOAP/JMS Web service over the SOAP/HTTP protocol.
- ▶ Use the functionality of the Web services gateway to expose our existing Web service at a new endpoint (introducing this indirection between the client and service enables us to move the actual target service in future without disruption to client requestors).
- ▶ Introduce a mediation to the gateway service to log all client requests to the target service.

Preparation

Important: We will be making use of Web services gateway functionality in this example. This functionality is only available in the Network Deployment version of WebSphere Application Server, so ensure that you are using this version before proceeding.

In order to use the sample application with the bus, make sure you have completed the following steps:

- ▶ The service integration bus must be configured for use with Web services, so ensure that you have completed the instructions for “Installation” on page 563.
- ▶ In this example, we will be exposing a SOAP/JMS version of the weather application (WeatherEJBJMSServer), so ensure that you have completed the steps to create this, as described in “Web services and JMS binding” on page 303.

Configuration steps

Note: The service integration bus can be used on a different server from the weather application, but if you do this, ensure that you review and complete the “Remote configuration steps” on page 616 before continuing.

To expose our weather forecast application as a Web service provided by the bus, you should complete the steps listed in this section.

Expose the weather JMS WSDL binding at a URL

To access a Web service, the bus has to know the WSDL information for the target Web service. In this case, we do this by making the SOAP/JMS WSDL binding for the weather service available at a URL:

- ▶ Using Application Developer, create a new Web project (select *File* → *New* → *Dynamic Web Project*) called WeatherEJBJMSWSDLWeb.
- ▶ Use the *Show Advanced* button to ensure that this project is associated with the WeatherEJBJMSServer EAR project and click *Finish* when done (Figure 22-14).

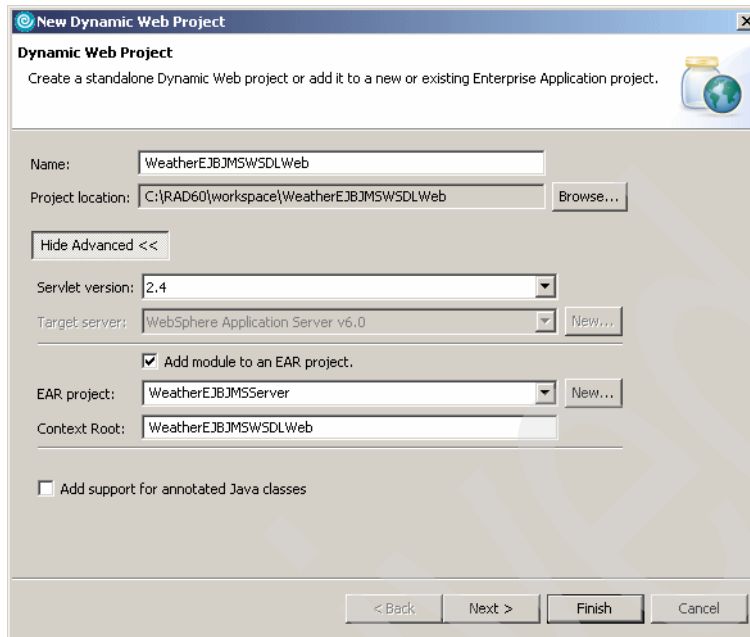


Figure 22-14 Creating a new JMS WSDL project

- ▶ Locate your existing WeatherEJBMS project in your workspace (under EJB Projects in the J2EE perspective) and copy the WeatherJMS.wsdl from *ejbModule* → *META-INF* to the WebContent directory of your newly created WeatherEJBMSWSDLWeb project.
- ▶ Finally, restart the WeatherEJBMSServer project on your application server, and verify that the WSDL is available at the following URL by navigating to it using a Web browser:

`http://machine_name:9080/WeatherEJBMSWSDLWeb/WeatherJMS.wsdl`

Create and configure a bus

You should now create and configure a bus to expose the weather service SOAP/JMS binding over the SOAP/HTTP protocol:

- ▶ Create a new bus called weatherJMSBus following the instructions detailed in “Creating a bus” on page 569.
- ▶ Create a SOAP/HTTP endpoint listener (named SOAPHTTPChannel1) following the instructions detailed in “Creating an endpoint listener” on page 571. Make sure that you connect this endpoint listener to the weatherJMSBus.
- ▶ Create a new gateway instance called weatherGateway as detailed in “Creating a gateway instance” on page 581.

- ▶ Create a new gateway service under the weatherGateway. Following the instructions detailed in “Creating a gateway service” on page 582, you should create a WSDL-defined Web service provider using the following values (leave all others at their defaults):
 - Name: weatherGatewayService
 - WSDL location:
 - http://machine_name:9080/WeatherEJBMSWSDLWeb/WeatherJMS.wsdl
 - Select port: WeatherJMSJMS
 - Endpoint listeners: SOAPHTTPChannel1
- ▶ Restart the application server to ensure that the messaging engine for our new bus is available.
- ▶ You should now be able to access WSDL for the new bus-provided weather service at the following URL:
 - http://machine_name:9080/sibws/wsdl/weatherJMSBus/weatherGatewayService

Create the logging mediation

Before testing the new weatherGatewayService, we first create a mediation to log requests to it.

Follow the instructions detailed in “Writing a mediation” on page 590 using the following values:

- ▶ Enterprise application project name: WeatherMediationServer
- ▶ Mediation handler class name: WeatherLoggingMediation
- ▶ Mediation handler package name: itso.mediations
- ▶ Mediation handler code should use the code shown in Example 22-2 (also available in the sample code):
 - \SG246461\sampcode\s-bus\WeatherLoggingMediation.java
- ▶ Mediation handler name: WeatherLoggingMediationList

Example 22-2 Mediation for weather forecast example

```
package itso.mediations;

import javax.xml.rpc.handler.MessageContext;
import com.ibm.websphere.sib.mediation.handler.MediationHandler;
import com.ibm.websphere.sib.mediation.handler.MessageContextException;
import com.ibm.websphere.sib.mediation.messagecontext.SIMessageContext;
import com.ibm.websphere.sib.SIMessage;
import commonj.sdo.DataGraph;
import commonj.sdo.DataObject;
```

```

public class WeatherLoggingMediation implements MediationHandler {

    final String loggingString = "WeatherLoggingMediation : ";
    public boolean handle(MessageContext context)
        throws MessageContextException {
        System.out.println(loggingString + "Started.");

        // Get the SIMessage from the context (this gives access to the payload)
        SIMessageContext siContext = (SIMessageContext) context;
        SIMessage msg = siContext.getSIMessage();
        try {
            // Get the root DataGraph and Info and Body dataObjects
            DataGraph graph = msg.getDataGraph();
            DataObject dataObject = graph.getRootObject();
            DataObject body = dataObject.getDataObject("info/body");

            // Write body info to log
            System.out.println(loggingString + "DataObjectBody output : "
                + body.toString());

            // Get the incoming client msg and write to log
            String clientString = body.getString("parameters/theDate");
            System.out.println(loggingString + "Client has requested weather
                information for date : " + clientString);

        } catch (Exception e) {
            System.out.println(loggingString + "ERROR : " + e.getMessage());
            e.printStackTrace();
            return false;
        }

        System.out.println(loggingString + "Ended.");
        return true;
    }
}

```

Deploy the logging mediation

We now have to deploy our new WeatherLoggingMediation to the weatherGatewayService following the instructions detailed in “Deploying a mediation” on page 593:

- Create the mediation on the weatherJMSBus using the following values:
 - Mediation name: WeatherLoggingMediation
 - Handler list name: WeatherLoggingMediationList

- ▶ Ensure that you deploy the mediation as a *Request Mediation* to the weatherGatewayService on the weatherJMSBus.

Execute a client

Our weatherGatewayService is now ready for use. We can execute a client to connect over the SOAP/HTTP protocol to the bus that executes the actual weather JMS target service over SOAP/JMS. The mediation we have deployed will log details of the incoming request:

- ▶ To test the new service, we can use the Web Services Explorer¹ in Application Developer to point at the new WSDL file (for details about using this tool, refer to “Web Services Explorer” on page 339):

`http://machine_name:9080/sibws/wsd1/weatherJMSBus/weatherGatewayService`

- ▶ Try executing a simple method such as `getDayForecast`. You should get an output in your status window similar to that shown in Figure 22-15.
- ▶ Also, check the application server log where the weatherJMSBus is defined—this will contain logging statements from the WeatherLoggingMediation.

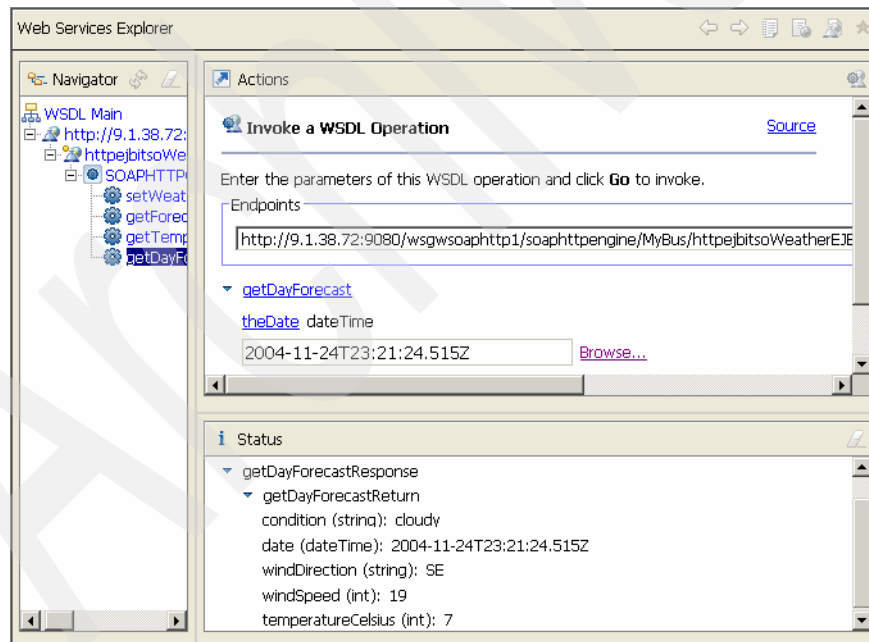


Figure 22-15 Results of connecting to weather JMS Web service through the bus

¹ This requires a WSDL fix for the Web Services Explorer (to be shipped in a fix pack).

Remote configuration steps

If you want to run your service integration bus on a different application server from the weather application, you should ensure that you have configured your systems for remote communication by completing the following steps:

- ▶ Ensure that you have configured the provider endpoints for you weather queue connection factory as detailed in “Setting up messaging for the JMS Web service” on page 713 (“Creating the queue connection factories” on page 716).
- ▶ When completing the steps in “Expose the weather JMS WSDL binding at a URL” on page 611, you should ensure that the bus knows where the service is located by editing the `WeatherJMS.wsdl` file and modifying the soap address to specify the exact JNDI provider URL as follows:

```
<wsdl:soap:address location="jms:/queue?destination=jms/weatherQ
&connectionFactory=jms/weatherQCF
&targetService=WeatherJMSJMS
&jndiProviderURL=iop://target_service_machine_name:2809/" />
```


Summary

The service integration bus acts as a powerful tool for integrating and managing Web services within your organization. It acts as a middleware component capable of bridging the gap between Internet and intranet environments during Web service invocations. It provides the following benefits:

- ▶ A central location for Web services definition and management.
- ▶ Web services integration with asynchronous messaging engines.
- ▶ Powerful tools managing the control, flow, and routing of messages.
- ▶ The ability to perform protocol transformation of Web services messages (allowing clients to connect to a SOAP/JMS Web service over SOAP/HTTP, for example).
- ▶ Powerful features for implementing security on Web services.
- ▶ Automated publishing and updating of bus-provided Web services to UDDI registries.
- ▶ Tools to enable the dynamic retargeting of Web services requests at runtime.
- ▶ Integrated GUI and command-line administration features.

Troubleshooting

To identify and resolve Web services problems in the service integration bus, we can use the standard WebSphere Application Server facilities:

- ▶ Check for error messages in the WebSphere administrative console and in the application server's SystemOut and SystemErr log files.
- ▶ For further analysis, enable the application server debug trace to provide a detailed exception dump. Useful trace strings to use are:

```
com.ibm.ws.sib.webservices.*=all=enabled  
com.ibm.ws.webservices.multiprotocol.*=all=enabled
```

More information

For more information, see:

- ▶ *Understand Enterprise Service Bus scenarios and solutions in Service-Oriented Architecture, Part 1*, available at:
<http://www.ibm.com/developerworks/webservices/library/ws-esbscen/>
- ▶ *WebSphere Application Server Version 6.0 Information Center*, available at:
<http://publib.boulder.ibm.com/infocenter/ws60help/index.jsp>



Implementing a private UDDI registry

This chapter describes how to implement a private UDDI registry.

You can implement a private UDDI registry either from the integrated testing environment with IBM Rational Application Developer or from IBM WebSphere Application Server Network Deployment. Because the private UDDI registry is implemented as an enterprise application, you also require an application server to deploy and run the registry.

With Version 6.0 of WebSphere Application Server, and therefore also the integrated test environment of Application Developer, IBM supports UDDI Version 3.0.

Installing a private UDDI registry

In this chapter, we describe how a private UDDI registry can be installed. To this end, we show how to install the registry using the integrated test environment and how to install it using the WebSphere Application Server Network Deployment product.

Installing using the integrated test environment

Installation of the private UDDI registry within the integrated test environment is straightforward and performed by a wizard.

You start the wizard by selecting *File* → *New* → *Other* → *Web Services* → *Unit Test UDDI*. Upon selection, the wizard asks you if you want to install a new registry, update an existing one, or remove an existing registry (Figure 23-1).

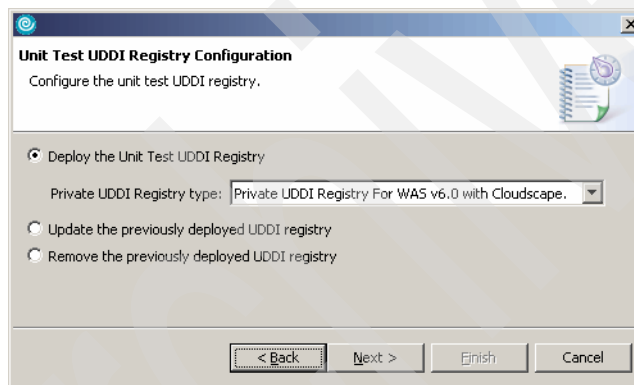


Figure 23-1 Installing a UDDI registry

Because we want to install a new registry, we select the option to create a new registry. Also, for the test environment, the integrated IBM Cloudscape database is a good choice.¹

After we choose that the registry should be using the integrated Cloudscape database and be installed in a WebSphere Application Server Version 6.0 environment, we proceed to the next panel where we can configure the registry (Figure 23-2).

¹ You can, however, also select DB2 as the target database system. And finally, you can also choose what application server version on which the UDDI should be installed.

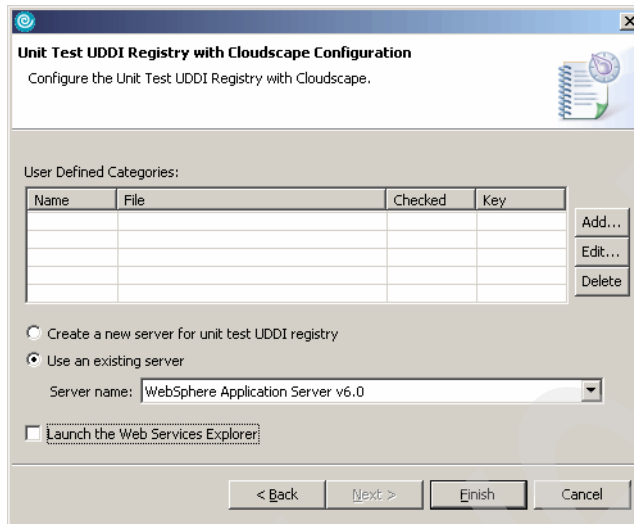


Figure 23-2 Configuring a Cloudscape-based UDDI registry

At this point in the installation, you can define new categories and choose if the registry should be installed on a new server of its own or an existing server. We recommend creating a new server for the registry if you do not want to interfere with any of the already installed applications. After these simple steps, the private UDDI registry can be used.

Installing using WebSphere Application Server Network Deployment

The installation of a registry using the Network Deployment product is not as straightforward as the installation using the integrated test environment. The following issues have to be considered when installing a private registry:

- ▶ If your registry is supposed to be used in a development environment only, you can install a default UDDI node. Otherwise, we recommend a user-customized UDDI node.
- ▶ Your UDDI registry cannot be installed in a clustered environment. Instead, it can only be deployed in a Network Deployment cell.
- ▶ The actual installation is run from the command line and can be targeted against Cloudscape, DB2, or Oracle.
- ▶ For a production environment, you must configure WebSphere security.

You can find more information about how to install a registry in the *WebSphere Application Server Information Center*, available at:

<http://publib.boulder.ibm.com/infocenter/ws60help/index.jsp>

You will find almost all of the information under *Learn about WebSphere Applications* → *Web Services* → *Using the UDDI Registry* or *Developing and deploying applications* → *Developing WebSphere applications* → *Setting up and deploying a new UDDI Registry*.

Using the UDDI registry

The UDDI registry can be programmatically accessed using the *IBM UDDI Version 3 Client for Java*, which is an JAX-RPC interface that you can use to access the registry services. This is the recommended way of accessing the UDDI registry. In addition, the registry can be also accessed using plain SOAP.

Note: The UDDI4J interface and the EJB interface that were shipped with the WebSphere Application Server Version 5.x product support only UDDI Version 2.0. They are, therefore, marked deprecated and should not be used.

More information about how to access the UDDI registry can be found in the *WebSphere Application Server Information Center*.

A UDDI GUI, also referred to as the UDDI User Console, is provided to enable users to familiarize themselves with the key concepts of UDDI. This UDDI GUI can be used both to publish and to inquire about UDDI entities. Because the GUI does not provide all the functions of the UDDI API, some of the more complex operations can be invoked using the programmatic interfaces instead.

And finally, the UDDI can be also explored using the UDDI Explorer that is part of the Web Services Explorer.

The remainder of this chapter focuses on how to use the private registry using the provided GUI tools. However, we also show how to use the command-line tools to publish and to remove entries from the registry.

Using the UDDI GUI

The following sections show how to perform some simple operations using the UDDI GUI.

Starting the UDDI GUI

Open a browser with the UDDI URL, and the private UDDI home page opens (Figure 23-3). If your registry is installed on the computer you are currently using, the URL is:

```
http://127.0.0.1:9080/uddigui
```

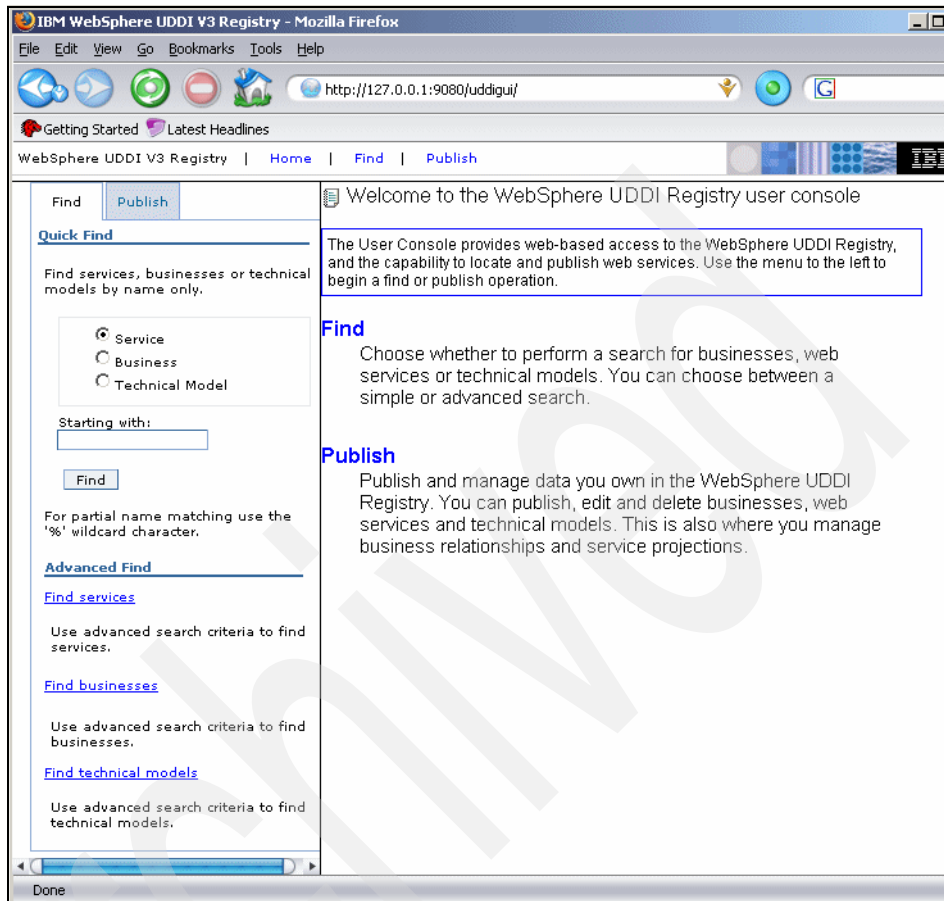
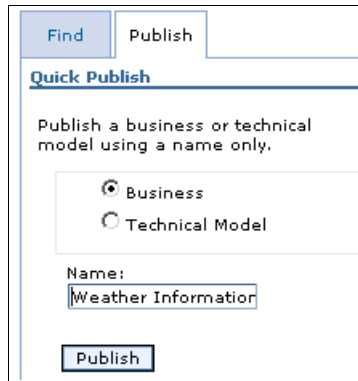


Figure 23-3 Private UDDI home page

Publishing businesses

To add a business, use the Publish tab (left side), enter a business name in the *Name* field, and click *Publish* (Figure 23-4). Throughout this chapter, we use the *Weather Information* business as an example.



Find Publish

Quick Publish

Publish a business or technical model using a name only.

☒ Business
☐ Technical Model

Name:
Weather Information

Publish

Figure 23-4 Publishing a business

Note that the *Business* radio button is selected. This operation will publish a business with the specified name using the *Quick Publish* option. With this option, no other information about the entity is provided.

If you want to provide further details when publishing a business, such as a description and some categorization, you should use the *Advanced Publish* option. However, you can always change the information later.

Locating entries

Before you can publish a new service, you have first to locate the business to which you want to add the service. To this end, you have to find the businesses that you own, which, of course, consist of one entry at the moment.

To locate and edit the entries you own, click *Show owned entities* (Figure 23-5).

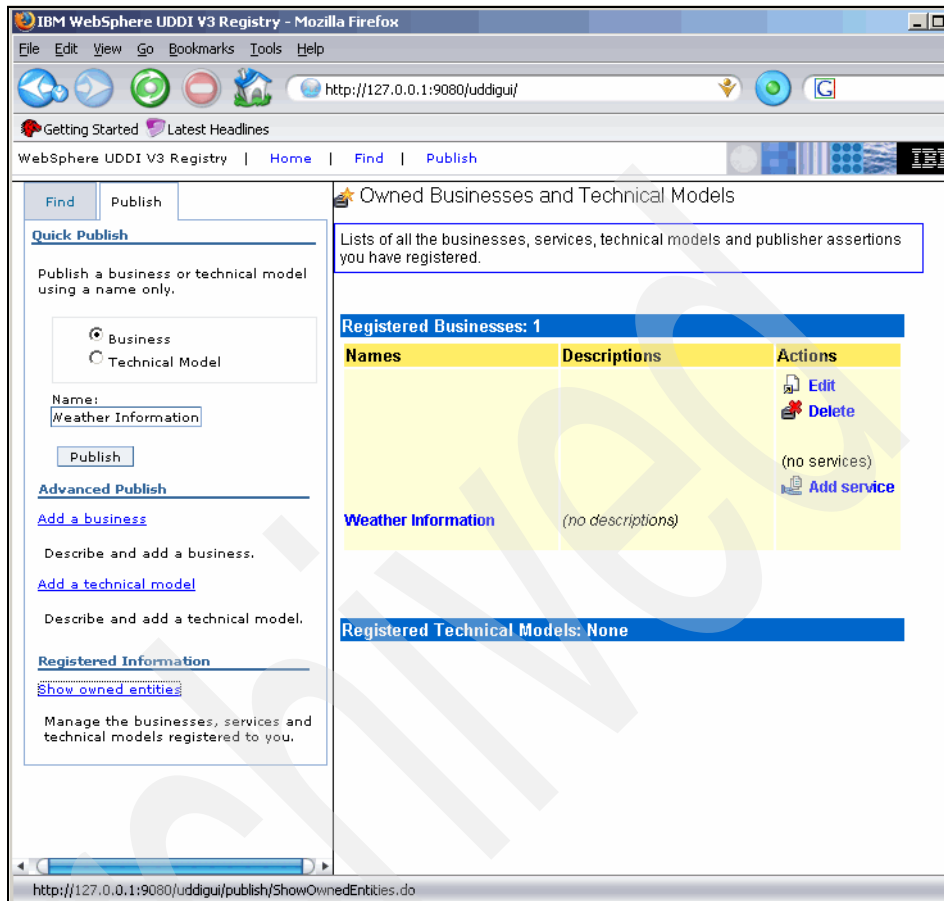


Figure 23-5 Owned entities

You can edit or delete the businesses that you own. Also, you can work with the technical model you defined, if any.

Adding services

You can add a service to the business by clicking *Add service* and completing the form (Figure 23-6). You have to click *Add* individually for each piece of information that you enter, and click *Add service* (bottom) when done.

Publish Service

Describe and publish a service.

Names

Names of the service.

☒ Predefined language code
 No language code

☐ Custom language code

Name [Add Name](#)

Descriptions

Free text to describe the service.

Description

☒ Predefined language code
 No language code

[Add Description](#)

Bindings

Bindings for the business service.

[Add a Service Binding](#)

Categorizations

Categorizations

[Show category tree](#)

Type	Key Name	Key Value	Action
New category:			Add Categorization

Figure 23-6 Adding a service to a business

To add binding information, you have to click *Add a Service Binding* and provide at least the service access point that is required for the service invocation. You can also provide a short description of the service.

Adding a technical model

Add a technical model using *Quick Publish* (Figure 23-7).

Quick Publish

Publish a business or technical model using a name only.

☐ Business
☒ Technical Model

Name:

Figure 23-7 Publishing a technical model

Editing a technical model

Edit the technical model from the owned entities panel (Figure 23-8):

- ▶ The provider URL that points to a WSDL file on a provider server can be edited by clicking *Add an Overview Document*, which opens a new window.
- ▶ Click *Update Technical Model* when done.
- ▶ You can, alternatively, provide the extra information about the technical model at the same time as you create it by using the *Advanced Publish* option.

Edit Technical Model

Describe and publish a technical model.

Name
Name of the technical model (only one name allowed).

Name	Language	Action
Weather Service	No language code	Remove

Descriptions
Free text to describe the technical model.

Description Predefined language code Add Description

Overview Documents
References to documents that provide overviews of the technical model.

Add an Overview Document

Categorizations
Categorizations

Show category tree

Type	Key Name	Key Value	Action
New category: UNSPSC v3.1			Add Categorization

Update Technical Model Publishes the technical model with the current form contents.

Figure 23-8 Updating a technical model

Matching service and model

Finally, you have to match the service and the model. To this end, go through the following steps:

- ▶ Click *Show owned entities*.
- ▶ Click *Show Services* for the business entity.
- ▶ Click *Edit* for the Weather Forecast service.
- ▶ Click *Add a Service Binding*.
- ▶ Click *Add Technical Model Instance Information*.

- Click *Add Technical Model* (Figure 23-9).

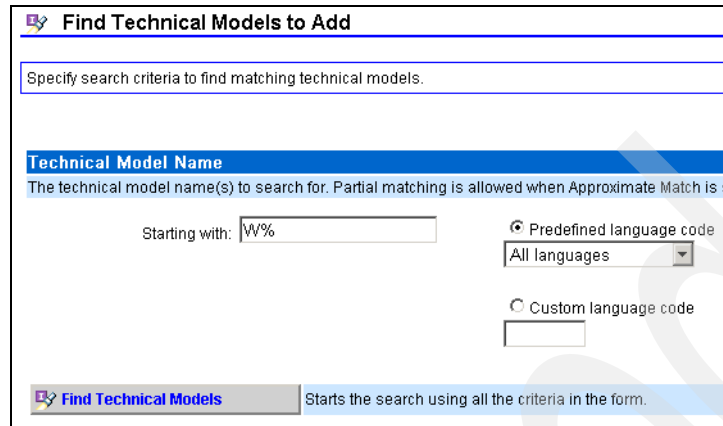
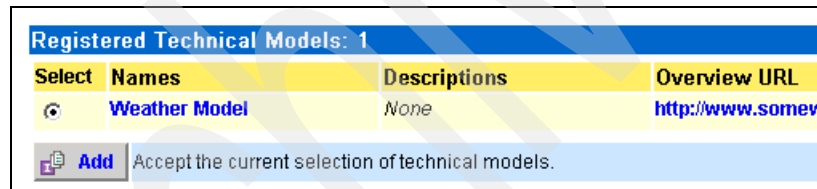


Figure 23-9 Finding a technical model

- Enter W% as the pattern to look for. Click *Find Technical Models*.
- Select the technical model from the list of results and click *Add* (Figure 23-10).



Select	Names	Descriptions	Overview URL
☐	Weather Model	None	http://www.someev...

Figure 23-10 Assigning a technical model to a service

- Click *Add Technical Model Instance* to save your changes.
- Click *Add Binding* to save your changes.

Using the UDDI Explorer with the private UDDI registry

When you are developing applications using Application Developer Version 6.0, you can also use the Web Services Explorer to work with the UDDI.

Start the UDDI Explorer by selecting *Run* → *Launch the Web Services Explorer*.

In the home panel, click *UDDI Main*, assign a name to the private registry, and define the inquiry API (Figure 23-11):

<http://127.0.0.1:9080/uddisoap/inquiryapi>

Click *Go* to connect to the registry.

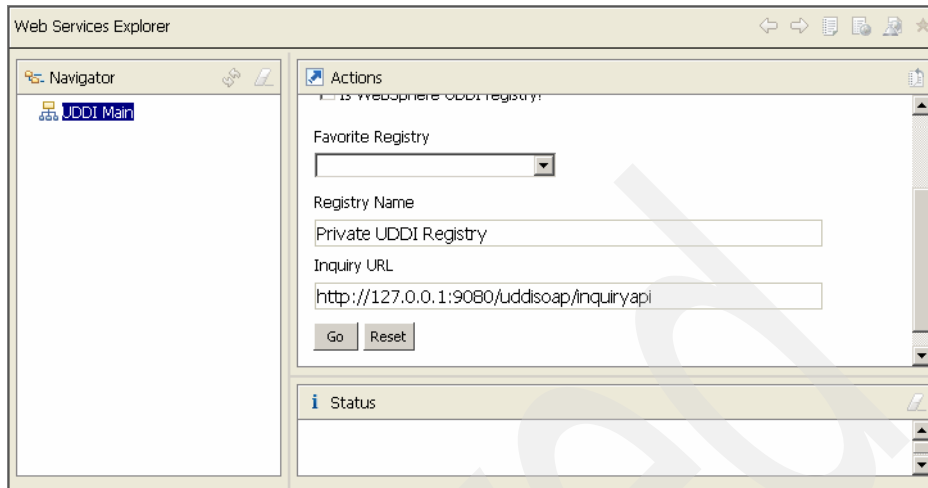


Figure 23-11 UDDI Explorer: Set up private UDDI registry

Publishing

Click the *Publish* icon  and fill in the form. You must provide the publishing API URL and a user ID and password (Figure 23-12). The user ID becomes the owner of the entry. Without WebSphere security enabled, you can type any user ID and leave the password empty.

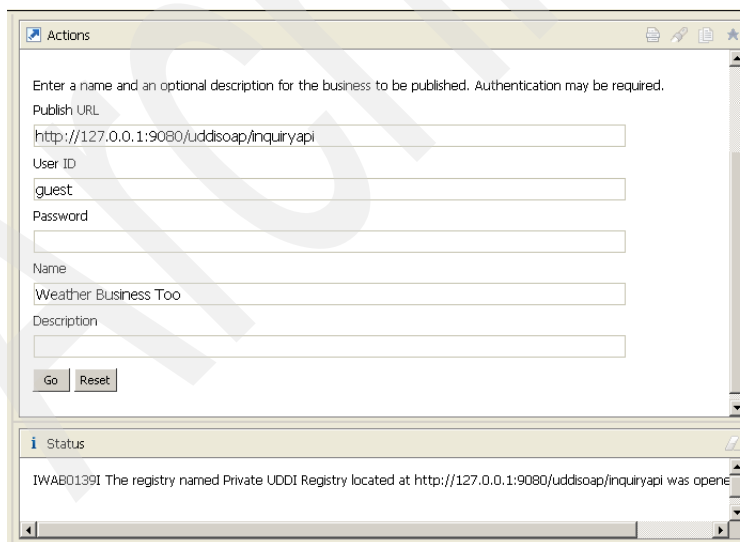


Figure 23-12 Publishing a business

Finding information

Use the *Find* icon  to execute queries against the registry (Figure 23-13).

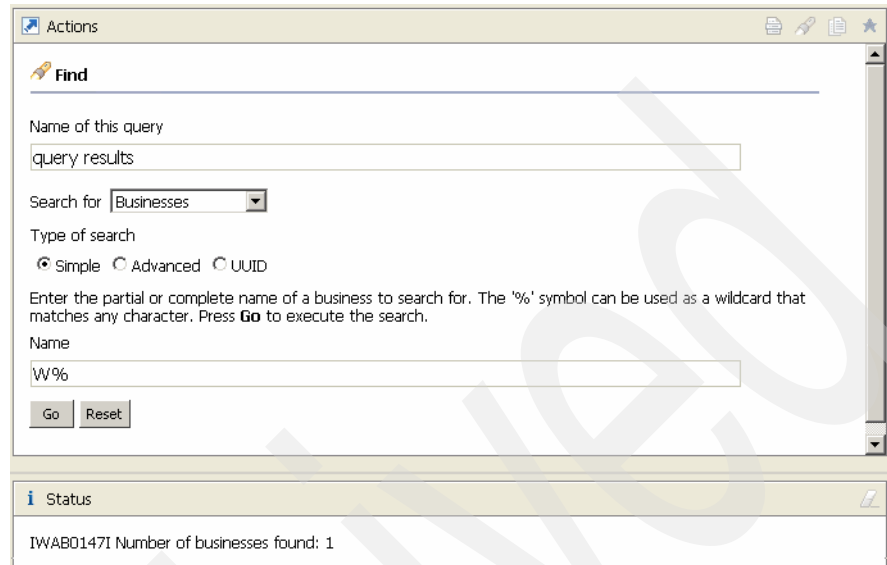


Figure 23-13 Exploring the UDDI registry

Starting from this point, you can easily navigate through the registry.

Using command-line tools

The creation of business and services in the UDDI registry can be also performed using the command-line tools that are installed together with RAD. If `<RAD60_HOME>` is pointing to the installation directory, the tools can be found in the `<RAD60_HOME>/bin` directory.

Using the command-line tools is more efficient if you are sure of what you are doing. Also, the tools can be used in automated tasks. In addition, the command to publish a service creates a technical model for you so that you do not have to do it yourself.

More details about how to use these tools and examples that publish the same information as described on the previous pages can be found in “UDDIPublish/UDDIUnpublish” on page 411.

Summary

In this chapter, we explained how to install a private UDDI registry and how to use it with the provided GUI tools. We also briefly covered the command-line tools to populate a registry.

More information

More information about UDDI can be found in the *WebSphere Application Server Information Center* and also at:

<http://www.uddi.org>

Archived

Web services caching

This chapter describes why and how to cache Web services. We explain the functions and techniques used to improve Web service performance by applying the IBM WebSphere Application Server dynamic cache service.

For general information about WebSphere cache services, refer to the IBM Redbook *WebSphere Application Server V6: Scalability and Performance Handbook*, SG24-6392.

Web services caching

One of the downsides of Web services is still the poorer performance compared to other distributed computing technologies, such as RMI-IIOP. This is especially true for SOAP/HTTP-bound Web services, where the main performance impact is the XML encoding (marshalling and de-marshalling).

There are several specific best practices about how to design and implement Web services, as described in Chapter 12, “Best practices” on page 225. Also, several J2EE and Java best practices should be applied to optimize service implementation performance. All these techniques do not improve the performance of a J2EE Web service as well as a properly defined and implemented cache.

Web services caching is based on the Application Server dynamic cache service. The dynamic cache service can be applied on the Web services server and on the Web services client side.

Note: Dynamic caching can only be used for HTTP-bound (and HTTPS-bound) Web services.

Using the dynamic cache service for Web services does not require a change to the Web service implementation or deployment descriptors. Defining the cache only requires a configuration file to be created. We describe how to create and configure this configuration file throughout this chapter.

When applying caching on the Web services client side, no changes on the server side are required. Similarly, for server-side caching on the Web service server, no changes on the client side are required.

Most benefits of client side caching can be expected when the Web services client/server communication is over a slow network.

When planning to apply dynamic caching for Web service applications, one of the main tasks is to define the cacheable service operations. This is because not all operations should be cached, for example, dynamic or confidential data. Depending on how large an application is, and how many operations are exposed, this is the most complex task when implementing a cache.

Even so, when using caching for Web services, the performance improvement can be very high. We cannot provide general performance numbers, but we show a performance improvement using a simple example in the sections that follow (see “Simple load test” on page 643).

The dynamic cache service runs in the application server Java virtual machine. All calls to servlet or commands are intercepted and handled based on the cache configuration. Web services caching works by intercepting the calls to the Web service router servlet.

Further information about the dynamic cache service concept and functions for other J2EE components can be found in the *WebSphere Application Server Information Center*.

Caching Web services

To cache Web services, the dynamic cache service and servlet caching must be enabled for the application server. Depending on where caching is used, these services must be enabled on the Web services client or server side, using the administrative console:

- ▶ **Activate dynamic cache service**—The dynamic cache service is enabled by default. To verify or activate the service, select *Servers* → *Application servers* → *server1* → *Container services* → *Dynamic cache service*.

Verify or select *Enable service at server startup*. Click *OK*, and save the changes to the server configuration.

- ▶ **Activate servlet caching**—Servlet caching is not enabled by default. To activate this service, select *Servers* → *Application servers* → *server1* → *Web Container Settings* → *Web container*.

Select *Enable servlet caching*, click *OK*, and save the changes.

The server has to be restarted to activate the changes.

Cache configuration

The cache configuration on the server and client side is defined in an XML file with the name `cachespec.xml`. The dynamic cache service searches for this file in specific locations. The configuration can be defined either globally at the application server level or at the enterprise application level.

Tip: We recommend defining the dynamic cache configuration file `cachespec.xml` at the application level, and not at the server level. This creates a well-separated and easier to maintain configuration.

Configuration at the server level

To define a cache configuration at the server level, create and configure the `cachespec.xml` file in the directory `<WAS_HOME>/properties`.

Note: <WAS_HOME> refers to the directory where the WebSphere Application Server is installed. For the test environment in Application Developer, the location is <RAD60_HOME>\runtimes\base_v6, where <RAD60_HOME> is where Application Developer is installed.

Configuration at the application level

To have the cache configuration defined at the application level, create the file `cachespec.xml` in the WEB-INF folder (for Web modules) or the META-INF folder (for EJB modules).

Creating the cache configuration file

To configure a Web service cache:

- ▶ Create a `cachespec.xml` file.

WebSphere Application Server ships with a sample cache configuration file. This file can be used to create application-specific configuration. The sample file `cachespec.sample.xml` is located in the <WAS_HOME>/properties directory. Copy this file to the appropriate module directory.

- ▶ Define the elements to cache.

Create the cache entry elements in `cachespec.xml` to identify the object to cache. Specific for Web services are the classes:

- `webservice`—This class extends the dynamic cache with special component types for Web services requests.
- `JAXRPCClient`—The `JAXRPCClient` class is used to define a cache entry for the Web services client cache.

- ▶ Define the cache identifier rule.

To cache an object, the cache service must generate a unique object identifier. The rule or rules for the ID generation are defined in the <cache-id> section. To define unique IDs for Web services, the following techniques are available in WebSphere Application Server:

- Calculating a hash of the SOAP envelope
- Using SOAPHeader entries
- Using operation and part parameters
- Using custom Java code to build the cache ID from an input SOAP message

Table 24-1 shows the Web service-specific cache ID components.

Table 24-1 Web service cache ID components

Type	Class	Description
header	servlet webservice	Retrieves the named request header.
SOAPEnvelope	webservice JAXRPCClient	Retrieves the SOAPEnvelope from a Web services request. id=Hash uses a hash of the SOAPEnvelope. id=Literal uses the received SOAPEnvelope.
SOAPAction	webservice	Retrieves the SOAPAction header (if available) for a Web services request.
serviceOperation	webservice	Retrieves the service operation for a Web services request.
serviceOperation-Parameter	webservice	Retrieves the specified parameter from a Web services request.
operation	JAXRPCClient	An operation defined in the service WSDL file. The id attribute is ignored, and the value is the operation or method name. If the namespace of the operation is specified, the value should be formatted as: namespaceOfOperation:nameOfOperation
part	JAXRPCClient	An input message part in the WSDL file or a request parameter. Its id attribute is the part or parameter name, and the value is the part or parameter value.
SOAPHeaderEntry	JAXRPCClient	Retrieves specific information from the request SOAP header. The id attribute specifies the name of the entry. In addition, the entry of the SOAP header in the SOAP request must have an actor attribute that contains: com.ibm.websphere.cache For example: <soapenv:Header> <getQuote soapenv:actor= "com.ibm.websphere.cache">IBM </getQuote> </soapenv:Header>

Important: When using Web services caching, application-defined JAX-RPC handlers are not executed on the side where caching is enabled.

Cache monitor

The *cache monitor* is a Web application that provides a real-time view of the current state of the dynamic cache. It can be used to verify the cache configuration and if the dynamic cache is working as expected. The cache monitor provides an easy to use Web interface to view and work with the dynamic cache.

The cache monitor application is shipped with WebSphere Application Server and is contained in following enterprise application:

```
<WAS_HOME>/installableApps/CacheMonitor.ear
```

The WebSphere Application Server test server environment shipped with Rational Application Developer also contains the cache monitor application. Therefore, caching can be configured and tested in the development environment.

The cache monitor is not installed by default. To install the application, use the common enterprise application installation steps. For more information, refer to the *WebSphere Application Server Information Center*.

To access the installed and started cache monitor, use the following URL:

```
http://<server_host_name>:<server_port_number>/cachemonitor  
http://localhost:9080/cachemonitor
```

Web services server cache

This section shows the steps to implement and activate a server cache for the sample weather forecast JavaBean service:

- ▶ Create a `cachespec.xml` file for the weather forecast Web service.
Copy the `cachespec.sample.xml` and `cachespec.dtd` files from the directory `<RAD60_HOME>\runtimes\base_v6\properties` to the Web service server project `/WeatherJavaBeanWeb/WebContent/WEB-INF`.
Either use the import or drag-and-drop function. Rename the sample file to `cachespec.xml`.
- ▶ Define the cache configuration.
For the weather forecast example, Example 24-1 shows the configuration `cachepsec.xml` file.

Example 24-1 Weather forecast server cache configuration

```
<?xml version="1.0" ?>
<!DOCTYPE cache SYSTEM "cachespec.dtd">
<cache>
  <cache-entry>
    <class>webservice</class>
    <name>/services/WeatherJavaBean</name>
    <sharing-policy>not-shared</sharing-policy>
    <cache-id>
      <component id="Hash" type="SOAPEnvelope" />
      <timeout>420</timeout>
    </cache-id>
  </cache-entry>
</cache>
```

- ▶ With this configuration, the dynamic cache service will intercept and cache all calls to the URL `/WeatherBeanWeb/service/WeatherJavaBean`. The cache service adds the Web projects context root (`/WeatherBeanWebproject`) to the value defined in the `name` section automatically. If the `name` value is defined with the context root, the cache service does not add the context root again.

In this example, the cache timeout is set to 420 seconds (`<timeout>` tag). All cache entries older than this time are discarded from the cache. The complete cache or specific cache entries can be cleared using the cache monitor.

The `<component>` tag defines that the dynamic cache service creates the cache ID by generating a hash for the whole SOAP envelope.

- ▶ Deploy and start the sample application.

Add the `WeatherJavaBeanServer` enterprise application to the server and start the application. The cache service is now active and is used for all further Web service requests to the weather forecast application.

Figure 24-1 shows the sample environment we accomplished with this configuration:

- The thick, red lines show the request flow when the object is already cached.
- The dotted, blue lines show the flow for objects that are not in the cache. The cache service generates the cache ID and puts those objects in the cache.

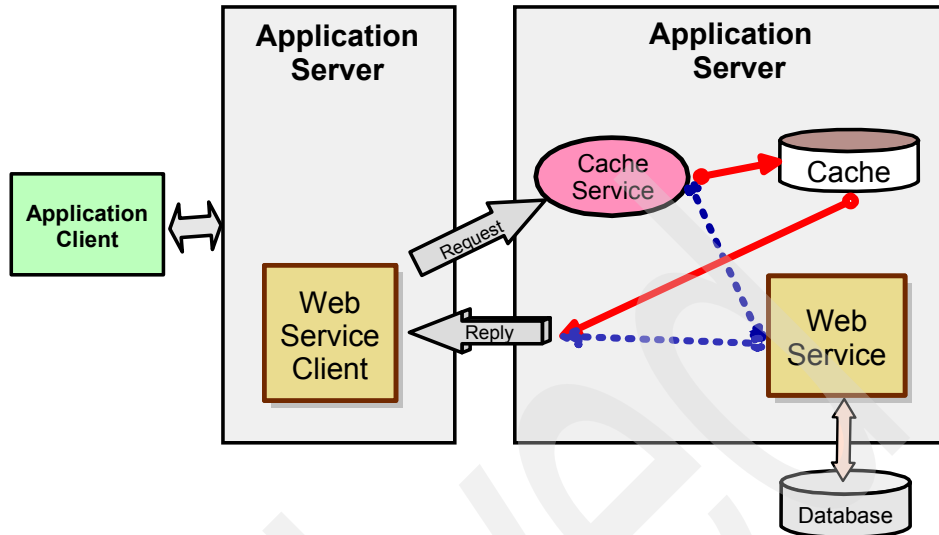


Figure 24-1 Server caching environment

- Verify the active cache configuration using the cache monitor.

Start the cache monitor (see “Cache monitor” on page 638), and click *Cache Policies*. The Current Cache Policies section shows the weather forecast cache template. Selecting a specific cache policy opens the details view that displays, for example, the rules for the cache ID (Figure 24-2).

Cache instance: baseCache

Current Cache Policies

Template	Class
/WeatherBeanWeb/services/WeatherJavaBean	webservice

Cache Policy Detail for /WeatherBeanWeb/services/WeatherJavaBean

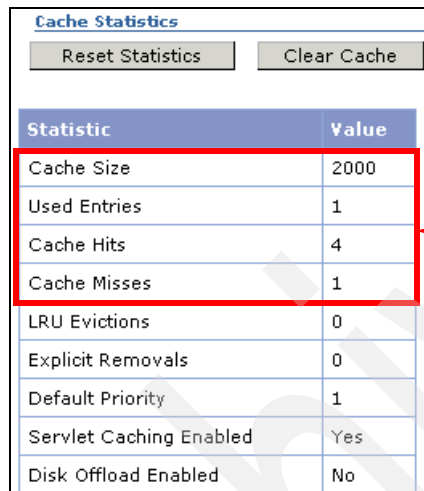
Component	ID	Type	Ignore Value	Method	Field	Required
Component 0	Hash	SOAPEnvelope	false	null	null	true

Figure 24-2 View the active cache policies with cache monitor

- Analyze the cache usage.

To monitor the cache usage, click *Cache Statistics* in the cache monitor.

Using a Web service client application, we requested the service `getForeCast` operation five times. The cache monitor statistic table shows one cache miss and four cache hits (Figure 24-3). The one cache miss is caused by the very first request; this request object is put in the cache and is represented by the “1” *Used Entries*.



Reset Statistics Clear Cache	
Statistic	Value
Cache Size	2000
Used Entries	1
Cache Hits	4
Cache Misses	1
LRU Evictions	0
Explicit Removals	0
Default Priority	1
Servlet Caching Enabled	Yes
Disk Offload Enabled	No

Monitor the cache size and the hit/miss ratio for tuning purposes.

Figure 24-3 Cache statistics for five service requests

Web services client cache

The Web service client cache is also part of the Application Server dynamic cache service and is used to increase the performance by caching responses from remote Web services.

After a response is returned from a remote Web service, the response is saved in the client cache on the application server. Any identical requests that are made to the same remote Web service are then responded to from the cache for a specified period of time.

The Web services client cache relies primarily on time-based invalidations, because the target Web service can be outside of your enterprise network and unaware of your client caching. Therefore, you can specify the amount of time in the cache and the rules to build cache entry IDs in the cache in your client application.

The Web services client cache is provided as a JAX-RPC handler on the application server. This JAX-RPC cache handler intercepts the SOAP requests that flow through it from application clients. It then identifies a cache policy based on the target Web service. After a policy is found, all the cache ID rules are evaluated one by one until a valid rule is detected.

To define and start a Web service cache on the client side for the weather forecast JavaBean Web service, perform these steps:

- ▶ Create a cachespec.xml file for the client weather forecast Web service.
Copy the cachespec.sample.xml and cachespec.dtd files from the directory <RAD60_HOME>\runtimes\base_v6\properties to the Web service client project /WeatherJavaBeanClientWeb/WebContent/WEB-INF.
Either use the import or the drag-and-drop function. Rename the sample file to cachespec.xml.
- ▶ Define the cache configuration.
For the weather forecast example, Example 24-2 shows the configuration cachespec.xml file.

Example 24-2 Weather forecast client cache configuration

```
<?xml version="1.0" ?>
<!DOCTYPE cache SYSTEM "cachespec.dtd">
<cache>
  <cache-entry>
    <class>JAXRPCClient</class>
    <name>
      http://<hostname>/WeatherBeanWeb/services/WeatherJavaBean
    </name>
    <sharing-policy>not-shared</sharing-policy>
    <cache-id>
      <component id="Hash" type="SOAPEnvelope" />
      <timeout>3600</timeout>
    </cache-id>
  </cache-entry>
</cache>
```

To define the cache configuration on the client side, the value for the class entry must be set to JAXRPCClient. In contrast to the server configuration, the Web service endpoint location must be inserted for the <name> tag.

The cache time out is set to 3600 seconds in the <timeout> tag. The cache service will discard all cache entries older than this time from the cache. The whole cache or just specific cache entries can be cleared using the cache monitor. The <component> tag specifies that cache entry IDs are generated as a hash of the complete SOAP envelope.

- Deploy and start the client sample application.
Add the client enterprise application (WeatherJavaBeanClient) to the test server and start the application. The cache service is now active and will be used for all further client Web service request on the client side.

Figure 24-4 shows the sample client environment:

- The thick, red lines show the request flow when the object is available in the cache.
- The dotted, blue lines show the flow for objects that are not in the cache. The cache service generates the cache ID and puts those objects into the cache.

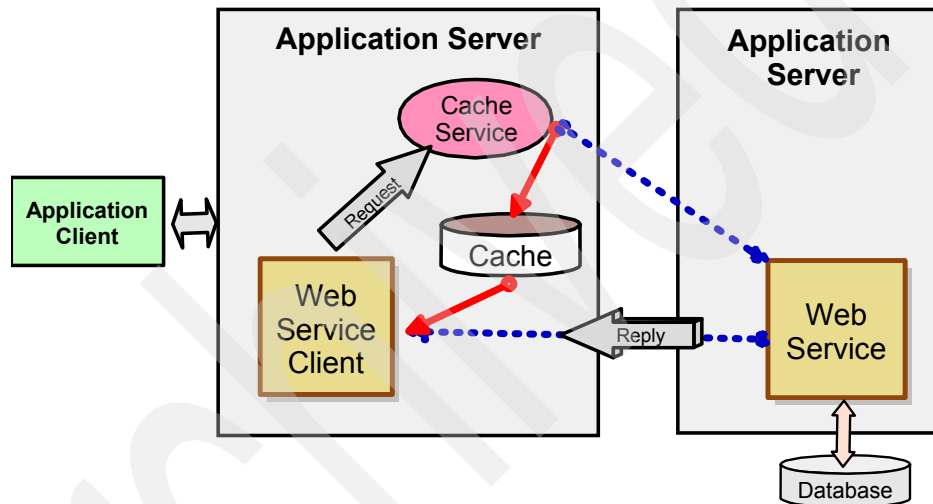


Figure 24-4 Client caching environment

To use the cache monitor on the client side, the cache monitor application must be installed (see “Cache monitor” on page 638).

Cache monitor usage and the data representation is the same as shown for “Web services server cache” on page 638.

Simple load test

To show the performance improvement, we run a simple load test. We created a servlet that uses the Web service client to get a weather forecast. The servlet calls the service `getForecast` operation with a specific date and requests the forecast for another 20 days. Example 24-3 shows an extract of the servlet code.

```
...
WeatherJavaBeanProxy proxy = new WeatherJavaBeanProxy();
proxy.setEndpoint("http://<hostname>/WeatherBeanWeb/services/WeatherJavaBean");
WeatherJavaBean bean = proxy.getWeatherJavaBean();

Weather[] weather = bean.getForecast(new GregorianCalendar(2004,9,17), 20);
for (int i = 0; i < weather.length; i++) {
    // Print out the Weather forecast conditions
    respOut.println("getForecast(): "+weather[i].getCondition()+"<br>");
}
...
```

We used two common desktop systems for the load test (Web service client and server). Because we only want to show the performance difference between *normal* Web service requests as compared to *cached* calls, this setup is sufficient.

We executed the following load tests:

- ▶ Test 1—Common scenario: 20 users, 5 seconds think time
- ▶ Test 2—High load scenario: 40 users, 3 seconds think time

For the test running with no caching, we ensured that the data source settings do not limit the database throughputs. The WEATHER database is the main limiting factor in this test scenario. But, this is also often true in real-world applications.

The test are executed with the following conditions:

- ▶ No caching—Normal Web service request
- ▶ Server caching—Web service server side caching
- ▶ Client caching—Web service client side caching

We executed the test runs for the different cache types three times to get reasonable measurements.

Test 1: 20 concurrent users and 5 seconds think time (Table 24-2)

Table 24-2 shows the results first test.

Table 24-2 Test 1—Web service load test results

Cache	Request per second	Mean response time (ms)
No cache	3.7	251
Server cache	3.9	49
Client cache	3.9	48

Test 2: 40 concurrent users and 3 seconds think time (Table 24-3)

For the load run without cache, this is the maximum load we could generate without having the test server bound to 100% CPU usage. Using caching, we could run test with a higher load until we reached the CPU limit. Running this test is only specific to our test system and would not provide further benefit for the cache to no cache comparison.

Table 24-3 Test 2—Web service load test results

Cache	Request per second	Mean response time (ms)
No cache	11.5	427
Server cache	12.9	62
Client cache	12.9	58

Test result analysis

Both test runs show a *high response time improvement by a factor of five to seven* for the cached Web service. The server-side and client-side caching response times are very similar. This is because the tests were executed in a lab environment. In real-world scenarios, where the client and the server are connected over different intranet segments—or even over the Internet—the response time improvement using client-side caching would be higher.

Because we used the same test load (number of users and think time) for the cached and uncached tests, the request per second results are almost the same. With caching enabled, we achieved almost the maximum possible throughput:

Test 1: 4 requests per second from 20 users with 5 second think time

Test 2: 13.3 requests per second from 40 users with 3 second think time

The resource allocations for the cached runs (server and client) are much lower (CPU, memory). Therefore, the number of requests that can be served with the same hardware are much higher. Stressing the cached Web services to the maximum number of request per second would only show our test environment specific maximum work load. Because this test is only meant to show the distinction between the different caching options, the maximum cache load test has not been done.

Summary

In this chapter, we showed how to use the WebSphere Application Server dynamic caching services for Web services. The performance and throughputs improvement has been shown exemplarily for the weather forecast application.

Based on the results of the load tests, we recommend that you spend the time to analyze Web service applications if caching can be applied. The potential performance increase and the system load decrease encourage the use of caching.

Important: The measurements presented in this chapter were obtained in a small and controlled test environment and are not representative of real-life applications. You will achieve the best throughput by monitoring the cache hit/miss information and by adjusting the cache entry size for maximum efficiency.

More information

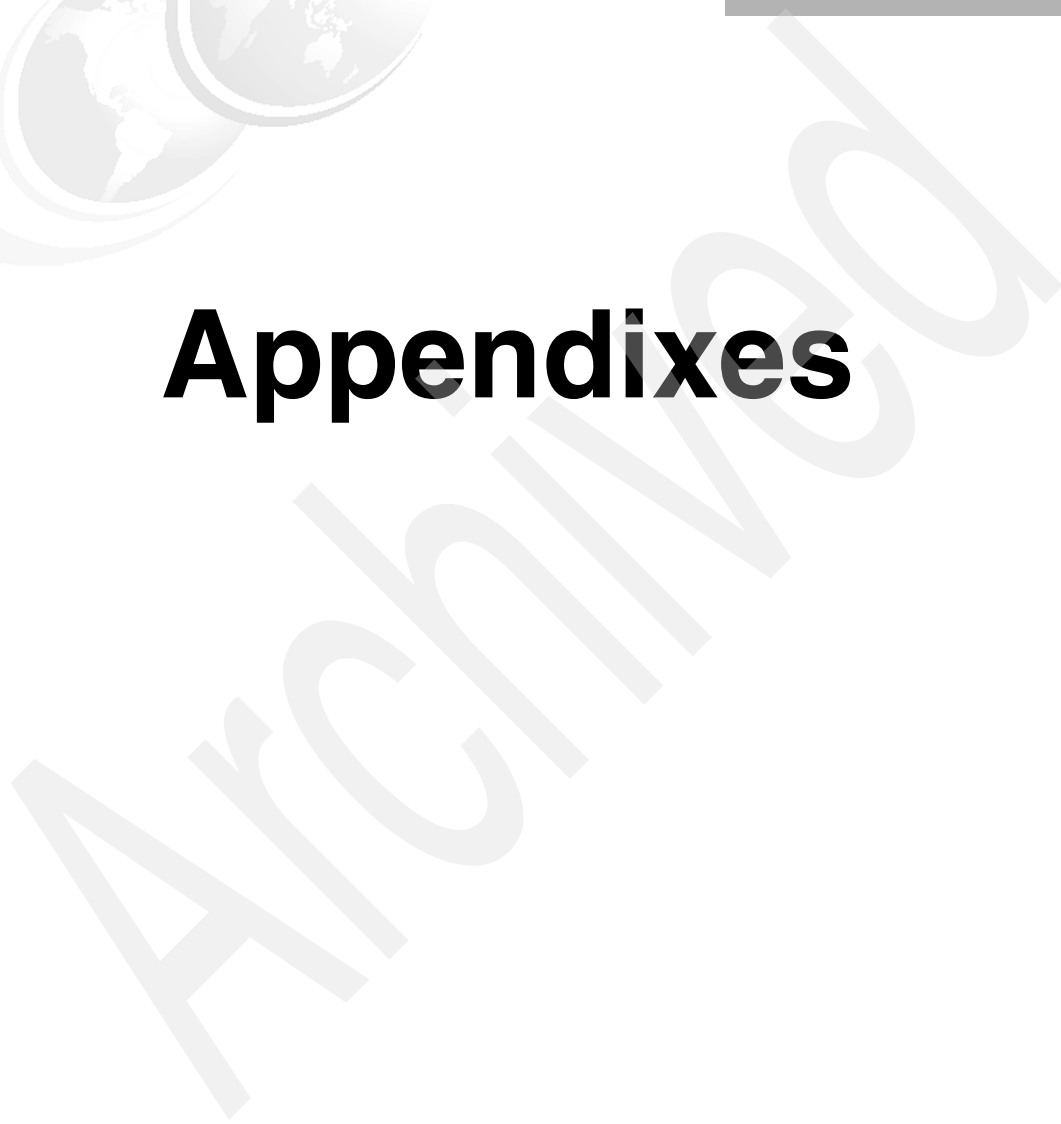
More information about the dynamic cache service can be found in the *WebSphere Application Server Information Center* and these other sources:

- ▶ *WebSphere Application Server Information Center*
<http://www.ibm.com/software/webservers/appserv/infocenter.html>
- ▶ The white paper, *WebSphere Dynamic Cache: Improving J2EE application performance*, available at:
<http://www.research.ibm.com/journal/sj/432/bakalova.html>
- ▶ The white paper, *Caching WebSphere Commerce pages with the WebSphere Application Server dynamic cache service*, available at:
http://www.ibm.com/developerworks/websphere/library/techarticles/0405_caching/0405_caching.html
- ▶ IBM Redbook *WebSphere Application Server V6: Scalability and Performance Handbook*, SG24-6392



Part 4

Appendixes



Installation and setup

This appendix provides brief instructions for installing the following products used in this publication:

- ▶ IBM WebSphere Application Server Version 6.0
- ▶ IBM Rational Application Developer Version 6.0
- ▶ IBM Rational Agent Controller

Following the installation instructions, we describe how to set up Rational Application Developer for the examples used in this document.

We also describe how to define multiple WebSphere Application Server profiles for use with Rational Application Developer.

Installation planning

This section provides installation planning information for the products used in this book.

WebSphere Application Server Version 6.0 requirements

IBM WebSphere Application Server Base and Network Deployment have the following hardware and software requirements. For updated information about the requirements, refer to the *WebSphere Application Server Information Center* and documentation:

<http://www.ibm.com/software/webservers/appserv/infocenter.html>

<http://www.ibm.com/software/webservers/appserv/doc/latest/prereq.html>

Hardware

Hardware requirements for Microsoft Windows servers include:

- ▶ Intel Pentium® processor at 500 MHz, or faster
- ▶ Minimum 990 MB free disk space for installation of Version 5.1
- ▶ Minimum 512 MB memory; 1 GB or more recommended
- ▶ CD-ROM drive

Software

The Microsoft Windows installation requires the following software to be installed:

- ▶ Microsoft Windows 2000 Server or Advanced Server SP4, Windows Server 2003, Windows XP Professional SP1a
- ▶ Mozilla 1.4 or 1.7, or Microsoft Internet Explorer 6.0 SP1

Database support

For the WebSphere Application Server installation, the database does not have to be configured. IBM Cloudscape can be used in a test environment, but other databases are required for the production environment.

Note: This publication includes sample code that uses a database. You can configure either Cloudscape or DB2 UDB Version 8.1 or Version 8.2.

Installing WebSphere Application Server V6.0

Note: An installation of WebSphere Application Server is not required to follow the development of the examples presented in this book. If you want to actually deploy the applications, you require WebSphere Application Server.

The installation of WebSphere Application Server V6.0 for a production environment is a complex task and is not covered in this document. We only installed a single copy of the base server to perform simple configuration and deployment tasks. Perform the following steps:

- ▶ Start the installation by running the `launchpad.bat` file. The Welcome page opens (Figure A-1).

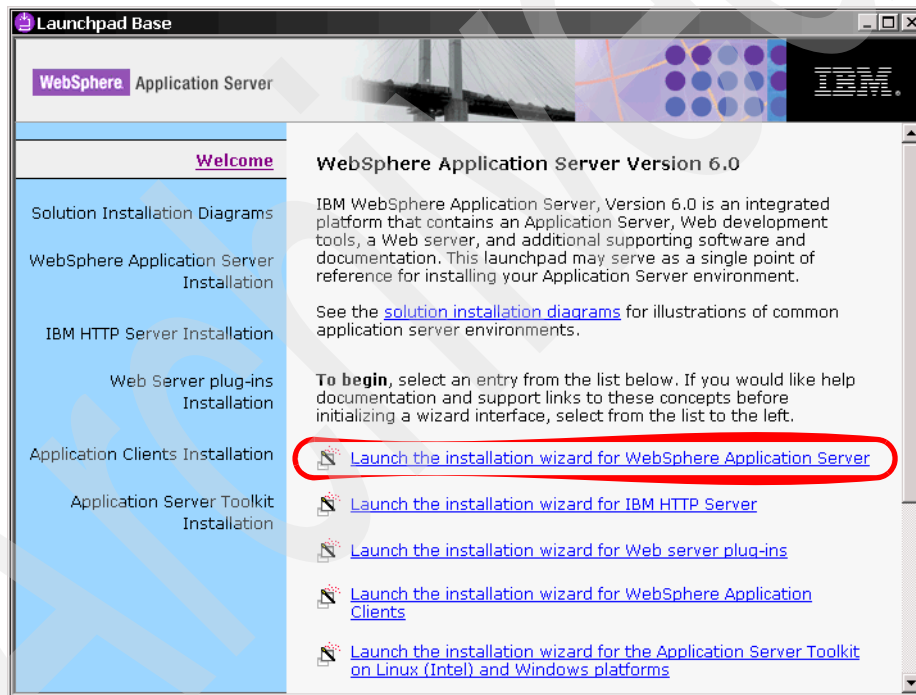


Figure A-1 WebSphere Application Server V6.0 installation launchpad

- ▶ Select *Launch the installation wizard for WebSphere Application Server*. The installation wizard opens.
- ▶ Read the Welcome panel and click *Next*.
- ▶ Accept the license agreement and click *Next*.

- ▶ The prerequisites are checked, and if successful, click *Next*.
- ▶ Select the installation directory, and then click *Next*:
 Default: C:\Program Files\IBM\WebSphere\AppServer
 Our choice: D:\WebSphere\AppServer60
- ▶ Select *Full installation* (includes samples and JavaDoc) and click *Next*.
- ▶ Click *Next* to install the base product. Be patient.
- ▶ When the installation is complete, you are prompted to *Launch the First steps console*. Click *Finish* and the First steps window opens (Figure A-2).



Figure A-2 WebSphere Application Server: First steps

- ▶ Select *Installation verification*. This starts the server and verifies that the installation is complete. Close the installation verification output log when finished.
- ▶ Select *Administrative console* to start the administrative console. Log in with your regular user ID. Expand *Servers*, and you should see the default server1. Select *Logout* in the top menu, and then close the browser.
- ▶ Select *Stop the server* and wait for the server to stop.
- ▶ Select *Exit*.

Installing IBM HTTP Server

In the launchpad, select *Launch the installation wizard for IBM HTTP Server*:

- ▶ Click *Next*.
- ▶ Accept the license agreement and click *Next*.
- ▶ Select an installation directory, for example, D:\WebSphere\IBMHTTPServer.
- ▶ Select *Typical* and click *Next*.
- ▶ Select *Run IBM HTTP Server as a Windows Service* and *Run IBM HTTP Administration as a Windows Service*. Enter an authorized user ID and password, and select the startup type (*Automatic* or *Manual*). Click *Next*.
- ▶ Click *Next* to start the installation.
- ▶ Click *Next* when the installation is finished.

You are prompted to install the *Launch the WebSphere Application Server - Plugin install*. Select the option and click *Finish*.

Installing the Web server plug-ins

The installation wizard for the plug-in starts:

- ▶ Optionally, select the documentation and click *Next*.
- ▶ Accept the license agreement and click *Next*.
- ▶ The prerequisites are checked, and if successful, click *Next*.
- ▶ Select *IBM HTTP Server V6* (to be configured) and click *Next*.
- ▶ Select *WebSphere Application Server machine (local)* and click *Next*.
- ▶ Select the installation directory, for example, D:\WebSphere\Plugins60.
- ▶ Confirm the installation location of WebSphere Application Server and click *Next*.
- ▶ Click *Browse* to locate the httpd.conf file, for example:
D:\WebSphere\IBMHTTPServer\conf\httpd.conf
- ▶ Leave the port as 80 and click *Next*.
- ▶ Enter a unique Web server name, for example, webserver1. Click *Next*.
- ▶ Accept the plug-in file name and click *Next*.
- ▶ Click *Next* to install the plug-in.
- ▶ Click *Next* when the installation is confirmed. Click *Finish*.

Close the launchpad.

Installing Rational Application Developer V6.0

The installation of Rational Application Developer is a very straightforward process. Perform the following steps:

- ▶ Double-click `launchpad.exe` and the installation launchpad window opens (Figure A-3).

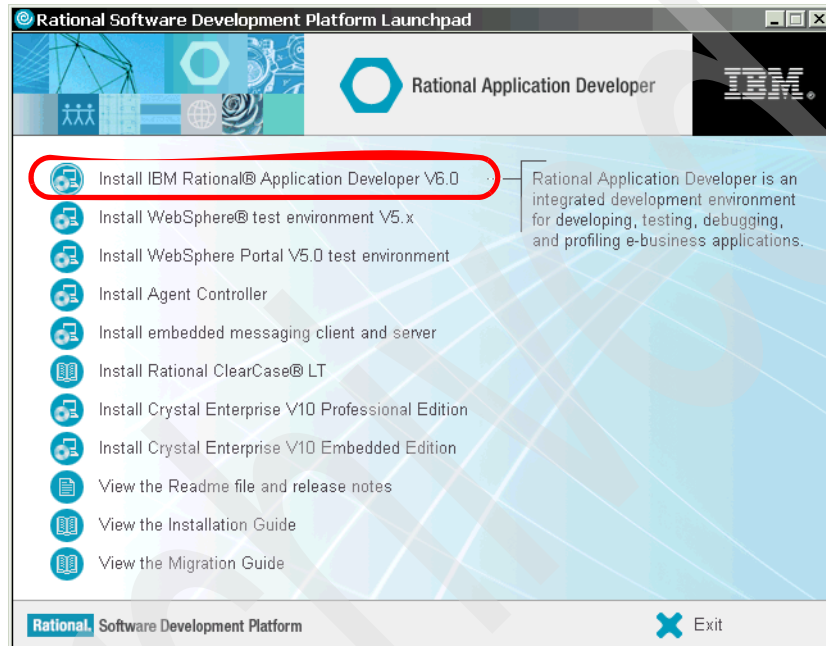


Figure A-3 Rational Application Developer V6.0 installation launchpad

The only component required for our examples is Rational Application Developer V6.0, which includes the WebSphere Application Server V6.0 test environment.

You can optionally install the WebSphere Application Server V5.x test environments, the Agent Controller (required for remote testing and deployment), and ClearCase LT (as a team repository). For a robust JMS messaging server, you can install the embedded messaging client and server.

- ▶ Select *Install IBM Rational Application Developer V6.0*. The installation program opens.
- ▶ Click *Next* on the Welcome panel.
- ▶ Accept the license agreement and click *Next*.

- ▶ Select the installation directory. The default is:
C:\Program Files\IBM\Rational\SDP\6.0
You can change this directory something simple as C:\RAD60. Click *Next*.
- ▶ Select the features to install (Figure A-4). Be sure to select the *IBM WebSphere Application Server V6.0 Integrated Test Environment*. No other features are required for our examples.



Figure A-4 Application Developer features

- ▶ Click *Next* and install the product. Be patient. After installing Application Developer, the WebSphere Application Server Version 6 test environment is installed.

Installing fixes

Investigate if there are interim fixes or new versions available for Application Developer. Refer to the *Download* section in:

<http://www.ibm.com/software/awdtools/developer/application/support/index.html>

Fixes are installed using the Rational Product Updater.

Installing the Agent Controller

The Agent Controller is only used for a few functions (for example, the component test) described in this book. However, it is a good idea to install the Agent Controller anyway.

To install the Agent Controller, perform these steps:

- ▶ Close Application Developer if it is running.
- ▶ Select *Install Agent Controller* in the launchpad (see Figure A-3 on page 654).
- ▶ Click *Next* on the Welcome panel.
- ▶ Click *Next* when you have confirmed that Application Developer is not running.
- ▶ Accept the license agreement.
- ▶ Accept or change the program location.
- ▶ Select all the features.
- ▶ Specify the Java runtime location, for example:
`C:\<RAD60_HOME>\runtimes\base_v6\java\jre\bin\java.exe`
- ▶ Specify the locations where the Version 5 servers have been installed. Skip this panel if you did not install the Version 5 servers.
- ▶ Specify hosts that can access the Hyades Data Collection Engine. You can select any computer, this computer, or a list. For a single workstation environment, select *This computer only*.
- ▶ Enable or disable security (enabling this forces authentication).
- ▶ Install the product.
- ▶ Click *Finish*.
- ▶ A service named *IBM Rational Agent Controller* is added to the services, and it is started automatically by default.

Setting up Rational Application Developer

In this section, we prepare Rational Application Developer for the examples used in this document.

Starting Application Developer

Start Application Developer from *Start* → *Programs*. You are prompted for a workspace location. Specify a location for the workspace, for example, as shown in Figure A-5. Click *OK*.

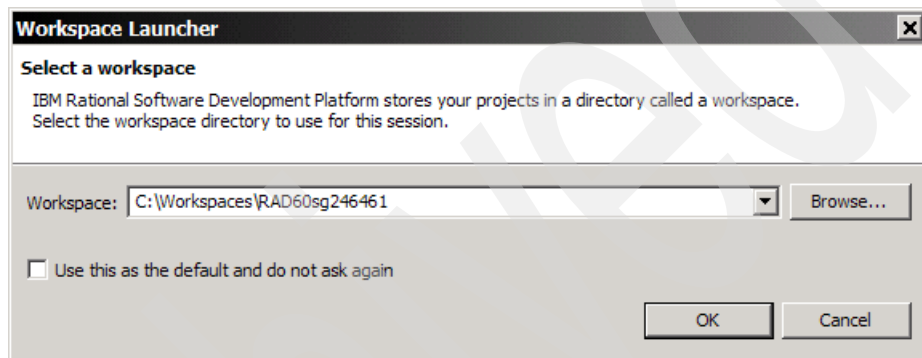


Figure A-5 Application Developer workspace

The Welcome panel opens (Figure A-6).



Figure A-6 Application Developer Welcome panel (compressed)

Close the Welcome view, and the J2EE perspective opens (Figure A-7).

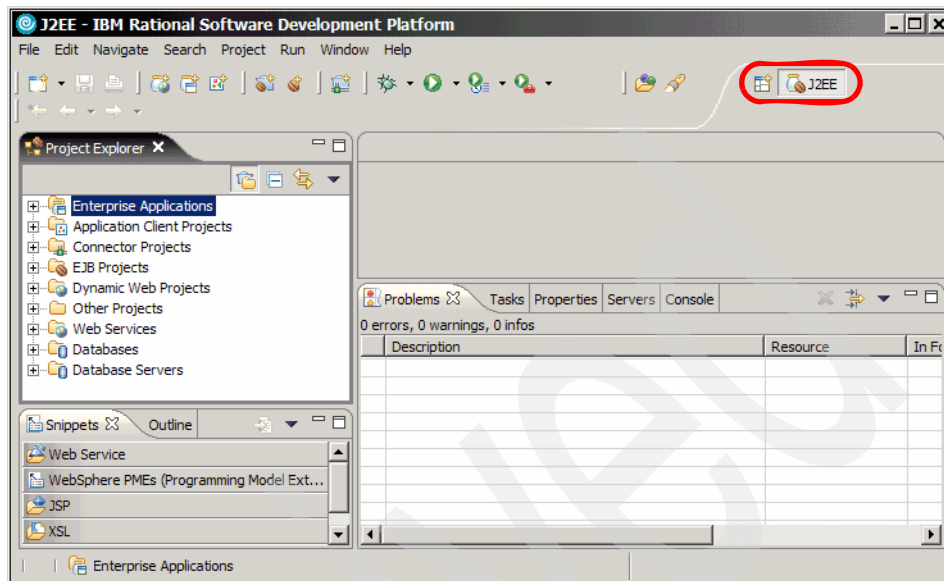


Figure A-7 Application Developer J2EE perspective

Open the Web perspective by clicking the  icon and selecting Web in the prompt (Figure A-8). We perform most of the work in the Web perspective.

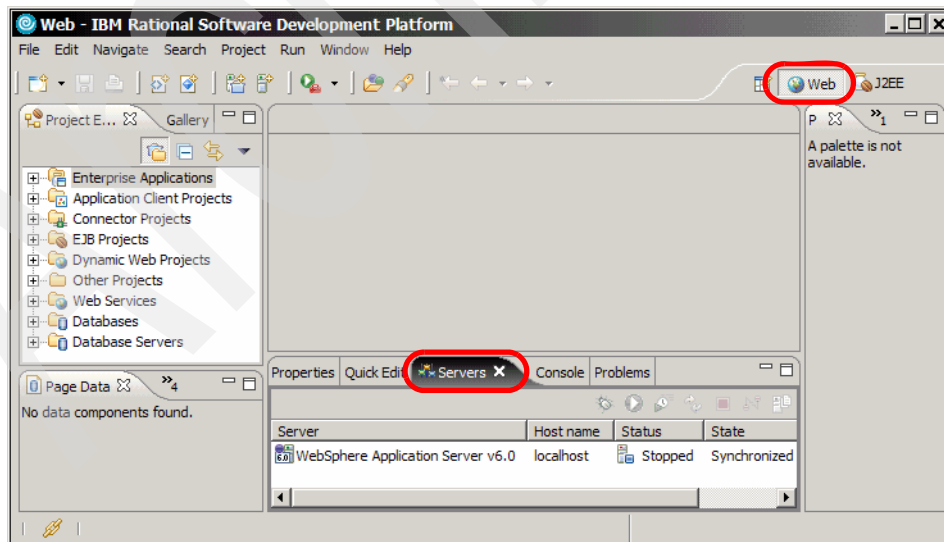


Figure A-8 Application Developer Web perspective

In the bottom pane, select the *Servers* view. After a while, the WebSphere Application Server v6.0 appears in the Stopped status.

Starting the WebSphere Application Server V6.0 test environment

Select *WebSphere Application Server v6.0* and click the start icon  or select *Start* from the context menu (right-click). Be patient, it takes a while to start the server. When the server is ready, the console view opens, and you can see the server's messages. Once started, you can keep the server running, because deploying applications to the server is dynamic.

Tip: You can also stop Application Developer without stopping the server. When you restart Application Developer, it will connect to the running server.

Configuring the server

Open (double-click) the WebSphere Application Server v6.0. You can only configure a few properties, such as:

- ▶ Hot method replace in debug mode
- ▶ Enabling the Universal Test Client
- ▶ Running the server with security enabled (for Web services security)
- ▶ Running the server with resources in the workspace (default) or server
- ▶ Enabling automatic publishing

Administrative console

All other server configurations must be performed using the administrative console. Select the server and *Run administrative console* (context). Log in with your usual user ID. You can explore the current settings.

For now, select *Logout* from the menu and close the administrative console view. We will configure the server when required for our sample applications.

Configuring Application Developer

Select *Window* → *Preferences*. Expand *Workbench* → *Capabilities*. On the right side expand *Web Service Developer* and select *Component Test for Web Services* and *Web Services Development*. Click *OK*.

Select *Window* → *Preferences*. Expand *Web Services* → *Code Generation* → *IBM WebSphere runtime*. Select *Do not overwrite loadable Java classes*. Click *OK*.

Refer to “Web services configuration settings” on page 241 for a description of these settings.

WebSphere Application Server profiles

WebSphere Application Server Version 6 provides the facility to run multiple servers by defining *profiles*. An initial profile, default, is created during the installation. A wizard is provided to create additional profiles. A profile consists of a set of commands (start, stop, and so forth), configuration files, log files, deployable applications, properties, and other information that defines a single application server environment. You can deploy an enterprise application into more than one profile.

Profiles can be defined for a real WebSphere Application Server and for the Rational Application Developer test environment:

- ▶ To create a profile for a real WebSphere Application Server, select *Start* → *Programs* → *IBM WebSphere Application Server V6.0* → *Profile creation wizard*. You can also invoke the wizard from the First steps menu (see Figure A-2 on page 652).
- ▶ To create a profile for the Application Developer test environment, run the profile creation program or use the First steps menu:

```
C:\<RAD60_HOME>\runtimes\base_v6\bin\ProfileCreator\pctWindows.exe  
C:\<RAD60_HOME>\runtimes\base_v6\profiles\default\firststeps\firststeps.bat
```

Multiple profiles for Application Developer

If you want to work with multiple Application Developer workspaces, it is best to define a profile for each workspace. Otherwise, you might have problems with the applications deployed from multiple workspaces to the same test server.

Creating a profile for the Application Developer test environment

The Profile Creation wizard guides you through a number of steps:

- ▶ Enter a unique name for the profile, for example, AppSrv01.
- ▶ Select a location; the default is <WAS60_HOME>\profiles\<profilename>, where <WAS60_HOME> is <RAD60_HOME>\runtimes\base_v6. You might want to use a dedicated disk area for your server configurations.
- ▶ Enter a node name; the default is <hostname>Node02. Enter a host name (use your host name or localhost or the IP address (if the address is static).
- ▶ Enter the port numbers. By default, all the ports are incremented by one for each server so that multiple servers can run in parallel. You can use the default ports for all servers, but you can only run one server at a time.
- ▶ Select whether you want to run the server as a Windows service using a system or user account.

Using the new server in Application Developer

To make a new server (profile) available in Application Developer, you have to define a new server:

- ▶ In the Servers view, select *New* → *Server* from the context menu.
- ▶ Select *WebSphere v6.0 Server* and click *Next*.
- ▶ Select the WebSphere profile that you created (AppSrv01).
- ▶ Enter the SOAP connector port of the profile that you created (this is the most important port that you entered in the Profile Creation wizard; Application Developer will get all the other information using this connection).
- ▶ Select *Run server with resources within the workspace*.
- ▶ Click *Finish* (Figure A-9).

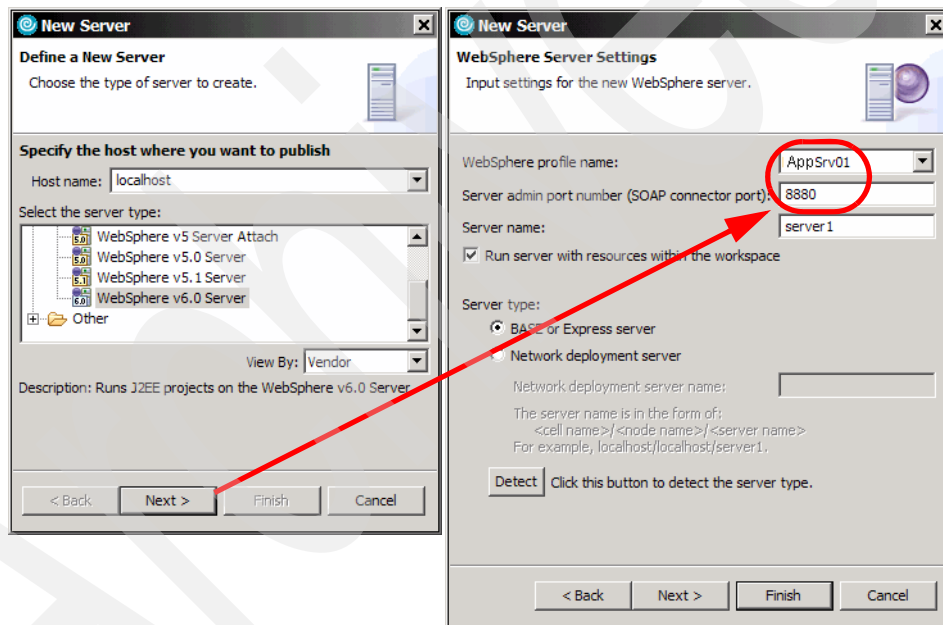


Figure A-9 Defining a server in Application Developer

You can now start the new server and deploy applications to the server.

Note: We used the default server for all applications for this book.

Using a remote server

If you have a separate test machine, you can install WebSphere Application Server V6 on that machine and also create multiple profiles (server instances). In addition, you have to install the Agent Controller on the remote machine.

To use a remote server instance, use the New Server window (Figure A-9 on page 661):

- ▶ Enter the fully qualified host name or the TCP/IP address in the Host name field and click *Next*.
- ▶ The WebSphere profile name pull-down list is disabled.
- ▶ Enter the SOAP connector port of the server instance, for example, 8881.
- ▶ Enter the Server name, for example, server1.
- ▶ *Run server with resources within the workspace* is disabled.
- ▶ Click *Finish*.

The remote server appears in the Servers view. Open the configuration of the new server and change the server name to something meaningful, for example, WebSphere V6 @ <hostname>-<profilename>.

You cannot start the remote server from Application Developer; you have to start the server on the remote machine. After the remote server is running, you can deploy enterprise applications (*Add and remove projects*), open the administrative console, and run applications. You can stop the remote server from Application Developer.

Installing the sample code

The sample code is available from the ITSO Redbooks Web site:

<http://www.redbooks.ibm.com>

Follow the link *Additional Materials* and look for the SG246461 directory matching this book. Download the sample code ZIP files and extract the files to your hard drive, creating the directory c:\SG246461.

In “How to use the Web material” on page 702, we describe the content of the sample code.

In “Installing the base weather forecast application” on page 705, we describe the instructions for installing and configuring the base code of the weather forecast application. This process includes:

- ▶ Importing the basic enterprise applications
- ▶ Setting up the WEATHER database in Cloudscape or DB2
- ▶ Selecting Cloudscape or DB2 for the enterprise applications
- ▶ Deploying the basic enterprise applications to the server
- ▶ Testing the basic enterprise applications

WS-Security configuration details: Mapping V5/V6, predefined properties, sample forms

In this appendix, we provide details about the WS-Security definition in the deployment descriptors.

There are three parts in this appendix:

- ▶ “Configuration mapping: Version 5.x to Version 6.0” on page 666—Guidelines for migration from a Version 5.x implementation to Version 6.0
- ▶ “List of predefined properties” on page 689—Properties of the security definitions
- ▶ “Forms for WS-Security configuration” on page 692—Sample forms for planning a security implementation

Configuration mapping: Version 5.x to Version 6.0

This section provides mappings between Version 5.x and Version 6.0 configurations. Refer to the tables in this section to migrate configurations.

Mapping for authentication

In this section, we show configuration mappings for two authentication configurations, basic authentication and authentication by X.509 certificate and identity assertion. For each configuration, we provide four tables of mappings: Deployment descriptors and binding configurations for the generator, and deployment descriptors and bindings for the consumer.

In the tables that follow, we show sample names of each element in *italics*. Additional explanations are shown in parenthesis.

Basic authentication

Table B-1 Basic authentication: Generator

V5.x extension	V6.0 extension
Login Config ▶ Authentication method: <i>BasicAuth</i> – Nonce – Nonce timestamp required	Security Token ▶ Name: <i>untoken</i> ▶ Token Type: <i>Username</i> – (Nonce and timestamp are configured in binding configuration.)
V5.x binding	V6.0 binding
Login Binding ▶ Authentication method: <i>BasicAuth</i> ▶ Callback handler: <i>NonPromptCallbackHandler</i> ▶ Basic authentication – User ID – Password	Token Generator ▶ Token generator class: <i>UsernameTokenGenerator</i> ▶ Security Token: <i>untoken</i> ▶ Callback handler: <i>NonPromptCallbackHandler</i> – User ID – Password ▶ Property – For Nonce • Name=<i>com.ibm.wsspi.wssecurity.token.userName.addNonce</i> • Value=<i>true</i> or <i>1</i> – For Timestamp • Name=<i>com.ibm.wsspi.wssecurity.token.userName.addTimestamp</i> • Value=<i>true</i> or <i>1</i>

Table B-2 Basic authentication: Consumer

V5.x extension	V6.0 extension
Login Config ▶ Authentication method: BasicAuth – Nonce settings – Nonce timestamp required	Required Security Token ▶ Name: <i>requntoken</i> ▶ Token Type: Username – (Nonce and timestamp are configured in binding configuration.)
	Caller Part ▶ Token type: Username
V5.x binding	V6.0 binding
Login Mapping ▶ Authentication method: BasicAuth ▶ Configuration name: WLogin ▶ Nonce settings – Maximum Age – Clock Skew ▶ Callback Handler Factory – <code>com.ibm.wsspi.wssecurity.auth.callback.WSCallbackHandlerFactoryImpl</code>	Token Consumer ▶ Token consumer class: <code>UsernameTokenConsumer</code> ▶ Security token: <i>requntoken</i> ▶ Use value type – Value type: Username Token ▶ Use jaas config – jaas.config name: <code>system.wssecurity.UsernameToken</code> ▶ Property – For verify Nonce • Name=<code>com.ibm.wsspi.wssecurity.token.username.verifyNonce</code> • Value=true or 1 – For verify Timestamp • Name=<code>com.ibm.wsspi.wssecurity.token.username.verifyTimestamp</code> • Value=true or 1

Authentication by X.509 certificate token

Table B-3 Authentication by X.509: Generator

V5.x extension	V6.0 extension
Login Config ▶ Authentication method: Signature	Security Token ▶ Name: <i>x509token</i> ▶ Token Type: X509 certificate token
V5.x binding	V6.0 binding
Login Binding ▶ Authentication method: Signature ▶ Callback handler: <code>NonPromptCallbackHandler</code>	Token Generator ▶ Token generator class: <code>X509TokenGenerator</code> ▶ Security Token: <i>x509token</i> ▶ Callback handler: <code>X509CallbackHandler</code> ▶ Use key store – (Specify key store for your X.509 certificate.) ▶ Key – (Specify key for the certificate.)

Table B-4 Authentication X.509: Consumer

V5.x extension	V6.0 extension
Login Config ▶ Authentication method: Signature	Required Security Token ▶ Name: <i>reqx509token</i> ▶ Token Type: X509 certificate token
	Caller Part ▶ Token type: X509 certificate token
V5.x binding	V6.0 binding
Login Mapping ▶ Authentication method: Signature ▶ Configuration name: <code>system.wssecurity.Signature</code> ▶ Callback Handler Factory – <code>com.ibm.wsspi.wssecurity.auth.callback.WSCallbackHandlerFactoryImpl</code>	Token Consumer ▶ Token consumer class: <code>X509TokenConsumer</code> ▶ Security Token: <i>reqx509token</i> ▶ Use value type – Value type: X509 certificate token ▶ Use jaas config – jaas.config name: <code>system.wssecurity.X509BST</code>

Authentication by LTPA token

Table B-5 Authentication by LTPA: Generator

V5.x extension	V6.0 extension
Login Config ▶ Authentication method: LTPA	Security Token ▶ Name: <i>ltpatoken</i> ▶ Token Type: LTPAToken
V5.x binding	V6.0 binding
Login Binding ▶ Authentication method: LTPA ▶ Callback handler: LTPATokenCallbackHandler	Token Generator ▶ Token generator class: LTPATokenGenerator ▶ Security Token: <i>ltpatoken</i> ▶ Callback handler: LTPATokenCallbackHandler

Table B-6 Authentication by LTPA: Consumer

V5.x extension	V6.0 extension
Login Config ▶ Authentication method: LTPA	Required Security Token ▶ Name: <i>reqltpatoken</i> ▶ Token Type: LTPAToken
	Caller Part ▶ Token type: LTPAToken
V5.x binding	V6.0 binding
Login Mapping ▶ Authentication method: LTPA ▶ Configuration name: WSLogin ▶ Callback Handler Factory – com.ibm.wsspi.wsssecurity.auth.callback.WSCallbackHandlerFactoryImpl	Token Consumer ▶ Token consumer class: LTPATokenConsumer ▶ Security token: <i>reqltpatoken</i> ▶ Use value type – Value type: LTPA Token

Identity assertion

Case1: ID Type=“Username” or “DN”, Trust Mode=(None)

Table B-7 Identity assertion: Case 1: Generator

V5.x extension	V6.0 extension
Login Config ▶ Authentication method: IDAssertion	Security Token ▶ Name: <i>ID_untoken</i> ▶ Token Type: Username
ID Assertion ▶ ID Type: Username or DN ▶ Trust Mode: (None)	
V5.x binding	V6.0 binding
Login Binding ▶ Authentication method: IDAssertion ▶ Callback handler: NonPromptCallbackHandler	Token Generator ▶ Token generator class: UsernameTokenGenerator ▶ Security token: <i>ID_untoken</i> ▶ Callback handler: NonPromptCallbackHandler ▶ User ID: (Specifies user ID of caller (client). Not necessary to specify Password.) ▶ Callback handler property: (Specifies properties in Table B-25 on page 689, “Binding: Token generator callback handler” on page 691 section.)

Table B-8 Identity assertion: Case 1: Consumer

V5.x extension	V6.0 extension
Login Config ▶ Authentication method: IDAssertion	Required Security Token ▶ Name: <i>requntoken</i> ▶ Token Type: Username
ID Assertion ▶ ID Type: Username or DN ▶ Trust Mode: (None)	Caller Part ▶ Token Type: Username ▶ Use IDAssertion – Trust method name: (None)
V5.x binding	V6.0 binding
Login Mapping ▶ Authentication method: IDAssertion ▶ Configuration name: “system.wssecurity.IDAssertion” ▶ Callback Handler Factory: com.ibm.wsspi.wssecurity.auth.callback.WSCallbackHandlerFactoryImpl	Token Consumer ▶ Token consumer class: UsernameTokenConsumer ▶ Security Token: <i>requntoken</i> ▶ Use value type – Value type: Username Token ▶ Use jaas.config – jaas.config name: system.wssecurity.IDAssertionUsernameToken

Case 2: ID Type="X509Certificate", Trust Mode=(None)

Table B-9 Identity assertion: Case 2: Generator

V5.x extension	V6.0 extension
Login Config ▶ Authentication method: IDAssertion	Security Token – Name: <i>ID_x509</i> – Token Type: X509 certificate token
ID Assertion ▶ ID Type: X509Certificate ▶ Trust Mode: (None)	
V5.x binding	V6.0 binding
Login Binding ▶ Authentication method: IDAssertion ▶ Callback handler: NonPromptCallbackHandler	Token Generator ▶ Token generator class: X509TokenGenerator ▶ Security token: <i>ID_x509</i> ▶ Callback handler: X509CallbackHandler ▶ Use key store: (Specify key store information for client's X.509 certificate.) ▶ Key: (Specify key information for client's X.509 certificate.)

Table B-10 Identity assertion: Case 2: Consumer

V5.x extension	V6.0 extension
Login Config ▶ Authentication method: IDAssertion	Required Security Token ▶ Name: <i>reqx509token</i> ▶ Token Type: X509 certificate token
ID Assertion ▶ ID Type: X509Certificate ▶ Trust Mode: (None)	Caller Part ▶ Token Type: X509 certificate token ▶ Use IDAssertion – Trust method name: (None)
V5.x binding	V6.0 binding
Login Mapping ▶ Authentication method: IDAssertion ▶ Configuration name: <i>system.wssecurity.IDAssertion</i> ▶ Callback Handler Factory: <i>com.ibm.wsspi.wssecurity.auth.callback.WSCallbackHandlerFactoryImpl</i>	Token Consumer ▶ Token consumer class: X509TokenConsumer ▶ Security Token: <i>reqx509token</i> ▶ Use value type – Value type: X509 certificate token ▶ Use jaas.config – jaas.config name: <i>system.wssecurity.X509BST</i>

Case 3: ID Type=“Username” or “DN”, Trust=“BasicAuth”

Table B-11 Identity assertion: Case 3: Generator

V5.x extension	V6.0 extension
Login Config ▶ Authentication method: IDAssertion	Security Token: Specify <i>two</i> security tokens. For ID Type ▶ Name: <i>ID_untoken</i> ▶ Token Type: Username For Trust Mode ▶ Name: <i>TM_untoken</i> ▶ Token Type: Username
V5.x binding	V6.0 binding
Login Binding ▶ Authentication method: IDAssertion ▶ Callback handler: NonPromptCallbackHandler ▶ Basic authentication – User ID: (Specifies User ID of intermediary.) – Password: (Specifies password of intermediary.)	Token Generator: Specify <i>two</i> Token Generators. For ID Type ▶ Token generator class: “UsernameTokenGenerator” ▶ Security token: <i>ID_untoken</i> ▶ Callback handler: NonPromptCallbackHandler ▶ User ID: (Specifies User ID of caller (client). Not necessary to specify Password.) ▶ Callback handler property: (Specify properties in Table B-25 on page 689, “Binding: Token generator callback handler” on page 691 section.) For Trust Mode ▶ Token generator class: UsernameTokenGenerator ▶ Security token: <i>TM_untoken</i> ▶ Callback handler: NonPromptCallbackHandler ▶ User ID: (Specify User ID of intermediary.) ▶ Password: (Specify password of intermediary.)

Table B-12 Identity assertion: Case 3: Consumer

V5.x extension	V6.0 extension
Login Config ▶ Authentication method: IDAssertion	Required Security Token: Specify <i>two</i> Required Security Tokens for both ID Type and Trust Mode. For ID Type ▶ Name: <i>ID_requntoken</i> ▶ Token Type: Username For Trust Mode ▶ Name: <i>TM_requntoken</i> ▶ Token Type: Username
ID Assertion ▶ ID Type: Username or DN ▶ Trust Mode: BasicAuth	Caller Part ▶ Token Type: Username ▶ Use IDAssertion – Trust method name: BasicAuth
V5.x binding	V6.0 binding
Login Mapping ▶ Authentication method: IDAssertion ▶ Configuration name: <code>system.wssecurity.IDAssertion</code> ▶ Callback Handler Factory: <code>com.ibm.wsspi.wssecurity.auth.callback.WSCallbackHandlerFactoryImpl</code>	Token Consumer: Specify <i>two</i> token consumers for both ID Type and Trust Mode. For ID Type ▶ Token consumer class: <code>UsernameTokenConsumer</code> ▶ Security Token: <i>ID_requntoken</i> ▶ Use value type – Value type: Username Token ▶ Use jaas.config – jaas.config name: <code>system.wssecurity.IDAssertionUsernameToken</code> For Trust Mode ▶ Token consumer class: <code>UsernameTokenConsumer</code> ▶ Security Token: <i>TM_requntoken</i> ▶ Use value type – Value type: Username Token ▶ Use jaas.config – jaas.config name: <code>system.wssecurity.UsernameToken</code>

Case 4: ID Type=“X509Certificate”, Trust Mode=“BasicAuth”

Table B-13 Identity assertion: Case 4: Generator

V5.x extension	V6.0 extension
Login Config ▶ Authentication method: IDAssertion	Security Token: Specify two security tokens for both ID Type and Trust Mode. For ID Type ▶ Name: <i>ID_x509</i> ▶ Token Type: X509 certificate token For Trust Mode ▶ Name: <i>TM_untoken</i> ▶ Token Type: Username
ID Assertion ▶ ID Type: X509Certificate ▶ Trust Mode: BasicAuth	
V5.x binding	V6.0 binding
Login Binding ▶ Authentication method: IDAssertion ▶ Callback handler: NonPromptCallbackHandler ▶ Basic authentication – User ID: (Specify User ID of intermediary.) – Password: (Specify password of intermediary.)	Token Generator: Specify two token generators for both ID Type. For ID Type ▶ Token generator class: X509TokenGenerator ▶ Security token: <i>ID_x509</i> ▶ Callback handler: X509CallbackHandler ▶ Use key store: (Specify key store information for client's X.509 certificate.) ▶ Key: (Specify key information for client's X.509 certificate.) For Trust Mode ▶ Token generator class: UsernameTokenGenerator ▶ Security token: <i>TM_untoken</i> ▶ Callback handler: NonPromptCallbackHandler ▶ User ID: (Specify User ID of intermediary.) ▶ Password: (Specify Password of intermediary.)

Table B-14 Identity assertion: Case 4: Consumer

V5.x extension	V6.0 extension
Login Config ▶ Authentication method: IDAssertion	Required Security Token: Specify <i>two</i> Required Security Tokens for both ID Type and Trust Mode. For ID Type ▶ Name: <i>ID_x509token</i> ▶ Token Type: X509 certificate token For Trust Mode ▶ Name: <i>TM_requntoken</i> ▶ Token Type: Username
ID Assertion ▶ ID Type: X509Certificate ▶ Trust Mode: BasicAuth	Caller Part ▶ Token Type: X509 certificate token ▶ Use IDAssertion – Trust method name: BasicAuth
V5.x binding	V6.0 binding
Login Mapping ▶ Authentication method: IDAssertion ▶ Configuration name: <code>system.wssecurity.IDAssertion</code> ▶ Callback Handler Factory: <code>com.ibm.wsspi.wssecurity.auth.callback.WSCallbackHandlerFactoryImpl</code>	Token Consumer: Specify two token consumers for both ID Type and Trust Mode. For ID Type ▶ Token consumer class: X509TokenConsumer ▶ Security Token: <i>ID_x509token</i> ▶ Use value type – Value type: X509 certificate token ▶ Use jaas.config – jaas.config name: <code>system.wssecurity.X509BST</code> For Trust Mode ▶ Token consumer class: UsernameTokenConsumer ▶ Security Token: <i>TM_requntoken</i> ▶ Use value type – Value type: Username Token ▶ Use jaas.config – jaas.config name: <code>system.wssecurity.UsernameToken</code>

Case 5: ID Type=“Username” or “DN”, Trust Mode=“Signature”

Table B-15 Identity assertion: Case 5: Generator

V5.x extension	V6.0 extension
Login Config ▶ Authentication method: IDAssertion	Security Token: Specify for ID Type ▶ Name: <i>ID_untoken</i> ▶ Token Type: Username
ID Assertion ▶ ID Type: Username or DN ▶ Trust Mode: Signature	Integrity: Specify for Trust Mode ▶ Integrity Name: <i>int_token</i> ▶ Message parts dialect: http://www.ibm.com/websphere/webservices/wssecurity/dialect-was ▶ Message parts keyword: securitytoken
V5.x binding	V6.0 binding
Login Binding ▶ Authentication method: IDAssertion ▶ Callback handler: NonPromptCallbackHandler	Token Generator: Specify <i>two</i> token generators for both ID Type and Trust Mode. For ID Type ▶ Token generator class: UsernameTokenGenerator ▶ Security token: <i>ID_untoken</i> ▶ Callback handler: NonPromptCallbackHandler ▶ User ID: (Specify User ID of caller (client). Not necessary to specify Password.) ▶ Callback handler property: (Specify properties in Table B-25 on page 689, “Binding: Token generator callback handler” on page 691 section.) For Trust Mode ▶ Token generator name: <i>TM_tgenerator</i> ▶ Token generator class: X509TokenGenerator ▶ Callback handler: X509CallbackHandler ▶ Use key store: (Specify key store information for intermediary’s X.509 certificate.) ▶ Key: (Specify key information for intermediary’s X.509 certificate.)

Table B-16 Identity assertion: Case 5: Consumer

V5.x extension	V6.0 extension
Login Config ▶ Authentication method: IDAssertion	Required Security Token: Specify for ID Type ▶ Name: <i>ID_reqtoken</i> ▶ Token Type: Username Integrity: Specify for Trust Mode ▶ Integrity Name: <i>reqint_token</i> ▶ Message parts dialect: http://www.ibm.com/websphere/webservices/wssecurity/dialect-was ▶ Message parts keyword: securitytoken
ID Assertion ▶ ID Type: Username or DN ▶ Trust Mode: Signature	Caller Part ▶ Token Type: Username ▶ Use IDAssertion – Trust method name: Signature – Integrity or Confidentiality part: <i>reqint_token</i>
V5.x binding	V6.0 binding
Login Mapping ▶ Authentication method: IDAssertion ▶ Configuration name: <code>system.wssecurity.IDAssertion</code> ▶ Callback Handler Factory: <code>com.ibm.wsspi.wssecurity.auth.callback.WSCallbackHandlerFactoryImpl</code>	Token Consumer: Specify <i>two</i> token consumers for both ID Type and Trust Mode. For ID Type ▶ Token consumer class: <code>UsernameTokenConsumer</code> ▶ Security Token: <i>ID_reqtoken</i> ▶ Use value type – Value type: Username ▶ Use jaas.config – jaas.config name: <code>system.wssecurity.IDAssertionUsernameToken</code> For Trust Mode ▶ Token consumer name: <i>TM_tconsumer</i> ▶ Token consumer class: <code>X509TokenConsumer</code> ▶ Use value type – Value type: X509 certificate token ▶ Use jaas.config – jaas.config name: <code>system.wssecurity.X509BST</code> ▶ Use trusted ID evaluator – TrustedID evaluator class: <code>com.ibm.wsspi.wssecurity.id.TrustedIDEvaluatorImpl</code> – Trusted ID evaluator property: (Specify trusted IDs. For example, Name=<code>trustedId_1</code> Value=(TrustedID's certificate Distinguished Name).)

V5.x binding	V6.0 binding
	Key Locators ▶ Key locator name: <i>TM_klocator</i> ▶ Key locator class: X509TokenKeyLocator
	Key information ▶ Key information name: <i>TM_keyinfo</i> ▶ Key information type: STRREF ▶ Use key locator – Key locator: <i>TM_klocator</i> ▶ Use token: <i>TM_tconsumer</i>
	Signing Information ▶ Canonicalization method algorithm: http://www.w3.org/2001/10/xml-exc-c14n# ▶ Signature method algorithm: http://www.w3.org/2000/09/xmldsig#rsa-sha1 ▶ Key locator information element: <i>TM_kinfo</i> ▶ Part Reference dialog – RequiredIntegrity part: <i>reqint_token</i> – Digest method algorithm: http://www.w3.org/2000/09/xmldsig#sha1 ▶ Transform dialog – Algorithm: http://www.w3.org/2001/10/xml-exc-c14n#

Case 6: ID Type=“X509Certificate”, Trust Mode=“Signature”

Table B-17 Identity assertion: Case 6: Generator

V5.x extension	V6.0 extension
Login Config ▶ Authentication method: IDAssertion	Security Token: Specify for ID Type ▶ Name: <i>ID_x509</i> ▶ Token Type: X509 certificate token
ID Assertion ▶ ID Type: X509Certificate ▶ Trust Mode: Signature	Integrity: Specify for Trust Mode ▶ Integrity Name: <i>int_token</i> ▶ Message parts dialect: http://www.ibm.com/websphere/webservices/wssecurity/dialect-was ▶ Message parts keyword: securitytoken
V5.x binding	V6.0 binding
Login Binding ▶ Authentication method: IDAssertion ▶ Callback handler: NonPromptCallbackHandler	Token Generator: Specify <i>two</i> token generators for both ID Type and Trust Mode. For ID Type ▶ Token generator class: X509TokenGenerator ▶ Security token: <i>ID_x509</i> ▶ Callback handler: X509CallbackHandler ▶ Use key store: (Specify key store information for client's X.509 certificate.) ▶ Key: (Specify key information for client's X.509 certificate.) For Trust Mode ▶ Token generator name: <i>TM_tgenerator</i> ▶ Token generator class: X509TokenGenerator ▶ Callback handler: X509CallbackHandler ▶ Use key store: (Specify key store information for intermediary's X.509 certificate.) ▶ Key: (Specify key information for intermediary's X.509 certificate.) Key Locators ▶ Key locator name: <i>TM_klocator</i> ▶ Key locator class: KeyStoreKeyLocator ▶ Use key store: (Specify key store information for intermediary's X.509 certificate.) ▶ Key: (Specify key information for intermediary's X.509 certificate.)

V5.x binding	V6.0 binding
	Key information ▶ Key information name: <i>TM_keyinfo</i> ▶ Key information type: STRREF ▶ Use key locator – Key locator: <i>TM_klocator</i> – Key name: (Specify key name for intermediary's X.509 certificate.) ▶ Use token: <i>TM_tgenerator</i>
	Signing Information ▶ Canonicalization method algorithm: http://www.w3.org/2001/10/xml-exc-c14n# ▶ Signature method algorithm: http://www.w3.org/2000/09/xmldsig#rsa-sha1 ▶ Key information element: <i>TM_kinfo</i> ▶ Part Reference dialog – Integrity part: <i>int_token</i> – Digest method algorithm: http://www.w3.org/2000/09/xmldsig#sha1 ▶ Transform dialog – Algorithm: http://www.w3.org/2001/10/xml-exc-c14n#

Table B-18 Identity assertion: Case 6: Consumer

V5.x extension	V6.0 extension
Login Config ▶ Authentication method: IDAssertion	Required Security Token ▶ Name: <i>ID_reqx509token</i> ▶ Token Type: X509 certificate token Integrity: Specify for Trust Mode ▶ Integrity Name: <i>reqint_token</i> ▶ Message parts dialect: http://www.ibm.com/websphere/webservices/wssecurity/dialect-was ▶ Message parts keyword: "securitytoken"
ID Assertion ▶ ID Type: X509Certificate ▶ Trust Mode: Signature	Caller Part ▶ Token Type: X509 certificate token ▶ Use IDAssertion – Trust method name: Signature – Integrity or Confidentiality part: <i>reqint_token</i>
V5.x binding	V6.0 binding
Login Mapping ▶ Authentication method: IDAssertion ▶ Configuration name: <code>system.wssecurity.IDAssertion</code> ▶ Callback Handler Factory: <code>com.ibm.wsspi.wssecurity.auth.callback.WSCallbackHandlerFactoryImpl</code>	Token Consumer: Specify two token consumers for both ID Type and Trust Mode. For ID Type ▶ Token consumer class: <code>X509TokenConsumer</code> ▶ Security Token: <i>ID_reqx509token</i> ▶ Use value type – Value type: X509 certificate token ▶ Use <code>jaas.config</code> – <code>jaas.config</code> name: <code>system.wssecurity.X509BST</code> For Trust Mode ▶ Token consumer name: <i>TM_tconsumer</i> ▶ Token consumer class: <code>X509TokenConsumer</code> ▶ Use value type – Value type: X509 certificate token ▶ Use <code>jaas.config</code> – <code>jaas.config</code> name: <code>system.wssecurity.X509BST</code> ▶ Use trusted ID evaluator – TrustedID evaluator class: <code>com.ibm.wsspi.wssecurity.id.TrustedIDEvaluatorImpl</code> – Trusted ID evaluator property: (Specify trusted IDs. For example, Name=<code>trustedId_1</code> Value=(TrustedID's certificate Distinguished Name).)

V5.x binding	V6.0 binding
	Key Locators ▶ Key locator name: <i>TM_klocator</i> (example) ▶ Key locator class: <i>X509TokenKeyLocator</i>
	Key information ▶ Key information name: <i>TM_keyinfo</i> ▶ Key information type: <i>STRREF</i> ▶ Use key locator – Key locator: <i>TM_klocator</i> ▶ Use token: <i>TM_tconsumer</i>
	Signing Information ▶ Canonicalization method algorithm: <i>http://www.w3.org/2001/10/xml-exc-c14n#</i> ▶ Signature method algorithm: <i>http://www.w3.org/2000/09/xmldsig#rsa-sha1</i> ▶ Key locator information element: <i>TM_kinfo</i> ▶ Part Reference dialog – RequiredIntegrity part: <i>reqint_token</i> – Digest method algorithm: <i>http://www.w3.org/2000/09/xmldsig#sha1</i> ▶ Transform dialog – Algorithm: <i>http://www.w3.org/2001/10/xml-exc-c14n#</i>

Mapping for integrity

Table B-19 Integrity: Generator

V5.x extension	V6.0 extension
Integrity ▶ Reference part: body or timestamp or securitytoken	Integrity ▶ Integrity name: <i>intpart</i> ▶ Message parts dialect: http://www.ibm.com/websphere/webservices/wssecurity/dialect-was ▶ Message parts keyword: body or timestamp or securitytoken
V5.x binding	V6.0 binding
	Token Generator ▶ Token generator name: <i>tgenerator</i> ▶ Token generator class: X509TokenGenerator ▶ Callback handler: X509CallbackHandler ▶ Use key store: (Specify key store information for client's X.509 certificate.) ▶ Key: (Specify key information for client's X.509 certificate.)
Key Locators ▶ Key locator name: <i>klocator5</i> ▶ Key locator class: <code>com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator</code> ▶ Use key store: (Specify key store information for your X.509 certificate.) ▶ Key: (Specify key information for your X.509 certificate.)	Key Locators ▶ Key locator name: <i>klocator</i> ▶ Key locator class: <code>com.ibm.wsspi.wssecurity.keyinfo.KeyStoreKeyLocator</code> ▶ (Version 6.0 KeyLocator interface is different from Version 5.x. Need to specify an appropriate KeyLocator implementation.) ▶ Use key store: (Specify key store information for your X.509 certificate.) ▶ Key: (Specify key information for your X.509 certificate.)
	Key information ▶ Key information name: <i>kinfo</i> ▶ Key information type: STRREF ▶ Use key locator ▶ Key locator: <i>klocator</i> ▶ Key name: (Specify key name for client's X.509 certificate.) ▶ Use token: <i>tgenerator</i>

V5.x binding	V6.0 binding
Signing Information ▶ Canonicalization method algorithm: http://www.w3.org/2001/10/xml-exc-c14n# ▶ Digest method algorithm: http://www.w3.org/2000/09/xmldsig#sha1 ▶ Signature method algorithm: http://www.w3.org/2000/09/xmldsig#rsa-sha1 ▶ Signing key locator: <i>klocator5</i>	Signing Information: Specify same algorithms as Version 5.x ▶ Canonicalization method algorithm: http://www.w3.org/2001/10/xml-exc-c14n# ▶ Signature method algorithm: http://www.w3.org/2000/09/xmldsig#rsa-sha1 ▶ Key information element: <i>keyinfo</i> ▶ Part Reference dialog – Integrity part: <i>intpart</i> – Digest method algorithm: http://www.w3.org/2000/09/xmldsig#sha1 ▶ Transform dialog – Algorithm: http://www.w3.org/2001/10/xml-exc-c14n#

Table B-20 Integrity: Consumer

V5.x extension	V6.0 extension
Required Integrity ▶ Reference part: body or timestamp or securitytoken	Required Integrity ▶ Integrity name: <i>reqintpart</i> ▶ Message parts dialect: http://www.ibm.com/websphere/webservices/wssecurity/dialect-was ▶ Message parts keyword: body or timestamp or securitytoken
V5.x binding	V6.0 binding
Trust Anchor ▶ Trust anchor name: <i>tanchor5</i>	Trust Anchor: Specify same trust anchors as Version 5.x ▶ Trust anchor name: <i>tanchor</i>
Certificate Store List ▶ Collection Certificate Store – Name: <i>certstore5</i>	Certificate Store List ▶ Collection Certificate Store: – Name: <i>certstore</i> – (Specify same X509 Certificate as Version 5.x.)

	Token Consumer ▶ Token consumer name: <i>tconsumer</i> ▶ Token consumer class: X509TokenConsumer ▶ Use value type – Value type: X509 certificate token ▶ Use jaas.config – jaas.config name: <code>system.wssecurity.X509BST</code> ▶ Use certificate path settings – Certificate path reference • Trust anchor reference: <i>tanchor</i> • Certificate store reference: <i>certstore</i> – Trust any certificate: (Specify if no Trust anchor and Certificate store specified.)
	Key Locators ▶ Key locator name: <i>klocator</i> ▶ Key locator class: <code>com.ibm.wsspi.wssecurity.keyinfo.X509TokenKeyLocator</code> – (Version 6.0 KeyLocator interface is different from Version 5.x. Need to specify an appropriate KeyLocator implementation.)
	Key information ▶ Key information name: <i>kinfo</i> ▶ Key information type: STRREF ▶ Use key locator – Key locator: <i>klocator</i> ▶ Use token: <i>tconsumer</i>
Signing Information ▶ Canonicalization method algorithm: <code>http://www.w3.org/2001/10/xml-exc-c14n#</code> ▶ Digest method algorithm: <code>http://www.w3.org/2000/09/xmldsig#sha1</code> ▶ Signature method algorithm: <code>http://www.w3.org/2000/09/xmldsig#rsa-sha1</code> ▶ Use certificate path reference – Trust anchor reference: <i>tanchor5</i> – Certificate store reference: <i>certstore5</i> ▶ Trust any certificate: (Specify if no trust anchors and certificate store references specified.)	Signing Information: Specify same algorithms as Version 5.x ▶ Canonicalization method algorithm: <code>http://www.w3.org/2001/10/xml-exc-c14n#</code> ▶ Signature method algorithm: <code>http://www.w3.org/2000/09/xmldsig#rsa-sha1</code> ▶ Key information element: <i>"keyinfo"</i> ▶ Part Reference dialog – RequiredIntegrity part: <i>reqintpart</i> – Digest method algorithm: <code>http://www.w3.org/2000/09/xmldsig#sha1</code> ▶ Transform dialog – Algorithm: <code>http://www.w3.org/2001/10/xml-exc-c14n#</code>

Mapping for confidentiality

Table B-21 Confidentiality: Generator

V5.x extension	V6.0 extension
Confidentiality ► Confidential Part: bodycontent or usernametoken	Confidentiality ► Confidentiality Name: <i>confpart</i> ► Message parts dialect: http://www.ibm.com/websphere/webservices/wssecurity/dialect-was ► Message parts keyword: bodycontent or usernametoken
V5.x binding	V6.0 binding
Key Locators ► Key locator name: <i>klocator5</i> ► Key locator class: <code>com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator</code> ► Use key store: (Specify key store information for your X.509 certificate.) ► Key: (Specify key information for your X.509 certificate.)	Key Locators ► Key locator name: <i>klocator</i> ► Key locator class: <code>com.ibm.wsspi.wssecurity.keyinfo.KeyStoreKeyLocator</code> – (Version 6.0 KeyLocator interface is different from Version 5.x. Need to specify an appropriate KeyLocator implementation.) ► Use key store: (Specify key store information for client's X.509 certificate.) ► Key: (Specify key information for client's X.509 certificate.)
	Key information ► Key information name: <i>kinfo</i> ► Key information type: KEYID ► Use key locator – Key locator: <i>klocator</i> – Key name: (Specify key name for client's X.509 certificate.)
Encryption Information ► Data encryption method algorithm: http://www.w3.org/2001/04/xmlenc#tripledes-cbc ► Key encryption method algorithm: http://www.w3.org/2001/04/xmlenc#rsa-1_5 ► Encryption key name: (Specify key used encryption.) ► Signing key locator: <i>klocator5</i>	Encryption Information: Specify same algorithms as Version 5.x ► Data encryption method algorithm: http://www.w3.org/2001/04/xmlenc#tripledes-cbc ► Key encryption method algorithm: http://www.w3.org/2001/04/xmlenc#rsa-1_5 ► Key information element: <i>keyinfo</i> ► Confidentiality part: <i>confpart</i>

Table B-22 Confidentiality: Consumer

V5.x extension	V6.0 extension
Required Confidentiality ► Confidential part: <code>bodycontent</code> or <code>usnametoken</code>	Required Confidentiality ► Integrity name: <i>reqconfpart</i> ► Message parts dialect: <code>http://www.ibm.com/websphere/webservices/wssecurity/dialect-was</code> ► Message parts keyword: <code>bodycontent</code> or <code>usnametoken</code>
V5.x binding	V6.0 binding
Key Locators ► Key locator name: <i>klocator5</i> ► Key locator class: <code>com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator</code> ► Use key store: (Specify key store information for client's decryption key.) ► Key: (Specify key information for client's decryption key.)	Key Locators ► Key locator name: <i>klocator</i> ► Key locator class: <code>com.ibm.wsspi.wssecurity.keyinfo.KeyStoreKeyLocator</code> – (Version 6.0 KeyLocator interface is different from Version 5.x. Need to specify an appropriate KeyLocator implementation.) ► Use key store: (Specify key store information for client's decryption key.) ► Key: (Specify key information for client's decryption key.)
	Key information ► Key information name: <i>kinfo</i> ► Key information type: KEYID ► Use key locator – Key locator: <i>klocator</i>
Encryption Information ► Data encryption method algorithm: <code>http://www.w3.org/2001/04/xmlenc#tripleDES-cbc</code> ► Key encryption method algorithm: <code>http://www.w3.org/2001/04/xmlenc#rsa-1_5</code> ► Encryption key name: (Specify key used encryption.) ► Encryption key locator: <i>klocator5</i>	Encryption Information: Specify same algorithms as Version 5.x ► Data encryption method algorithm: <code>http://www.w3.org/2001/04/xmlenc#tripleDES-cbc</code> ► Key encryption method algorithm: <code>http://www.w3.org/2001/04/xmlenc#rsa-1_5</code> ► Key information element: <i>keyinfo</i> ► RequiredConfidentiality part: <i>reqconfpart</i>

Mapping for timestamp

In WebSphere Application Server Version 5.x, a timestamp element can be inserted into both the request message and the response message. The timestamp element of the request message has the created date, and that of the response message has the received date.

WebSphere Application Server Version 6.0 supports only a request message timestamp with a created date because the specification is updated. Table B-23 and Table B-24 show the mapping of Version 5.x to Version 6.0. There is no configuration for receiving a request timestamp in Version 5.x, but as for Version 6.0, it is necessary to configure timestamp at the consumer side to receive a request timestamp.

Table B-23 *Timestamp: Generator*

V5.x extension	V6.0 extension
Add Created Timestamp ► Add Created Timestamp – Expires	Add Timestamp ► Use Add Timestamp – Use Expires under Expires: (specify same expires date as Version 5.x.)

Table B-24 *Timestamp: Consumer*

V5.x extension	V6.0 extension
	Add Timestamp ► Use Add Timestamp

List of predefined properties

Here is a list of predefined properties that can be specified in using a GUI editor of WS-Security configuration in Rational Application Developer.

Property list of deployment descriptor extension

Note: All properties are in the `com.ibm.wsspi.wssecurity` package, for example:

`com.ibm.wsspi.wssecurity.timestamp.SOAPHeaderElement`

Table B-25 Predefined properties of WS-Security extensions and bindings

Name	Value	Usage
Extension: Request and response generator properties: AddTimestamp		
timestamp. SOAPHeaderElement	1 or true	To set the mustUnderstand flag
timestamp. dialect	(Same as message parts dialect)	To indicate insertion point
timestamp. keyword	SecurityFirst, SecurityLast, SOAPHeaderFirst, SOAPHeaderLast as the Application Server proprietary keyword, or specify a parent of the element to be inserted using XPath or WS-Policy function	To indicate insertion point
Extensions: Request and response consumer properties: AddTimestamp		
timestamp. dialect	(Same as message parts dialect)	To indicate insertion point
timestamp. keyword	SecurityFirst, SecurityLast, SOAPHeaderFirst, SOAPHeaderLast as the Application Server proprietary keyword, or specify a parent of the element to be inserted using XPath or WS-Policy function	To indicate insertion point
Binding: Generator and consumer: Transform		
dsig. XPathExpression	XPath expression: for example: not(ancestor-or-self::*[namespace-uri()='http://www.w3.org/2000/09/xmldsig#' and local-name()='Signature'])	To specify the node by XPath in case transform is XPath Transform

Name	Value	Usage
dsig. XPath2Expression_*	XPath expression for XPath filter 2	To specify the node applied XPath filter 2
dsig. XPath2Filter_*	intersect, subtract, union	To specify how is the node treated
dsig. XPath2Order_*	Processing order of each XPath	To specify the processing order
Binding: Key information properties		
keyinfo. IDTypeNS	A namespace URI of the QName. The default is an empty string: ""	To specify the namespace URI part of the QName that represents the calculation method
keyinfo. IDTypeLN	A local name of the QName. The choices are (first is default): ▶ http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#ITSHA1 ▶ http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#IT60SHA1	To specify the localName part of the QName that represents the calculation method
keyinfo. EncodingNS	A namespace URI of the QName. The default is an empty string: ""	To specify the namespace URI part of the QName that represents the encoding method
keyinfo. EncodingLN	A local name of the QName. The choices are (first is default): ▶ http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#Base64Binary ▶ http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#HexBinary	To specify the localName part of the QName that represents the encoding method
keyinfo. ValueTypeNS	A namespace URI of the QName. The default is an empty string: ""	To specify the namespace URI part of the QName that represents the ValueType

Name	Value	Usage
keyinfo. ValueTypeLN	A local name of the QName. Typical choices are: ▶ http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509 ▶ http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#UsernameToken	To specify the localName part of the QName that represents the ValueType
Binding: Token generator properties		
token. username. addNonce	true or 1	To add <Nonce> under the <UsernameToken> element in UsernameTokenGenerator
token. username. addTimestamp	true or 1	To add <Created> under the <UsernameToken> element in UsernameTokenGenerator
Binding: Token generator callback handler		
token. IDAssertion. isUsed	true or 1	To explicitly declare the use of the identity assertion in TokenGeneratorComponent
token. IDAssertion. useRunAsIdentity	true or 1	To explicitly declare the use of the RunAs identity assertion in UsernameToken
Binding: Token consumer properties		
token. username. verifyNonce	true or 1	To process verification of the Nonce
token. username. verifyTimestamp	true or 1	To process verification of the Timestamp
Binding: Token consumer JAAS configuration		
token. x509. issuerName	The issuer's SubjectDN or the IssuerDN of the X.509 certificate	To specify the issuer or the certificate in TokenConsumerComponent
token. x509. issuerSerial	The serial number of the X.509 certificate	To specify the issuer serial number of the certificate in TokenConsumerComponent

Forms for WS-Security configuration

Here, we provide forms for preparation of the WS-Security configuration. Before you start to configure WS-Security using Rational Application Developer, it is good to make a list of configuration items to prevent misconfiguration.

The forms are for the generator and consumer configurations. The consumer configuration should match the generator configuration, so first fill in the generator forms according to your security configuration and then fill the consumer forms to match the generator configuration.

Deployment descriptor configuration

Table B-26 to Table B-29 show the list of configuration items in the deployment descriptor with the predefined choices.

Table B-26 Integrity configuration items

Generator deployment descriptor		Consumer deployment descriptor	
Integrity		Required Integrity	
Name		Name	
Order			
Message parts dialect	- dialect-was - XPath	Message parts dialect	- dialect-was - XPath
Message parts keyword	- body - timestamp - securitytoken - dsigkey - enckey - messageid - to - action - relatesto - XPath []	Message parts keyword	- body - timestamp - securitytoken - dsigkey - enckey - messageid - to - action - relatesto - XPath []
Nonce		Nonce	
Nonce dialect	- dialect-was - XPath	Nonce dialect	- dialect-was - XPath

Generator deployment descriptor		Consumer deployment descriptor	
Nonce keyword	- body - timestamp - securitytoken - dsigkey - enckey - messageid - to - action - relatesto - XPath []	Nonce keyword	- body - timestamp - securitytoken - dsigkey - enckey - messageid - to - action - relatesto - XPath []
Timestamp		Timestamp	
Timestamp dialect	- dialect-was - XPath	Timestamp dialect	- dialect-was - XPath
Timestamp keyword	- body - timestamp - securitytoken - dsigkey - enckey - messageid - to - action - relatesto - XPath []	Timestamp keyword	- body - timestamp - securitytoken - dsigkey - enckey - messageid - to - action - relatesto - XPath []
Timestamp expires		Timestamp expires	

Table B-27 Confidentiality configuration items

Generator deployment descriptor		Consumer deployment descriptor	
Confidentiality		Required Confidentiality	
Name		Name	
Order			
Message parts dialect	- dialect-was - XPath	Message parts dialect	- dialect-was - XPath
Message parts keyword	- bodycontent - usernametoken - digestvalue - XPath []	Message parts keyword	- bodycontent - usernametoken - digestvalue - XPath []
Nonce		Nonce	

Generator deployment descriptor		Consumer deployment descriptor	
Nonce dialect	- dialect-was - XPath	Nonce dialect	- dialect-was - XPath
Nonce keyword	- bodycontent - usenametoken - digestvalue - XPath []	Nonce keyword	-bodycontent -usenametoken -digestvalue -XPath []
Timestamp		Timestamp	
Timestamp dialect	- dialect-was - XPath	Timestamp dialect	- dialect-was - XPath
Timestamp keyword	- bodycontent - usenametoken - digestvalue - XPath []	Timestamp keyword	- bodycontent - usenametoken - digestvalue - XPath []
Timestamp expires		Timestamp expires	

Table B-28 Security token configuration items

Generator deployment descriptor		Consumer deployment descriptor	
Security Token		Required Security Token	
Name		Name	
Token type	- Username - X.509 certificate - X.509 (PKIPath) - X.509 (PKCS#7) - LTPA - Custom	Token type	- Username - X509 certificate - X509 (PKIPath) - X509 (PKCS#7) - LTPA - Custom
URI		URI	
Local name		Local name	
		Usage type	- Required - Optional

Table B-29 Timestamp configuration items

Generator deployment descriptor		Consumer deployment descriptor	
Add Timestamp		Add Timestamp	
Use Add Timestamp	- OFF - ON	Use Add Timestamp	- OFF - ON
Expires	- OFF - ON		
Expires date			
Property	Name= Value=	Property	Name= Value=

Binding configuration

Table B-30 to Table B-38 show list of configuration items in a binding configuration. The list shows the predefined choices.

Table B-30 Trust anchor configuration items

Generator binding configuration	Consumer binding configuration	
	Trust Anchor	
	Trust Anchor	
	Key store storepass	
	Key store path	
	Key store type	

Table B-31 Collection certificate store configuration items

Generator binding configuration		Consumer binding configuration	
Collection certificate store		Collection certificate store	
Name		Name	
Provider		Provider	
		X509 Certificate path	
CRL path		CRL path	

Table B-32 Token generator and consumer configuration items

Generator binding configuration		Consumer binding configuration	
Token Generator		Token Consumer	
Token generator name		Token consumer name	
Token generator class	- UsernameToken - X509Token - LTPAToken	Token consumer class	- UsernameToken - X509Token - LTPAToken - IDAssertionUserNameToken
Security token		Security token	
Value type	- Username - X.509 certificate - X.509 (PKIPath) - X.509 (PKCS#7) - LTPA - Custom	Value type	- Username - X.509 certificate - X.509 (PKIPath) - X.509 (PKCS#7) - LTPA - Custom
Local name		Local name	
URI		URI	
Callback handler	- NonPrompt - GUIPrompt - X509 - PkiPath - PKCS7 - LTPAToken - StdinPrompt	JAAS config	
User ID		JAAS config property	Name= Value=
Password		TrustedIDevaluator class	
Key store storepass		TrustedIDevaluator property	Name= Value=
Key store path		Trusted ID evaluator reference	
Key store type	- JKS - JCEKS - PKCS11 - PKCS12	Trusted ID evaluator reference property	Name= Value=

Generator binding configuration		Consumer binding configuration	
Key Alias			
Key pass			
Key name			
Callback handler property	Name= Value=		
Property	Name= Value=	Property	Name= Value=
Trust anchor reference		Trust anchor reference	
Certificate store reference		Certificate store reference	
Trust any certificate	- OFF - ON	Trust any certificate	- OFF - ON

Table B-33 Key locator configuration items

Generator binding configuration		Consumer binding configuration	
Key Locator		Key Locator	
Key locator name		Key locator name	
Key locator class	- KeyStoreKey Locator	Key locator class	- KeyStoreKey Locator - SignerCertKey Locator - X509TokenKey Locator
Key store storepass		Key store storepass	
Key store path		Key store path	
Key store type	- JKS - JCEKS - PKCS11 - PKCS12	Key store type	- JKS - JCEKS - PKCS11 - PKCS12
Key Alias		Key Alias	
Key pass		Key pass	
Key name		Key name	

Generator binding configuration		Consumer binding configuration	
Property	Name= Value=	Property	Name= Value=

Table B-34 Key information configuration items

Generator binding configuration		Consumer binding configuration	
Key Information		Key Information	
Key information name		Key information name	
Key information type	- STRREF - EMB - KEYID - KEYNAME - X509ISSUER	Key information type	- STRREF - EMB - KEYID - KEYNAME - X509ISSUER
Key information class		Key information class	
Key locator		Key locator	
Key name		Key name	
Token		Token	
Property	Name= Value=	Property	Name= Value=

Table B-35 Signing information configuration items

Generator binding configuration		Consumer binding configuration	
Signing Information		Signing Information	
Signing information name		Signing information name	
Canonicalization method algorithm	- exc-c14n - exc-c14n with comments - c14n - c14n with comments	Canonicalization method algorithm	- exc-c14n - exc-c14n with comments - c14n - c14n with comments
Signature method algorithm	- RSA-SHA1 - DSA-SHA1 - HMAC-SHA1	Signature method algorithm	- RSA-SHA1 - DSA-SHA1 - HMAC-SHA1

Generator binding configuration		Consumer binding configuration	
Key information name		Key information name	
Key information element		Key information element	
Use key information signature type		Use key information signature type	
Key information signature property	Name= Value=	Key information signature property	Name= Value=
Property	Name= Value=	Property	Name= Value=

Table B-36 Part reference configuration items

Generator binding configuration		Consumer binding configuration	
Part reference		Part reference	
Part reference name		Part reference name	
Integrity part		RequiredIntegrity part	
Digest method algorithm	- SHA1	Digest method algorithm	- SHA1

Table B-37 Transform configuration items

Generator binding configuration		Consumer binding configuration	
Transform		Transform	
Name		Name	
Algorithm	- exc-c14n - XPath - XPath Filter2 - STR-Transform - Decrypt XML - Enveloped signature	Algorithm	- exc-c14n - XPath - XPath Filter2 - STR-Transform - Decrypt XML - Enveloped signature
Transform property	Name= Value=	Transform property	Name= Value=

Table B-38 Encryption information configuration items

Generator binding configuration		Consumer binding configuration	
Encryption Information		Encryption Information	
Encryption name		Encryption name	
Data encryption method algorithm	- 3DES-CBC - AES128-CBC - AES256-CBC - AES192-CBC	Data encryption method algorithm	- 3DES-CBC - AES128-CBC - AES256-CBC - AES192-CBC
Key encryption method algorithm	- RSA-1_5 - 3DES - AES128 - AES256 - AES192 - (None)	Key encryption method algorithm	- RSA-1_5 - 3DES - AES128 - AES256 - AES192 - (None)
Key information name		Key information name	
Key information element		Key information element	
Confidentiality part		RequiredConfidentiality part	
Property	Name= Value=	Property	Name= Value=

Additional material

This redbook refers to additional material that can be downloaded from the Internet as described in this appendix.

Locating the Web material

The Web material associated with this redbook is available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser to:

<ftp://www.redbooks.ibm.com/redbooks/SG246461>

Alternatively, you can go to the IBM Redbooks Web site at:

ibm.com/redbooks

Select the **Additional materials** and open the directory that corresponds with the redbook form number, SG246461.

Using the Web material

The additional Web material that accompanies this redbook includes the following files:

<i>File name</i>	<i>Description</i>
sg246461code.zip	Sample code for following the samples in the book
sg246461solution.zip	Solution code (applications)
6461corrections.txt	Corrections to the book after publishing

System requirements for downloading the Web material

The following system configuration is recommended:

Hard disk space	3 GB
Operating System	Windows 2000 or Windows XP
Processor	1.5 MHz or better
Memory	1 GB, recommended 1.5 GB

How to use the Web material

Unzip the contents of the Web material files (**sg246461code.zip** and **sg246461solution.zip**) onto your hard drive. This creates a directory **c:\SG246461\sampcode** with these subdirectories:

_setup	Database setup and initial EAR files
ant	Sample Ant build file
clients	Helper code to build Web services clients
commandline	Helper code for command-line examples
commandtest	Instructions to run the command-line examples
componentTest	Helper code for component test examples
interop	Helper code for the interoperability example
multiprotocol	Helper code for the multiprotocol example
mykeystore	Helper code for security signing and encryption
servers	Helper code to build Web services servers
si-bus	Helper code for the weather forecast meditation
weather_images	Sample images for weather conditions
ws_at	Helper code for atomic transaction
zSolution	Exported code of implemented samples (sg246461solution.zip)

List of enterprise applications

Table C-1 shows all the J2EE enterprise applications and Java applications developed in this rebook. The code is available in the directory:

\SG246461\sampcode\zSolution

Table C-1 Overview of all enterprise applications and Java projects

Application and projects		Description
WeatherJavaBeanServer		JavaBean Web service
	WeatherJavaBeanWeb	Web project with WeatherJavaBean
	WeatherBase	Base code for weather forecast
WeatherEJBServer		EJB Web service
	WeatherEJBClientClasses	EJB client classes
	WeatherEJB	EJB project with WeatherEJB session bean
	WeatherEJBRouterWeb	Web router project
	WeatherBase	Base code for weather forecast
WeatherEJBJMSServer		EJB Web service (over JMS)
	WeatherEJBJMS	EJB project with WeatherJMS session bean
	WeatherEJBJMSRouter	EJB router project with MDB
	WeatherBase	Base code for weather forecast
WeatherEJBAppClientEAR		J2EE client application
	WeatherEJBAppClient	Java application for WeatherEJB
	WeatherBase	Base code for weather forecast
	WeatherEJBClientClasses	EJB client classes
WeatherJavaBeanClient		Client application for WeatherJavaBean
	WeatherJavaBeanClientWeb	Generated Web client
WeatherClientStandalone		Java client for WeatherJavaBean
WeatherClientEAR		Client applications
	WeatherClientJSF	Web client using JSF for WeatherJavaBean
	WeatherTopDownServerWebClient	Web client for WeatherEJBTopDownServer
	WeatherClientAppJMS	Java client for WeatherEJBJMS
WeatherEJBTopDownServer		Web service top-down
	WeatherEJBTopDownServerWeb	Generated from WSDL
WeatherEJBJMSSClient		Client application for WeatherEJBJMS
	WeatherEJBJMSSClientWeb	Generated Web application

Application and projects		Description
WeatherAttachmentServer		Application with attachments
	WeatherAttachmentWeb	Web application with attachment
	WeatherBase	Base code for weather forecast
WeatherAttachmentClient		GUI client using attachments
WeatherJavaBeanComponentTest		Java project for component test
WeatherCommandLineJava		Starting code for command-line samples
WeatherServiceEAR		Command-line tools: Bean2WebService
	WeatherService	Generated Web application
WeatherServiceClient		Java client from command-line tools
WeatherService2EAR		Command-line tools: WSDL2WebService
	WeatherService2	Generated Web application
WeatherService3EAR		Command-line tools: WSDL2Client
	WeatherService3	Generated Web application
WeatherServiceAntEAR		Application using Ant
	WeatherServiceAnt	Web application with Ant
	WeatherServiceAntClient	Generated client application
WeatherEJBMultiProtocolEAR		Application using EJB binding
	WeatherEJBMultiProtocolWeb	Web application with EJB binding
	WeatherBase	Base code for weather forecast
	WeatherEJBClientClasses	EJB client classes
WeatherJavaBeanAxisEAR		Application using Axis runtime
	WeatherJavaBeanAxisWeb	Web application using Axis
WeatherEJBJMSServer1		Copy of WeatherEJBJMSServer for SIB
	WeatherEJBJMS	EJB project with WeatherJMS session bean
	WeatherEJBJMSRouter	EJB router project
	WeatherEJBJMSWSDLWeb	Web project with WSDL file
WeatherMediationServer		Service integration bus (SIB) application
	WeatherMediationServerEJB	EJB for service integration bus
WeatherJavaBeanServer-xxxx		Web service servers with security options
WeatherJavaBeanClient-xxxx		Web service clients with security options
WeatherSecurityConfigSave		Security: Saved deployment descriptors

Installing the base weather forecast application

Here are brief instructions for how to install the starting sample code in IBM Rational Application Developer Version 6.0.

Create a work space for the samples as described in “Starting Application Developer” on page 657, for example, `c:\workspaces\RAD60sg246461`.

Importing the base applications

The base applications are provided as enterprise application EAR files in `\SG246461\sampcode\_setup\EARbase`:

<code>WeatherJavaBeanServer.ear</code>	Implementation of the weather forecast using a JavaBean (Web service SOAP/HTTP)
<code>WeatherEJBServer.ear</code>	Implementation of the weather forecast using a session EJB (Web service SOAP/ HTTP)
<code>WeatherEJBJMSServer.ear</code>	Implementation of the weather forecast using a session EJB (Web service SOAP/JMS)
<code>WeatherEJBAppClient.ear</code>	Implementation of an EJB client (for testing the EJB session bean)

Start Application Developer V6.0 and import the EAR files in the sequence of the enterprise applications listed above:

- ▶ Select *File* → *Import*.
- ▶ Select *EAR file* and click *Next*.
- ▶ The Import panel opens (Figure C-1):
 - For the EAR file, click *Browse* and locate the EAR file to import, for example:
`C:\SG246461\sampcode\_setup\EARbase\WeatherJavaBeanServer.ear`
 - The EAR project name is already filled in.
 - Select *Import EAR Project*.
 - If you repeat the import operation, select *Overwrite existing resources without warning* (and correct the EAR project name).
 - Select *WebSphere Application Server v6.0* as the Target server.
 - Click *Next*.

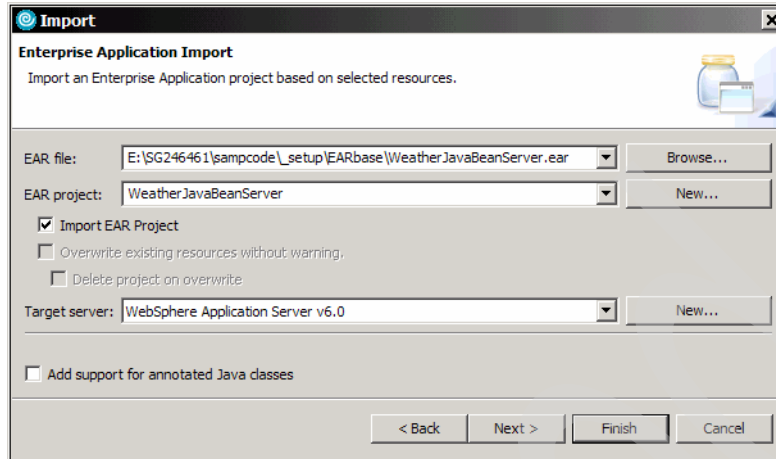


Figure C-1 Enterprise Application Import (1)

- Select *Allow nested project overwrites*. This is necessary because all EAR files contain the WeatherBase utility module.

Select all the utility JARs that are listed (WeatherBase.jar always, and WeatherEJBClientClasses.jar for the WeatherEJBAppClient.ear).

Click *Next* (Figure C-2).

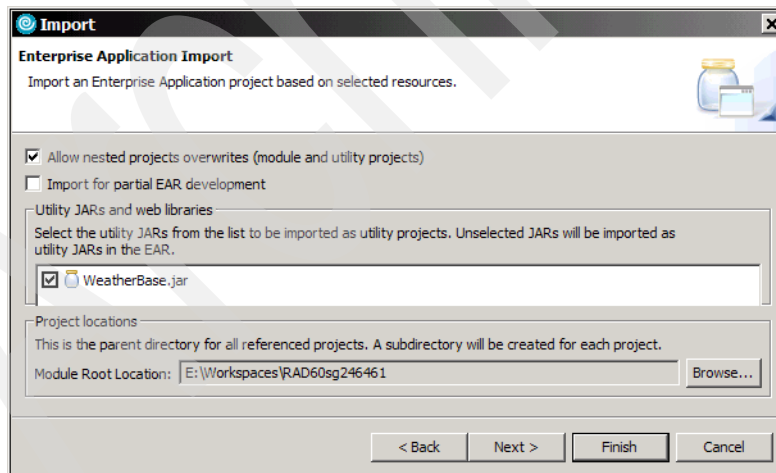


Figure C-2 Enterprise Application Import (2)

- Verify all the module names. *Make sure that you overwrite existing modules by correcting the generated name*, such as WeatherBase1 to WeatherBase (and WeatherEJBClientClasses1 to WeatherEJBClientClasses).

Click *Finish* (Figure C-3).

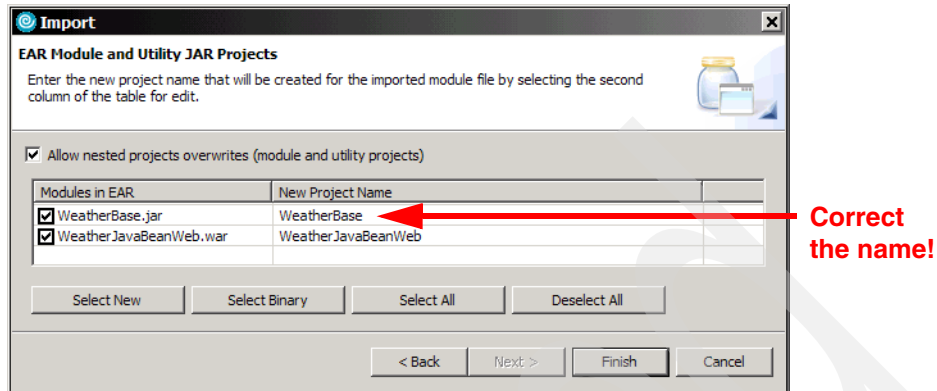


Figure C-3 Enterprise application import (3)

- When prompted to switch the perspective, click *No*.

The J2EE Project Explorer should now contain the projects shown in Figure C-4.

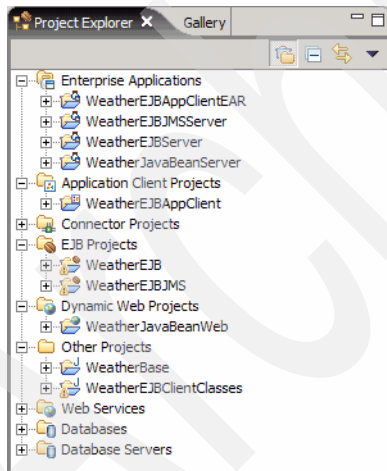


Figure C-4 Enterprise Application Import result

- Note that the WeatherEJB project was configured with an EJB client JAR (WeatherEJBClientClasses project) that makes client development easier. The WeatherEJBJMS project was not configured with a client JAR.
- The Problems view shows many warning about unused imports. You can click the Filters icon and select *Where severity is Error*. Also select *On any resource in same project* to make the error list shorter.

Setting up the WEATHER database

We provide two implementations of the WEATHER database, Cloudscape and DB2. You can choose to implement either or both databases and then set up the enterprise applications to use one of the databases.

Cloudscape

Command files to define and load the WEATHER database in Cloudscape are provided in \SG246461\sampcode_setup\cloudscape\databases:

- ▶ Copy the databases directory to C:\SG246461\cloudscape (create this).
- ▶ In the C:\SG246461\cloudscape\databases directory:
 - Execute the CloudCreate.bat file to define the database and table.
 - Execute the CloudLoad.bat file to delete the existing data and add records.
 - Execute the CloudList.bat file to list the contents of the database.

These command files use the SQL statements and helper files provided in:

- ▶ Weather.ddl—Database and table definition
- ▶ WeatherLoad.sql—SQL statements to load sample data
- ▶ WeatherList.sql—SQL statement to list the sample data
- ▶ tables.bat—Command file to execute Weather.ddl statements
- ▶ load.bat—Command file to execute WeatherLoad.sql statements
- ▶ list.bat—Command file to execute WeatherList.sql statements

The Cloudscape WEATHER database is now stored under:

C:\SG246461\cloudscape\databases\WEATHER

Note: The data source definition for Cloudscape assumes that the Cloudscape WEATHER database is on the C drive. You have to change the data source if you create the database on another drive.

DB2

DB2 command files to define and load the WEATHER database are provided in \SG246461\sampcode_setup\DB2V8:

- ▶ Execute the DB2Create.bat file to define the database and table.
- ▶ Execute the DB2Load.bat file to delete the existing data and add six records.
- ▶ Execute the DB2List.bat file to list the contents of the database.

These command files use the SQL statements provided in:

- ▶ Weather.ddl—Database and table definition
- ▶ WeatherLoad.sql—SQL statements to load sample data
- ▶ WeatherList.sql—SQL statement to list the sample data

Selecting DB2 or Cloudscape

The enterprise applications contain two JDBC drivers with a data source for the WEATHER database, one for DB2 and one for Cloudscape. The JNDI names are jdbc/weather for Cloudscape and jdbc/weather2 for DB2.

By default, Cloudscape will be used for the WEATHER database. To use the DB2 database, rename the Cloudscape data source to jdbc/weather1, and the DB2 data source to jdbc/weather:

- ▶ Open the deployment descriptor of the three server enterprise applications (WeatherJavaBeanServer, WeatherEJBServer, and WeatherEJBMServer).
- ▶ Select the *Deployment* tab (Figure C-5).

The screenshot shows the 'WebSphere v6.0 Deployment' window with the 'Deployment' tab selected. The 'Data Sources' section is expanded, showing a table of JDBC providers. The 'DB2 Legacy CLI-based Type 2 JDBC Driver (XA)' is selected. Below this, the 'Data source defined in the JDBC provider selected above:' section shows a table with one entry: 'WeatherDataSourceDB2' with JNDI Name 'jdbc/weather' and Type 'V5'. This entry is circled in red. Below that, the 'Resource properties defined in the data source selected above:' section shows a table with three entries: 'databaseName' with value 'WEATHER', 'description', and 'connectionAttribute'. The 'Application' section shows 'WeatherJavaBeanServer' selected. The 'Substitution Variables' section shows a table with one entry: 'DB2_JDBC_DRIVER_PATH' with value 'C:\SQLLIB\java', which is circled in red. The 'Authentication' section shows a table with one entry: 'DB2user' with User ID 'db2admin', which is circled in red. The bottom of the window has tabs for 'Overview', 'Module', 'Security', 'Deployment', and 'Source'.

Name	Implementation Class Name	Add...	Edit...	Remove
Cloudscape JDBC Provider (XA)	com.ibm.db2.jdbc.DB2XADataSource			
DB2 Legacy CLI-based Type 2 JDBC Driver (XA)	COM.ibm.db2.jdbc.DB2XADataSource			

Name	JNDI Name	Type	Add...	Edit...	Remove
WeatherDataSourceDB2	jdbc/weather	V5			

Name	Value	Type	Add...	Edit...	Remove
databaseName	WEATHER	java.lang			
description		java.lang			
connectionAttribute	cursorhold=0	java.lang			

Name	Value	Des	Add	Edit	Remove
DB2_JDBC_DRIVER_PATH	C:\SQLLIB\java				

Alias	User ID	Des	Add	Edit	Remove
DB2user	db2admin				

Figure C-5 Deployment Descriptor data source configuration

- ▶ Select the *Cloudscape JDBC Provider (XA)*.
- ▶ From the data source list, select the *WeatherDataSourceCloud* data source and click *Edit*. Change the JNDI name to `jdbc/weather1` and click *Finish*.
- ▶ Select the *DB2 Legacy CLI-based Type 2 JDBC Driver (XA)*.
- ▶ In the data source list, select the *WeatherDataSourceDB2* data source and click *Edit*. Change the JNDI name to `jdbc/weather` and click *Finish*.
- ▶ Expand *Substitution Variables* (bottom). Select the `DB2_JDBC_DRIVER_PATH` variable and click *Edit*. Change the value to the location of your DB2 installation (`C:\<db2-installation-dir>\java`). Click *OK*.
- ▶ Expand *Authentication* (bottom). Select the `DB2user` alias and click *Edit*. Enter the user ID and password you used to install DB2. Click *OK*. Be sure to change the password (the shipped value is not what you expect).
- ▶ Save and close the deployment descriptor.

Tip: You can switch database implementations at any time by changing the JNDI names of the weather data sources in the deployment descriptor of the enterprise applications. The applications use the data source with the name `jdbc/weather`.

Deploying the enterprise applications to the server

In the Servers view, select *WebSphere Application Server v6.0* and start the server. Wait until the server is ready.

Select *WebSphere Application Server v6.0* and *Add and remove projects* (context). Add the three server projects and click *Finish* (Figure C-6). Wait for the deployment to complete.

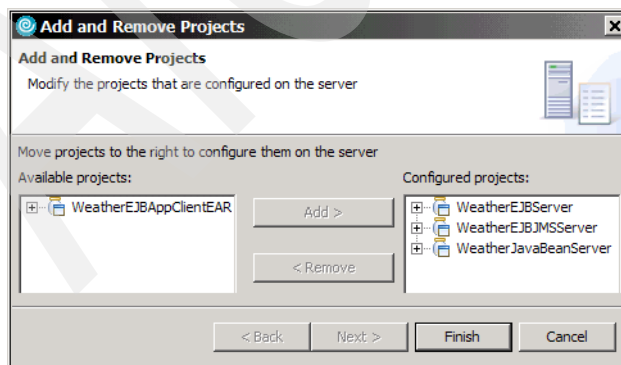


Figure C-6 Adding projects to the server

Testing the enterprise applications

To test the basic functionality of the applications, a simple servlet and an EJB client are included in the base code.

Testing the JavaBean implementation

To test the JavaBean implementation, expand the WeatherJavaBeanWeb project, Deployment Descriptor: WeatherBean → Servlets. Select the WeatherServlet and *Run* → *Run on Server*.

When prompted for Server Selection, the *WebSphere Application Server v6.0* is preselected. Select *Set server as project default* and click *Finish*.

The servlet should produce sample output in the Web browser (Figure C-7).

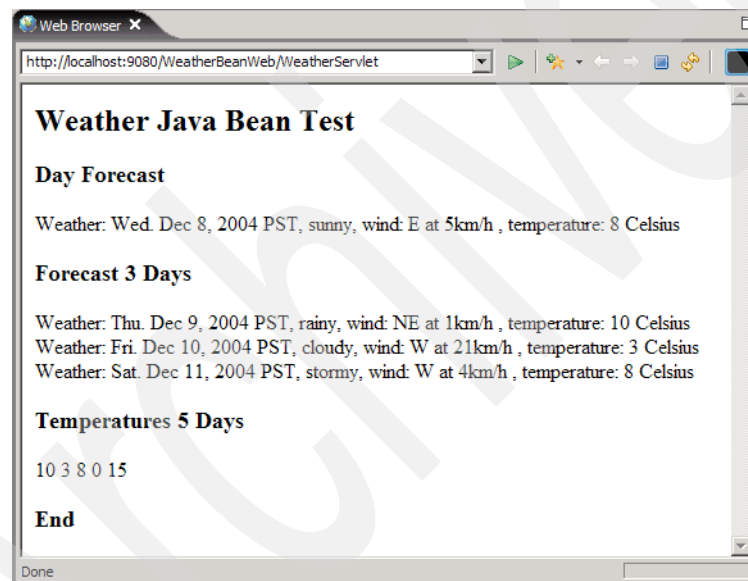


Figure C-7 Adding projects to the server result

Testing the EJB implementation

To test the EJB implementation, we provide the WeatherEJBAppClientEAR enterprise application with a Java program in the WeatherEJBAppClient module:

- ▶ Select the WeatherEJBAppClientEAR project and *Run* → *Run*.
- ▶ In the Run configuration panel, select *WebSphere v6.0 Application Client* and click *New*. Overtyping the name with WeatherEJBAppClient. For the Application client module, select the WeatherEJBAppClient from the pull-down menu. Click *Apply*, and then click *Run* (Figure C-8).

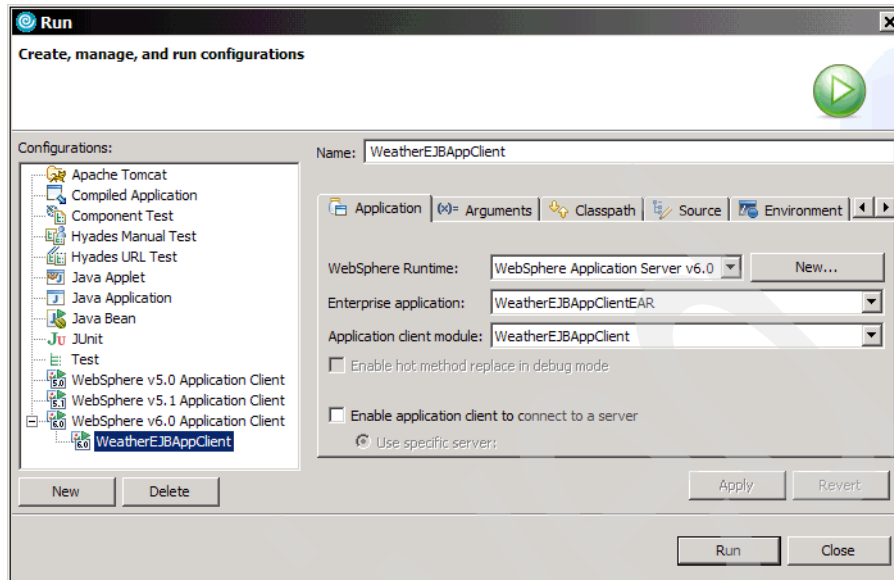


Figure C-8 EJB client run configuration

- In the Console view, you should see the application starting and displaying one result weather forecast, such as:

EJB client got:

Weather: Wed. Dec 8, 2004 PST, sunny, wind: E at 5km/h , temperature: 8 Celsius

Tip: Switch between different console output listings by using the Console icon or pull-down .

The basic setup is now ready to create Web services using SOAP over HTTP. For Web services using SOAP over JMS, you have to configure the server with a JSM server, as described in “Setting up messaging for the JMS Web service” on page 713.

Important: One of the three base applications must be deployed to the server when testing other enterprise applications. This is to ensure that the JDBC data source jdbc/weather is available in the server. Alternatively, you can define the data source in the server configuration using the administrative console.

Setting up messaging for the JMS Web service

For the SOAP over JMS Web service, the server must be configured for JMS messaging. This configuration can be performed as required when developing the JMS Web service. It is not required for the other Web services.

Incoming SOAP/JMS calls are routed to the session enterprise bean through a message-driven bean (MDB). We create a SOAP over JMS Web service in “Web services and JMS binding” on page 303. Therefore, we have to set up a JMS configuration for the Web service with JMS binding.

Starting with WebSphere Application Server V6, there are two options to configure the support for MDBs:

- ▶ **Listener port**—The listener port must be used when using JMS providers, WebSphere V5 default messaging, WebSphere MQ, or a generic JMS provider. This is the same support for MDBs used in WebSphere Application Server Version 5.x.
- ▶ **Java 2 Connection Architecture (J2C) activation specification**—Applications using the default messaging provider (or any J2C inbound resource adapter), must use the activation specification. Access to the default messaging in Application Server Version 6 is provided through a resource adapter.

More information about J2C, MDBs, JMS in WebSphere Application Server, and the activation specification is available in the *WebSphere Application Server Information Center*.

The objects to configure for Web services over JMS in Application Server Version 6 are:

- ▶ Service integration bus.
- ▶ JMS queue connection factories (two are required).
- ▶ JMS queue.
- ▶ Activation specification.

Starting the administrative console

All server configuration must be performed using the administrative console:

- ▶ Make sure that WebSphere Application Server V6.0 is running.
- ▶ Open the administrative console by selecting the server in the Servers view and *Run administrative console* (from the context menu).
- ▶ Log in with your regular user ID.

Creating a service integration bus

To configure a service integration bus, perform these steps:

- ▶ Expand *Service integration* in the left pane and select *Buses*.
- ▶ Click *New*. Enter *weatherBUS* as the Name and click *Apply* (Figure C-9).

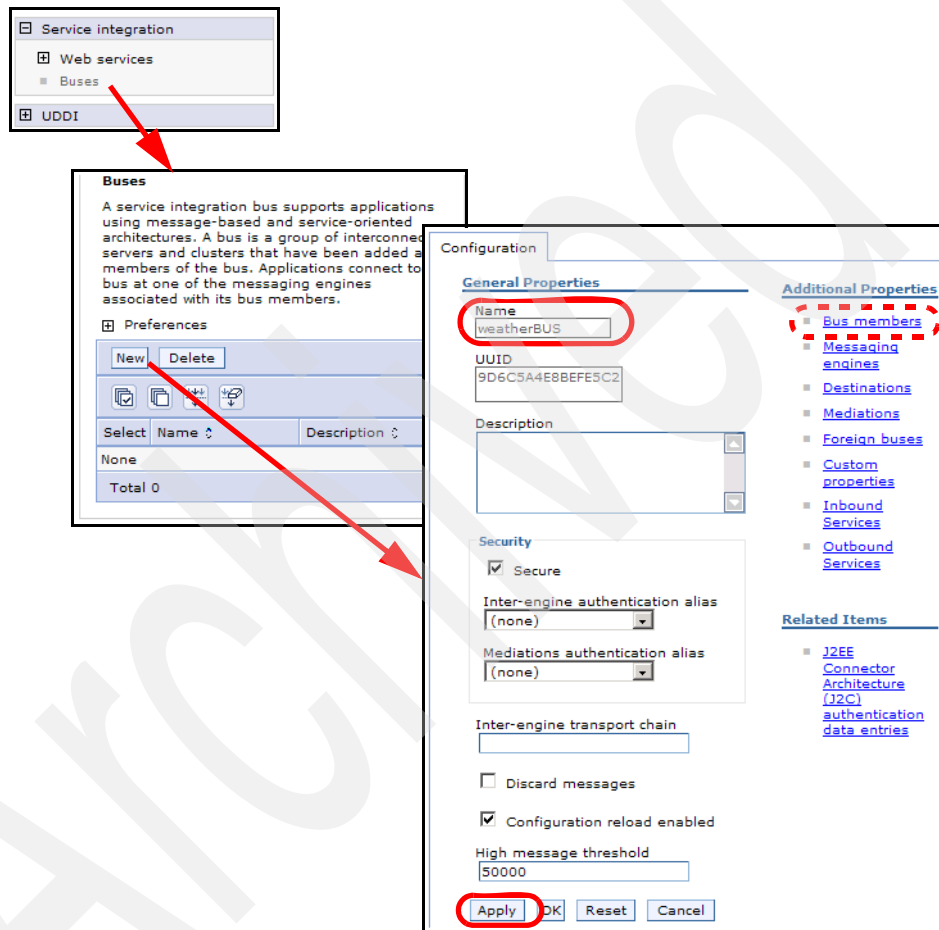


Figure C-9 Create a service integration bus (1)

- ▶ Select *Bus members*. Click *Add*. The server is already filled in, and all the settings are fine. Click *Next*, and then on the next page (Figure C-10), click *Finish*.

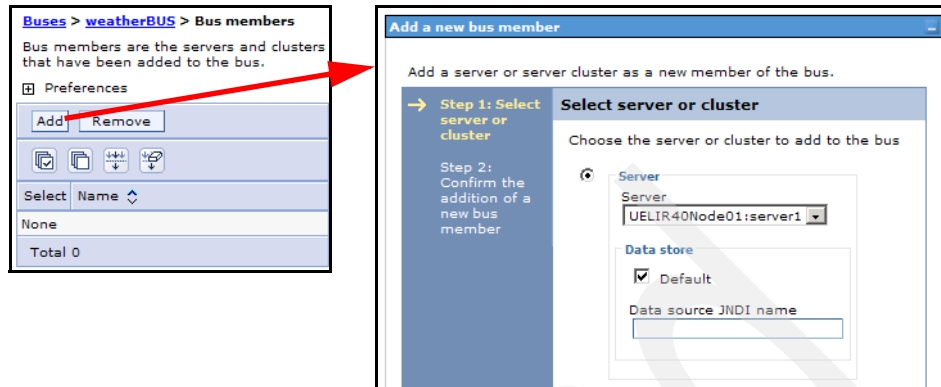


Figure C-10 Create a service integration bus (2)

Defining the destination

To define the destination, complete these steps:

- ▶ Navigate to weatherBUS. Select weatherBUS in the link sequence (*Buses > weatherBUS > Bus members*) or by selecting *Service integration* → *Buses* → *weatherBUS*.
- ▶ Select *Destinations* under Additional Properties. Click *New* (Figure C-11).

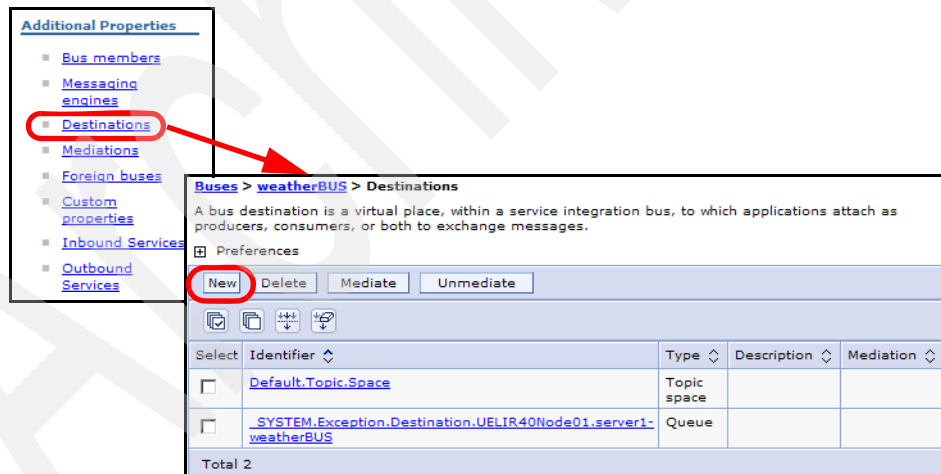


Figure C-11 Create a service integration bus: Destination (1)

- ▶ Go through the pages of the wizard. Select *Queue* as the destination type. Enter WeatherDestQ as the Identifier. The rest of the settings are fine. Click *Finish* (Figure C-12).

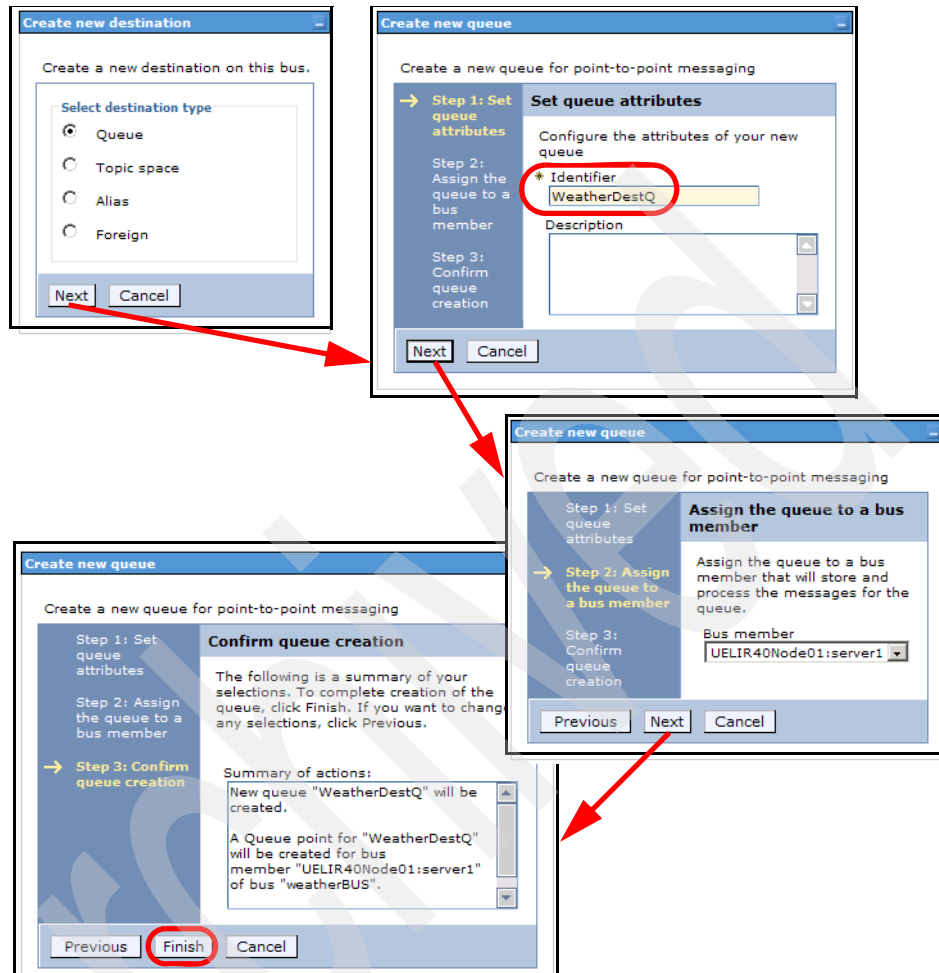


Figure C-12 Create a service integration bus: Destination (2)

Creating the queue connection factories

A SOAP over JMS Web service requires two queue connection factories and two queues; one each for the request and for the response. The reply queue is dynamically defined and must not be configured.

First, we configure the queue connection factories:

- Expand *Resources* → *JMS Providers* and select *Default messaging* as the JMS provider. Click *JMS queue connection factory* (Figure C-13).

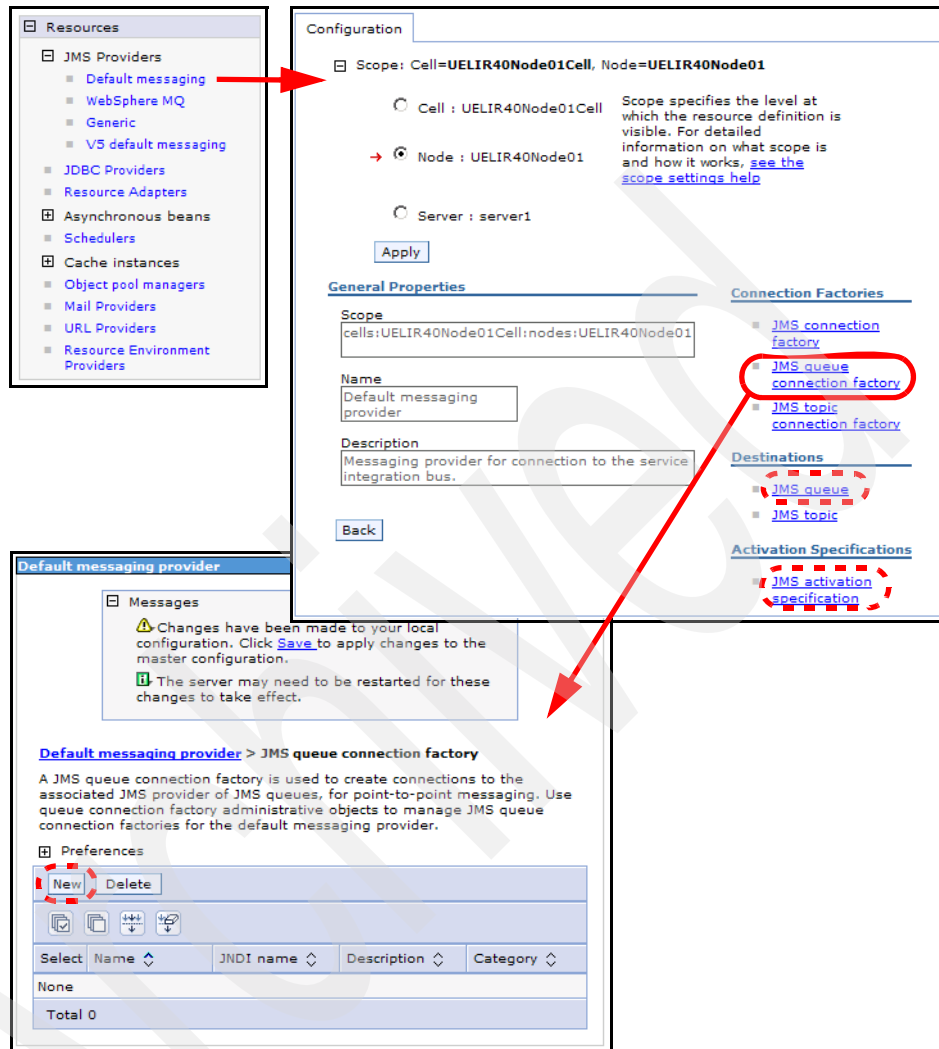


Figure C-13 Define queue connection factory (1)

- Click **New** to define a queue connection factory. In the panel, enter Weather QCF as the Name and jms/weatherQCF as the JNDI name. Select the weatherBUS from the Bus name drop-down menu. Leave all other parameters with their default settings. Click **OK** (Figure C-14).

The screenshot shows the 'Configuration' window for a queue connection factory. The 'General Properties' tab is active, showing fields for 'Name' (Weather QCF), 'JNDI name' (jms/weatherQCF), 'Description', and 'Category'. The 'Connection' tab is also visible, showing fields for 'Bus name' (weatherBUS), 'Target', 'Target type' (Bus member name), 'Target significance' (Preferred), 'Target inbound transport chain', 'Provider endpoints', and 'Connection proximity' (Bus). A red circle highlights the 'Name' and 'JNDI name' fields. Another red circle highlights the 'Bus name' dropdown. A red arrow points to the 'Provider endpoints' field with the text 'for remote clients over a service integration bus'. The 'OK' button is also circled in red.

Figure C-14 Create queue connection factories (2)

Note: If you plan to connect to these JMS resources from a remote machine over a service integration bus (when the bus runs on a machine other than the Web service), you should also configure the *Provider endpoints* field for your queue connection factory as follows:

```
server_hostname:7276:BootstrapBasicMessaging
```

See “Using the bus with the weather forecast application” on page 610.

- Redo the same steps to create the Web services reply queue connection factory:

Name: Weather Reply QCF
JNDI name: jms/WebServicesReplyQCF
Bus name: weatherBUS (select from the drop-down list)

Important: The name for the JMS Web services reply queue connection factory is a reserved, case-sensitive name: **jms/WebServicesReplyQCF**

This queue connection factory name cannot be changed and must not be used for application purposes.

Creating the queue

To create the queue for the weather forecast service, perform these steps:

- For the Default messaging provider, select *JMS queue* under Destinations. Click *New*. In the panel, enter Weather Queue as the Name, jms/weatherQ as the JNDI name, select *weatherBUS* for the Bus name, and select *WeatherDestQ* as the Queue name. Click *OK* (Figure C-15).

The screenshot shows the 'Configuration' dialog box for a JMS queue. The 'General Properties' tab is selected. The 'Administration' section shows the 'Scope' as 'cells:UELIR40Node01Cell:nodes:UELIR40Node01'. The 'Name' field is 'Weather Queue', the 'JNDI name' is 'jms/weatherQ', and the 'Description' is empty. The 'Connection' section shows the 'Bus name' as 'weatherBUS' and the 'Queue name' as 'WeatherDestQ'. The 'Delivery mode' is 'Application', 'Time to live' is empty, 'Priority' is empty, and 'Read ahead' is 'Enabled'. The 'OK' button is highlighted with a red circle.

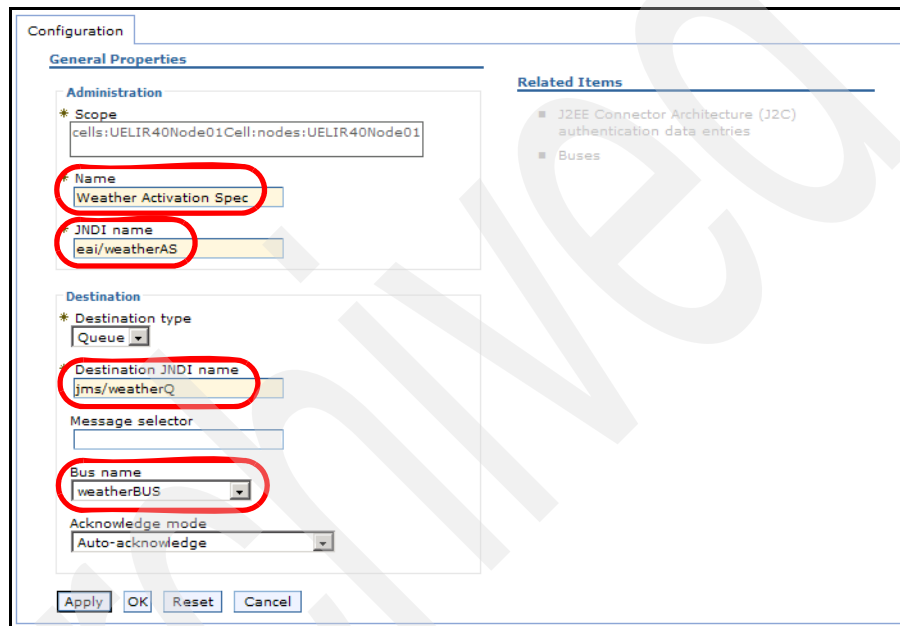
Figure C-15 Create queue

Creating an activation specification

To create an activation specification, perform these steps:

- For the Default messaging provider, select *JMS activation specification* under Activation Specifications. Click *New*.

In the panel, enter *Weather Activation Spec* as the Name, *eai/weatherAS* as the JNDI name, select *weatherBUS* for the Bus name, and enter *jms/weatherQ* as the Destination JNDI name. Click *OK* (Figure C-16).



The screenshot shows the 'Configuration' window with the 'General Properties' tab selected. The 'Administration' section contains a 'Scope' field with the value 'cells:UELIR40Node01Cell:nodes:UELIR40Node01'. Below it, the 'Name' field is set to 'Weather Activation Spec', the 'JNDI name' field is set to 'eai/weatherAS', and the 'Destination JNDI name' field is set to 'jms/weatherQ'. The 'Destination' section shows 'Destination type' set to 'Queue', 'Message selector' is empty, 'Bus name' is set to 'weatherBUS', and 'Acknowledge mode' is set to 'Auto-acknowledge'. The 'Related Items' section on the right lists 'J2EE Connector Architecture (J2C) authentication data entries' and 'Buses'. At the bottom, there are buttons for 'Apply', 'OK', 'Reset', and 'Cancel'.

Figure C-16 Create an activation specification

Important JNDI names

This completes the definitions for SOAP over JMS.

We have to remember the important JNDI names to be used when configuring the SOAP over JMS protocol for our Web service:

- *eai/weatherAS*—Activation specification
- *jms/weatherQCF*—Queue connection factory
- *jms/weatherQ*—Queue

Saving the changes

Save the WebSphere configuration (Figure C-17). Leave the administrative console by clicking *Logout* in the administrative console tool bar.

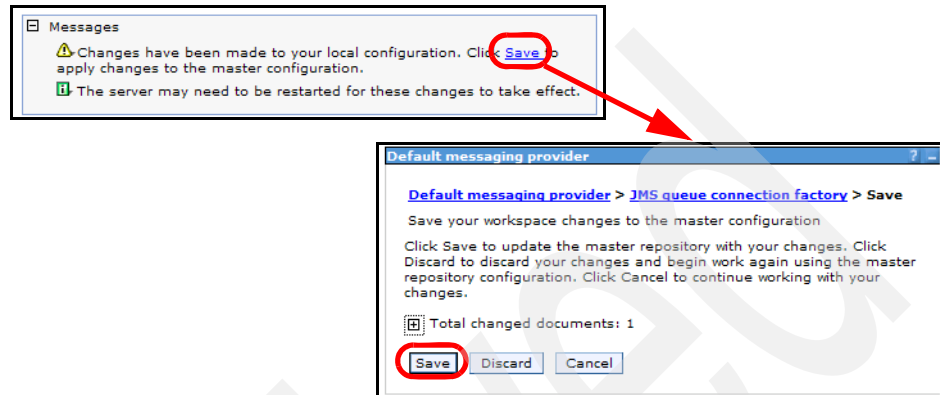


Figure C-17 Save configuration changes

Note: For each service integration bus with persistence (which is the default), a data source for a Cloudscape database is created:

- ▶ You can find this data source in the administrative console under *Resources* → *JDBC Providers*. Select *Server* as scope and click *Apply*. Then, select the *Cloudscape JDBC Provider* and click *Data sources*. The name of the data source is <nodename>.server1-weatherBUS.
- ▶ Reset the scope of JDBC Providers to *Node* and click *Apply*.

Configuring JMS using a JACL script

Instead of manually configuring the server using the administrative console, we also provide a JACL script to perform the configuration using the `wsadmin` tool:

- ▶ The JACL script and command files to execute the script are provided in:
 - `\SG246461\sampcode\_setup\serverJMS`
 - `JMSforWebServ.jacl`—JACL script
 - `wsadminConfigJMS.bat`—Command file to execute the script
 - `setupPath.bat`—Command file to set up the PATH (must be tailored)
- ▶ The server must be running while executing the JACL script.

Restarting the server

To activate the configuration changes, restart the application server in the Servers view. In the console, you should see a message that the bus is started:

```
[date] 00000018 SibMessage [weatherBUS:<node>.server1-weatherBUS]  
CWSID0016I: Messaging engine <node>.server1-weatherBUS is in state Started.
```

When the server is ready, open the administrative console:

- Expand *Servers* and select *Application servers*. Select the *server1* entry. Select *Messaging engines*, and you can see that the weatherBUS is started (Figure C-18).

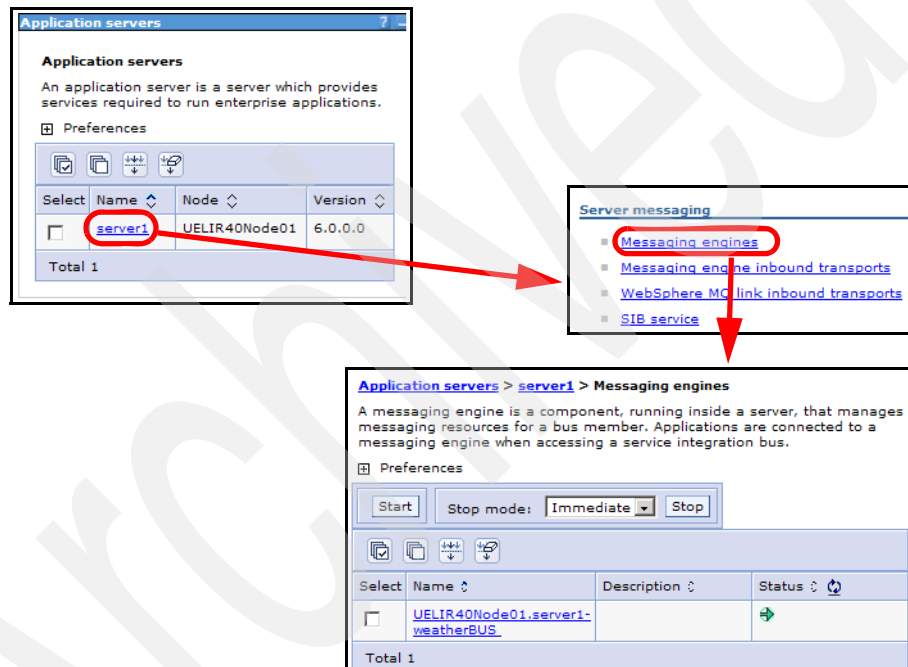


Figure C-18 Examine the server messaging engines

The server is now configured for JMS and ready for development of JMS Web services. The Cloudscape database for persistent messages is created in the directory:

```
<RAD-HOME>\runtimes\base_v6\profiles\default\databases\com.ibm.ws.sib
```

Importing solutions

When importing enterprise application solution EAR files (from zSolution), follow the instructions in “Importing the base applications” on page 705 and make sure that you do not create copies of the imported modules (WeatherBase).

You can import a solution and replace your own development. Sometimes, it might be better to delete your projects before importing a solution.

Archived

Abbreviations and acronyms

AAT	Application Assembly Tool	DTD	Document type definition
ANSI	American National Standards Institute	DTO	Data transfer object
API	Application programming interface	EAI	Enterprise application integration
ASAP	Asynchronous Service Access Protocol	EAR	Enterprise application archive
ASTK	Application Server Toolkit	EDI	Electronic data interchange
Axis	Apache Extensible Interaction System	EIS	Enterprise information system
B2B	Business-to-business	EJB	Enterprise JavaBeans
B2C	Business-to-consumer	EJS	Enterprise Java Server
B2E	Business-to-employee	EPI	External presentation interface
BPEL	Business process execution language	ERP	Enterprise resource planning
CICS	Customer Information Control System	ESB	Enterprise Service Bus
CORBA	Common Object Request Broker Architecture	ETTK	Emerging Technologies Toolkit
CRM	Customer relationship management	FAQ	Frequently asked questions
CTG	CICS Transaction Gateway	FTP	File Transfer Protocol
CVS	Common Versions System	GUI	Graphical user interface
DAD	Document access definition	HTML	Hypertext Markup Language
DADX	Document access definition extended	HTTP	Hypertext Transfer Protocol
DBMS	Database management system	IBM	International Business Machines Corporation
DCOM	Distributed common object model	IDE	Integrated development environment
DII	Dynamic invocation interface	IDL	Interface definition language
DNS	Domain Name System	IETF	Internet Engineering Task Force
DNS-EPD	Domain Name System End Point Discovery	IIOP	Internet Inter-ORB Protocol
DOM	Document object model	IMS	Information Management System
		ITSO	International Technical Support Organization
		IWA	International Weather Association (fictitious)

J2C	J2EE Connector Architecture	OMG	Object Management Group
J2EE	Java 2, Enterprise Edition	PKI	Public key interface
J2SE	Java 2, Standard Edition	PMI	Performance monitor interface
JAAS	Java Authentication and Authorization Service	QoS	Quality of service
JAR	Java Archive	RAD	Rapid application development
JAXB	Java Architecture for XML Binding	RAR	Resource adapter archive
JAXM	Java API for XML Messaging	RDBMS	Relational database management system
JAXR	Java API for XML Registries	REL	Rights Expression Language
JAX-RPC	Java API for XML-based RPC	RMI	Remote Method Invocation
JCA	J2EE Connector Architecture	RMI-IIOP	Remote Method Invocation/Internet Inter-ORB Protocol
JCE	Java Cryptography Extension	RPC	Remote procedure call
JCP	Java Community Process	RSA	Rivest, Shamir, and Adelman (encryption)
JDBC	Java Database Connectivity	SAAJ	SOAP with Attachments API for Java
JDK	Java Developer's Kit	SAML	Security assertion markup language
JKS	Java Keystore	SAX	Simple API for XML
JMS	Java Messaging Service	SCM	Supply chain management
JMX	Java Management Extension	SCT	security context token
JNDI	Java Naming and Directory Interface	SDK	Software Development Kit
JSP	JavaServer Page	SEI	Service endpoint interface
JSR	Java Specification Request	SMTP	Simple Mail Transfer Protocol
JSSE	Java Secure Socket Extension	SOA	Service-oriented architecture
JTA	Java Transaction API	SOAP	Simple Object Access Protocol (also known as Service Oriented Architecture Protocol)
JTS	Java Transaction Service	SPML	Service Provisioning Markup Language
LDAP	Lightweight Directory Access Protocol	SQL	Structured query language
LTPA	Lightweight Third-Party Authentication	SSL	Secure Sockets Layer
MDB	Message-driven bean	SwA	SOAP with Attachments
MTOM	(SOAP) Message Transmission Optimization Mechanism	SWAM	Simple WebSphere Authentication Mechanism
MVC	Model-view-controller		
NAICS	North American Industry Classification System		

TCP/IP	Transmission Control Protocol/Internet Protocol	WS-BA	Web Services Business Activity
TDT	Test data tables	WS-BPEL	Web Services Business Process Execution Language
TGT	Ticket granting ticket (Kerberos)	WS-CAF	Web Services Composite Application Framework
UDDI	Universal Description, Discovery, and Integration	WS-CDL	Web Services Choreography Description Language
UNSPSC	Universal Standard Products and Services Classification	WS-COOR	Web Services Coordination
URI	Uniform resource identifier	WS-I	Web services interoperability
URL	Uniform resource locator	WWW	World Wide Web
URN	Uniform resource name	XMI	XML metadata interchange
UTC	Universal Test Client	XML	Extensible Markup Language
UUID	Universal unique identifier	XSD	XML Schema Definition
WAR	Web application archive	XSL	Extensible Stylesheet Language
WORF	Web service object runtime facility		
WSCA	Web services conceptual architecture		
WSCI	Web Service Choreography Interface		
WSDL	Web Services Description Language		
WSDM	Web Services Distributed Management		
WSEE	Web services for J2EE		
WSGW	Web Services Gateway		
WSIF	Web Services Invocation Framework		
WSIL	Web Services Inspection Language (also known as WS-Inspection)		
WSRM	Web Services Reliable Messaging		
WSRP	Web Services for Remote Portlets		
WSS	Web Services Security		
WSTK	Web Services Toolkit		
WS-AT	Web Services Atomic Transaction		

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

IBM Redbooks

For information about ordering these publications, see “How to get IBM Redbooks” on page 731. Note that some of the documents referenced here may be available in softcopy only.

- ▶ *WebSphere Application Server V6: Planning and Design*, SG24-6446
- ▶ *WebSphere Application Server V6: Security Handbook*, SG24-6316
- ▶ *WebSphere Application Server V6: System Management and Configuration Handbook*, SG24-6451
- ▶ *Rational Application Developer V6 Programming Guide*, SG24-6449
- ▶ *WebSphere Application Server V6: Scalability and Performance Handbook*, SG24-6392
- ▶ *WebSphere Application Server V6 Technical Overview*, REDP-3918
- ▶ *Patterns: Implementing an SOA Using an Enterprise Service Bus*, SG24-6346
- ▶ *Patterns: Service-Oriented Architecture and Web Services*, SG24-6303
- ▶ *Using Web Services for Business Integration*, SG24-6583
- ▶ *WebSphere Version 5.1 Application Developer 5.1.1 Web Services Handbook*, SG24-6891
- ▶ *WebSphere and .NET Interoperability Using Web Services*, SG24-6395
- ▶ *WebSphere Studio 5.1.2 JavaServer Faces and Service Data Objects*, SG24-6361
- ▶ *WebSphere Version 5 Application Development Handbook*, SG24-6993
- ▶ *WebSphere Studio Application Developer Version 5 Programming Guide*, SG24-6957
- ▶ *EJB 2.0 Development with WebSphere Studio Application Developer*, SG24-6819
- ▶ *Web Services Wizardry with WebSphere Studio Application Developer*, SG24-6292

- ▶ *Software Configuration Management: A Clear Case for IBM Rational ClearCase and ClearQuest UCM*, SG24-6399

Other publications

These publications are also relevant as further information sources:

- ▶ Zimmermann, O., et al., *Perspectives on Web Services: Applying SOAP, WSDL, and UDDI to Real-World Projects*, Springer-Verlag, 2003, ISBN 3540009140
- ▶ Gamma, E., et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional, 1995, ISBN 0201633612

Online resources

These Web sites and URLs are also relevant as further information sources:

- ▶ IBM software
<http://www.ibm.com/software>
<http://www.ibm.com/software/websphere>
- ▶ *IBM WebSphere Application Server Information Center*
<http://www.ibm.com/software/webservers/appserv/was/library/>
http://publib.boulder.ibm.com/infocenter/ws60help/index.jsp?topic=/com.ibm.websphere.base.doc/info/welcome_base.html
- ▶ IBM developerWorks
<http://www.ibm.com/developerworks>
- ▶ IBM Research
<http://www.research.ibm.com>
- ▶ Web Services Interoperability Organization
<http://ws-i.org>
- ▶ World Wide Web Consortium
<http://www.w3.org>
<http://www.w3c.org>
- ▶ Web services standards
<http://www.w3.org/2002/ws/>
- ▶ OASIS Consortium
<http://www.oasis-open.org>

- ▶ The Internet Engineering Task Force
<http://www.ietf.org>
- ▶ Sun Java Web site
<http://java.sun.com/>
- ▶ Java Community Process
<http://www.jcp.org>
- ▶ Apache XML Project
<http://xml.apache.org>
- ▶ Microsoft Developer Network
<http://msdn.microsoft.com>
- ▶ UDDI Organization and Registries
<http://www.uddi.org>
<http://www.ibm.com/software/solutions/webservices/uddi/>
<http://uddi.microsoft.com>
<http://uddi.sap.com>
<http://www.ntt.com/uddi>
- ▶ Borland
<http://www.borland.com>

How to get IBM Redbooks

You can search for, view, or download Redbooks, Redpapers, Hints and Tips, draft publications and Additional materials, as well as order hardcopy Redbooks or CD-ROMs, at this Web site:

ibm.com/redbooks

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services

Index

Symbols

.NET

- exceptions 203
 - interoperability 201
 - interoperability example 434
 - nillable primitives 205
 - UTF-16 203
 - Web service proxy 439
- .NET Framework SDK 434

A

- access point 129
- activation specification 713
 - create 720
- activity session 235
- adaptor 223
- administrative console 380
 - Application Developer 659
 - default binding 547
 - global security 531
 - JAAS configuration 551
 - modify binding 551
 - PMI 363
 - save 721
 - service integration bus 566
 - tracing 540
- Agent Controller
 - component test 352
 - installation 656
- Ant
 - editor 413
 - Java2WSDL 415
 - properties 414
 - run 418
 - script 413
 - target 415
 - task 415
 - Web services tasks 412
 - WSDL2Java 416
- Apache
 - Axis 41
 - Web site 67
 - SOAP
 - implementation 57
 - server 57
- application
 - profiling 235
- Application Assembly Tool 382
- application client 306
 - run 306
 - outside 307
- Application Developer 237
 - administrative console 659
 - generate WS-Security 543
 - import EAR files 705
 - installation 654
 - interoperability tools 428
 - new server 661
 - overview 274
 - preferences 659
 - profiles 660
 - server configuration 532
 - server security 532
 - start 657
 - test environment 660
 - Web services 238
 - WSDL support 239
 - WS-I compliance 428
- Application Server
 - dynamic cache service 634
 - global security 185
 - installation 651
 - profiles 660
 - TCP/IP Monitor 372
- Application Server Toolkit 382, 446
- application-to-application 11
- array
 - empty 204
 - interoperability 205
- ASAP 20
- assembly tools 382
- ASTK 382, 446
- asymmetric key 455
- asynchronous
 - beans 235
 - message 217
 - messaging 560

- Asynchronous Services Access Protocol 20
- atomic transaction
 - deployment descriptor 328, 332
 - example 328
 - SOAP message 333
 - testing 332
- attachments 316
- Attachments Profile 316
- auditing 174, 181
- authentication 119, 174, 181, 448
 - alias 577
 - alias for DB2 710
 - basic 184, 448
 - caller part 489
 - client 485
 - data entries 384
 - LTPA token mapping V5/V6 669
 - mapping V5/V6 666
 - sample 484
 - server 489
 - SOAP Message 535
 - token consumer 493
 - token generator 486
 - X.509 certificate mapping V5/V6 668
- authorization 119, 174, 181, 600
- availability 175
- Axis 41, 61–62, 431
 - client architecture 63
 - component test 355
 - interoperability client 431
 - server architecture 63
 - subsystems 64

B

- B2B 124
- base 64 encoding 316
- base applications 705
- basic authentication 184, 448
 - mapping V5/V6 666
 - sample 486
- Bean2WebService 396
 - client 403
 - deployment 402
 - example 397
 - help 402
 - output 400
- best practices 225
- binary security token 448–449

- binding
 - configuration
 - default 547
 - HTTP 85
 - MIME 86
 - SOAP 84
 - stub 96
 - template 125, 128
 - UDDI 166
 - WSDL 81
 - WSIL 164
 - WS-Inspection 164
- Bindings page 479
- body 41, 48
 - SOAP 48
- Borland
 - C#Builder Personal Download Edition 434
 - client development 435
- bottom-up development 263–264
- BPEL 25
- BPEL4WS 25
- broker 5
 - public 5
- build phase 263
- build.xml 413
- bus
 - see service integration bus
- business
 - entity 125
 - process 6
 - registry 134
 - relationships 126, 132
 - rule 6
 - service 125
- Business Process Execution Language 25
- business-to-business 124

C

- C# 435
- cache
 - certificate 466, 554
 - configuration 635
 - file 636
 - handlers 637
 - JAX-RPC handler 642
 - load test 643
 - monitor 638
 - nonce 465

- replication 554
- result analysis 645
- servlet 635
- statistics 641
- throughput 645
- Web services 633
 - client 641
 - server 638
- cachespec.xml 635
- caching 228
- callback handler classes 487
- CallbackHandler 450, 462
- cell 378
- certificate caching 466, 554
- certification agency 192
- chain 63
- choreography 218, 220
- class reloading 387
- client
 - Borland 435
 - confidentiality 516
 - identity assertion 494
 - integrity 500
 - programming model 105
 - security token 485
 - timestamp 528
 - WS-Security 476
- Cloudscape
 - WEATHER database 708
- collaboration 4
- collection classes 226
- command line
 - service integration bus 567
 - tools 396
 - UDDI 630
- communication
 - protocol 6
 - style 42, 52
- compliance level 428
- component test 339
 - create 352
 - data 356
 - enable 353
 - export 361
 - failure 361
 - HTML 361
 - input data 357
 - project 354
 - return data 358

- run 360
- composite message 220
- compound type 50
- confidentiality 120, 174, 181, 190, 450
 - client 516
 - dialog 517
 - encryption information 520, 526
 - key information 519, 525
 - key locator 518, 525
 - keywords 464
 - mapping V5/V6 686
 - required confidentiality 522
 - response message 527
 - sample 516
 - server 521
 - SOAP message 538
 - token consumer 523
- configuration
 - mapping V5/V6 666
- conformance
 - WS-I 202
- connection factories 716
- Console view 659
- CORBA 40, 45, 227
- custom token 450, 462

D

- DADX 238
- data
 - model
 - SOAP 49
 - source
 - configuration 383
 - weather forecast 253
- DataHandler class 321
- DB2
 - authentication alias 710
 - WEATHER database 708
 - Web services 240
- DB2 Legacy CLI-based Type 2 JDBC Driver 383
- DB2user 384, 710
- DCOM 45
- default binding configuration 547
- deploy phase 263
- deployment
 - enterprise application 382
 - manager 378
 - SOAP 58

- Web services 392
- deployment descriptor 109
 - atomic transaction 328
 - data source 709
 - SOAP 55
- Deployment Descriptor Editor 476
- description language 7
- deserialization 54
- deserializer 51, 57, 60
- design patterns 730
- developerWorks
 - Web site 67
- development
 - phases 262
 - process 262
 - scenarios 274
- digital signature 130, 174
- DII 98, 108, 271, 609
- DIME 316
- Direct Internet Message Encapsulation 316
- disableTestDataTable method 359
- DNS Endpoint Discovery 19
- document
 - style
 - SOAP 52
 - type definition 48
 - WSDL 71
- document/encoded 55
- document/literal 55, 201
 - wrapped 201
- DTD 48
- dynamic
 - binding 269
 - cache
 - activate 635
 - client 269
 - invocation interface 95, 98, 106, 108, 271, 609
 - proxy 97, 106, 108
 - query 235
 - Web services 10, 124, 269

E

- eai/weatherAS 305, 720
- eavesdropping 176
- e-business 10
- ebXML 235
- Eclipse 236
- EDI

- see electronic data interchange
- EGL 236
- EJB
 - container
 - programming model 113
 - QL 30
 - router project 303
 - testing 351
- EJB2WebService 396
 - example 404
- EJBDeploy tool 387
- electronic data interchange 4
- e-marketplace UDDI 149
- embedded configuration 387
- empty array 204
- enableTestDataTable method 359
- encapsulation 7
- encoding 42
 - SOAP 54
- encryption 176
- endpoint
 - change address 341
 - interface 108
 - listener 561, 571
 - application 565
 - authentication 598
 - security settings 598
- end-to-end security 177
- enterprise
 - application integration 211
 - service bus 210–211
 - Web services 103
- enterprise application
 - deployment 382, 710
 - EAR file 384
 - installation 384
 - list of samples 703
 - solutions 723
 - start/stop 391
 - testing 711
- Enterprise Developer 237
- Enterprise Generation Language 236
- entity promotion 129
- envelope 41
 - SOAP 45
- error handling 48
- Express server 234
- Extensible Markup Language
 - see XML

Extensions page 478
extranet UDDI 150

F

facade 223
fault
 code 48
 element 48
 message 49
find_binding 147
find_business 144
find_service 145
find_tModel 146
forms
 WS-Security 692
FTP 40, 62, 192

G

garbage collection 40
gateway
 coexistence 605
 instance 561, 581
 service 561, 582
generated files 283
getEndpoint method 348
global security 185, 531
GSS-API 454
GUIPromptCallbackHandler 487

H

handler 63, 118, 309
 chain 315
 client side 312
 server side 309
 stand-alone client 313
 testing 311
Handler interface 309
header 46
 SOAP 43
history
 Web services 10
HTTP 192
 binding 85
 extension framework 40
 header 43
 TCP/IP Monitor 371
 Server

 installation 653
 server plug-in 391
human-to-application 11

I

IBM
 Web services 36
ibm-webservices-bnd.xmi 471
ibm-webservicesclient-bnd.xmi 471
ibm-webservicesclient-ext.xmi 471
ibm-webservices-ext.xmi 471
IDAssertionUsernameTokenConsumer 491
identification 174, 181
identity assertion 184, 450, 460
 architecture 461
 caller part 496
 client 494
 mapping V5/V6 670
 required integrity 495
 sample 494
 server 495
 SOAP Message 536
IDL 71
IETF 34
Implementing Enterprise Web Services 31, 34, 103
import
 WSDL 75
inbound service 561, 573
information set 88
inheritance 204
installation
 Agent Controller 656
 Application Developer 654
 HTTP Server 653
 planning 650
 sample code 663
 WebSphere Application Server 650–651
 WebSphere plug-in 653
integration 6
Integration Edition 237
integrity 120, 174, 176, 181, 185, 450
 client 500
 key information 505, 513
 key locator 504, 512
 keywords 463
 mapping V5/V6 683
 part reference 507, 514
 required integrity 509

- response message 515
- sample 498
- server 509
- signature 501
- signing information 506, 513
- SOAP message 537
- token consumer 511
- token generator 503
- transform 509, 515
- interface definition language 71
- intermediary 182, 222
- internationalization 133
- Internet Engineering Task Force 34
- interoperability 6, 49
 - .NET 201
 - .NET client 434
 - arrays 205
 - definition 196
 - examples 431
 - profiles 196
 - WebSphere 200
- interoperable 9
- intra-application communication 227
- IWeather interface 249

J

J2EE

- application client 306
- client container 106
- Connector Architecture 30
- Deployment 1.1 API 30
- Management 1.0 API 30
- perspective 658

J2EE 1.4 30, 234

J2EE Migration Wizard 557

J2ME Web Services 32

JAAS configuration 551

jaas.config 491, 523

JACL script 721

Java

- API for XML Binding 243
- API for XML Registries 32
- API for XML-based RPC 31, 33, 91
- APIs for WSDL 32, 87
- APIs for XML Messaging 32
- Architecture for XML Binding 33
- Architecture for XML Data Binding 31
- Business Integration 33

- Community Process 34
- JSRs 30
 - Messaging Service 30, 71, 562
- Java 2 security 531
- Java2WSDL 396, 415
 - multiprotocol 420
- JavaServer Faces 235
 - Web services client 291
- JAXB 31, 33
- JAXM 32
- JAXR 32, 153, 235
- JAX-RPC 31, 33, 91
 - basics 92
 - client 92, 94, 323
 - deployment descriptor 109
 - handler 118, 560, 562, 588
 - list 562
 - mapping 99
 - mapping deployment descriptor 110
 - multiprotocol 418
 - programming 95
 - server 93
 - terminology 92
- JB1 33
- JCP 34
- jdbc/weather 331, 383, 388, 709
- JMS 30, 121
 - Web service
 - messaging setup 713
- jms/weatherQ 305, 720
- jms/weatherQCF 305, 717, 720
- jms/WebServicesReplyQCF 719
- JNDI 107
- JSF
 - see JavaServer Faces
- JSR 101 31, 91
- JSR 109 31, 103, 240
- JSR 110 32, 87
- JSR 127 235
- JSR 172 32
- JSR 173 33
- JSR 181 33
- JSR 208 33
- JSR 222 33
- JSR 224 33
- JSR 31 31
- JSR 47 235
- JSR 67 32
- JSR 921 34, 103

JSR 93 32
JTA 235
JUnit 352

K

Kerberos 447, 454
key store 480
 invalid 541
KeyLocator 503
KeyStoreKeyLocator 504
keytool 481
keywords
 confidentiality 464
 encryption 464
 integrity 463
 signing 463

L

language-independent 9
launchpad 651, 654
legacy systems 7
listener port 713
literal
 encoding 54
 XML 54
location transparency 211
logging
 handler 309
 mediation 613
LoggingHandler class 310
LoginModule 450, 462
loosely coupled 10
LTPA token 184, 448, 464
LTPATokenCallbackHandler 487
LTPATokenConsumer 491
LTPATokenGenerator 486

M

manage phase 263
managed
 client 99
 process 378
mapping
 deployment descriptor 117
 JAX-RPC 100
 SOAP 51
marshalling 54

MDB 121, 303, 418
mediation 223, 560, 562, 590
 logging example 613
MediationHandler 590

message

 confidentiality 178
 exchange
 composite 220
 one-way 215
 publish-subscribe 219
 request-response 217
 two-way 216
 workflow 218
 exchange patterns 215
 integrity 178
 manipulation 588
 oriented style 52
 SOAP 41, 44
 validation 429
 WSDL 78
Message Transmission Optimization Mechanism
198, 317
MessageContext 309
message-driven bean 303, 713
message-driven EJB 121
message-level security 176, 447
message-oriented style 42
messaging setup
 JMS 713
Microsoft
 Web services 36
Microsoft SOAP Toolkit 41, 65
migratewsgw 603
MIME 81
 binding 86
 multipart message 316
modular 9
monitor
 cache 638
MTOM 20, 198, 316
multiprotocol binding 418
 client 422
 example 419
mustUnderstand 46

N

NAICS 126, 132
namespaces

- WSDL 76
- WS-Inspection 159
- Network Deployment 132, 151, 379, 563
 - server 234
- nillable primitive 226
- node agent 378
- nonce 465
 - cache 465, 542, 552
 - cluster 542
 - error 542
- NonPromptCallbackHandler 487
- non-repudiation 174, 181

O

- OASIS 35, 124, 178
- observer 219
- one-way message 215
- onion 213
- operation 70
 - overloading 201
- outbound service 561, 575

P

- Page Data view 292
- Page Designer 292
- Palette view 296
- pattern
 - adaptor 223
 - facade 223
 - message exchange 215
 - observer 219
- performance 177
- Performance Monitoring Infrastructure
 - see PMI
- perspective
 - J2EE 658
 - Test 353
 - Web 275, 658
- Perspectives on Web Services 730
- PKCS7CallbackHandler 487
- PkiPathCallbackHandler 487
- pluggable token architecture 462
- PMI 363
 - counters 366
 - enable 363
 - metrics 364
 - Web services 365
- point-to-point security 192

- port 70, 83
 - type
 - WSDL 79
- precompile JSP 386
- preferences 659
- private UDDI registry 148
- processing model 222
- profile 379
 - Application Server 660
 - interoperability 196
 - wizard 660
- programmatic access 10
- programming
 - language 6
 - model 104
 - EJB 113
 - Web 114
 - style 95
- Project Explorer
 - view 275
 - Web services 289
- properties
 - WS-Security 689
- Properties view 294
- protocol 181
 - mapping 223
 - transformation 562
- provider endpoints 718
- proxy
 - class 96
 - server 223
 - authentication 601
 - service 561, 584
 - UDDI 143
- public
 - broker 5
 - key encryption 174
- publisher
 - assertions 126
- publishing 128
- publish-subscribe message 219

Q

- quality of service 209
- queue
 - connection factory 716
- Quick Edit view 294

R

Rational

- Application Developer 237
 - see Application Developer
- ClearCase LT 237
- Software Architect 237
- software development products 236
- Web Developer 236

Rational Product Updater 655

Redbooks Web site 731

- Contact us xxvi

registry 5

- management 131
- UDDI 124

relationships 132

reload interval 387

remote

- procedure call 42
- server 662

replication 134

request security handler 467

request-response 217

requirements 6

response security handler 467

RMI 40

RMI binding 419

RPC 42, 52, 57

- style 84

RPC/encoded 55

RPC/literal 55

- interoperability 201

rule engine 6

run phase 263

S

SAAJ 198, 317

- client 322

SAML 29

sample code 663

SampleTrustedIDEvaluator 492

save_business 148

SAX 62

scalable vector graphics 367

scope 59

SDO 235

Secure Sockets Layer 192

security 7

- company policy 174

- context 455
- context token 454
- developer 446
- enterprise 174
- global 185, 531
- header 183
- message level 176
- overview 174
- point-to-point 192
- role references 112
- timestamp 528
- token 185
- transport level 176, 192
- Web services 175

Security Assertion Markup Language 29

SEI 96, 279

self-contained 9

self-describing 9

serialization 54

serializer 51, 57, 60

server

- confidentiality 521
- configuration 659
- configuration for global security 533
- container responsibilities 116
- deployment descriptor 116
- enable security 531
- identity assertion 495
- integrity 509
- programming model 111
- remote 662
- security token 489
- timestamp 529
- WS-Security 477

Servers view 365, 372, 662

service

- broker 5
- destination name 609
- endpoint interface 96, 105, 108, 279
- gateway 561
- implementation bean 112–113
- inbound 561
- interface 96, 105
 - mapping 110
- locator 96
- outbound 561
- projection 132
- provider 5
- proxy 561

- registry 5
- requestor 6, 78, 88, 124, 127–128, 154, 267
- WSDL 82
- WSIL 162
- WS-Inspection 162
- Service Data Objects 235
- service integration bus 559
 - administrative tasks 566
 - advanced configuration 607
 - authorization 600
 - commands 567
 - configure 569
 - core application 565
 - endpoint listener 571
 - gateway 580
 - HTTPS 601
 - inbound service 573
 - JAX-RPC handler 588
 - mediation 590
 - outbound service 575
 - proxy service 584
 - remote configuration 616
 - troubleshooting 617
 - UDDI reference 577
 - UDDI registry 579
 - weather forecast 610, 714
 - WSDL 578
 - WS-Security 595
- Service Provisioning Markup Language 24
- service-oriented architecture 3
 - characteristics 6
 - components 210
 - concepts 4
 - definition 210
 - requirements 6
- servlet caching 635
- session EJB 113
- setEndpoint method 348
- sg246461code.zip 702
- sg246461solution.zip 702
- SignerCertKeyLocator 504
- simple API for XML 62
- Simple Object Access Protocol
 - see SOAP
- Simple WebSphere Authentication Mechanism 531
- SMTP 40, 62, 192
- SOA
 - see service-oriented architecture
- SOAP 8
 - 1.1 specification 67
 - binding 84
 - body 41, 48
 - Call object 61
 - client API 59
 - communication style 52
 - data model 49
 - deployment 58
 - deployment descriptor 55, 58
 - encoding 42, 54
 - engine
 - WebSphere 64
 - envelope 41
 - error handling 48
 - fault 48
 - fault response 468
 - header 43, 46
 - implementation 56
 - implementations 56
 - intermediary 46
 - mapping 51
 - message 41, 44
 - atomic transaction 333
 - authentication 184, 535
 - confidentiality 538
 - identity assertion 536
 - integrity 185, 189, 537
 - timestamp 539
 - Message Transmission Optimization Mechanism
 - 20
 - Microsoft 41, 65
 - over HTTP 120
 - over JMS 121
 - overview 40
 - processing model 209, 221
 - specification
 - Web site 67
 - standard 16
 - Toolkit 41
 - SOAP Messages with Attachments 198
 - SOAP over JMS 303
 - SOAP with Attachments 20, 316
 - SOAP with Attachments API for Java 198, 317
 - SOAP/HTTP 215
 - SOAP/JMS 215
 - SOAP4J 57, 62, 66
 - soapearenabler tool 392
 - SoapException

- .NET 203
- Software Architect 237
- Software Development Platform 236
- SPML 24
- spoofing 175
- SSL 192
- standards
 - ASAP 14, 20
 - BPEL 25
 - BPEL4WS 14
 - DNS-EPD 19
 - Implementing Enterprise Web Services 15
 - Implementing Enterprise Web Services 1.1 15
 - industry 37
 - J2ME Web Services 15
 - JAXB 15
 - JAXB 2.0 15
 - JAXM 15
 - JAXR 15
 - JAX-RPC 31
 - JAX-RPC 2.0 15
 - JB1 15
 - JSR 101 15
 - JSR 109 15
 - JSR 110 15
 - JSR 172 15
 - JSR 173 15
 - JSR 181 15
 - JSR 208 15
 - JSR 222 15
 - JSR 224 15
 - JSR 31 15
 - JSR 67 15
 - JSR 921 15
 - JSR 93 15
 - Message Transmission Optimization Mechanism 20
 - MTOM 20
 - SAML 15, 29
 - Security Assertion Markup Language 29
 - Service Provisioning Markup Language 24
 - SOAP 14, 16
 - SOAP Messages with Attachments 20
 - SOAP Messages with Attachments (SwA) 14
 - SOAP MTOM 14
 - SPML 14, 24
 - StAX 15
 - SwA 20
 - UDDI 14, 17

- Web services 13
- Web Services for Remote Portlets 29
- Web Services Metadata for Java 15
- WS Reliable Messaging 14
- WS-Addressing 14, 21
- WS-AtomicTransaction 15, 27
- WS-BaseFaults 14
- WS-BaseNotification 14
- WS-BrokeredNotification 14
- WS-BusinessActivity 15, 27
- WS-CAF 14, 25
- WS-CDL 14, 25
- WS-Coordination 15, 26
- WS-Discovery 14, 18
- WSDL 14, 17
- WSDL4J 15
- WSDM 14, 24
- WS-Enumeration 14, 22
- WS-Eventing 14, 21
- WS-Federation 15, 29
- WSIL 14, 18
- WS-Inspection 14
- WS-Manageability 14, 24
- WS-MessageDelivery 14, 22
- WS-MetadataExchange 14, 18
- WS-Notification 14, 21
- WS-Policy 14, 18
- WS-PolicyAssertions 19
- WS-PolicyAttachment 19
- WS-Provisioning 14, 24
- WS-Reliability 14, 22
- WS-ReliableMessaging 23
- WS-ResourceLifetime 14
- WS-ResourceProperties 14
- WS-Resources 14, 23
- WSRP 15, 29
- WS-SecureConversation 15, 28
- WS-Security 15, 28
- WS-SecurityPolicy 15, 28
- WS-ServiceGroup 14
- WS-Topics 14
- WS-Transaction 15, 26
- WS-Transfer 14, 23
- WS-Trust 15, 29
- XML 14, 17
- XML-Encryption 15, 27
- XML-Signature 15, 27
- stateless session EJB 113
- static

- stub 96, 106–107
- Web service 124, 268
- StdinPromptCallbackHandler 487
- Streaming API for XML 33
- style 55
- subscription API 131
- SwA 316
 - interoperability 202
- symmetric key 455
- synchronous 217
- system
 - capacity 177
 - resources 177

T

- tampering 176
- taxonomy 7, 126, 128, 132
- TCP/IP
 - monitoring 62
- TCP/IP Monitor 368
 - configuration 368
 - HTTP header 371
 - preferences 368
 - view 368–369
 - WebSphere Application Server 372
 - WS-I validation 429
- tcpmon 372
 - configuration 373
 - see also TCP/IP Monitor
 - SOAP request 373
 - start 373
- template WSDL 574
- test
 - approaches 339
 - automated 338
 - client 287
 - component 339
 - performance 339
 - registry 134, 136
 - regression 339
 - suite 356
 - implementation class 357
- Test Data Comparator view 360
- Test Data Table view 357
- Test Navigator view 361
- Test perspective 353
- test registry 135
- TestClient.jsp 287

- timestamp 465, 528
 - client 528
 - expired 541
 - generate 529
 - mapping V5/V6 688
 - receive 530
 - response message 530
 - server 529
 - SOAP message 539
- TimingHandler class 312
- Tivoli Performance Viewer 363
- TLS 192
- tModel 125
- TokenConsumerComponent 450, 462
- TokenGeneratorComponent 450, 462
- tools
 - command line 396
- top-down development 263–265
- topology 130
- TPV
 - see Tivoli Performance Viewer
- trace
 - security 540
- transport
 - listener 63
- Transport Layer Security 192
- transport-level security 176, 192
- trust mode 461
- TrustedIIDEvaluator 461
- two-way message 216
- type
 - mapping 110

U

- UBR 124
- UDDI 9, 153
 - administrative console extension 152
 - API 141
 - binding 166
 - binding example 169
 - browser 438
 - business 623
 - Business Registry 124, 134, 301
 - client 143
 - cloud 148
 - command-line tools 630
 - data model 127
 - Explorer 622, 628

- extranet 150
- find 137
- find_binding 147
- find_business 144
- find_service 145
- find_tModel 146
- GUI 622
- history 11
- IBM Web site 154
- inquiry API 131
- internationalization 133
- introduction 123
- library 143
- match service and model 627
- policies 131
- policy 130
- pollution 149
- private registry 148, 619
- proxy 143
- publish 139
- publishing 128, 629
- query 630
- reference 562, 577
- registry 127, 161
 - administrative interface 153
 - API 622
 - GUI 622
 - integrated test environment 620
 - Network Deployment 621
 - structure 125
- replication 134
- security 130
- service 625
- standard 17
- taxonomy 126, 132
- technical model 626
- topology 130
- User Console 622
- Utility Tools 152
- Version 2 132
- Version 3 154
- Web front-end 136
- Web site 154
- WS-Inspection relationship 161
- UDDI.org 36
- UDDI4J 142, 270, 622
- UDDIPublish 396, 411
- UDDIUnpublish 396, 411
- UML
 - Testing Profile 352
- unified resource name
 - see URN
- Universal Description, Discovery, and Integration
 - see UDDI
- universal resource identifier
 - see URI
- Universal Test Client 349
 - EJB testing 351
- unmarshalling 54
- UNSPSC 126, 132
- URI 45
- URN 45, 59
- username token 448–449
- UsernameTokenConsumer 185, 491
- UsernameTokenGenerator 185, 486
- UTC
 - see Universal Test Client
- utility JAR 706
- UUID 138, 148

V

- vertical organization standards 37
- view
 - Console 659
 - Page Data 292
 - Palette 296
 - Project Explorer 275
 - Properties 294
 - Quick Edit 294
 - Servers 365, 372
 - servers 662
 - TCP/IP Monitor 368–369
 - Test Data Comparator 360
 - Test Data Table 357
 - Test Navigator 361
- virtual host 389

W

- W3C 35, 40
- WEATHER
 - database 250, 383
 - setup 708
- Weather class 254
- weather forecast 247
 - .NET client 434
 - application components 248
 - application implementation 252

- attachments 317
- back-end module 250
- bus 612
- business module 249
- caching 638
- command-line tools 398
- data module 249
- data source 253, 382
- information flow 250
- installation 705
- queue 719
- service integration bus 610, 714
- service modules 248
- source code 253
- Weather 249
- WeatherAttachmentClient 321
- WeatherAttachmentGUIClient class 322
- WeatherAttachmentServer 320
- WeatherAttachmentSoapBindingImpl class 320
- WeatherAttachmentWeb 320
- WeatherBase 252
- WeatherClientAppJMS 306
- WeatherClientEAR 291, 306
- WeatherClientJSF 291
- WeatherClientStandAlone 393
- WeatherClientStandalone 290
- WeatherCSApp 436
- WeatherCSApp.exe 442
- WeatherDAO class 256, 330
- WeatherDAOclass 249
- WEATHERDataSourceDB2 383
- WeatherDataSourceDB2 384
- WeatherDataSourceReference 253, 331, 388
- WeatherEJB class 248, 259
- WeatherEJBClientClasses 252
- WeatherEJBJMSSClient 304
- WeatherEJBJMSSClientWeb 304
- WeatherEJBJMSSRouter 303
- WeatherEJBJMSSServer 252, 303, 382, 392
- WeatherEJBMultiprotocolEAR 423
- WeatherEJBMultiprotocolWeb 423
- WeatherEJBRouterWeb 297
- WeatherEJBServer 252, 331, 382, 388
- WeatherForecast class 249, 254
- WeatherJavaBean class 248, 252, 259, 277
- WeatherJavaBean interface 285
- WeatherJavaBean_mapping.xml 283
- WeatherJavaBeanAxisEAR 432
- WeatherJavaBeanAxisWeb 432–433
- WeatherJavaBeanClient 279, 346, 384, 474, 643
- WeatherJavaBeanClientWeb 279, 285, 346, 431, 474, 545, 642
- WeatherJavaBeanProxy class 285
- WeatherJavaBeanServer 252, 382, 384, 431, 474, 639
- WeatherJavaBeanService interface 285
- WeatherJavaBeanServiceInformation class 285
- WeatherJavaBeanServiceLocator class 285
- WeatherJavaBeanSoapBindingImpl class 299, 329
- WeatherJavaBeanSoapBindingStub class 285
- WeatherJavaBeanWeb 252, 275, 474, 638
- WeatherJavaClient class 290
- WeatherJMS class 248
- WeatherJMSSClient class 306
- WeatherMultiServlet class 423
- WeatherPredictor class 250, 258
- WeatherReport JSF page 292
- WeatherServiceAnt 413
- WeatherServiceAntClient 413
- WeatherServiceAntEAR 413
- WeatherTopDownServer 299, 331
- WeatherTopDownServerWeb 299, 331
- WeatherTopDownServerWebClient 331
- Web
 - container
 - programming model 114
 - perspective 275, 658
- Web Developer 236
- Web Service wizard
 - client 289
 - generate client 345
 - JavaBean 275
 - JMS client 306
 - session bean 297
 - skeleton JavaBean 299
 - SOAP over JMS 303
 - with security 543
 - WSDL with attachments 325
- Web services
 - Ant tasks 412
 - Application Developer 238
 - architectures 209
 - asynchronous 217
 - atomic transaction 326
 - attachments 316
 - best practices 225
 - build 263
 - caching 228, 633

- client 267
- client deployment descriptor 109
- code generation 243
- component test 338, 352, 354
- composition 266
 - example 301
- configuration settings 241
- core standards 16
- core technologies 8
- creating clients 288
- DB2 240
- definition 7, 215
- deployment 263, 377, 392
- deployment descriptor 116
- development 261
- enabling 392
- enterprise 103
- Explorer 239
- fine grained 226
- from URL 308
- gateway 222, 559, 580
 - migration 602
- generated files 283
- handler 309
- history 10
- IBM 36
- information sources 12
- introduction 3
- J2EE 104
- JavaBean 275
- JMS 303
- management 263
- Microsoft 36
- migration 212
- monitor 277
- monitoring 363
- onion 213
- organizations 34
- PMI counters 366
- Project Explorer 289
- properties 9
- protocol stack 209, 213
- proxy 281
- references 242
- request monitoring 365
- runtime 200, 263, 279
- runtime environments 240
- sample test JSP 345
- scope 59
- security 11
 - model 181
 - trace 540
- service integration bus 559
- session bean 297
- standards 13, 213
 - online 15
- TCP/IP Monitor 369
- testing 338
- tooling 238
- top-down 299
- Universal Test Client 349
- versus service-oriented architectures 212
- Web Services Choreography Description Language 25
- Web Services Client Editor 557
- Web Services Composite Application Framework 25
- Web Services Description Language
 - see WSDL
- Web Services Distributed Management 23
- Web Services Editor 477
- Web Services Explorer 281, 339, 615
 - edit SOAP message 343
 - example 286
 - SOAP messages 344
 - start 340
 - test example 341
 - UDDI 622
- Web Services for Remote Portlets 29
- Web Services Inspection Language
 - see WSIL
- Web Services Interoperability Organization 35
 - see WS-I
- Web Services Invocation Framework 271
- Web Services Metadata for the Java Platform 33
- Web Services Security UsernameToken Profile 179
- Web Services Security X.509 Certificate Token Profile 179
- webservices.xml 116, 283, 298, 477
- webservicesclient.xml 285
- WebSphere
 - administration 378
 - administrative console 380
 - Application Server 57
 - Network Deployment 132, 151, 379
 - requirements 650
 - Application Server Version 6.0 234

- atomic transaction support 327
- best practices 228
- Business Integration Server Foundation 237
- interoperability 200
- key stores 480
- SOAP Engine 41, 64
- Studio
 - Application Developer 237
 - Application Developer Integration Edition 237
 - Enterprise Developer 237
 - Site Developer 236
- topology 378
- upgrade 380
- Web service support 200
- Web Services Engine 64, 228
- WS-Security support 458
- WinForm.cs 436
- workflow 6, 264
 - message 218
- World Wide Web Consortium 35
- wrap applications 10
- WS Binding page 477
- WS Extension page 476
- WS-Addressing 21, 234
- wsadmin 721
- WS-AT 27
- WS-AtomicTransaction 27, 234–235, 326
- WS-Attachments 316
- WS-Authorization 182
- WS-BA 27
- WS-BaseFaults 23
- WS-BPEL 25
- WS-BusinessActivity 27, 216
- WS-CAF 25
- WS-CDL 25
- WS-COOR 26
- WS-Coordination 26
- wsdeploy tool 387
- WS-Discovery 18
- WSDL 8
 - API 86
 - Application Developer 239
 - attachment document 319
 - binding 70, 81, 164
 - binding example 168
 - conversion 201
 - document 70–71
 - files 75
 - history 11
 - import 202
 - introductory 69
 - message 78
 - operation 70, 79
 - overview 70
 - port 70, 83
 - port type 79
 - publishing 578
 - service 82
 - service integration bus 578
 - specification 429
 - Web site 89
 - standard 17
 - template 574
 - UTF-16 203
 - validation 239, 429
- WSDL2Client 396
 - client 409
 - example 408
- WSDL2Java 396, 416
- WSDL2WebService 396
 - example 405
- WSDL4J 32, 86
- WSDM 23
- WSEE 104
- WS-Enumeration 22
- WS-Eventing 21
- WS-Federation 29, 182
- WS-I 35, 196
 - Attachments Profile 196
 - Basic Profile 44, 47, 92, 196, 234
 - compliance 239
 - compliant 226
 - conformance 199, 202
 - message validation 429
 - profiles 35
 - Simple SOAP Binding Profile 196
 - tools 198
 - validation tools 428
 - Web site 196
- WSIF 271
- WSIL 9, 155, 239, 269
 - binding 164
 - example 168
 - link
 - link 163
 - see also WS-Inspection
 - service 162

- standard 18
- WSIL4J 170
- WS-Inspection 155
 - API 170
 - binding 164
 - definition 161
 - document 157
 - example 159
 - Java API 170
 - link 163
 - namespaces 159
 - overview 157
 - service 162
 - specification
 - Web site 172
 - UDDI relationship 160
- WS-Manageability 24
- WS-MessageDelivery 22
- WS-MetadataExchange 18
- WS-Notification 21, 220
- WS-Policy 18, 181
- WS-PolicyAssertions 19
- WS-PolicyAttachement 19
- WS-Privacy 181
- WS-Provisioning 24
- WS-Reliability 22
- WS-ReliableMessaging 22
- WS-ResourceLifetime 23
- WS-ResourceProperties 23
- WS-Resources 23
- WSRF 23
- WSRP 29
- WSS
 - see Web services security
- WS-SecureConversation 28, 182, 447
- WS-Security 28, 119
 - application testing 534
 - authentication 184
 - client configuration 476
 - configuration
 - files 470
 - forms 692
 - deployment descriptor 469
 - development 474
 - evolution 179
 - example 182
 - extensions 460
 - implementation architecture 467
 - implementation sequence 446

- interoperability 178, 205
- key store 480
- migration 556
- platform-level configuration 472
- predefined properties 689
- road map 181
- sample configurations 543
- see also Web services security
- server configuration 477
- service integration bus 595
- specification 177
- support in Application Server 180
- testing 531
- timestamp 528
- tracing 540
- typical errors 541
- typical scenario 447
- Version 5.x 556
- Web site 194
- WebSphere server 547
- WebSphere Version 6 458
- WS-SecurityKerberos 454
- WS-SecurityKerberos architecture 447
- WS-SecurityPolicy 28
- WS-ServiceGroup 23
- WS-Transaction 26, 235
- WS-Transfer 23
- WS-Trust 29, 181, 447
- WS-TX 26

X

- X.509 180
 - certificate 448
- X509TokenConsumer 188, 191, 491
- X509TokenGenerator 188, 486
- X509TokenKeyLocator 504
- X590CallbackHandler 487
- XMethod 172
- XMI 55
- XML 8, 14
 - encryption 178
 - Web site 194
 - Information Set 88
 - metadata interchange
 - see XMI
 - Protocol Working Group 40
 - schema descriptor 42
 - signature 178

Web site 194
standard 17
structures 227
XML-Encryption 27
XML-Signature 27
XPath expression 463–464
XSD
type 49



Redbooks

WebSphere Version 6 Web Services Handbook Development and Deployment

(1.0" spine)
0.875" <-> 1.498"
460 <-> 788 pages



Redbooks

WebSphere Version 6 Web Services Handbook Development and Deployment

**WebSphere
Application Server V6.0
Rational Application
Developer V6.0**

**Latest Web services
technologies and
standards**

**Best practices and
advanced techniques**

This IBM Redbook describes the concepts of Web services from various perspectives. It presents the major building blocks on which Web services rely. Here, well-defined standards and new concepts are presented and discussed.

While these concepts are described as vendor independent, this book also presents the IBM view and illustrates with suitable demonstration applications how Web services can be implemented using the IBM product portfolio, especially IBM WebSphere Application Server Version 6.0 and IBM Rational Application Developer for WebSphere Software Version 6.0.

This redbook is a rewrite of the redbook *WebSphere Version 5.1 Application Developer 5.1.1 Web Services Handbook*, SG24-6891-01. The new book covers the latest specifications in regard to Web services and Web services security. The new book uses the same weather forecast application as the base for all examples, but updated to Version 6 of the products.

This book is structured into three parts:

- ▶ Part 1 presents the underlying concepts, architectures, and specifications for the use of Web services.
- ▶ Part 2 shows how Web services can be implemented and deployed using the latest IBM products. Here, we introduce the weather forecast application, which we use in many ways to demonstrate these concepts and features.
- ▶ Part 3 shows some advanced techniques, such as Web services security, interoperability, and the service integration bus.

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