

Web Services Wizardry with WebSphere Studio Application Developer

Creating dynamic e-business with Web services

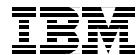
Using the IBM toolset for Web services

Introduction to WebSphere Studio



Ueli Wahli
Mark Tomlinson
Olaf Zimmermann
Wouter Deruyck
Denise Hendriks

Redbooks



International Technical Support Organization

**Web Services Wizardry with WebSphere Studio
Application Developer**

April 2002

Take Note! Before using this information and the product it supports, be sure to read the general information in “Special notices” on page 583.

First Edition (April 2002)

This edition applies to WebSphere Studio Application Developer Version 4.0.2 and WebSphere Application Server Version 4.0.1 for use with the Windows 2000 and Windows NT operating system.

Comments may be addressed to:

IBM Corporation, International Technical Support Organization
Dept. QXXE Building 80-E2
650 Harry Road
San Jose, California 95120-6099

When you send information to IBM, you grant IBM a non-exclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 2002. All rights reserved.

Note to U.S Government Users – Documentation related to restricted rights – Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

Contents

Preface	xvii
The team that wrote this redbook.....	xviii
Special notice.....	xx
IBM trademarks	xx
Comments welcome.....	xx
Part 1. Introduction	1
Chapter 1. Auto parts sample application	3
Use cases for the application	4
Local dealership inquiry	4
Inquiry on vehicle manufacturer system	5
Dynamic inquiry on other parts manufacturers	5
Cross dealership inquiry enablement	6
System design	6
Stage 1: Local dealership inquiry	7
Stage 2: Inquiry on vehicle manufacturer system.....	7
Stage 3: Dynamic inquiry on other manufacturers	8
Stage 4: Cross dealership inquiry enablement	10
System diagram	11
Data model	12
Products used for the application	13
Summary	13
Part 2. WebSphere Studio Application Developer	15
Chapter 2. Application Developer overview	17
Product	18
WebSphere Studio Workbench	18
Workbench window	20
Perspectives	20
Views.....	22
Navigator view.....	22
Editors.....	22
Outline view.....	23
Properties view	23
Tasks view	23
Customizing perspectives.....	24
Web perspective	25

Java perspective	25
J2EE perspective	27
Data perspective	28
XML perspective	29
Server perspective	30
Debug perspective	31
Help perspective	32
Projects	33
Java project	33
EAR project	38
Web project	40
EJB project	44
Server project	47
Application Developer internal system files	52
Summary	53
Chapter 3. Web development with Application Developer	55
Solution outline for auto parts sample Stage 1	56
Class and sequence diagrams	57
Preparing for development	58
Creating a new workspace	58
Project configuration	58
Viewing the default Web project files	60
Creating the ITSOWSAD sample database	61
Creating a suitable Java package for the generated servlets	62
Building the application	62
Generating the basic Web application	63
Designing the forms and generation options	68
Investigating the generated types	71
Deployment descriptor web.xml file	73
Changing the background	75
Adding support for displaying the image	75
Modifications required if using code prior to Version 4.0.2	80
Deploying and testing the Web application	81
Creating a server project	82
Creating a server instance and configuration	82
Associating the project with a server configuration	83
Adding the JDBC data source	84
Executing the Web application	86
Summary	88
Chapter 4. XML support in Application Developer	91
An XML primer	92

Background	92
XML base concepts	93
XML namespaces	97
XML schema	98
Processing XML	101
XML parser	102
XML processor	104
Summary	104
XML tool support in Application Developer	105
Solution outline for example	106
Class diagram	107
Sequence diagram	108
Preparing for development	109
Generating an XML file and an XML schema from SQL	111
Browsing and editing XML and XSD files	113
Editing an XML file	113
Edit an XSD file	114
Validate the XML file	115
Editing XML schema and data files	115
Modifying the XML to match the new schema	120
Editing the XML directly	120
Creating an XML to XML mapping	120
XML processing	125
Transforming XML to XML with a style sheet	125
Transforming XML to HTML with a style sheet	127
Parsing an XML file	128
Creating an XML file from a Java program	131
Outlook	131
Chapter 5. EJB development with Application Developer	133
Solution outline for auto parts sample Stage 2a	134
Class diagrams	135
Sequence diagram	136
Preparing for development	137
Project configuration	137
EJB mapping approaches review	137
Building the entity EJBs	138
Creating the entity EJBs	138
Investigating the generated files	141
EJB classes review	144
Generated EJB metadata	145
Generated mapping	145
Adding getter methods for the key fields	147

Defining getter methods as read-only	148
Binding the EJBs to a JDBC data source	149
Validating the project	150
Generating the deployed code	151
Testing the entity EJBs	152
Associating the EAR project with a server configuration	152
Executing the EJB test client	153
Developing the session EJB facade	158
Creating the session EJB	158
Completing the session entity facade	160
Promoting the new methods to the EJB remote interface	164
Defining the EJB references	165
Validating the project	166
Generating the deployed code	166
Testing the session EJB	167
Summary	168
Chapter 6. Deployment of Web and EJB applications to WebSphere	169
Differences between WebSphere AEs and AE	170
Deployment to WebSphere AEs	170
Exporting the EAR files	171
Starting the AEs server	172
Starting the Administrator's Console	173
Creating the JDBC driver and data source	174
Installing the Almaden Autos enterprise application	176
Installing the Mighty Motors enterprise application	178
Saving the configuration	181
Stopping the AEs server	182
Deploying an enterprise application with SEAppInstall	182
Deploying the EJB test client	183
Verifying the enterprise applications	185
Regenerating the plug-in	186
Testing the deployed applications in AEs	188
Testing using the embedded HTTP server	189
Testing using the IBM HTTP Server (AEs)	189
Deployment to WebSphere AE	190
Starting the Admin Server	190
Starting the Administrator's Console	190
Creating the JDBC driver and data source	191
Creating an application server	192
Installing the Almaden Autos enterprise application	194
Installing the Mighty Motors enterprise application	195
Starting the WebSphere AE application server	198

Testing the deployed applications in WebSphere AE	198
Testing using the embedded HTTP server	199
Testing using the IBM HTTP Server (AE)	200
Remote unit testing from Application Developer	200
Summary	205
Chapter 7. Working in a team	207
Team overview	208
An extensible architecture	208
Differences between CVS and ClearCase	209
Workspace	210
Local history	210
Configuring multiple workspaces	211
Executing Application Developer with the binaries on the server	211
Team terminology introduction	212
Streams	212
Optimistic concurrency model	212
Branching	213
Versions	214
Terminology matrix	214
Team perspective	215
Perspective overview	215
Connecting to a CVS repository	215
Viewing the projects in the HEAD stream	217
Browsing the resource history	217
Comparing two versions of a file in the repository	218
Adding a project to the workspace	220
Changing project types	221
Configuring the project	221
Installing a local CVS repository	222
Downloading and installing CVS	222
Creating a new repository	222
Creating a Windows service	223
Creating new users	223
Team development simulation	224
Configuration	224
Sequential development scenario	224
Parallel development in a single stream scenario	227
Branching using multiple streams scenario	231
Additional team topics	235
Determining which files are managed	235
Backing up the CVS repository	235
Repository management	236

Implementing security	236
Build scripts	236
Managing class paths	237
Using CVS macros	238
Watching a file for changes	238
Other CVS commands	238
Part 3. Web services	239
Chapter 8. Web services overview and architecture	241
Motivation	242
Technical foundation	242
Introduction to the service oriented architecture	245
Service roles	245
SOA stack	246
First examples	248
Implementations of the SOA	252
What is next	253
Developing Web services	254
Development steps	254
Development strategies for provider and requestor	255
Service life cycle	258
What is next	260
An introduction to SOAP	261
Overview	261
Anatomy of a SOAP message	262
URN	263
Envelope	264
Advanced concepts	266
Apache SOAP implementation	270
SOAP summary	274
WSDL primer	277
Overview	277
Anatomy of a WSDL document	279
Service implementation document	283
Service interface document	284
Bindings	286
WSDL API	290
WSDL summary	291
UDDI overview	293
UDDI registry structure	293
Identification and categorization	297
UDDI API overview	298

Anatomy of a UDDI registry	300
Existing registries	300
UDDI summary	301
Summary	304
Chapter 9. Product support for Web services	307
WebSphere Studio Application Developer	308
Web service wizard	308
Web service client wizard	310
Web service skeleton JavaBean wizard	311
Web service DADX group configuration wizard	312
UDDI browser (import, export)	313
Web services toolkit (WSTK)	313
Relations between WSTK, Application Developer, and other IBM tools ..	313
WSTK architecture	313
Design-time components	314
Runtime components	315
Other components and functionality	316
WebSphere Studio (classic) Version 4	316
DB2	317
Versata	317
Versata XML and Web services toolkit	318
Exposing rules in the VLS as a Web service	318
Calling a Web service from business rules	319
Summary	320
Chapter 10. Static Web services	321
Solution outline for auto parts sample Stage 2b	322
Class and sequence diagrams	323
Preparing for development	324
Creating a new ITSOMightyWeb project	324
Web service types	325
Creating the Mighty Motors InquireParts Web service	326
Creating the JavaBean wrapper	326
Creating the package for the JavaBean	326
Creating the InquireParts JavaBean	327
Implementing the InquireParts JavaBean	327
Testing the InquireParts JavaBean	332
Creating the Web service	334
Using the Web service wizard	336
Configuring the Web service identity	337
Web service scope	338
Enable SOAP security	338

Selecting JavaBean methods and encoding styles	338
XML /Java mappings	340
Defining Java to XML mappings	341
Proxy generation	342
Defining the XML to Java mappings	343
Verifying the SOAP bindings	344
Web service test client	344
Generating a sample client	344
Publishing the Web service	345
Investigating the generated files	345
Working with the previously created XML schema	346
Web service WSDL files	347
Changing the service implementation	347
Changing the service interface	348
Generating SOAP deployment descriptors	350
SOAP router servlets	351
Viewing the deployed Web services	352
Web service client proxy	354
Web service sample client	356
Using the TCP/IP Monitoring Server to view message contents	358
Creating the Almaden Autos Web service client	363
Copying the WSDL service implementation	364
Creating the Web service client	364
Examining the generated client files	365
Testing the Web service requestor	365
Building the client application	366
Creating the XSL style sheet	366
Creating the new servlet	368
Completing the servlet	368
Linking the new servlet into the Web application	370
Testing the Web service client	373
Considerations when enhancing this sample	374
Creating a Web service from a session EJB	374
Create the session EJB Web service using the wizard	375
Testing the session EJB Web service	375
Summary	377
Chapter 11. Dynamic Web services	379
Solution outline for auto parts sample Stage 3	380
Class diagram	381
Sequence diagram	382
Preparing for development	383
Installing the Plenty Parts Web service	384

Static Web services (revisited)	384
Working with the Application Developer UDDI browser	386
Publishing a UDDI business entity	387
Exporting the service WSDL to the UDDI registry	390
Using a Web browser to access the Test Registry	393
Importing the service WSDL from the UDDI registry	394
Unpublishing and updating entries in the UDDI registry	396
Generating a SOAP client proxy from a WSDL file	397
Working with the UDDI APIs	398
Finding the service providers in the UDDI registry	399
Updating the Almaden Autos Web application	405
UDDI lookup servlet	405
Creating the XSL style sheet	407
Updating the existing Web application	408
Unit testing	408
Deploying the enhanced application to WebSphere 4.0	410
Summary	411
Chapter 12. Composed Web services	413
Solution outline for auto parts sample Stage 4	414
Class and interaction diagrams	415
Preparing for development	417
Creating a new dealer	418
Creating the Almaden Autos Web service	418
Creating the JavaBean skeleton	419
Investigating the generated files	421
Implementing the JavaBean skeleton	423
Testing the Web service	425
Creating the Santa Cruz Sports Cars Web application	426
Copying the WSDL file to the ITSOSantaWeb project	426
Starting the server	426
Creating the Web service client	426
Testing the proxy	427
Adding the XSL style sheet	427
Creating the servlet to invoke the Web service	427
Linking the servlet into the Web application	429
Testing the Santa Cruz Sports Cars Web application	429
Summary	431
Chapter 13. Deployment of Web services to WebSphere	433
Preparing for deployment	434
URL considerations	434
Export the enterprise applications	435

Export EAR files	435
Add the SOAP admin client.....	435
Deployment to WebSphere AEs	438
Uninstall the previous enterprise applications.....	438
Install the enterprise applications	439
Define port 8080 for AEs.....	439
Regenerate the plug-in and save the configuration	441
Add the necessary JAR files to the AEs class path	441
Test the applications with AEs	441
Deployment to WebSphere AE	443
Uninstall the previous enterprise applications.....	443
Install the enterprise applications	443
Define port 8080 for AE.....	444
Add the necessary JAR files to the AE class path	445
Test the applications with AE	445
SOAPEAREnabler tool	445
Summary	446
Chapter 14. Web services advanced topics	447
Advanced SOAP programming	448
Programmatic deployment.....	448
Encodings and type mapping alternatives	448
Overview of encodings and mappings	449
Implementing a Web service with a JavaBean as parameter.....	451
Create a custom mapping.....	458
Messages with XML element parts and other parameters	464
Summary.....	465
Message-oriented communication.....	466
Prepare for development.....	469
Accessing the message service from the Web service client	471
Test the message service	477
Summary.....	478
Additional Web service support.....	479
Creating a Web service from a DADX file.....	479
Generating a Web service from an URL	488
Summary.....	488
Advanced UDDI topics	489
UDDI operator cloud	489
e-marketplace UDDI	490
Portal UDDI.....	491
Partner catalog UDDI	492
Internal enterprise application integration UDDI	493
Test bed UDDI	494

Managing the relationships between UDDI nodes	494
IBM WebSphere UDDI Registry	496
Business process management with Web services	497
IBM WSFL proposal	497
WSFL flow model	499
Web services and MQSeries Workflow	503
Web services and MQSeries Integrator	507
Summary	511
Chapter 15. Architecture and design considerations	513
Architecture	514
What are common usage scenarios for Web services?	514
What is the appropriate architectural positioning of Web services?	515
What is the impact on the operational model?	518
How do Web services relate to similar technologies?	519
What about security?	520
What about management and quality of service?	521
What about performance?	522
What about interoperability?	523
Are there any gaps in the current version of the SOA?	523
How do I get started in my project?	524
Design	525
Early best practices	527
Troubleshooting	530
Chapter 16. IBM's jStart program	531
jStart provides the edge	532
About the program	532
Where competitive advantage begins	533
jStart engagement model	533
Take the step	534
Part 4. Appendixes	535
Appendix A. Product installation and configuration	537
DB2 Version 7.2	538
Create sample database	538
JDBC Version 2	538
WebSphere Application Server Advanced Single Server	539
WebSphere Application Server Advanced	539
WebSphere Studio Application Developer	540
IBM Agent Controller	540
IBM WebSphere UDDI Registry	541
Define and load the ITSOWSAD database	542

Sample data	542
Appendix B. State transitions for resources in Application Developer	545
Stream lifecycle.....	546
Project lifecycle.....	548
File lifecycle	553
Appendix C. Quiz answers	559
Introduction.....	560
Application Developer overview.....	560
Web development with Application Developer.....	561
XML support in Application Developer.....	561
EJB development with Application Developer	562
Deployment of Web and EJB applications.....	563
Working in a team	563
Web services architecture and overview	564
Product support for Web services	565
Static Web services	565
Dynamic Web services.....	566
Composed Web services	566
Deployment of Web services	567
Web services advanced topics.....	567
Appendix D. Additional material	569
Locating the Web material	569
Using the Web material	570
System requirements for downloading the Web material	570
How to use the Web material	570
Directories.....	571
Instructions	572
Setup directory	572
WSADWeb directory	573
WSADXML directory	573
WSADEJB directory	573
WSADDDeploy directory	573
WSStatic directory.....	574
WSDynamic directory	574
WSComposed directory	575
WSDeploy directory	575
WSEnhanced directory	576
Related publications	577
IBM Redbooks	577
Other resources	578

Referenced Web sites	580
How to get IBM Redbooks	581
IBM Redbooks collections.....	581
Special notices	583
Abbreviations and acronyms	585
Index	587

Preface

This IBM Redbook explores the new WebSphere Studio Application Developer for J2EE application development and Web services. The WebSphere Studio Application Developer basic tooling and team environment is presented along with the development and deployment of Web Applications (JSPs and servlets), XML, data, EJBs, and Web services.

WebSphere Studio Application Developer is the new IBM tool for Java development for client and server applications. It provides a Java integrated development environment (IDE) that is designed to provide rapid development for J2EE-based applications. It is well integrated with WebSphere Application Server Version 4, and provides a built-in single server that can be used for testing of J2EE applications.

Web services are a new breed of Web applications. Web services are self-contained, self-describing, modular applications that can be published, located, and invoked across the Web. Web services perform callable functions that can be anything from a simple request to complicated business processes. Once a Web service is deployed and registered, other applications can discover and invoke the deployed service. The foundation for Web services are the simple object access protocol (SOAP), the Web services description language (WSDL), and the Universal Description, Discovery, and Integration (UDDI) registry.

This redbook consists of three parts: An introduction of the sample auto parts application that is used throughout the book; J2EE development and deployment with WebSphere Studio Application Developer; and Web services technology, along with the development and deployment of Web services in WebSphere Studio Application Developer.

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, San Jose Center.



Ueli Wahli is a consultant IT specialist at the IBM International Technical Support Organization in San Jose, California. Before joining the ITSO eighteen years ago, Ueli worked in technical support at IBM Switzerland. He writes extensively and teaches IBM classes worldwide on application development, object technology, VisualAge for Java, WebSphere Studio, and WebSphere Application Server products. Ueli holds a degree in Mathematics from the [Swiss Federal Institute of Technology](#).

Mark Tomlinson is a member of the IBM Application and Integration Middleware technical sales team covering northern Europe, and is based in London, England. After joining the company in 1995, he worked for the IBM Insurance Business, IBM Global Services, IBM Global Network, and now in the IBM Software Group. His last three years have involved presenting and demonstrating various VisualAge and WebSphere technologies to IBM customers. He holds a degree in Physics from Warwick University, England.

Olaf Zimmermann is a consulting IT architect at IBM Global Services, BIS e-business Integration Services, Heidelberg, Germany. He has been working with IBM since 1994, in various areas such as product development, technical pre-sales consulting, teaching, and systems integration. His areas of expertise

include distributed computing and middleware in general, as well as Enterprise Application Integration (EAI), XML, and Java 2 Enterprise Edition (J2EE). He holds a degree in Computer Science from the Technical University in Braunschweig, Germany.

Wouter Deruyck is a consultant for the EMEA AIM Partner Technical Enablement Team based in La Hulpe, Belgium. He has been working for IBM since 1999. He started in IBM Global Services as an IT specialist, but he then found a new challenge in the software group. He has two years of experience in object technology. He holds a degree in Computer Science from the University of Leuven.

Denise Hendriks is a managing director and WebSphere Architect with Perficient, Inc. Denise has a BS in Physics and a BS degree in Computer Science, followed by a MS in Electrical and Computer Engineering. She joined IBM and spent ten years as a lead developer for VisualAge and WebSphere in various capacities. She has recently joined Perficient, Inc., where she makes extensive use of her skills as a consultant in WebSphere and J2EE technologies.

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

Tom Bellwood
UDDI Development Lead, Emerging Internet Technologies, Austin, TX

Peter Brittenham
Web Services Toolkit Architect, Emerging Technologies, Raleigh, NC

Barbara McKee
AIM Architecture, Software Group, Austin, TX

Frank Mueller
IBM Global Services, J2EE Architect, e-business Integration, Heidelberg, Germany

Dan Gisolfi
Engagement Manager/Architect, jStart Emerging Technologies, Somers, NY

Deborah Cottingham
International Technical Support Organization, San Jose Center

Special notice

This publication is intended to help Java developers create client and server applications on the WebSphere platform, including the creation and usage of Web services. The information in this publication is not intended as the specification of any programming interfaces that are provided by WebSphere Studio Application Developer. See the PUBLICATIONS section of the IBM Programming Announcement for WebSphere Studio Application Developer for more information about what publications are considered to be product documentation.

IBM trademarks

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

e (logo)® 
Redbooks Logo 
CICS®
IBM Registry™
NetRexx™
Planet Tivoli®
Tivoli Enterprise™
WebSphere®
Notes®

IBM®
AIX®
DB2®
IMS™
NetView®
S/390®
TME®
z/OS™

Redbooks™
AlphaWorks®
IBM Global Network®
MQSeries®
OS/390®
Tivoli®
VisualAge®
Lotus®

Comments welcome

Your comments are important to us!

We want our IBM 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 Internet note to:
redbook@us.ibm.com
- ▶ Mail your comments to the address on page ii.



Part 1

Introduction

In part one we introduce the sample application that is used throughout the redbook.

The auto parts sample application starts with parts management from a dealer point-of-view, then evolves to include the manufacturer and other parts providers.

Web services are used to connect the dealer information system to the manufacturer information system in a static predetermined way. When other parts providers are added, the application becomes more dynamic in nature. In the last stage, the dealer itself provides a Web service to other dealerships.



Auto parts sample application

This chapter contains an overview of the sample application we are building in this redbook. The example is based on an automobile dealership's parts department inventory system.

The application will grow in a number of stages, each demonstrating different aspects of IBM's WebSphere Studio Application Developer (Application Developer) and the Web services support in WebSphere Application Server Version 4.0.

For the time being, we introduce the business functionality aspects of the stages only. We will discuss the technical aspects in a subsequent section of this chapter.

The stages of the application are ordered as follows:

1. *Local dealership inquiry*—Develop a simple Web application that enables mechanics to browse the parts currently in the dealerships inventory.
2. *Inquiry on vehicle manufacturer's system*—The dealership extends its current inventory application to search the manufacturer's national warehouse if they do not have a part in stock locally. The search operation is implemented as a Web service; there is a static binding between the dealer and the vehicle manufacturer.

3. *Dynamic inquiry on other parts manufacturers*—Dissatisfied with the number of times the national parts depot is also out of stock, the dealership implements a system that searches the Internet and finds if any other third party parts manufacturers have the part in stock (dynamic Web service).
4. *Cross dealership inquiry enablement*—Finally, the network of local dealerships decide to integrate their own parts inventory system to allow them to search each others inventories (composed Web service).

Use cases for the application

The following sections show the use cases that apply to the sample application. Each stage is discussed separately. The basic implementation is to allow mechanics in the dealership to look up parts in various inventories. As the application develops through its stages, each stage adds another option for the mechanic to look up parts in the inventory.

Local dealership inquiry

The first stage of the application is the base application, a classic list and details application. A mechanic can look up parts in the dealership's inventory. The mechanic enters the desired part number into a browser and is presented with a list of the parts in the inventory matching the requested part number. The mechanic can then ask for the details of a part selected in the list.

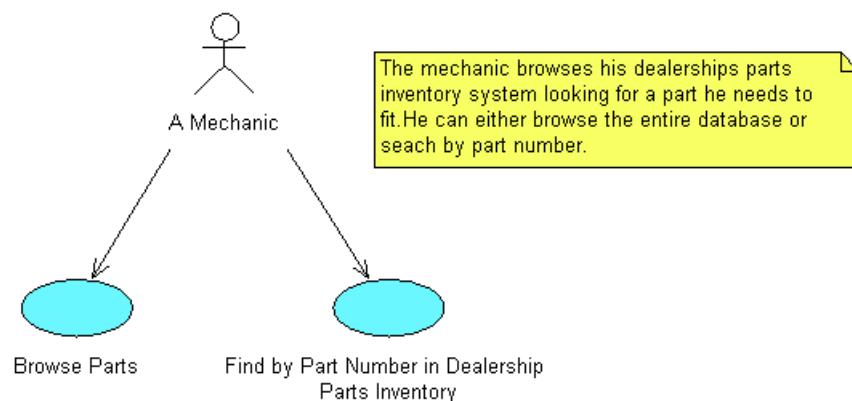


Figure 1-1 Use case: Local dealership inventory inquiry

Inquiry on vehicle manufacturer system

In the second stage of the application, the mechanic now has another option for looking up parts. In this stage, if the mechanic does not find any parts in the local inventory, he is presented with the option of looking for the part in the vehicle manufacturer's inventory.

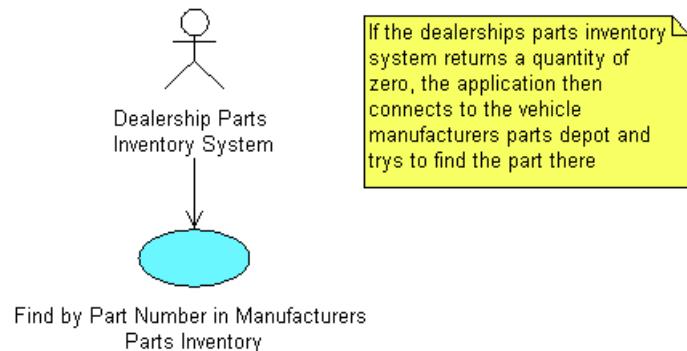


Figure 1-2 Use case: Dealership inquiry of manufacturer's inventory

Dynamic inquiry on other parts manufacturers

The third stage of the application provides the mechanic with another option. In the case that no parts are found in the local inventory, the mechanic now has the option to search all inventory systems registered by the Auto Parts Association. This includes the vehicle manufacturer and other parts manufacturers.

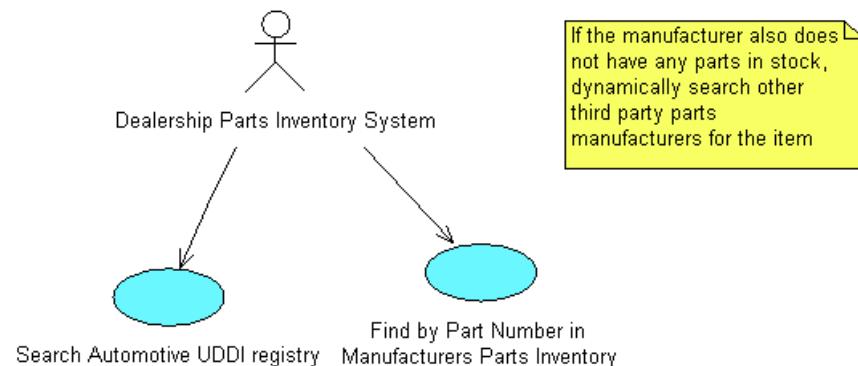


Figure 1-3 Use case: Dealership dynamically determines the inventory systems

Cross dealership inquiry enablement

The fourth and final stage of the application integrates the local dealerships. The mechanics now have the option of searching other dealerships. The dealership's inventory service includes searching their own local inventory, and in turn, all vehicle manufacturer inventories that are available to the dealer.

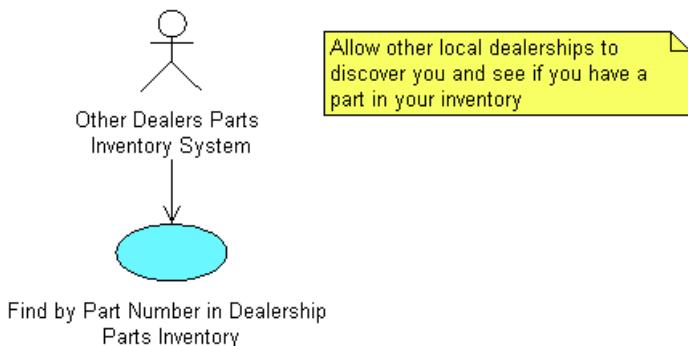


Figure 1-4 Use case: Dealerships provide their inventory services to each other

System design

The sample application described in the previous section will be implemented using IBM WebSphere Studio Application Developer. Emphasis is on these Application Developer tools:

- ▶ Java, Web, XML, and EJB development for the implementations of the original dealer and manufacturer applications
- ▶ Web services development:
 - WSDL generation from an existing JavaBean
(WSDL = Web services description language)
 - Generation of a Java stub implementation from a WSDL document
 - Deployment of Web service into the Application Developer WebSphere test environment
- ▶ Web services client development:
 - Generation of a client side SOAP proxy from a WSDL document
(SOAP = simple object access protocol)
 - Generation of a test client for a Web Service (from WSDL)
- ▶ UDDI browser (UDDI = universal description, discovery, and integration):
 - Export operation (publish)
 - Import operation (find)

Stage 1: Local dealership inquiry

The implementation of Stage 1 of the auto parts system is a standard J2EE Web application. We develop the **Almaden Autos** local dealership Web application, which we continue to enhance throughout the development process. As our application grows in sophistication, search functionality will be added to the Almaden Autos dealership application.

Using the Database Web pages wizard we develop a Web application with the following characteristics:

- ▶ Servlets, JSPs, HTML to structure presentation and control layer
 - MVC pattern
 - no EJBs (EJB development is shown on the vehicle manufacturer system)
- ▶ DB2 database
 - JDBC 2.0 access

Figure 1-5 shows the operational model for Stage 1 of the auto parts system.

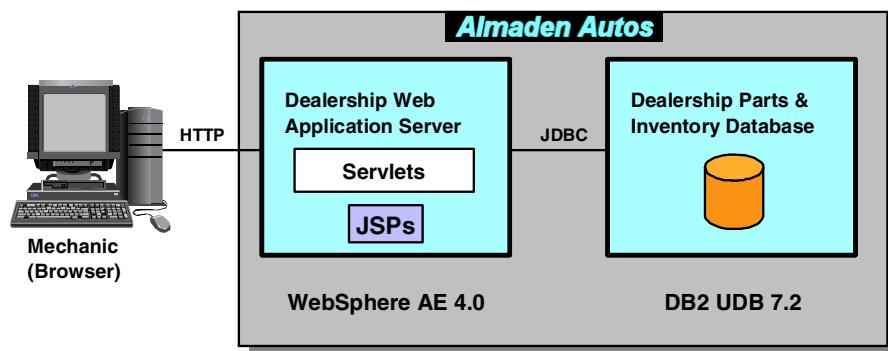


Figure 1-5 Stage 1: Web application

Stage 2: Inquiry on vehicle manufacturer system

The dealership and manufacturer agree to define and use a Web service to implement the use cases for this stage, allowing the dealer to electronically inquire about the inventory levels in the national warehouse of the manufacturer.

The dealership takes the role of the service requestor; the manufacturer is the service provider. There is a fixed, pre-configured relationship between the two of them; hence, the upper layers of the Web services protocol stack (UDDI for example) are not yet involved.

Using the Application Developer Web service wizard, the WSDL definition of the Web service is derived from the already existing Java interface, to the manufacturer's part inventory system. A standards body in the auto industry approves the definition for parts inventory inquiries that are jointly submitted by the manufacturer and the dealership.

The vehicle manufacturer, **Mighty Motors**, implements and deploys the approved Web service; the dealer, Almaden Autos, enhances its local inquiry system from Stage 1 to allow the mechanic to optionally extend a part search. This functionality is implemented as a separate button on the mechanic's browser GUI. A SOAP client is added to the Stage 1 Web application. This client is implemented using the SOAP client proxy that is generated by the Application Developer Web service client wizard.

Figure 1-6 shows the operational model for Stage 2 of the auto parts system.

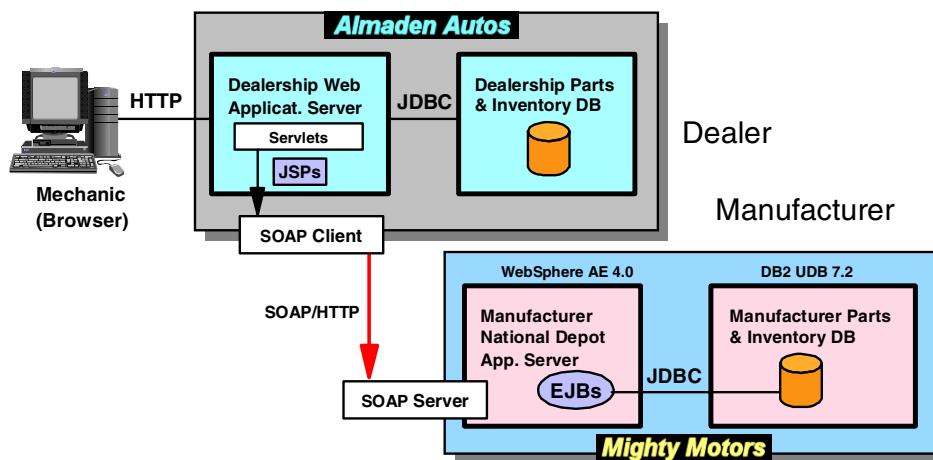


Figure 1-6 Stage 2: Statically bound Web service invocation

Stage 3: Dynamic inquiry on other manufacturers

In this stage, we add a parts manufacturer, **Plenty Parts**. The part manufacturers also implement the Web service; they become additional service providers. We use the Application Developer Web service wizard to generate a Java server side stub from the WSDL document.

We implement and deploy the Web service (the part manufacturer implementation is different from the implementation of the vehicle manufacturer in stage 2). Additionally, we publish the Web service interface and at least two implementations to a UDDI registry (we show how to use both the IBM Test Registry as well as a stand-alone registry).

The **Auto Parts Association**, has approved the services of Mighty Motors and Plenty Parts for publication.

We enhance the dealership Web application, Almaden Autos, once again. Dynamic inquiry is added as an additional option on the mechanic's GUI. The dealership Web application, acting as a service requestor, dynamically queries the UDDI registry to locate all the services provided by the Auto Parts Association. The dealership Web application then invokes the inquiry Web service of each service provider discovered in the UDDI registry to locate the desired auto part.

The UDDI operations (publish and find) required in this stage are performed using the UDDI browser that is part of Application Developer. The find operations also need to be executed programmatically by the Web application.

Figure 1-7 shows the operational model for Stage 3 of the auto parts system.

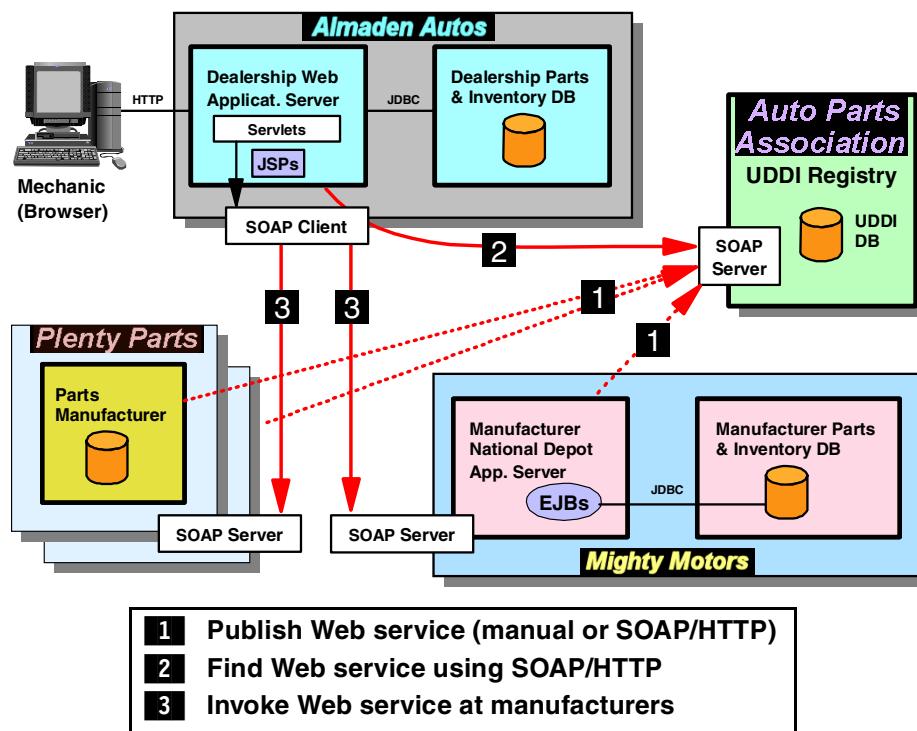


Figure 1-7 Stage 3: Dynamic lookup of Web services in the UDDI registry

Stage 4: Cross dealership inquiry enablement

In the final stage, a new dealership, **Santa Cruz Sports Cars**, opens up. To help them get up and going, the Almaden Autos dealership decides to provide them an inventory service. Almaden Auto's inventory service looks in its own local inventory, and if the desired part is not found, it looks (on behalf of Santa Cruz Sports Cars) in the parts manufacturers (Mighty Motors) inventory using the Web service provided by Mighty Motors.

To implement the desired functionality, the Almaden Autos dealership must create and publish a Web services interface to their own internal systems. This Web service must in turn wrap their already existing invocation of the Mighty Motors Web service. Hence, the dealership now becomes a hybrid service, or a provider and requestor.

Figure 1-8 shows the operational model for stage 4 of the auto parts system.

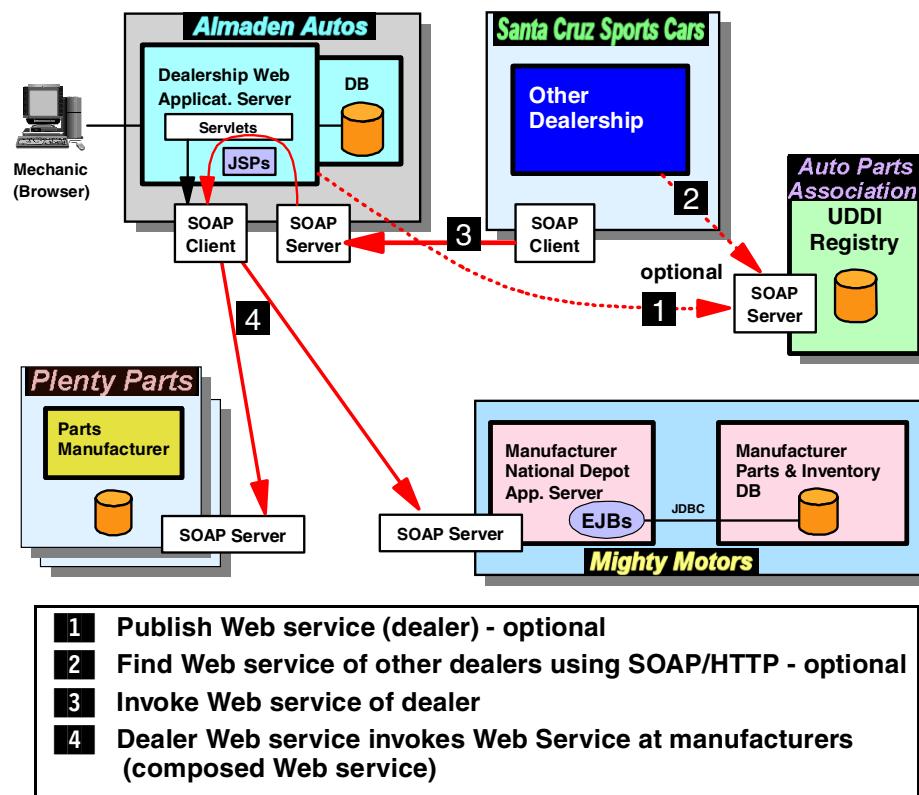


Figure 1-8 Stage 4: Dealership provides composed Web service

System diagram

Figure 1-9 shows the conceptual system diagram of the components involved.

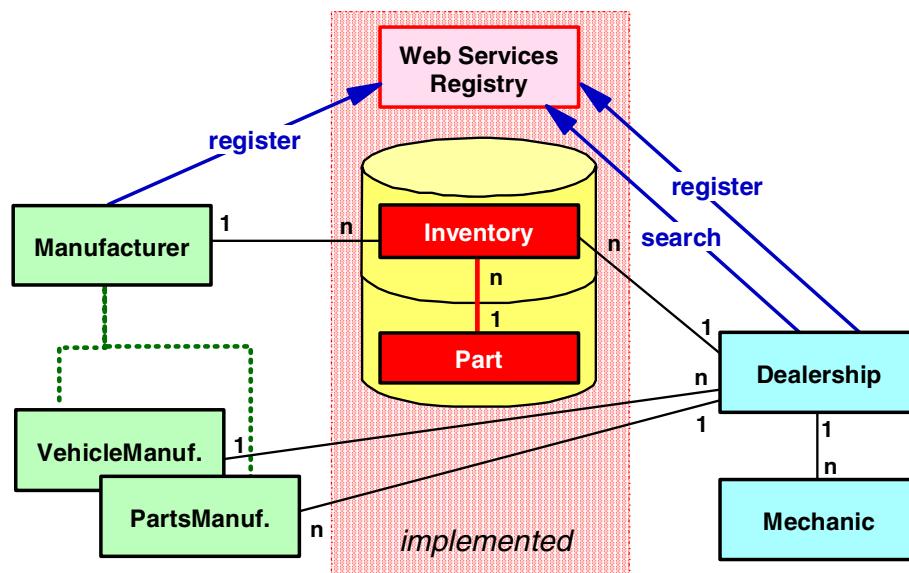


Figure 1-9 System diagram

There are two kind of manufacturers, the vehicle manufacturer and independent parts manufacturers. Vehicle manufacturers have many dealerships that sell their cars, and a dealership does business with many parts manufacturers. A dealership employs many mechanics.

Dealerships and manufacturers have many inventory records. Multiple inventory records may exist for a part; a manufacturer may have multiple warehouses across the country for storage of parts.

Web services are registered by manufacturers and dealerships in a UDDI registry. Dealerships search the registry for implementations of the inventory Web service.

Our sample system implements a database with parts and inventory tables and we use the UDDI registry for Web service registration and search.

Data model

The auto parts application is based on a very simple data model. All data is stored in one database for simplicity.

Each participant in the application, Almaden Autos, Mighty Motors, Plenty Parts, and Santa Cruz Sports Cars has a set of two tables, a parts table and an inventory table. The inventory table points to the parts table though a foreign key.

Figure 1-10 shows the implementation of the tables.

PARTS				
Dealer: AAPARTS, SSPARTS				
Manufacturer: MMPARTS, PPPARTS				
char(10)	char(30)	varchar(100)	double	varchar(100)
partNumber	name	description	weight	image_URL
M100000001	CR-MIRROR-L-01	Large mirror left...	10.5	mirror01.gif

INVENTORY					
Dealer: AAINVENTORY, SSINVENTORY					
Manufacturer: MMINVENTORY, PPINVENTORY					
bigint	char(10)	integer dec(10,2)	char(2)	varchar(100)	
itemNumber	partNumber	quantity	cost	shelf	location
21000002	M100000001	10	89.99	2A	AA - Almaden

Figure 1-10 Auto parts application database implementation

The prefixes stand for Almaden Autos (AA), Mighty Motors (MM), Plenty Parts (PP), and Santa Cruz Sports Cars (SS).

A number of different data types were used in the tables for an illustration of functionality in the products. Our choices may not be appropriate for a real production application.

A command stream is provided to define the ITSOWSAD database with the tables and for loading sample data. Refer to “Define and load the ITSOWSAD database” on page 542 for instructions.

Products used for the application

The sample application was implemented using these products:

- ▶ DB2 UDB Version 7.2
- ▶ WebSphere Application Server Advanced Edition (full and single server)
- ▶ WebSphere Studio Application Developer (beta code, Version 4.0, and Version 4.0.2)
- ▶ WebSphere UDDI Registry, a stand-alone UDDI registry

See Appendix A, “Product installation and configuration” on page 537 for installation instructions.

Summary

In this chapter we introduced the auto parts scenario that is used throughout the book.

Quiz: To test your knowledge of the auto parts scenario, try to answer the following five questions. The answers are provided in Appendix C, “Quiz answers” on page 559.

1. What is stored in WSDL?
2. What is SOAP used for?
3. What is stored in a UDDI registry?
4. What brand of cars does Almaden Autos sell?
5. What is the name of the manufacturer of the cars?



Part 2

WebSphere Studio Application Developer

In part two we introduce the new IBM tool for Java client and server application development, WebSphere Studio Application Developer.

We often abbreviate the product name as Application Developer.



Application Developer overview

This chapter provides an overview of WebSphere Studio Application Developer (Application Developer), which we use to develop the sample application. We provide a tour of the integrated development environment (IDE) in which we discuss:

- ▶ WebSphere Studio Workbench open tooling platform
- ▶ The use of the different types of projects within Application Developer, such as the Java, EAR, Web, EJB and Server projects
- ▶ The use of the different perspectives available in Application Developer, such as the Web, Java, J2EE, Data, XML, server, debug, and help perspectives

Product

Application Developer brings together most of the functionality offered by VisualAge for Java and WebSphere Studio *classic edition*. Besides the functionality of these two products, new features were added, as shown in Figure 2-1. You learn about the new and adopted features when you explore this chapter and you can gain some practical experience when you start develop the sample applications described in the chapters that follow.

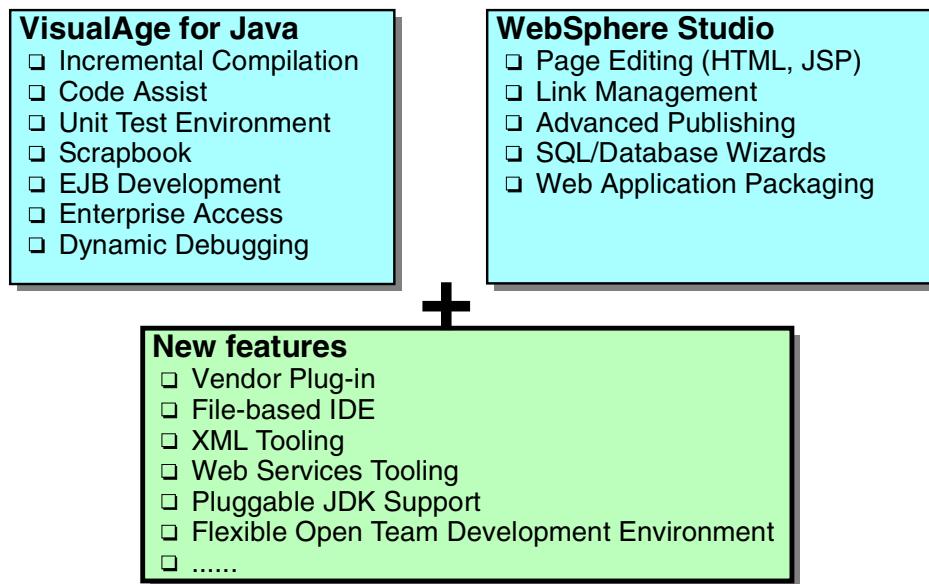


Figure 2-1 Application Developer features

WebSphere Studio Workbench

WebSphere Studio Workbench is the brand name for the new open, portable universal tooling platform and integration technology from IBM. It forms the base for the new WebSphere Studio suite (WebSphere Studio Site Developer and WebSphere Studio Application Developer).

The Workbench is not meant for customers, but for tool builders who want to plug-in their tools into the WebSphere Studio Workbench. This open source project (<http://www.eclipse.org>) enables other tool vendors to develop plug-ins for the WebSphere Studio Workbench. The tool providers write their tool as a plug-in for the Workbench, which operates on files in the workspace.

When the Workbench is launched, the user sees the integrated development environment composed of the different plug-ins. WebSphere Studio Workbench provides APIs, building blocks, and frameworks to facilitate the development of new plug-ins. There can be interconnections between plug-ins by means of extension points. Each plug-in can define extension points that can be used by other plug-ins to add function. For example, the Workbench plug-in defines an extension point for user preferences. When a tool plug-in wants to add items in that preferences list, it just uses that extension point and extends it.

The Workbench user interface (UI) is implemented using two toolkits:

- ▶ Standard widget toolkit (SWT)—a widget set and graphical library integrated with the native window system but with an OS-independent API.
- ▶ JFace—a UI toolkit implemented using SWT.

The whole Workbench architecture is shown in Figure 2-2.

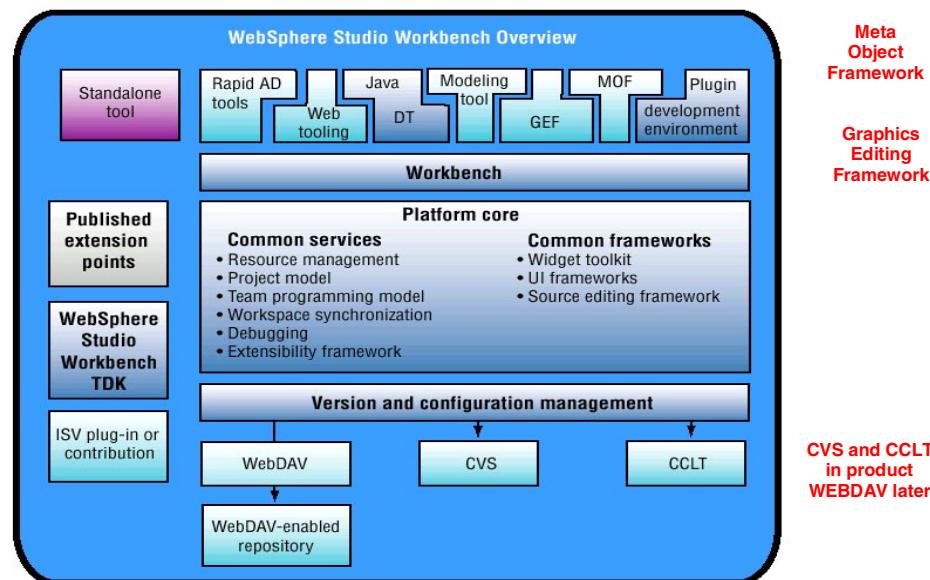


Figure 2-2 The Workbench architecture

Here are some explanations about the acronyms used in Figure 2-2.

Meta Object Framework (MOF)

MOF is an architecture for describing and defining meta-data. More information about MOF can be found at <http://www.omg.org>. MOF is used to share information in memory by plug-ins.

Graphics Editing Framework (GEF)

GEF is a free Java library for use in the development of new connected graph user interfaces. GEF can be used in any Java program that requires the ability to display and edit connected graphs or structured graphics. More info at <http://www.ics.uci.edu/pub/arch/gef/>.

Concurrent Versions System (CVS)

CVS is the open standard for version control systems. More information on CVS can be found at <http://www.cvshome.org>. and in Chapter 7, "Working in a team" on page 207.

ClearCase LT (CCLT)

ClearCase LT is from Rational and is shipped with the Application Developer. More information can be found at <http://www.rational.com>.

Other versioning systems will be supported in future versions of the Application Developer product or by code shipped by other vendors (Merant, for example).

Workbench window

We will refer to the interface of Application Developer as the Workbench. It's an integrated development environment that promotes role-based development. For each role in the development of your e-business application, it has a different and customizable perspective. A perspective is the initial set and layout of the views in the Workbench. Each perspective is related to a different kind of project. For example, if you want to develop Java applications, you first create a Java project. When you work in a Java project, you probably use the Java perspective because that is the most useful perspective to do Java developing. We give an overview of the different perspectives and project in the next sections.

Perspectives

Perspectives are a way to look through different glasses to a project. Depending on the role you are in (Web developer, Java developer, EJB developer) and/or the task you have to do (developing, debugging, deploying) you open a different perspective. The Workbench window can have several perspectives opened, but only one of them is visible at one point in time.

Note: The screen shots shown in this chapter were taken after most of the applications in this redbook were developed.

Figure 2-3 shows the Web perspective. You can switch easily between perspectives by clicking on the different icons in the perspective toolbar:

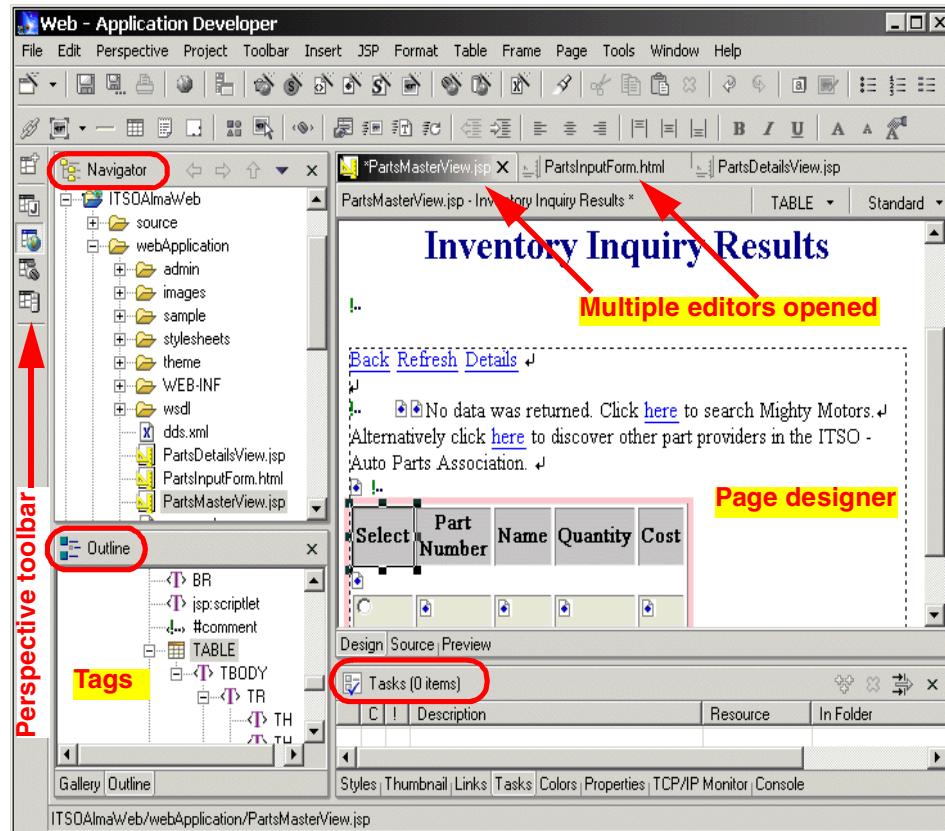


Figure 2-3 Web perspective

- ▶ You can open a new perspective by clicking on the top icon of the perspective toolbar.
- ▶ Each perspective has its own views and editors that are arranged for presentation on the screen (some may be hidden at any given moment). Several different types of views and editors can be open at the same time within a perspective.
- ▶ There are several perspectives predefined (resource, Java, Web, J2EE, Data, XML, server, help, debug, team) in the Workbench. You can customize them easily by adding, deleting, or moving the different views.
- ▶ You can also compose your own perspective by defining the views it should contain.

Views

We first discuss the different views that appear in most perspectives and we then take a closer look at some of the most used perspectives. Some of the views appear in most of the perspectives. We will give you a short overview of those views.

Navigator view

The small navigator panel shows you how your different resources are structured into different folders. There are three kinds of resources:

- | | |
|-----------------|---|
| Files | Files correspond to files in the file system. |
| Folders | Folders are like directories in the file system. They can contain files as well as other folders. |
| Projects | You use projects to organize all your resources and for version management. When you create a new project, you will have to assign a physical location for it on the file system. |

Editors

By double-clicking on a resource the associated editor opens and allows you to modify it. In Figure 2-3 the active editor is the page designer, associated with JSP and HTML files. If no editor is currently associated with a particular file extension, the Workbench checks if there is one associated in the operating system and uses that editor to edit the file. You can also open OLE document editors such as Word, which is associated with the *.doc* extension.

You can change or add editors associated with a file extensions:

- ▶ From the menu bar select *Window -> Preferences*.
- ▶ In the left panel, select *File editors* under the *Workbench* hierarchy.
- ▶ You can then select a file extension and associate an internal or external editors for it.

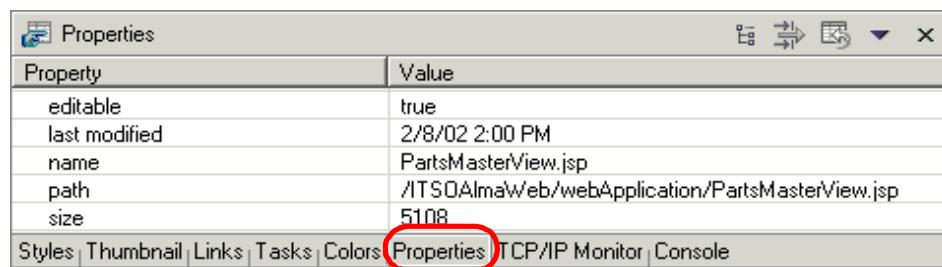
When you double-click another resource, a different editor shows up. You can easily switch between the different opened resources by selecting them on the top bar above the editor area. If the tab of your editor contains an asterisk (*), it means that it contains unsaved changes.

Outline view

The Outline view gives you an overview of the key elements that make up the resource that is being edited. It allows quick and easy navigation through your resource. By selecting one of the elements in the Outline view, the line in the editor view that contains the selected element gets highlighted.

Properties view

When you click on a resource in the Navigator view and then click on the properties tab at the bottom of the screen you can view the different properties of that resource. The Properties view contains general things such as the full path on the file system, the date when it was last modified, and the size, as shown in Figure 2-4.



Property	Value
editable	true
last modified	2/8/02 2:00 PM
name	PartsMasterView.jsp
path	/ITSOAlmaWeb/webApplication/PartsMasterView.jsp
size	5108

Styles | Thumbnail | Links | Tasks | Colors | **Properties** | TCP/IP Monitor | Console

Figure 2-4 Properties view

Tasks view

The Tasks view is shown in Figure 2-3 on page 21. The Tasks view contains a list of two types of elements:

- | | |
|-----------------|---|
| Problems | Problems are tool determined issues that have to be resolved. Example problems are Java compile errors, or broken links for HTML/JSP files. They are automatically added to the Task view when working with the tool. When you double-click on a problem, the editor for the file containing the problem opens and the cursor is pointed at the location of the problem. |
| Tasks | You can manually add tasks yourself. For example, you can add a task that reminds you that you have to implement a Java method. Place the cursor in the method's implementation, right-click and select <i>Add -> Task</i> . When you double-click, the file opens and the cursor is located in the method. You can also add general tasks that do not refer to a specific file. |

You can set up several filters to show only the tasks you really want to see. For example, by clicking the filter icon , you can specify to show only the Java compile errors from a particular Java class.

Customizing perspectives

You can highly customize the different perspectives by:

- ▶ Closing or opening views.
- ▶ Maximizing the view by double-click on the title bar. You do this when you need a large window for code editing. Double-click again to restore the layout.
- ▶ Moving views to other panes or stack them behind other views. To move a view:
 - Select in the view's title bar and start dragging the view.
 - While you drag the view, the mouse cursor changes into a drop cursor. The drop cursor indicates what will happen when you release the view you are dragging:



The floating view appears below the view underneath the cursor.



The floating view appears to the left of the view underneath the cursor.



The floating view appears to the right of the view underneath the cursor.



The floating view appears above the view underneath the cursor.



The view becomes a floating view, that is a separate window.



The floating view appears as a tab in the same pane as the view underneath the cursor.



You cannot dock the floating view at this point.

- ▶ Adding views. You can add a view by doing the following:
 - From the main menu bar select *Perspective -> Customize*.
 - Select the view you want to add and click *OK*.
 - Select *Perspective -> Show View* and select the view you just added.

When you want to reset a perspective to its original state, select *Perspective -> Reset* from the main menu.

Web perspective

In Figure 2-3 on page 21, you see the Workbench opened in the Web perspective. You use the Web perspective when you want to develop Web applications. The Web perspective is the best perspective to add and organize static content (HTML, images) and dynamic content (servlets and JSPs) to a Web application.

On top of the perspective, you see the Workbench toolbar. The contents of the toolbar change based on the active editor for a particular resource. The current editor is the page designer for editing our JSP page. The toolbar now reflects JSP development and contains icons to add JSP tags and a JSP menu item.

The Outline view shows the outline of a JSP page. It contains all the tags from which the JSP page is constructed. When you switch to the source tab of the page designer and you select a tag in the Outline view, the matching line in the Source view is highlighted.

We use the Web perspective in the chapters that follow, where we develop the sample Web application and the Web services.

Currently, there is no Web service perspective. You have to define a Web application before you run the Web service wizard, because the wizard adds WSDL files, a proxy class, a sample test client, and the necessary SOAP router servlets to the Web application. More information about the creation of Web services can be found in “Creating the Mighty Motors InquireParts Web service” on page 326.

Java perspective

When you want to develop Java applications you use the Java perspective. The Java perspective is shown in Figure 2-5. It contains a lot of useful editors and views which help you in your Java development.

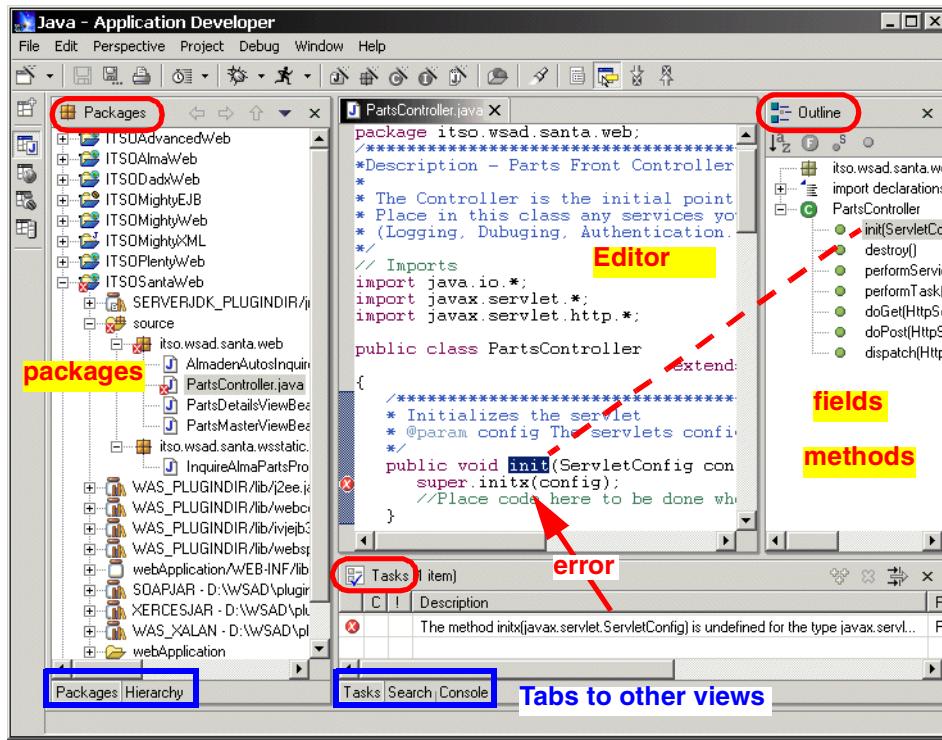


Figure 2-5 Java perspective

You navigate in the Java perspective through the Packages view. The Packages view allows you to define and manage Java packages and the Java classes defined in the packages.

When you right-click a Java class in the Packages view and you select *Open Type Hierarchy*, the hierarchy view for that Java class opens. The Hierarchy view allows you to see the full hierarchy of a Java class. In Figure 2-5, the Hierarchy view is currently hidden by the Packages view.

When you double-click on a Java file the Java editor opens. You can open multiple Java files at the same time. The Java editor features syntax highlighting and a code assistant by pressing *Ctrl-space*.

The Outline view in the Java perspective gives an overview of all the methods and fields for the Java file that is currently opened. When you click on a method in the Outline view, the cursor is positioned in the method signature in the Java editor. The toolbar at the top contains filters to include or exclude static methods or fields, and to sort the Outline view.

In the Java perspective, the Workbench toolbar contains several icons to add new packages, new Java classes, new Java interfaces, or to create a new Scrapbook page.

Clicking the search icon  invokes the search dialog as shown in Figure 2-6. Now you can either do a full text search, or a more intelligent Java search, to look, for example, for a particular method declaration or references to it.

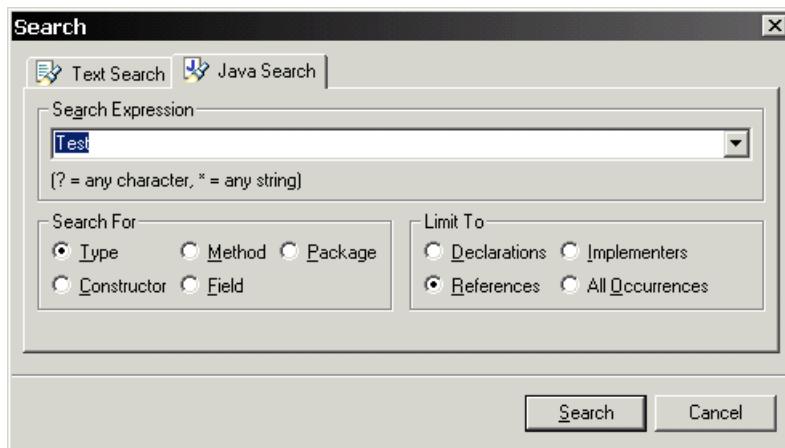


Figure 2-6 Search dialog

J2EE perspective

The J2EE perspective provides useful views for the J2EE or EJB developer. The J2EE view on Figure 2-7 shows you a list of all the different modules such as Web modules, EJB modules or server instances and configurations that make up your enterprise application. You can expand the module you want to explore and can edit the associated deployment descriptors for that module by double-clicking.

In Figure 2-7, the EJB deployment descriptor (`ejb-jar.xml`) is currently opened in the EJB editor. You will use the J2EE perspective most of the time when you do EJB development as illustrated in Chapter 5, “EJB development with Application Developer” on page 133.

The Navigator view, hidden by the J2EE view in Figure 2-7, shows a hierarchical view of all the resources in the workspace. When you double-click a resource, the registered editor for that file extension opens and the Outline view shows the outline for the file you are editing.

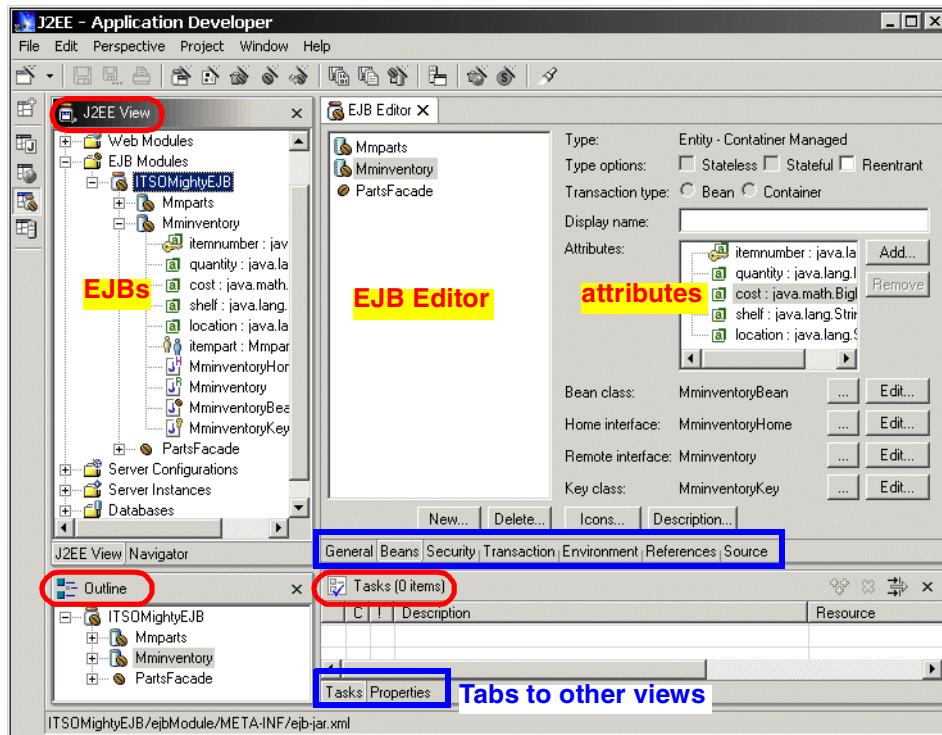


Figure 2-7 J2EE perspective

Data perspective

You use the data perspective (Figure 2-8) for relational database design for your application. You can either create a relational database schema yourself, or import it from an existing database. Afterwards, you can browse, query or modify it. The data perspective provides the views to manage and work with database definitions. You can find more information on using the data perspective in “Building the entity EJBs” on page 138.

In the DB Explorer view, you can create a connection to an existing database and browse its schema. When you want to modify or extend the schema, you have to import it into the Data view.

The Data view allows you to define new tables, or to modify existing tables. If you double-click on a table in the Data view, the table editor opens and you can add or change columns and primary or foreign keys.

The Navigator view shows all the resources in the folder structure.

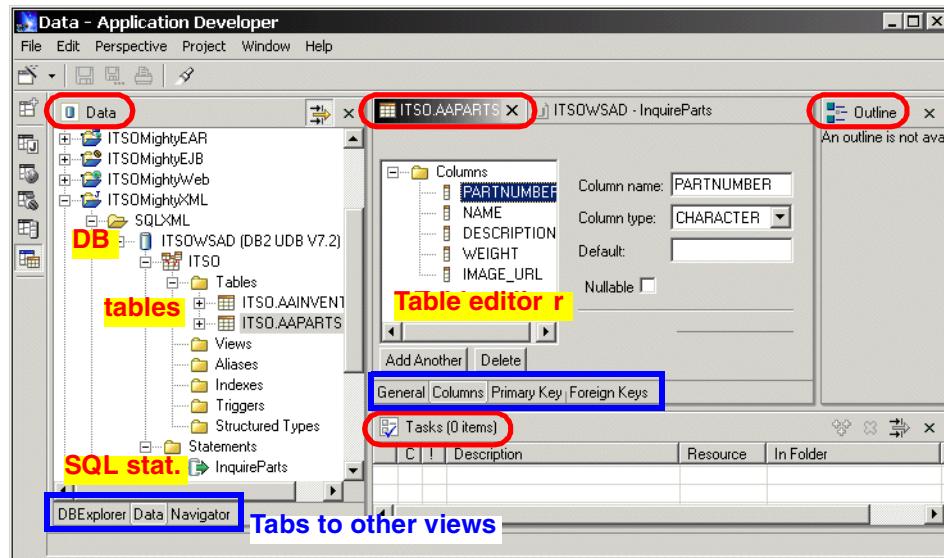


Figure 2-8 Data perspective

XML perspective

The XML perspective is the perspective for XML development. The XML perspective contains several editors and views that help you in building XML, XML schemas, XSD, DTD and integration between relational data and XML.

In Figure 2-9, the XML editor is opened. You can switch between the design and source panel of the editor to develop your XML file. The Outline view contains all the XML tags that make up the XML document that is currently opened in the XML editor.

More information about the XML support in Application Developer can be found in “XML support in Application Developer” on page 91.

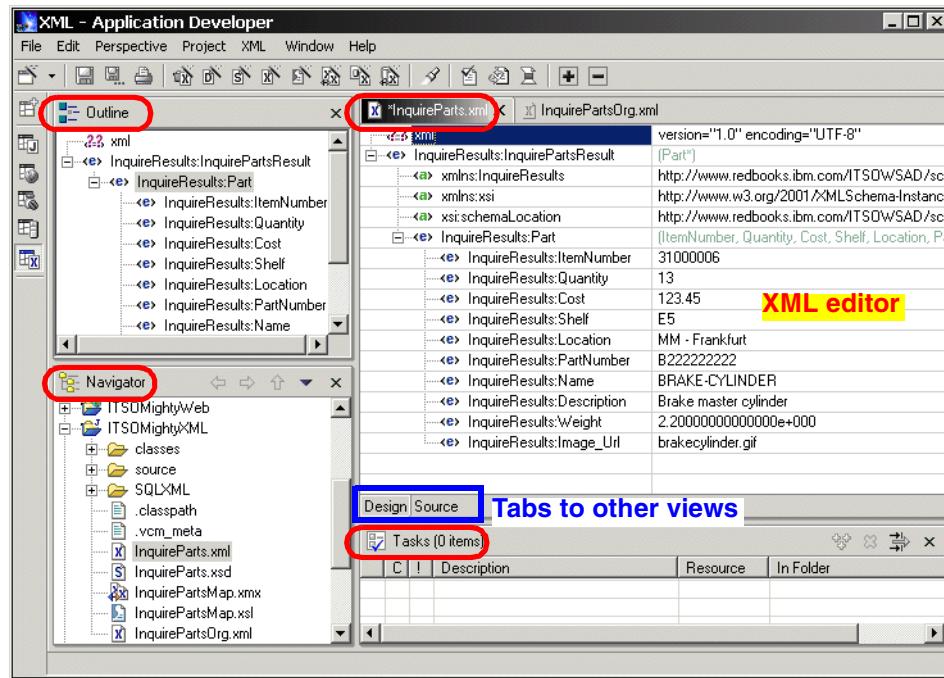


Figure 2-9 XML perspective

Server perspective

When you want to test a Web application or EJB module you need the server perspective (Figure 2-10). The server perspective contains views and editors that enable you to define, configure, and manage server instances and configurations.

The Server Configuration view (left bottom) enables you to define or modify server instances and configurations, and bind them to a project. When you right-click on the server configuration file in the Server Configuration view, and select *Edit*, the Server Configuration editor opens. More info about this topic can be find in “Creating a server instance and configuration” on page 82.

The Servers view (right bottom) lists all the currently defined server instances. Here you can start or stop their execution, or assign another server configuration to a server instance.

The Console view (currently hidden by the Servers view) shows all the output listed by a running server instance.

The Processes view shows all the processes that are started.

The Debug view allows you to step thought the code when debugging.

The Variables view allows you to inspect the values of variables when debugging.

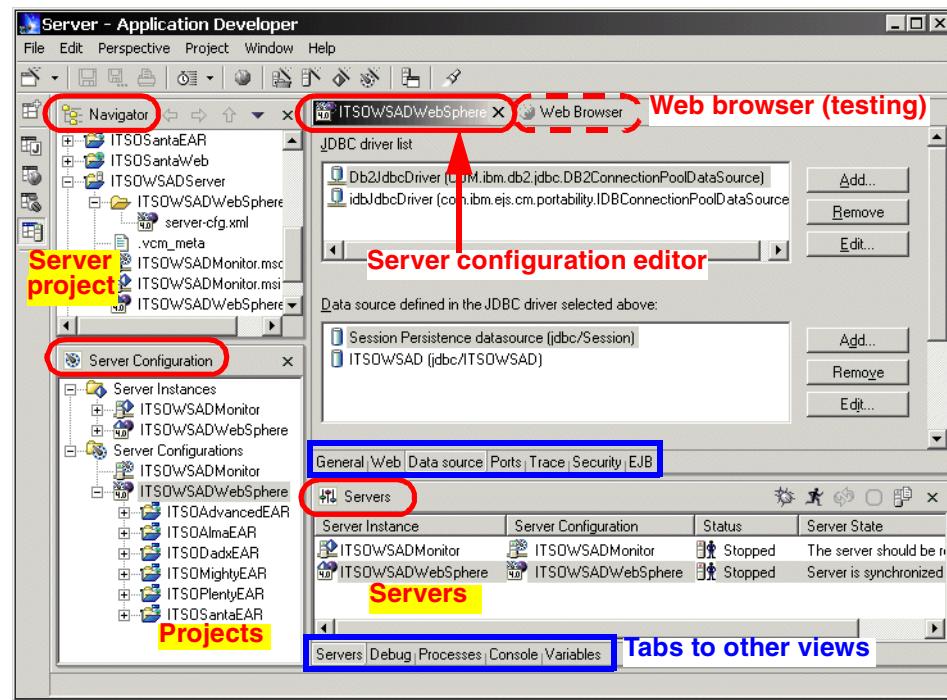


Figure 2-10 Sever perspective

Debug perspective

Use the debug perspective (Figure 2-11) when you want to debug your code. The debug perspective automatically opens when you click the Debug icon  in the Java perspective to run an application in debug mode. It allows you to step through your code, inspect the values of variables, modify your code and resume execution.

The debug perspective is built from the following views:

- ▶ The Processes view lists all running processes.
- ▶ The Debug view lists all threads within the different processes and shows you where the execution is halted when reaching a breakpoint.
- ▶ Beneath the Debug view there is a Java editor which shows the source of the code you are stepping into.

- ▶ The Breakpoint view lists all currently defined breakpoints. The *exception* icon on top of the Breakpoint view allows you to define exceptions that will halt execution when thrown.
- ▶ The Inspector view allows you to inspect variables.
- ▶ The Variables view lists all variables defined currently in the running thread. You can view and modify their values and set up filters to exclude for example static fields.
- ▶ The Console view (bottom) shows the output of your application.

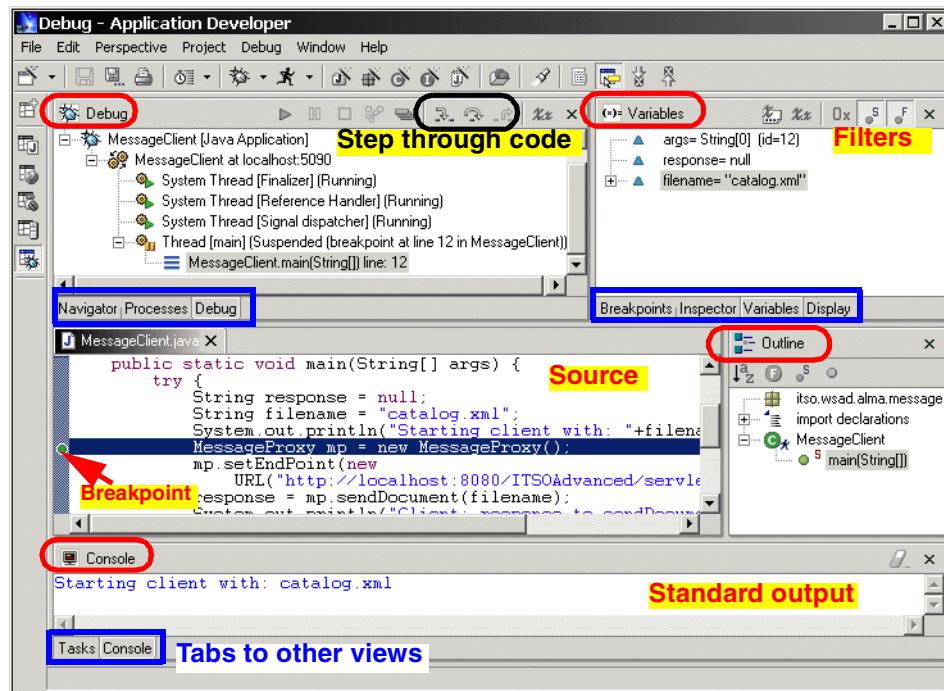


Figure 2-11 Debug perspective

Help perspective

The help perspective contains a lot of useful information about Application Developer. It provides information about the different concepts used by Application Developer, the different Tasks you can do within Application Developer and some useful samples. The search tab allows you to do a search in the help. You can open the help perspective just as any other perspective.

Important: Perspectives and memory considerations

After working with Application Developer for some time, you will have opened several perspectives. You might have the impression that Application Developer is working slower. It is good practice to close down the perspectives you have not been using for a while, because they can consume a lot of memory, and hence, slow down the overall performance. One of the reasons for this is, that when you open the same file in different perspectives, that file is synchronized among the different perspectives. Therefore, it is a good practice to edit a file using one perspective at a time.

Tips:

- ▶ Close perspectives that you do not need.
- ▶ Do not open multiple perspectives of the same type (it is allowed).
- ▶ Close editors before switching perspectives (unless you know you come back quickly).

Projects

A project is the top level construct for organizing the different resources. It contains files as well as folders. In Application Developer you can create different kind of projects, and they will have a different structure. A Web project, for example, has a different nature than a Java project, therefore it will have a different folder structure.

We will now briefly discuss the types of projects we use in the sample application:

- ▶ Java project
- ▶ EAR project
- ▶ Web project
- ▶ EJB project
- ▶ Server project

Java project

When you want to create a Java application, you first have to create a Java project to contain the Java files. Each Java project has a Java builder and builder path associated with it, which are used to compile the Java source files.

Creating a Java project

Here are the steps to create a Java project:

- ▶ Select *File -> New -> Project*.
- ▶ Select *Java* in the left panel and *Java Project* in the right panel.
- ▶ Specify a name for the project and an output folder for the compiled resources. In the default setup the output folder is the project, which means that the compiled class files are in the same directory as the source files.
- ▶ The *Java build settings* panel (Figure 2-12) contains four tabs to specify the folders, projects, and libraries used for compilation.

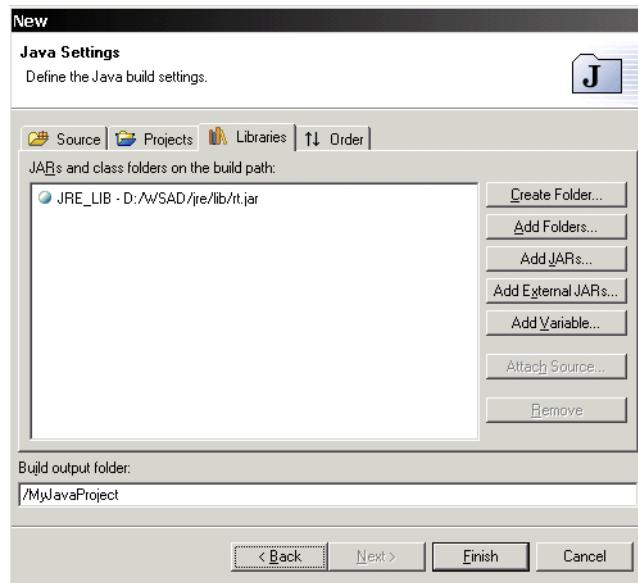


Figure 2-12 Java build settings

- | | |
|------------------|--|
| Source | In the <i>Source</i> tab you specify where the source files should be stored; either within the normal project folders or in folders designated by you. |
| Projects | In the <i>Projects</i> tab you specify whether other projects are required in the build path. For example, a Web project requires an EJB project. |
| Libraries | In the <i>Libraries</i> tab you can add internal and external JAR files to the build path: <ul style="list-style-type: none">▶ An internal JAR is contained in the Workbench. The advantage of an internal JAR is that it can be treated like a normal |

resource within the Workbench, which allows version management.

- ▶ External JARs are referenced with absolute paths in the file system so that you cannot share them in a team environment. Variables can be used to alleviate the issue introduced by absolute paths. An example of an external JAR file would be the db2java.zip file used for JDBC access to DB2.

Order	The <i>Order</i> tab enables you to specify the order in which the different items in the build path are accessed, when loading Java classes.
--------------	---

Tip: Use the predefined variables instead of adding external JARs with absolute paths to your build path whenever possible. Application Developer contains various predefined variables such as the DB2JAVA variable which defines the db2java.zip file.

You can add variables for other JAR files through the *Windows -> Preferences* dialog.

You can modify the Java build path after you have created a project through the *Properties* context menu of the project.

When you are finished creating the Java project, the Workbench switches automatically to the Java perspective.

Creating a package and a class

To create a Java package select *File -> New -> Java Package* or click the *New Package* icon  in the toolbar. Enter the package name, click *Finish*. The package appears in the Packages view.

To create a class in the new package select the package and select *File -> New -> Java Class* or press the *New Class* icon  in the toolbar. In the SmartGuide, check the package name and enter the desired class name and superclass. If you want a *main* method in your class, select the *main* method under *Which method stubs would you like to create*. Click *Finish*. The class appears under the package and a Java editor opens and you can start coding.

Java editing

The following useful features are available when you edit Java code:

- ▶ Double-clicking in the title bar of the Java editor maximizes the editor so that it occupies the whole perspective. Double-click again to restore its original size.
- ▶ Use *Ctrl-space* to launch the code assist in the Java editor when coding.

- ▶ If you select the edited Java source and you click the *Show Source of Selected Element Only* button in the toolbar  (4th from right) then only the source of the selected element in the Outline view are displayed in the editor.
- ▶ If you place the cursor in the Java editor on a variable then the full package name of that variable displays in the hover help (a small window that opens at the cursor location over the text). Hide the hover help by clicking on the *Text Hover* button in the toolbar  (third from right).
- ▶ If you select a method in the Outline view and then select *Replace from Local History* from the context menu, a dialog opens and shows all the states of the method that you saved. You can replace the method with an older version. The same can be done for the class itself from the Navigator view.

Refactoring code

Refactoring your code means reorganizing it so that the behavior of the application stays the same. There is refactoring support available from the context menu to rename packages, classes, and methods. When you rename an element, Application Developer automatically fixes all dependencies for the renamed element. For example, if you change the name of a package, all import statements referencing that package will be updated.

Another refactoring action is extracting a method out of an existing method. For this you select the lines of code you want to extract, and select *Extract Method* from the context menu. Provide a name for the new method and press *Finish* to accept the necessary changes. A new method is created, and a call is inserted into the old method.

All refactoring actions display a dialog with all the changes that will occur, and you can decide which of the changes should actually be performed.

Running a Java application

A Java class must contain a *main* method in order to be runnable. To run a class, click on the *Run* icon .

- ▶ The first time a class is run, you are prompted to select the launcher. Select *Java Application* click *Next*. Select your class and click *Finish*. The launcher is then associated with the project.
- ▶ The debug perspective opens and the Console displays the output for your application if any. If you do not want the debug perspective to be opened when running a class you can modify it by selecting *Window -> Preferences*, item *Debug*.

Debugging a Java application

To set a breakpoint in your code double-click in the left border of the Java editor on the line you want the execution to be halted.

- ▶ Click on the *Debug* icon  in the toolbar to run the application in debug mode. The debug perspective opens at the breakpoint.
- ▶ You can step through the code using the icons in the Debug view. The debug perspective is described in “Debug perspective” on page 31.

Attention: In VisualAge for Java there is no special debug mode. Whenever you put a breakpoint in your code, the debugger halts execution when it hits that breakpoint. In Application Developer however, you have to run your application in debug mode in order to stop the execution at the breakpoint.

Changing the build path

If your code refers to an external JAR file, you have to add it in the build path in order to run your application.

For example, suppose that you want to use the SAXParser class to parse an XML document. To use this class, you have to add the xalan.jar that contains that class to the build path. We could add the file to the project itself, but it is better to refer to the file:

- ▶ In the Java perspective select the project. From the context menu select *Properties*. Select the *Java Build Path* property, then the *Libraries* page.
- ▶ We could now add the file by clicking on *Add External JARs* as discussed in “Creating a Java project” on page 34, but it is better to see if Application Developer defined a variable to refer to that jar file.
- ▶ Click on *Add Variable*, click *Browse*, select WAS_XALAN and click *OK*. The variable is now added to the build path and we are able to use the SAXParser.

Specifying alternate JREs

One of the new features of Application Developer compared to VisualAge for Java is that you can specify multiple Java runtimes to be used in your Workbench. To add a different JRE or to change the current JRE:

- ▶ Select *Window -> Preferences*.
- ▶ Expand *Java* at the left and select *installed JREs* (Figure 2-13).
- ▶ Click *Add* to add a different JRE or select one and click *Edit* to modify it.

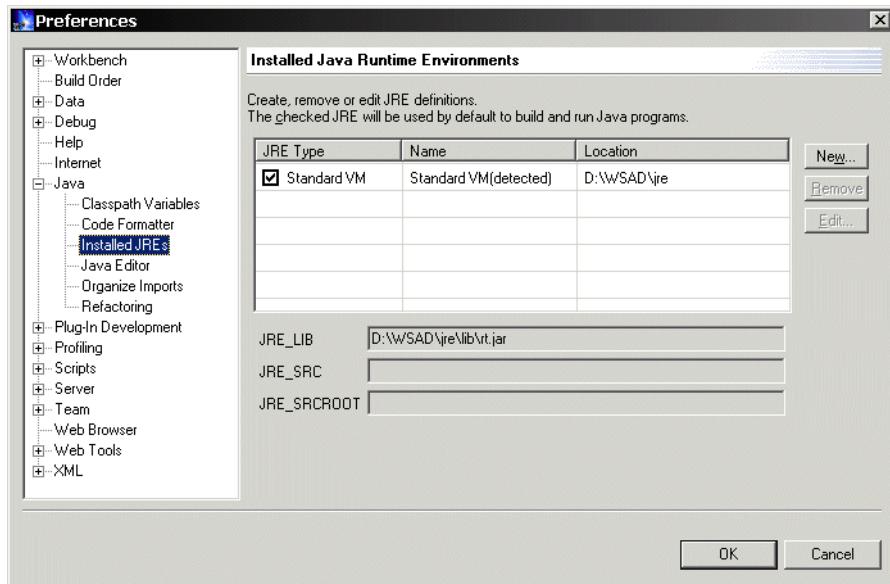


Figure 2-13 Adding JRES to the Workbench

Once you have added a JRE, you can assign a particular JRE to a project to be launched. Open the *Properties* menu of the context menu of the Java project and select *JRE*.

Note that specifying another JRE for a project to be launched does not affect the compilation. Compilation is still determined by the *Java build path*.

EAR project

To develop a J2EE enterprise application you have to create an Enterprise Application project (EAR project). An EAR project usually consists of one or more EJB modules, one or more Web applications, and an application client.

Creating an EAR project

To create an EAR project do the following:

- ▶ Select *File -> New Project*.
- ▶ Select *J2EE* in the left panel and *EAR project* in the right panel, click *Next*.
- ▶ Specify a *Name* for the EAR project.
- ▶ Specify contained modules (client, EJB, and Web projects) that you want to include in the EAR project, and click *Finish*.

EAR deployment descriptor (application.xml)

When you create an EAR project, a deployment descriptor (application.xml) is created in the /META-INF folder. The EAR deployment descriptor defines all modules in the EAR file. To open the EAR deployment descriptor do the following:

- ▶ Open the J2EE perspective and J2EE view.
- ▶ Right-click the EAR project and select *Open With -> Application Editor*.

IBM also provides extensions to the standard deployment descriptor for facilities that are not defined in the J2EE specification (<http://java.sun.com/j2ee>), such as Web application reloading. The ibm-application-ext.xmi contains the IBM extensions. To open the extensions:

- ▶ Open the J2EE perspective and in the *J2EE* view expand *Enterprise Applications*.
- ▶ Right-click the EAR project and select *Open With -> Application Extension Editor*.

Note: The IBM extensions for the different deployment descriptors (application, Web, EJB) are saved in the XML metadata interchange format (XMI). The XMI format specifies an open information interchange model that is intended to give developers working with object technology the ability to exchange programming data over the Internet in a standardized way. More information about XMI can be found at:

<http://www-4.ibm.com/software/ad/library/standards/xmi.html>
<http://www.omg.org/technology/documents/formal/xmi.htm>

J2EE packaging

An EAR project can be packaged as an enterprise archive file (EAR file). An enterprise application consists of the following modules:

- ▶ Web applications, which are packaged in .WAR files. The WAR file contains the resources that compose the Web application and a deployment descriptor (web.xml). A Web application is contained in a Web project, which we discuss in “Web project” on page 40.
- ▶ EJB modules, which are packaged in .JAR files. The EJB JAR file contains all the EJBs and a deployment descriptor (ejb-jar.xml). EJBs are contained in an EJB project, which we discuss in “EJB project” on page 44.
- ▶ Optionally we can have a stand-alone client application that uses EJBs, for example. An application client is also packaged in a JAR file. The application client JAR contains all the Java classes of the application client and a deployment descriptor.

Figure 2-14 shows how WAR files and JAR files together constitute the EAR file, which also contains the application deployment descriptor (application.xml).

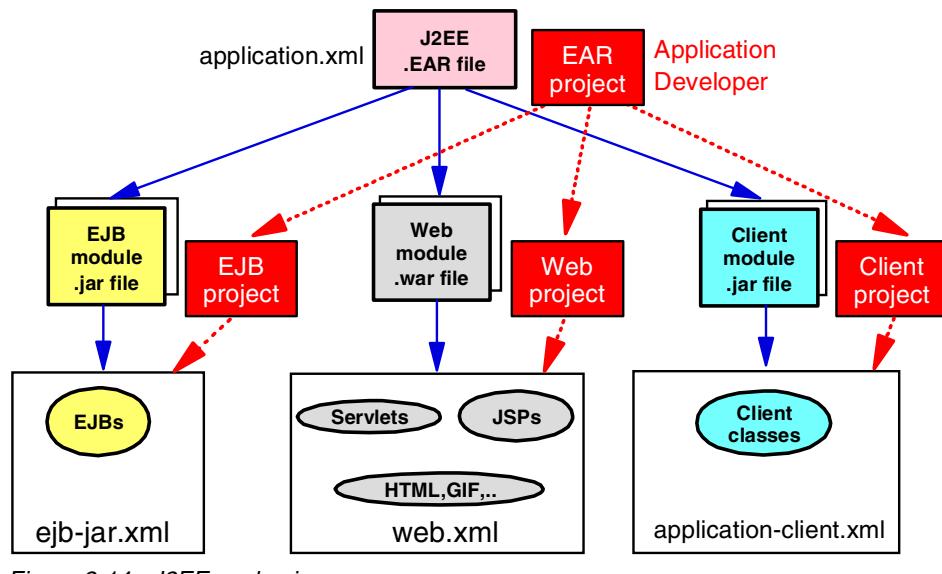


Figure 2-14 J2EE packaging

Web project

You use a Web project when you want to create and maintain resources that compose a Web application. A Web project contains the structure of folders to contain all files that are needed to build a Web application. Typically, a Web application consists of HTML pages, images, XML, servlets, JSPs and JavaBeans. How to build a Web application is described in Chapter 3, “Web development with Application Developer” on page 55.

Creating a Web project

To create a Web project do the following:

- ▶ Select *File* -> *New* -> *Project*.
- ▶ Select *Web* at the left side of the panel, *Web Project* at the right side of the panel, and click *Next* to start the wizard.
 - Specify the *Project name* and the workspace location for the project. You must also specify an EAR project name that will contain the Web project. You can select an existing EAR project or create a new one.
 - Specify the *Context root*, the alias that will be used in URLs to access project resources.

- Optionally, select *Create CSS file* (HTML style sheet).
- Click *Next*.
- ▶ On the *Module Dependencies* panel you can specify JAR files required by the Web application, for example EJB modules within the same EAR project.
- ▶ On the *Java Build Settings* panel you can add JAR files to your build path. Refer to *Java build settings panel* on page 34 for the definition of the different tabs.
 - The default setup includes a source folder for servlets and JavaBeans, and a webApplication/WEB-INF/classes folder for compiled class files.
 - The *Libraries* page includes several JAR file by default.
- ▶ Click *Finish*. Your Web project is automatically opened in the Web perspective.

When you create a new Web project a default directory structure is created that reflects the J2EE view of a Web application. A Web deployment descriptor `web.xml` is generated in the `/webApplication/WEB-INF` folder. You can find a detailed description about the generated folders and files in “Viewing the default Web project files” on page 60.

Web application archive files (WAR files)

As defined in the J2EE specification, a WAR file is an archive format for Web applications. The WAR file is a packaged format of your Web application that contains all the resources (HTML, servlets, JavaBeans, JSPs, XML, XML Schemas) that compose your Web application.

You can deploy a Web application by itself to an application server by creating a WAR file. Select *File -> Export -> WAR* and specify the output file and directory.

In general, however, it is easier to have Application Developer create the EAR file that contains the WAR file and deploy the EAR to an application server.

More information about deploying Web applications can be found in Chapter 6, “Deployment of Web and EJB applications to WebSphere” on page 169.

Import existing Web sites

Once you created a Web project, you can import existing Web resources in the Web project:

- ▶ Select *File -> Import*.
 - Select *HTTP* if you want to import resources using HTTP. Select the project to contain the resources and the URL from which you want to start the import. You can limit the import by setting the depth for the links to

follow. You can bound the import by setting a maximum depth for the HTTP links. If you want to import servlets, JSPs, or CGI scripts, you have to use the FTP protocol.

- Select FTP if you want to import resources using the FTP protocol. Specify the destination folder for the import, and provide a FTP login/password.

Import an existing WAR file

You can start developing a Web application by importing an existing WAR file:

- Create a Web project.
- Select *File -> Import -> WAR file*, and click *Next*.
- Specify the location of the WAR file, the destination Web project, and the destination EAR project.
- Click *Next* if you want to specify additional *Java build settings*, or *Finish* to end the wizard and start the import.

Web application deployment descriptor (web.xml)

When you create a Web project a web.xml file is created for the Web application in the /webApplication/WEB-INF folder. The web.xml file is the deployment descriptor for the Web application. When you double-click the web.xml file the Web deployment descriptor editor opens (Figure 2-15).

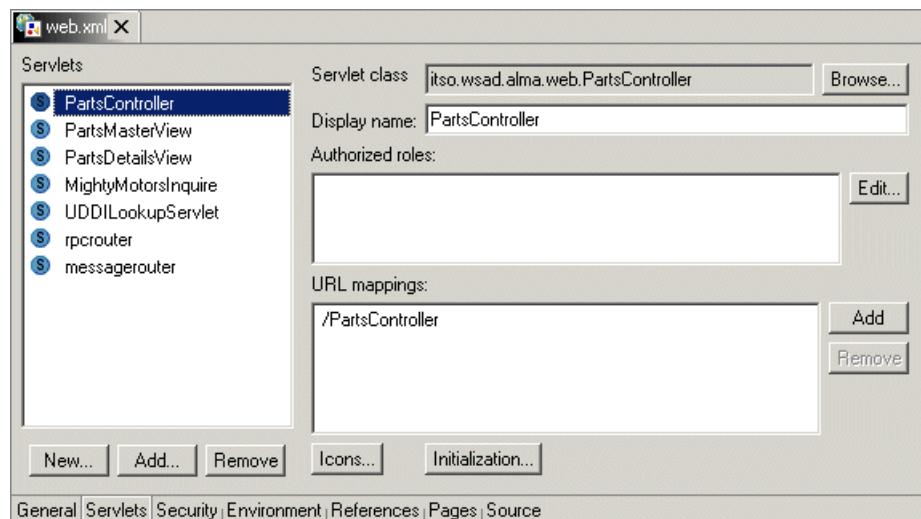


Figure 2-15 Web deployment descriptor editor

Select the different tabs at the bottom to define the properties for the Web application:

- ▶ *General* tab—in the *General* tab you define basic setting for the Web application such as:
 - Name and the description of the Web application
 - Session time-out (in minutes) for the Web application
 - Web application context initialization parameters
- ▶ *Servlets* tab—in the *Servlets* tab you can add servlets to the Web application deployment descriptor. To add a servlet you have to specify the *Display name*, the *Servlet class* or *JSP file*, and the *URL mapping* (the URL to invoke your servlet). Additionally you can define *initialization* parameters and *authorized roles* for your servlet. Authorized roles are roles that have access to the servlet or JSP. These roles must be defined in the *security* page.
- ▶ *Security* tab—the *Security* tab allows you to define security roles and security constraints for the Web application.
- ▶ *Environment* tab—the *Environment* tab allows you to specify environment variables for the Web application.
- ▶ *References* tab—references are logical names used to locate external resources for enterprise applications. At deployment, the references are bound to the physical location (JNDI name) of the resource in the target operational environment.

The *References* tab lets you specify the following references:

- *EJB references*: an EJB reference is a logical name used to locate the home interface of an enterprise bean. EJB references are made available in the `java:comp/env/ejb` subcontext of the JNDI. The use of EJB references is illustrated in “Defining the EJB references” on page 165.
- Resource factory references: references to external resources
- JSP tag libraries references: references to tag libraries
- ▶ *Pages* tab—the *Pages* tab allows you to specify:
 - A welcome page, for example, `index.html`
 - Login settings for defining an authentication method. In the case of a *form* based login you have to define a login and error page.
 - Error mapping between HTTP error codes or Java exceptions and a path to a resource of your Web application.
- ▶ *Source* tab—the Source tab shows you the source of the `web.xml`. Here you can manually edit the source.

EJB project

If you want to develop Enterprise JavaBeans (EJBs) you have to create an EJB project first. An EJB project is a logical group for organizing the EJBs. To create an EJB project:

- ▶ Select *File -> New-> Project*.
- ▶ Select *EJB* on the left panel and *EJB project* on the right panel and click *Next*.
- ▶ Specify the *Name* of the EJB project and the workspace location. You also have to specify an EAR project name that will contain your EJB project. You can select an existing EAR project or create a new one. Click *Next*.
- ▶ On the *Module Dependencies* panel you can specify JAR files required by the Web application, for example EJB modules within the same EAR project.
- ▶ On the *Java Build Settings* panel you can add JAR files to your build path. Refer to *Java build settings panel* on page 34 for the definition of the different tabs.
 - The default setup includes an *ejbModule* folder for EJBs and a *bin* folder for compiled class files.
 - The *Libraries* page includes several JAR file by default.
- ▶ When you click *Finish*, the EJB project opens in the J2EE perspective. The deployment descriptor for the EJB module (*ejb-jar.xml*) is created in the */YourProject/ejb-module/META-INF* folder.

The use of EJB projects and the use of the EJB wizard is illustrated in Chapter 5, “EJB development with Application Developer” on page 133.

EJB deployment descriptor (*ejb-jar.xml*)

An EJB module requires a deployment descriptor (*ejb-jar.xml*) in the same way a Web application requires a deployment descriptor (*web.xml*).

In addition to the standard deployment descriptor, Application Developer also defines EJB bindings and extensions. Both binding and extension descriptors are stored in XMI files, *ibm-ejb-jar-bnd.xmi* and *ibm-ejb-jar-ext.xmi*, respectively.

Attention: Please note that the *bin* folder also contains the *META-INF* folder with the *ejb-jar.xml* deployment descriptor. You should never open *ejb-jar.xml* from this folder, because the *bin* folder is the build output folder. The build output folder can be modified by selecting *Properties* from the EJB project context menu.

EJB editor

To edit the deployment descriptor for the EJB module:

- ▶ In the *J2EE* view of the *J2EE* perspective, expand *EJB Modules*.
- ▶ Right-click the EJB module and select *Open With -> EJB Editor*, or just double-click on the module.
- ▶ The *ejb-jar.xml* deployment descriptor opens in the EJB editor (Figure 2-16).

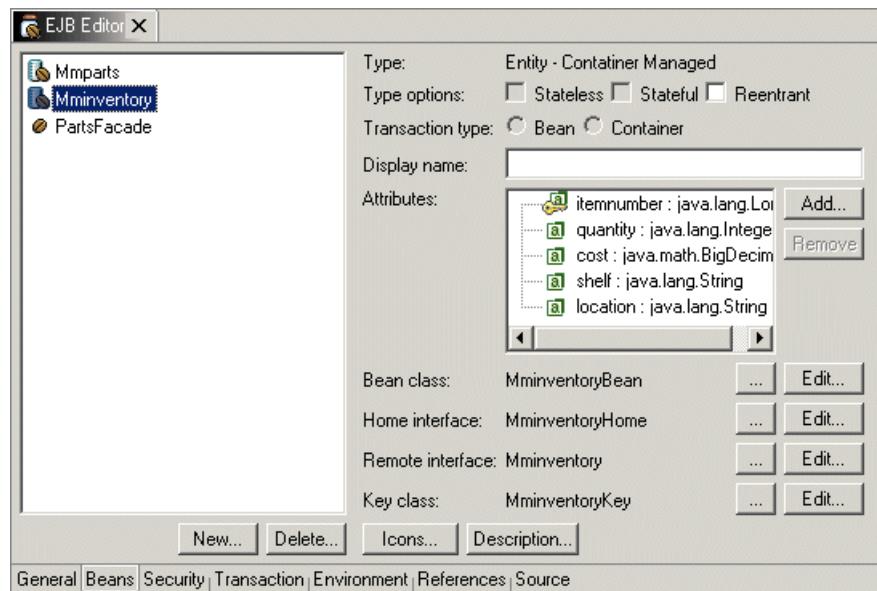


Figure 2-16 EJB editor: Beans tab

The *ejb-jar.xml* is presented in several sections defined by the tabs at the bottom of the EJB editor:

- ▶ *General* tab—allows you to define the *Display Name* and an EJB client jar for the EJB module.
- ▶ *Bean* tab— allows you to edit the attributes of the EJBs or to add or delete new EJBs.
- ▶ *Security* tab—allows you to define security roles and permissions for the different methods of the EJBs.
- ▶ *Transactions* tab—allows you to specify the transaction type attribute for the different methods of the EJBs.
- ▶ *Environment* tab— allows you to add, delete or modify environment variables.

- ▶ *EJB Reference* tab—allows you to define EJB references. EJB references are explained in “Web application deployment descriptor (web.xml)” on page 42, and illustrated in “Defining the EJB references” on page 165.
- ▶ *Source* tab—you can view and edit the XML source code of ejb-jar.xml.

EJB extension editor

To edit the EJB bindings and extensions in the EJB extension editor:

- ▶ In the J2EE view of the J2EE perspective, expand EJB Modules.
- ▶ Select *Open With -> EJB Extension Editor* from the context menu of the EJB module.
- ▶ The deployment descriptor opens in the EJB extension editor (Figure 2-17).

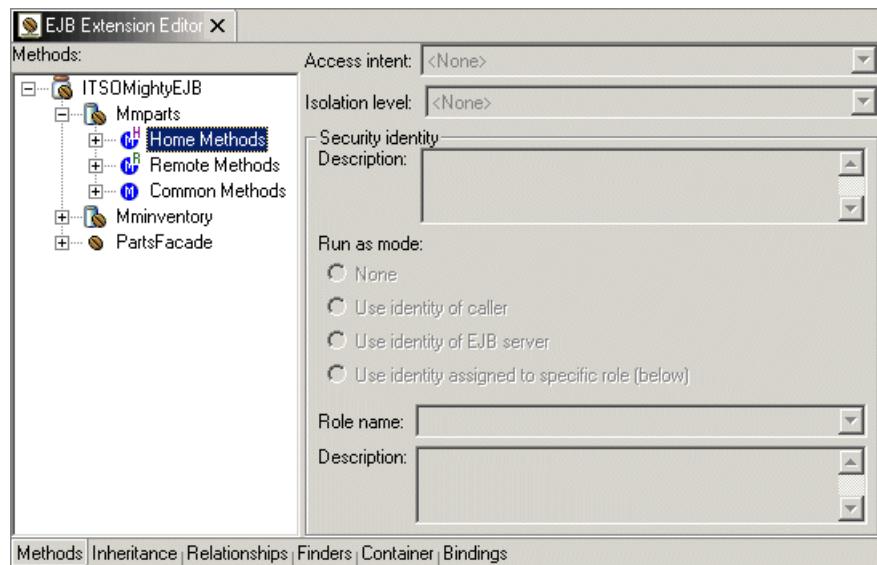


Figure 2-17 EJB extension editor: Methods tab

The ejb-jar.xml extensions are presented in several sections defined by the tabs at the bottom of the EJB editor.

- ▶ *General* tab—displays a hierachal view of the EJBs in the EJB module. Here you can change the supertype of an EJB and generate deployed code for it.
- ▶ *Methods* tab—allows you to define for each EJB on a per-method level:
 - Access intent (read or update)
 - Isolation level (repeatable_read, read_committed, read_uncommitted, serializable)
 - Security identity settings

- ▶ *Relationships* tab—allows you to define associations between EJBs.
- ▶ *Finders* tab—allows you to define additional finder methods for the home interface.
- ▶ *Container* tab—allows you to define cache settings per EJB and local transaction settings.
- ▶ *Bindings* tab—allows you to define the real JNDI names for the EJB references and for the data sources the EJBs are using.

Server project

To test an EJB or Web project, you have to define a server instance and a server configuration to publish and run the code. Server instances and server configurations are defined in server projects.

Creating a server project

To create a new server project:

- ▶ Select *File -> New -> Project*.
- ▶ Select *Server* on the left panel and *Server project* on the right panel.
- ▶ Specify a *Name* for your project and click *Finish*.

After creating a project the server perspective opens and you can now add a server configuration and a server instance. This is demonstrated in “Creating a server instance and configuration” on page 82.

Server instance

A server instance identifies the server used to test your application. Unlike VisualAge for Java, Application Developer has the option to deploy to and test with both local and remote instance of the WebSphere application server, and additionally Apache Tomcat. Here is a brief explanation of each of the server instances:

WebSphere v4.0 Test Environment

This enables the developer to work with an integrated version of WebSphere Application Server Advanced Edition Single Server v4.0.1, which supports the entire J2EE programming model. This is the best option for testing EJB-based applications.

WebSphere v4.0 Remote Server

This option allows the developer to define a WebSphere v4.0 Advanced Edition Single Server instance either locally or on a remote machine. When the application is executed, Application Developer publishes the code to the external server and attempts to start the application using the IBM Agent Controller

service, which is supplied with Application Developer. For further details on this option, see Chapter 6, “Deployment of Web and EJB applications to WebSphere” on page 169. This feature provides a very efficient approach to remotely deploying an application.

Tomcat v3.2 Local Server

Tomcat v3.2 is the current production-ready release of the Apache open source servlet and JSP engine. For more information on Tomcat and the Apache Jakarta project see <http://jakarta.apache.org>. This release supports the servlet 2.2 and JSP 1.1 specifications. Application Developer does not ship with the Tomcat binaries, only a toolkit to support its execution. You must already have a working Tomcat instance installed in order for this to work. The most current release is v3.2.3.

Tomcat v3.2 Test Environment

Permits publishing and execution of the Web application to an external version of Tomcat v3.2. Unlike the WebSphere Remote Test option, this is only supported for a local instance on the same machine.

Tomcat v4.0 Local Server

Tomcat v4.0 has been developed by the Apache group on a completely separate code base from the v3.2 release, and will provide the reference implementation for the Servlet 2.3 and JSP 1.2 specifications, which are currently in public review stage. The current release is a beta-level driver and it is not recommended for production use.

Tomcat v4.0 Test Environment

As described in the Tomcat v3.2 Remote Test option, this supports publishing to local external instances of Tomcat.

TCP/IP Monitoring Server

This is a simple server that forwards requests and responses, and monitors test activity. This run-time environment can only be run locally, and it only supports Web projects. You cannot deploy projects to the TCP/IP Monitoring Server. The TCP/IP Monitoring Server is illustrated in “Using the TCP/IP Monitoring Server to view message contents” on page 358.

Because Tomcat does not have EJB support, you cannot deploy EAR files to it, only WAR files containing servlets and JSPs.

Attention: Before you can do a remote unit test you have to install and run the IBM Agent Controller, which comes with Application Developer, on the remote machine. IBM Agent Controller is a process that runs on the remote machine and which enables client applications to start new host processes.

Limitations of the WebSphere Unit Test Environment:

The Secure Socket Layer (SSL) and the secure HTTP (HTTPS) features that are offered by WebSphere Application Server are not enabled in the WebSphere test environment.

Should your application require access to this feature, use the WebSphere 4.0 Remote Server with a local instance of a WebSphere Application Server. These are beta code limitations; in the product only the first limitation will remain.

Server configuration

A server configuration contains the information about the server. It defines:

- ▶ Port numbers for the different processes such as the naming service
- ▶ Mime types
- ▶ JDBC drivers
- ▶ Data sources
- ▶ Security enablement

All the information is stored in the `server-cfg.xml`, which is created when you add a server configuration. The properties can be set by opening (double-click) the configuration in an editor.

A server configuration can be reused by multiple server instances but each server instance will only point to one server configuration. Each server configuration can point to multiple projects as shown in Figure 2-18.

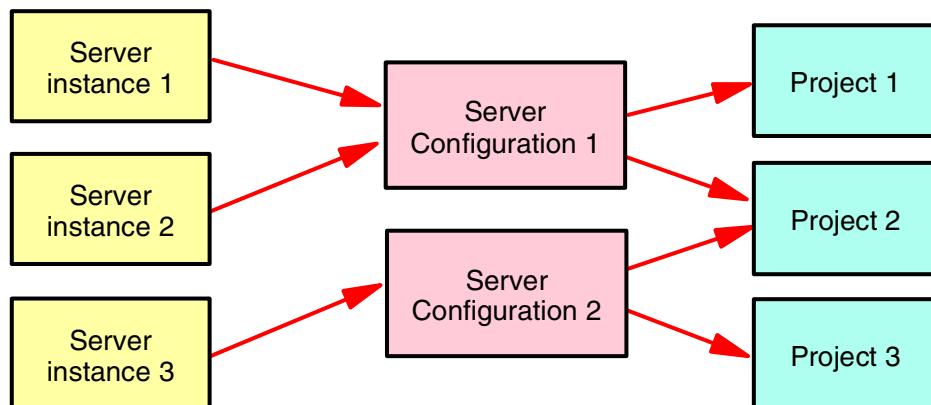


Figure 2-18 Relationship between server instance, configurations, and projects

Each project has a preferred server configuration that is used when the project is run by selecting *Run on Server* from its context menu.

Creating a server instance and configuration

In the server perspective, select *New -> Server -> Server instance and configuration* and complete the dialog as shown in Figure 2-19.

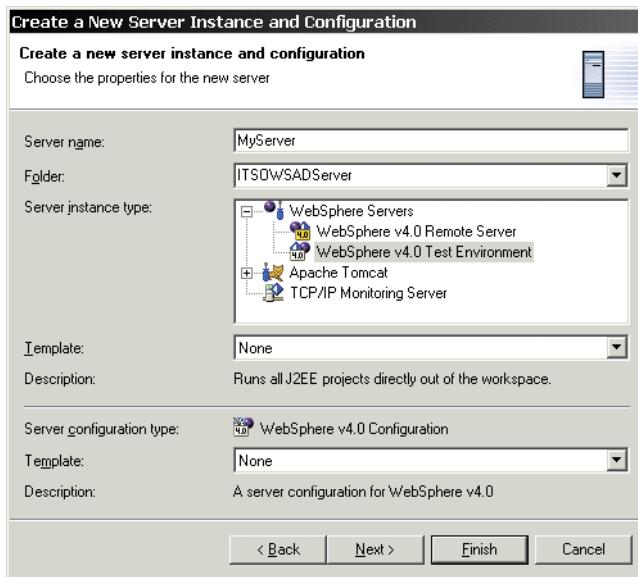


Figure 2-19 Creating a server instance and configuration

On the next page you can set the port (default is 8080) and click *Finish*.

The new server appears in the server perspective and you can assign EAR projects to the server. Those projects will be loaded when the server is started.

Automatic creation of a server configuration and instance

In the case you try to run a project for which you did not define a server instance and configuration, Application Developer creates a default WebSphere 4.0 Test Environment instance and starts it. This default server instance is then set as the project preferred server instance.

Later, when you define your own server instance and configuration, you can change the project preferred server instance by selecting *Properties* in the projects context menu and then selecting *Server Preference* (Figure 2-20).

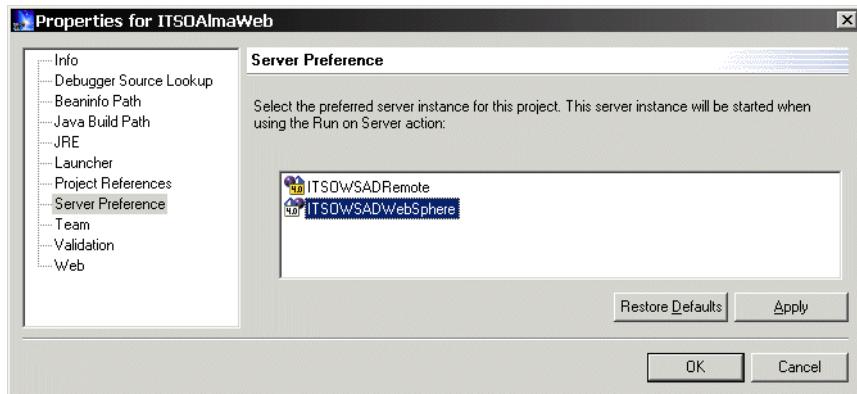


Figure 2-20 Changing the preferred server instance

Server instance templates

When you have to create several similar server instances or configurations, you can create them from a template to save time. You could also share a template across a team so that team members can start personalizing their server configuration or instance starting from a template. To create a template:

- ▶ Select *Window -> Preferences*.
- ▶ Expand *Server* on the left pane and select *Templates*.
- ▶ Click *Add* and the template is stored in a server project.

Publishing

Publishing means copying all the resources that are required to test a project to the right place so that the server can find them. In cases when you are testing within the Workbench, the resources might already be at the right place.

However, when you are testing with WebSphere Application Server on a local or remote machine, or with Tomcat on a local machine, publishing means to copy the resources outside of the Workbench.

By default the *Automatically publish before starting server* option is turned on. This option can be found in the *Window -> Preferences -> Server* item. When this option is turned on, all files and folders on the server are synchronized with the Workbench whenever starting or restarting that server.

You can manually publish by selecting the server instance in the server control panel of the server perspective and selecting *publish* from the context menu as shown in Figure 2-21.

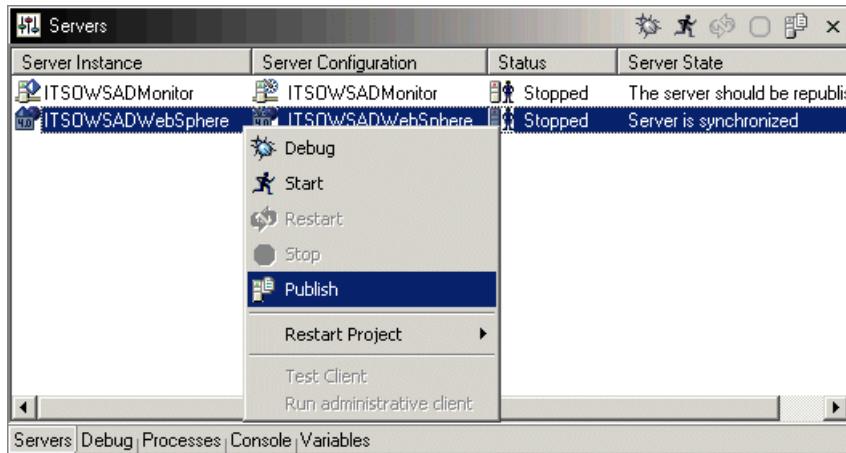


Figure 2-21 Publish to a server

Application Developer internal system files

There are several files that Application Developer uses to for its own system internal use. These files will be generated by Application Developer and have a naming convention such that the file name starts with a period. It is not recommended that you change these files.

Examples of these files are:

- | | |
|--------------------------|--|
| .classpath | Application Developer uses this file to store metadata about the build path for the project when compiling Java classes and executing Java applications. It is not recommended to edit this file directly—to change the path information, simply click on the <i>Properties</i> context menu for the project and modify the <i>Java Build Path</i> item. |
| .serverpreference | The Application Developer server preference for the project. |
| .websettings | Global Web application settings, for example, the context root. |
| .modulemaps | Composition of an EAR project |
| .vcm-meta | Information about builders used to compile the project. |

Summary

In this chapter we introduced WebSphere Studio Application Developer. We introduced WebSphere Studio Workbench, the new open tooling platform, and we gave a small tour of the IDE itself. The different project and perspectives within the IDE are summarized in Table 2-1.

Table 2-1 Projects and perspectives

Projects	Perspectives	Purpose
Web	Web	Web application and Web services development
EJB	J2EE, Java	EJB development
EAR	J2EE	J2EE packaging
Java	Java	Java development
Server	Server	Deployment and testing
All	Debug	Debugging
All	Team	Repository management
Web, EJB, Java, other	Data	Relational DB management
Web, EJB, Java, other	XML	XML development

Quiz: To test your knowledge of general features of Application Developer, try to answer the following five questions. The answers are provided in Appendix C, “Quiz answers” on page 559.

1. Which of the following perspectives does not exist?
Web, XML, Java, Team, Web services, J2EE
2. Which file format is used to store the IBM extensions to the deployment descriptors?
3. Name at least three different types of projects in Application Developer.
4. How can you add a JAR file to the class path for a Java, Web, or EJB project?
5. What is the difference between a task and a problem?



Web development with Application Developer

In this chapter we introduce the Web application development capabilities of Application Developer by developing the first stage of our auto parts sample application. These capabilities will include:

- ▶ Developing a dynamic HTML client to the sample database
- ▶ Publishing the pages to a test instance of the WebSphere Application Server executing inside Application Developer
- ▶ Enhancing the appearance of the application using the page designer

Solution outline for auto parts sample Stage 1

An overview of the sample application can be found in Chapter 1, “Auto parts sample application” on page 3. This section will discuss the design in more detail before we progress onto building the application.

The application at this stage is intentionally very simple, allowing basic read access to a DB2 database in order for a mechanic to browse the parts inventory at the Almaden Autos car dealership, as illustrated in Figure 3-1.

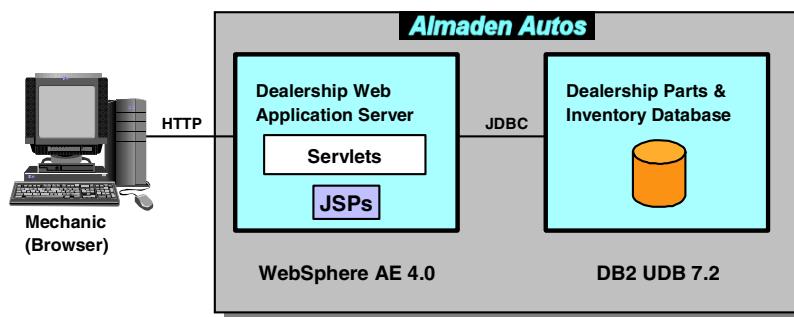


Figure 3-1 Basic design for the Almaden Autos application

The intention of this chapter is not to provide an overview of the servlet and JSP programming models, but to give the reader an opportunity to understand how such components can be developed using Application Developer.

Please refer to “Related publications” on page 577 if you require a primer on basic J2EE programming.

Class and sequence diagrams

Figure 3-2 shows the class diagram for stage 1.

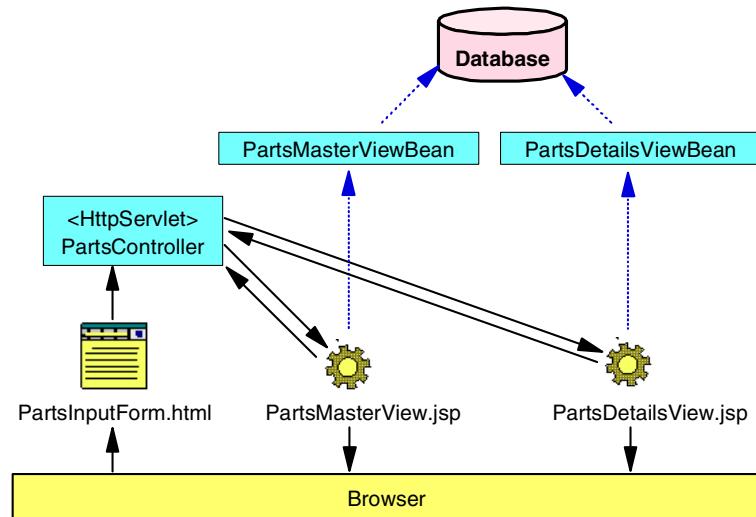


Figure 3-2 Class diagram for Stage 1 of the auto parts sample

Figure 3-3 shows the sequence diagram for Stage 1.

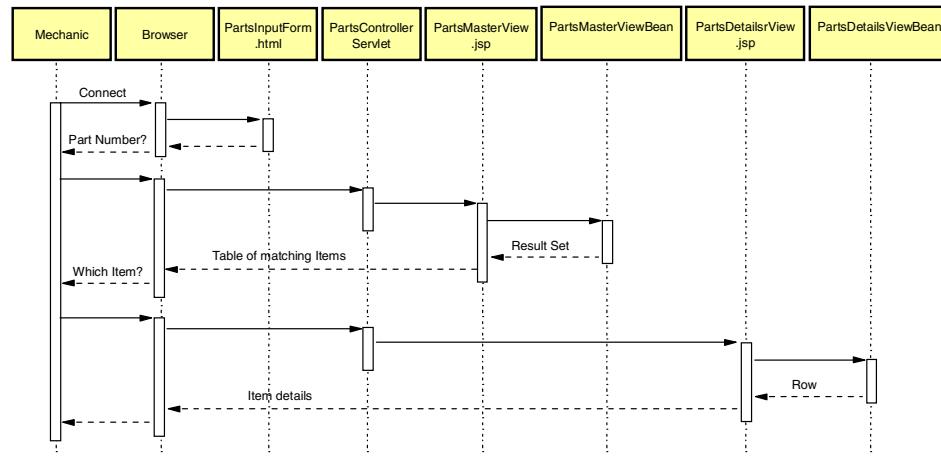


Figure 3-3 Sequence diagram for Stage 1 of the auto parts sample

Preparing for development

Before we start developing J2EE components, a number of configuration steps must be completed. These will include the following:

- ▶ Creating a new workspace
- ▶ Creating and configuring a Web project
- ▶ Understanding the files generated by default
- ▶ Creating the sample database from the supplied DDL
- ▶ Creating a Java package for the Java code

This section assumes that Application Developer and DB2 7.2 have already been installed by the user using the steps outlined in Appendix A, “Product installation and configuration” on page 537.

Creating a new workspace

When developing applications in Application Developer, we recommend that you create a new workspace to store the application artifacts outside the installation directory. To do this, we must first exit Application Developer and create a new directory in the file system. In this example, we will call the workspace directory WSADsg246292. From a command line window on a data drive, type the following:

```
md /WSADsg246292  
x:/wsad/wsappdev -data y:/WSADsg246292
```

Where x:/wsad is the drive and directory used to install Application Developer, and y: is the drive used to create the workspace. It may be worth creating a shortcut on your Windows desktop (or in the IBM WebSphere Studio Application Developer folder) for launching the command in future, because the default parameters to launch Application Developer used by the current start menu icon revert your settings back to the default workspace in the installation directory.

Project configuration

First we must build a Web project to contain the application code. To do this, launch Application Developer if it is not already running, and select the *File -> New -> Project* menu from the workbench.

In the New Project dialog, select the *Web* category, then *Web Project*. Click on the *Next* button.

- ▶ Enter ITS0A1maWeb in the Project name field and ITS0A1maEAR as the Enterprise Application project name.

The context root for the Web application is also set here. The default context root is the Web project name, in this case ITSOAlmaWeb.

- ▶ Change the context root to ITSOAlma.
- ▶ Select the *Create CSS file* checkbox. This will create the a default style sheet to be used by the Web project.

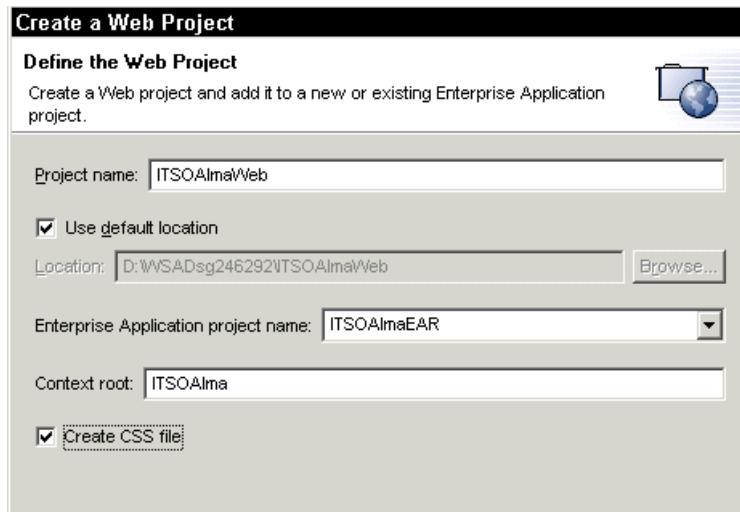


Figure 3-4 Create a Web project

Click on the *Finish* button to complete the project creation.

Project naming convention:

The naming convention we will be using throughout the samples for our Application Developer projects is as follows:

- ▶ Web Projects: ITSO*ApplicationName*Web
- ▶ EJB Projects: ITSO*ApplicationName*EJB
- ▶ EAR Projects: ITSO*ApplicationName*EAR

This allows us to see related projects grouped together in the main Navigator view. While some project types (for example, Java projects) allow spaces in their names, this is not currently supported for Web or EJB projects.

Viewing the default Web project files

Once we have generated the Web project in Application Developer, it is worth taking a few moments to review the files which have been generated by the wizard. After clicking the *Finish* button, a navigator similar to Figure 3-2 in the opened Web perspective should be visible.

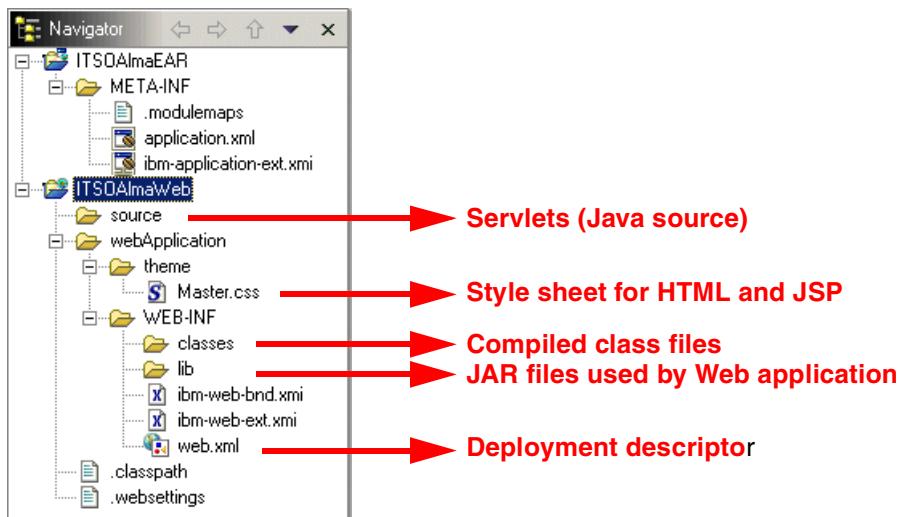


Figure 3-5 Generated elements after Web project creation

For Web application development, the key files managed in a Web project are:

- web.xml** Deployment descriptor for the Web module, as defined in J2EE 1.2. This file contains general information about the Web application, including servlet mappings, security information, and the welcome page. Application Developer provides a special `web.xml` editor for modifying this file which is launched by default when the file is opened in the workbench.
- ibm-web-bnd.xml** WebSphere bindings for the Web application. This file contains bindings to references used at the Web module level as defined by IBM. J2EE provides a mechanism to use local names for external resource objects (for example EJBs and JMS), however, the J2EE specification does not define how these objects are referenced at runtime. This implementation is left to the vendor to implement in their own way, and this file is IBM's implementation.
- ibm-web-ext.xml** IBM-specific extensions to the Web module. This file is used by WebSphere to support additional options beyond

the J2EE specification, such as the reloading interval, and the default error page (if file serving is allowed, and if servlet class invocation is enabled.)

An additional file is also generated in this project.

Master.css This is the default cascading style sheet that the Application Developer page designer will use for any new HTML and JSP pages.

In the EAR project, two files are created:

application.xml Deployment descriptor for the enterprise application, as defined in J2EE 1.2. This file is responsible for associating Web and EJB projects to a specific EAR file. An editor is also provided for this file, which is launched either by double-clicking on the file in the navigator, or selecting the EAR file in the J2EE view, and choosing the *Open in Editor* item from the context menu.

ibm-application-ext.xml IBM-specific extensions to the enterprise application. Similar to those defined in `ibm-web-ext.xml` Web module extension file, but with scope across the whole application.

Adding the IBM-specific J2EE extensions in separate files ensures portability of the application between application servers, while also enabling the application to take advantage of WebSphere's advanced features, if they are available.

Creating the ITSOWSAD sample database

First, ensure that DB2 is installed and running.

If you have not created the ITSOWSAD database as described in “Define and load the ITSOWSAD database” on page 542, open a DB2 command window, navigate to the directory where the sample setup files have been downloaded and extracted (`..\sampcode\setup\dd1`), and execute the following command:

```
db2 -tf ITSOWSAD.ddl
```

This DDL script creates a local database called ITSOWSAD, which has a number of tables, two of which we are interested in at this time: AAPARTS and AAINVENTORY. Each table is populated with some sample data.

DB2 and JDBC 2.0:

It is important to note that both WebSphere 4.0 and Application Developer do not function correctly using the default JDBC 1.0 drivers installed with DB2. To make all of the examples work, you must switch to JDBC 2.0. To do this:

- ▶ Stop all DB2 services and close all applications using the database.
- ▶ Open a command window and navigate to the directory where DB2 is installed: x:/sqllib.
- ▶ Change to the java12 directory.
- ▶ Execute usejdbc2.bat.

This overwrites the JDBC driver file, db2java.zip with a new version containing JDBC 2.0 drivers.

To verify that the database has been created successfully, open a DB2 command window and execute these commands:

```
db2 connect to itsowsad  
db2 select * from itso.aaparts  
db2 select * from itso.aainventory  
db2 connect reset
```

Creating a suitable Java package for the generated servlets

Use the *File -> New -> Other* menu and select *Java package* from the *Java* category to add a new Java package to the Web project. Click *Next*. In the *Folder* field, browse to */ITSOA1maWeb/source*. In the *Package* field enter *itso.wsad.alma.web* and click *Finish*.

The sample project configuration is now complete and we are ready to start developing the application itself.

Building the application

The next section steps through the processes required to build the necessary artifacts for the Almaden Autos Web application. We will be performing the following:

- ▶ Creating a Web application using the Database Web Pages wizard
- ▶ Investigating the generated types
- ▶ Customizing the Web pages using the page designer

Generating the basic Web application

We use the Database Web Pages wizard to create a database Web application. This wizard builds a Web application based on one of two models, *View Bean* or *Tag Lib*. Both models make use of JSP and servlet technology to generate Web pages from SQL queries. Business logic is managed by *view helpers* and presentation is managed by *views*. In the case of the *Tag Lib* model, the view helpers are JSP tags. For the *View Bean* model the view helper is a JavaBean.

We use the *View Bean* model to create our Web application. The Web application generated by the Database Web Pages wizard using the View Bean model is shown in Figure 3-6.

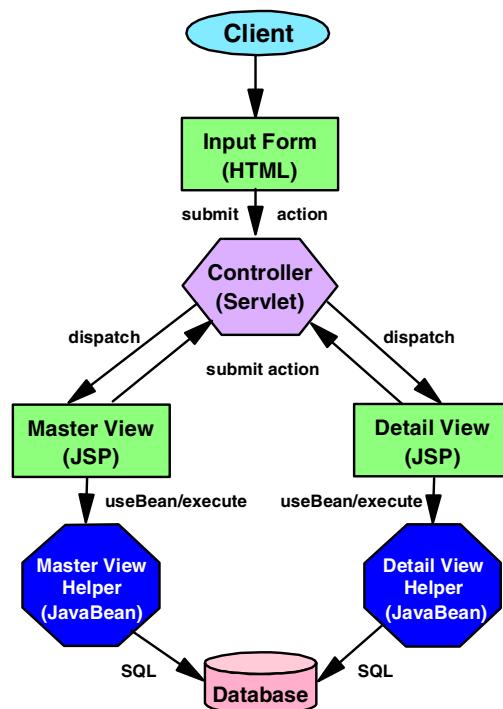


Figure 3-6 Database Web Pages wizard: View Bean model

Once the environment has been successfully configured, we can execute the *Database Web Pages* wizard.

From the Web perspective, first select the ITS0A1maWeb project in the Navigator view, then from the *Open The New Wizard* button  on the Workbench toolbar and select *Database Web Pages* from the *Web* category. (There is also a toolbar icon *Create Web Pages that access and display database fields*.)

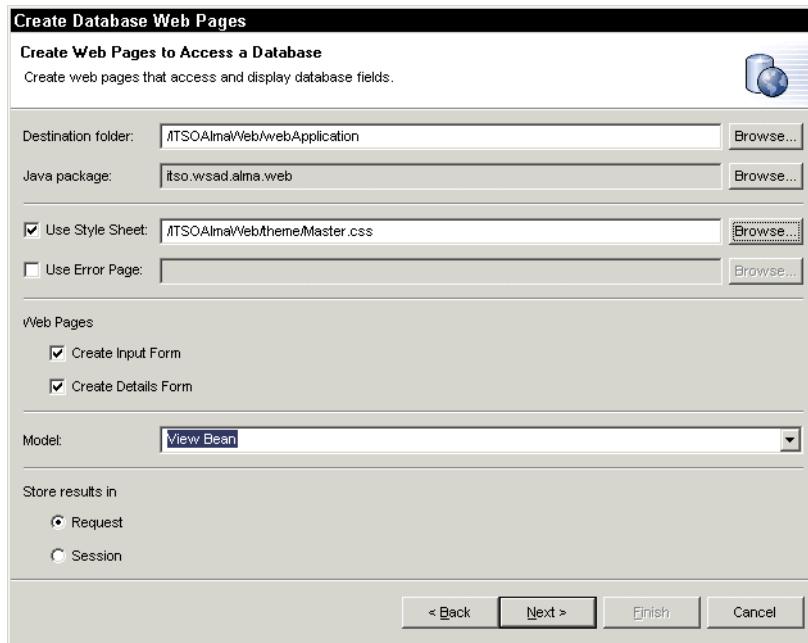


Figure 3-7 Database Web Pages wizard

Complete the first panel of the wizard as shown in Figure 3-7:

- ▶ Enter the *Destination folder* is /ITSOAlmaWeb/webApplication.
- ▶ Enter the *Java package* is itso.wsad.alma.web.
- ▶ Ensure *Use Style Sheet* is selected and points to the style sheet created when the Web project was created (/ITSOAlmaWeb/theme/Master.css).
- ▶ Ensure *Create Input Form* and *Create Details Form* are selected.
- ▶ Ensure the *Model* is *View Bean*.

Note: The Database Web Pages wizard allows two models, *View Bean* and *Tag Lib*. *Tag Lib* generates the JSPs using a tag library that contains specialized tags for database connection and access.

- ▶ Ensure that *Request* is selected for the *Store Results* option.
- ▶ Click *Next*.

We will not be using a predefined SQL statement so skip the Choose an existing SQL Statement panel.

- ▶ Click *Next*.

We have not defined the ITSOWSAD database to Application Developer before, therefore, we must import the schema during this step. On the specify SQL statement information wizard panel, accept the default value *Be Guided through creating an SQL statement* and switch to *Connect to a database and import a new database model*. Click *Next*.

On the Database Connection panel we have to specify how to connect to the database at development time using JDBC. This is different than the approach we will use when we execute the application, because we will later define a JDBC 2.0 data source in the WebSphere test environment server configuration. Complete the panel as shown in Figure 3-8.

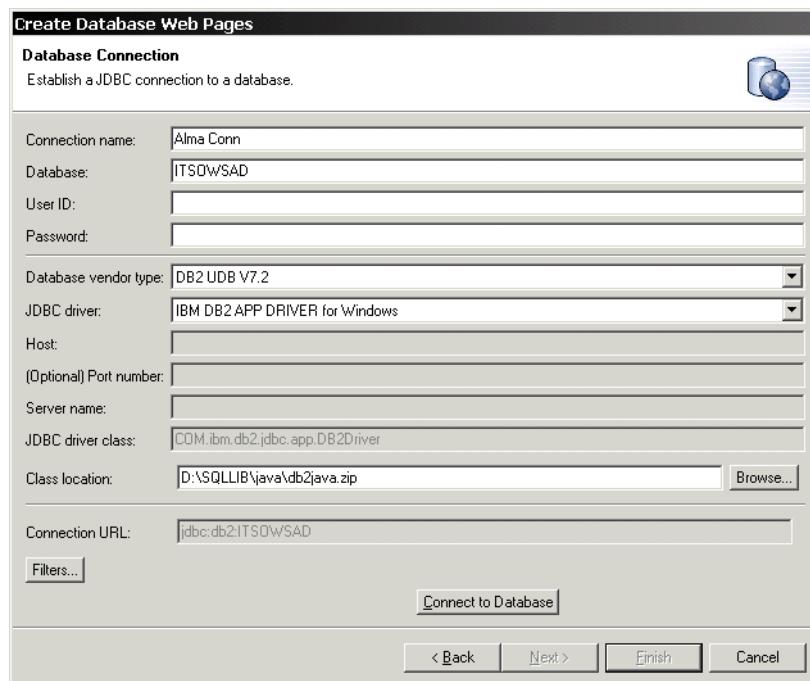


Figure 3-8 Connecting to the ITSOWSAD database

- ▶ Enter a connection name of Alma Conn and a database name of ITSOWSAD.
- ▶ Type a valid user ID and password for DB2 (empty fields usually work as well if your own user ID is allowed to connect to DB2).
- ▶ Leave the *Database Vendor* and *JDBC Driver* as the default, *DB2 UDB 7.2* and *IBM DB2 App Driver for Windows*.

Click on *Filters* so that we can limit which tables are imported. In the Connection Filters dialog (Figure 3-9), enter AA% as New Filter and click *Add Filter*. Then change the Predicate from NOT LIKE to LIKE using the drop-down menu in that column.

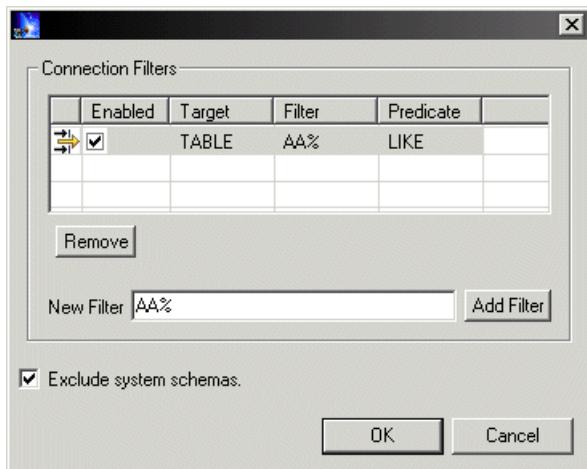


Figure 3-9 Import filter for tables

Click *OK* to close the dialog.

Click on the *Connect to Database* button. This takes you to the *Construct an SQL Statement* panel (Figure 3-10).

- ▶ On the *Tables* tab, select the AAINVENTORY and AAPARTS tables from the ITSO schema and click on the > button.
- ▶ Switch to the *Columns* tab:
 - Expand the AAINVENTORY table, select all the columns except PARTNUMBER and click on the > button to add all columns from the AAINVENTORY table to the query.
 - Expand the AAParts table and add all the columns in the table to the query.
- ▶ In the *Joins* tab (Figure 3-10), draw a connection between the two PARTNUMBER columns to indicate the join you wish to use in the result set (click on PARTNUMBER and drag it to the other table on top of the matching PARTNUMBER column).

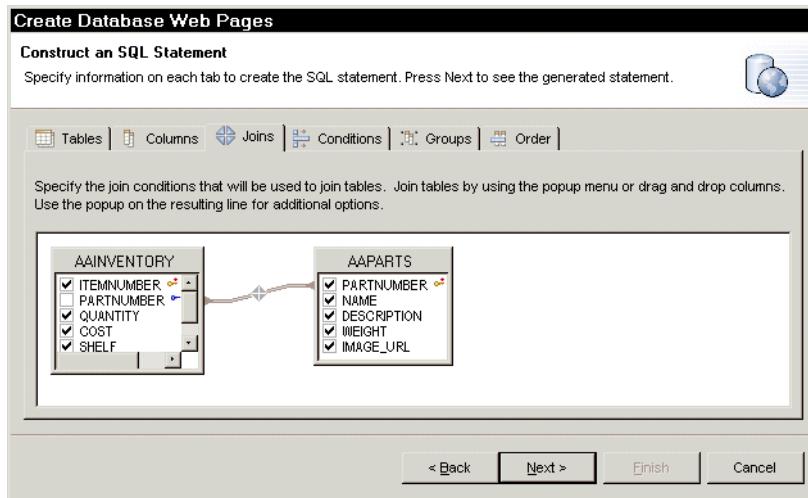


Figure 3-10 Defining the join in the SQL query

Finally, we need to define a parameter in the SQL query that allows us to search by a part number passed into the query from a browser:

- ▶ Select the *Conditions* tab and click on the first column of the first row in the table. A drop-down list appears, containing column names. Select *ITSO.AAPARTS.PARTNUMBER*.
- ▶ In the *Operator* column, select the = option from the drop-down list.
- ▶ In the *Value* field type :partNo as a host variable.

Alternatively select the *Build Expression* option from the drop down list. This opens the *Expression builder* dialog, which provides a number of options on how to define the WHERE clause in the SQL query. To pass a parameter:

- Select the *Constant - numeric, string or host variable* option, click *Next*.
- We select *String Constant* in the following panel because PARTNUMBER is stored as a CHARACTER(10) field in our database. Click *Next* again.
- First select *Host variable name* as the string constant type. In the field at the top of the dialog, which should now be preceded by a colon, type a variable name of partNo. Click *Finish*.

We are now returned to the Construct an SQL Statement panel. Click *Next*.

The resulting SQL query should appear as follows:

```
SELECT
    ITSO.AAINVENTORY.ITEMNUMBER,
    ITSO.AAINVENTORY.QUANTITY,
    ITSO.AAINVENTORY.COST,
    ITSO.AAINVENTORY.SHELF,
    ITSO.AAINVENTORY.LOCATION,
    ITSOAAPARTS.PARTNUMBER,
    ITSOAAPARTS.NAME,
    ITSOAAPARTS.DESCRIPTION,
    ITSOAAPARTS.WEIGHT,
    ITSOAAPARTS.IMAGE_URL
FROM
    ITSOAAPARTS, ITSO.AAINVENTORY
WHERE
    ITSOAAPARTS.PARTNUMBER = ITSO.AAINVENTORY.PARTNUMBER
    AND ITSOAAPARTS.PARTNUMBER = :partNo
```

We can test the query by clicking on the *Execute* button. This displays the *Execute SQL statement* dialog. Complete the following:

- ▶ Click *Execute* and you are prompted for a value for the partNo parameter.
- ▶ In the Value column of the table type 'M100000001' (with quotes) and click *Finish*. (If the *Finish* button is not enabled, click elsewhere in the dialog first to confirm the changes to the table cell.)
- ▶ A single record should appear in the Query results table. Close this window.

Back in the *SQL statement* panel, click *Next* to specify the runtime database connection information. For this example we will be using a JDBC 2.0 data source and connection pool to connect at runtime instead of a direct DB2 JDBC connection:

- ▶ Select the *Use datasource connection* radio button and in the *JNDI name* field type `jdbc/ITSOWSAD`.
- ▶ Enter the *User ID* and *Password* of db2admin (or what was used when DB2 was installed, or leave empty). Click *Next*.

Designing the forms and generation options

The next stage in the *Database Web Pages* wizard enables you to tailor the generated Web application to suit your requirements.

Input form

The first screen displayed relates to the input HTML form which will capture the assigned parameters to the SQL query. The only modifications necessary to this page is to change the label on the input field, and the page title:

- ▶ Select the *Fields* tab from the property sheet and change the *Label* to Part Number.
- ▶ Optionally change the size and max length to 10.

Important: Note the ID of this field. This is the ID that will be used to place the part number in the request and the session data. This value can be changed, however, the rest of the code examples assume it is **partNo**. Code generated prior to WSAD v 4.0.2 may have the ID as partNo suffixed with a number, for example **partNo1**. For information on modifying this in the generated code to match the rest of the code in the book, see “Modifications required if using code prior to Version 4.0.2” on page 80.

- ▶ To change the page title, select the *Page* tab in the property sheet and edit the *Page Title* field to Parts Inventory Inquiry.
- ▶ To change the background color, select the *Page* tab in the property sheet and change the *Background Color* field to white (255,255,255). Note that the color does not change in the dialog.
- ▶ Click *Next*.

Results table

The submit link on the input form passes the part number to a controller servlet generated by the wizard. The controller servlet then invokes the Master View JSP. This result table Master View JSP is the next page we have to tailor.

The default format for the result table contains all of the columns returned by the query. In order to make this more attractive, we want to reduce the amount of detail in the table, as it will be available in the details page. Suitable columns to include on this page could include PARTNUMBER, NAME, QUANTITY and COST.

- ▶ Ensure only the following fields are selected and their column labels match Table 3-1.

Table 3-1 Result table column properties

Column name	Label
AAPARTS.PARTNUMBER	Part Number
AAPARTS.NAME	Name
AAINVENTORY.QUANTITY	Quantity
AAINVENTORY.COST	Cost

- ▶ Clicking the up and down arrows to the right of the column list allows you to sort the change the order in which they are displayed.
- ▶ When changing the label, ensure you either press *Enter* on your keyboard or click elsewhere in the property sheet before selecting another column, because the changes are not committed if focus is immediately lost from the property sheet. You can verify these changes in the preview pane.
- ▶ In the *Page* tab of the property sheet, change the value of the *Page Title* property to *Inventory Inquiry Results*.
- ▶ To change the background color, select the *Page* tab in the property sheet and change the *Background Color* field to white (255,255,255).
- ▶ Click *Next*.

Details page

When the SQL query returns multiple records, the user has the option to select an instance and look at the record in further detail. By clicking on the *Details* link generated on the Master View page, the controller servlet is invoked again. The controller servlet invokes a second JSP, the Detail View JSP, to display the details of the record.

We now tailor the details page. Edit and sequence the properties of the fields as shown in Table 3-2.

Table 3-2 Details form properties

Field name	Label
AAPARTS.PARTNUMBER	Part Number
AAINVENTORY.ITEMNUMBER	Item Number
AAPARTS.NAME	Name
AAPARTS.DESCRIPTION	Description
AAINVENTORY.QUANTITY	Quantity

Field name	Label
AAINVENTORY.COST	Cost
AAPARTS.WEIGHT	Weight
AAINVENTORY.LOCATION	Location
AAINVENTORY.SHELF	Shelf
AAPARTS.IMAGE_URL	Image

- ▶ For now we leave the AAPARTS.IMAGE_URL field in the details view, but we will use this column later to display the actual image.
- ▶ Also change the page title to Inventory Item Details.
- ▶ Change the background color, to white (255,255,255).
- ▶ Finally, click *Next*.

Controller page

The *Specify Front Controller and View Bean Choices* page presents the options for generating the controller servlet and the view beans. In order to adhere as closely as possible to a model-view-controller architecture, take the defaults.

- ▶ Select *Create new Front Controller* to generate a new controller servlet.
- ▶ Select *Create view bean(s) to wrapper your data object(s)* to generate view bean wrappers for the data objects.
- ▶ Click *Next*.

Prefix page

The *Specify Prefix* page displays the prefix used in generating the database Web page files.

- ▶ Change the prefix for the application to Parts.
- ▶ Click *Finish*.

Investigating the generated types

Once you have completed the wizard, you should be returned to the Web perspective. The Navigator view is shown in Figure 3-11.

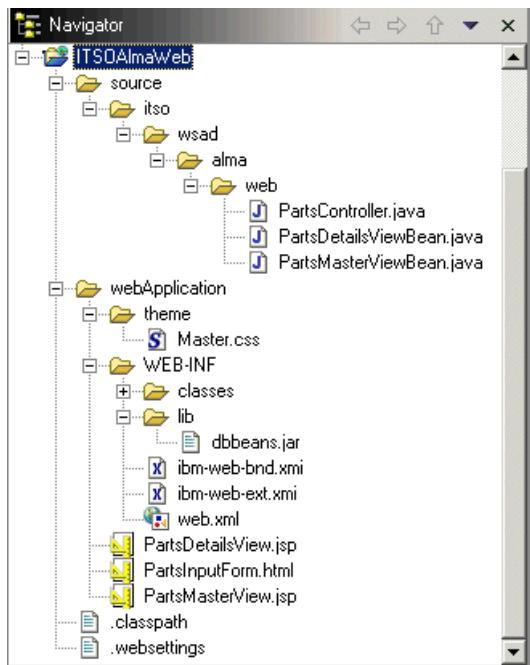


Figure 3-11 Generated types by the Database Web Pages wizard.

The following six types have been generated:

PartsInputForm.html	The HTML input page to start the query.
PartsMasterView.jsp	A JSP to display the result set of the query.
PartsDetailsView.jsp	A JSP to display the details of a selected item.
PartsController.java	The controller servlet that handles the submit from the HTML and JSPs and determines the next page to be invoked based on a command passed on the submit.
PartsMasterViewBean.java	This is the view bean for the master view JSP. It interacts with the database to collect the appropriate data based on the part number submitted in the input form.
PartsDetailsViewBean.java	This is the view bean for the details view JSP. It interacts with the database to collect the appropriate data based on the item selected in the master view.

In addition to the generated types, the wizard performed two more operations:

- ▶ First, it adds a JAR file called dbbeans.jar to the *lib* folder and to the application build path (it will also be included in the resulting WAR file). This JAR provides helper classes to manage the database connection, query and result set, and is used extensively by the view beans that are generated by the wizard.
- ▶ The second operation is related to the Web application deployment descriptor, *web.xml*. This is worth further investigation.

Deployment descriptor *web.xml* file

Open the *web.xml* file from the Web perspective. A number of tabs appear at the bottom of the window.

Select the *Servlets* tab to see the deployment information for the generated servlets and the JSPs. A typical panel is shown in Figure 3-12.

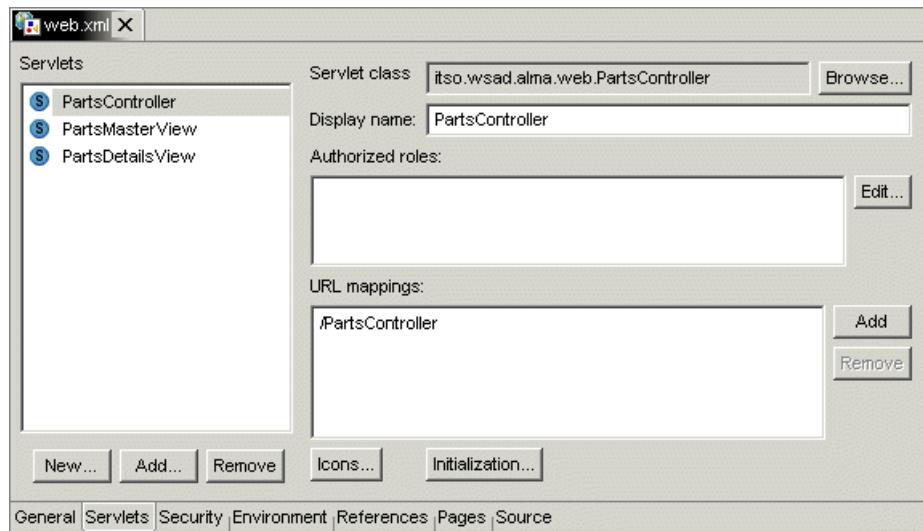


Figure 3-12 Servlet information in the *web.xml* file

The generated servlet and each of the generated JSPs has an entry in this file. The entry provides a number of important pieces of information for the Web application server:

- ▶ The most obvious is the *URL mappings* field, which defines the URL that the Web application can use to invoke the servlet or JSP.

- ▶ The second, less obvious, entry is only available when you click the button marked *Initialization* on the form. Figure 3-13 shows the pop-up dialog if PartsController servlet was selected. Figure 3-14 shows the pop-up dialog if the PartsMasterView.jsp is selected.

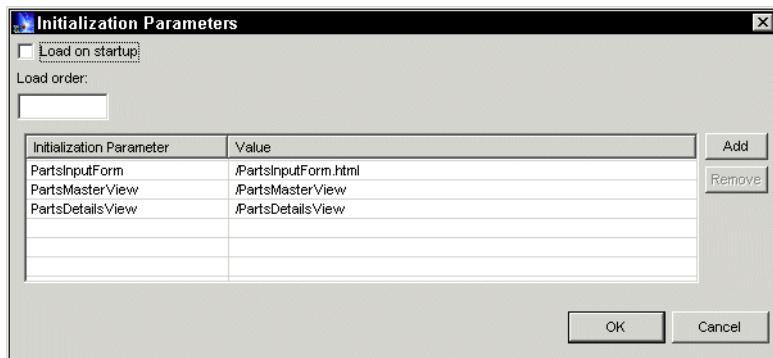


Figure 3-13 Initialization parameters for the generated servlet

The three parameters used by controller servlet define the command values that control navigation from the controller servlet.

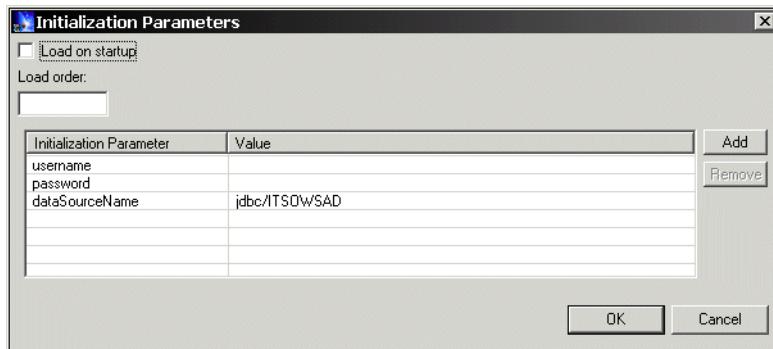


Figure 3-14 Initialization parameters for the generated JSPs

Also note that the initialization parameters for the JSPs contain the database user ID and password parameters along with the data source name that we specified in the Database Web Pages wizard.

Changing the background

When customizing the Web pages in the *Database Web Pages* wizard, we changed the background color to white. If we edit one of the JSPs or the HTML we will see that the background color is not white. This appears to be a side effect of using a style sheet. Even though the JSPs and HTML set the background color to white the style sheet background color overrides it. The background color in the style sheet is BACKGROUND-COLOR: #FFE4B5. We need to change this to white:

- ▶ Open the file /webApplication/theme/Master.css.
- ▶ Modify the background color to be white as shown in Figure 3-15.
- ▶ Save and close the file.

```
BODY
{
    BACKGROUND-COLOR: white;
    COLOR: black;
    FONT-FAMILY: 'Times New Roman'
}
....
```

Figure 3-15 Style sheet Master.css

Adding support for displaying the image

Currently, the database query returns only the image name and displays it on the details page. A common requirement is to display the associated part picture on the page instead of the text. There are a number of alternatives to dynamically display an image from a database query, including storing the image itself in the database as a BLOB type. However, the simplest way to implement is to store the image name in the database record, and provide the file in the application server's file system.

A number of images are provided as part of the sample code of this redbook. Here are the steps to import the GIF images into a new folder of the Web application:

- ▶ Select the *webApplication* folder in the ITS0A1maWeb project.
- ▶ Select *New -> Folder* item from its context menu.
- ▶ In the new folder dialog, enter a folder name of *images*. Click *Finish*.

Importing the images

Once we have created the folder, we can import the images.

- ▶ Import the parts images:
 - Select the new folder then the *File* -> *Import* menu.
 - Choose *File system* in the next panel and click *Next*.
 - Browse to the directory containing the sample code and navigate to the setup images directory (..\\sampcode\\setup\\images).
 - Click the *Select All* button and then click *Finish*.
- ▶ Import the Almaden Autos image:
 - Select the new folder then the *File* -> *Import* menu.
 - Choose *File system* in the next panel and click *Next*.
 - Browse to the directory containing the Web code and navigate to the setup images directory (..\\sampcode\\wsadweb\\images).
 - Click the *Select All* button and then click *Finish*.

The image files should now appear under the webApplication/images folder of the Web project.

The next step is to modify the PartsDetailsView.jsp file. Open the page designer from the navigator by double-clicking on the JSP file. The appearance of the page in the *Design* tab of the *Page Designer* view should be similar to Figure 3-16.

A number of icons appear in the page, relating to embedded JSP and JavaScript code:

- ▶ The  icon represents an HTML comment.
- ▶ The  icon represents a JavaBean, which has been inserted into the page through the JSP <jsp:useBean> tag.
- ▶  refers to a JSP expression, which is surrounded by <% and %> tags in the source. Most of these expressions extract or set a property value from the JavaBean (for example the part number).

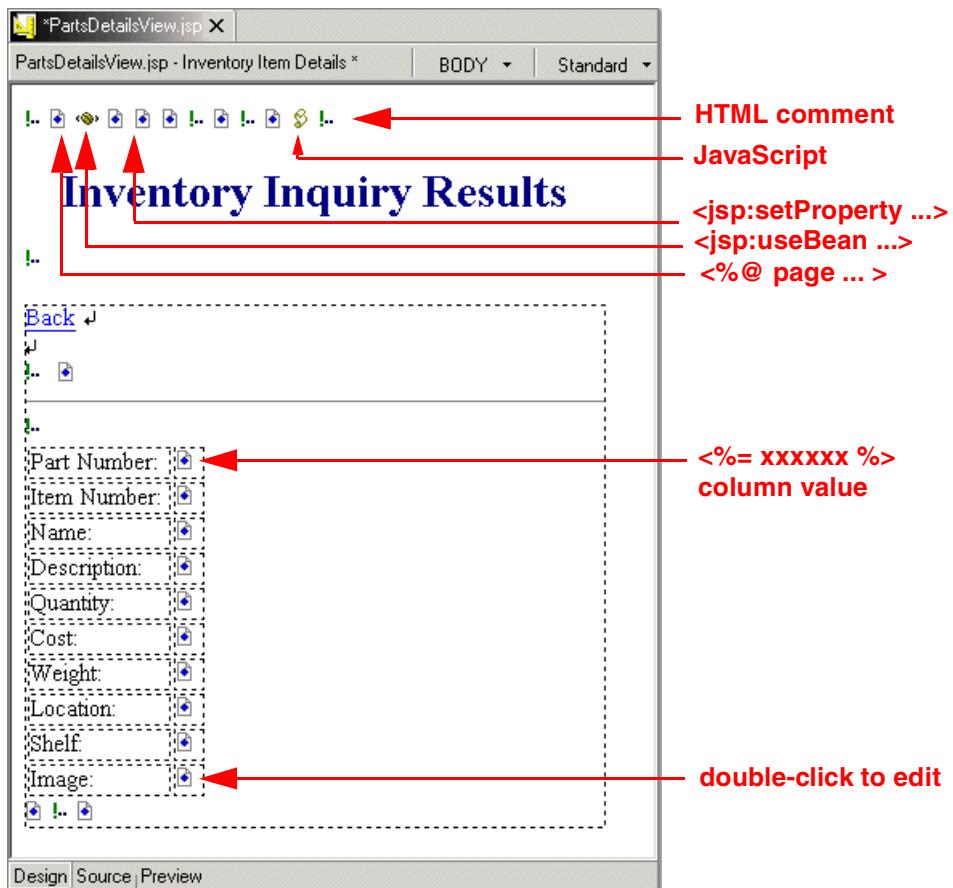


Figure 3-16 Generated item detail page

Modifications

First we add the Almaden Autos image logo to the top of the page:

- ▶ Drag the almadenautos.gif file from the images folder in the Navigator view to the page designer view, just after the HTML comment icon  , or
- ▶ On the *Design* tab, place the cursor after the HTML comment icon and click on *Insert -> Image File -> From File* menu. Browse to the images folder of the Web application and select the almadenautos.gif file.

Image URL

The field of interest at this time is the code to retrieve the image from the result set. Double-click on the JSP expression icon next to the Image label. This opens the script editor (Figure 3-17).

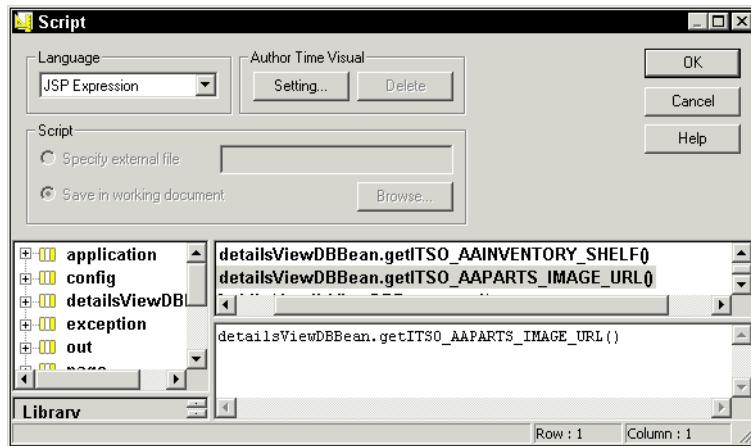


Figure 3-17 Script expression editor for the image field

We can see that the image name is retrieved by the expression:

```
detailsViewDBBean.getITSO_AAPARTS_IMAGE_URL()
```

It would be better if we could move the image display to the left of the information retrieved:

- ▶ Close the script expression editor and select one of the labels on the table.
- ▶ Select the *Table -> Add Column -> Add to Left* menu.
- ▶ To make one large cell that spans the entire column, select all of the cells and select the *Table -> Join Selected Cells* menu.
- ▶ The next step is to add the new dynamic image into the page:
 - Select the new column, and click on the *Insert -> Dynamic Elements -> Dynamic Image* menu.
 - The *Attributes* window appears. In the *SRC attribute* field of the *Dynamic* page, enter the following:
 - "images/" + detailsViewDBBean.getITSO_AAPARTS_IMAGE_URL()
 - Switch to the *Image* tab and enter a *Horizontal spacing* of 50 to separate the image from the text descriptions. Click *OK*.
- ▶ Finally remove the image row from the table. Select the Image label in the table, then select *Table -> Delete Row*.

The completed page as it appears in the *Design* tab of the page designer is shown in Figure 3-18.

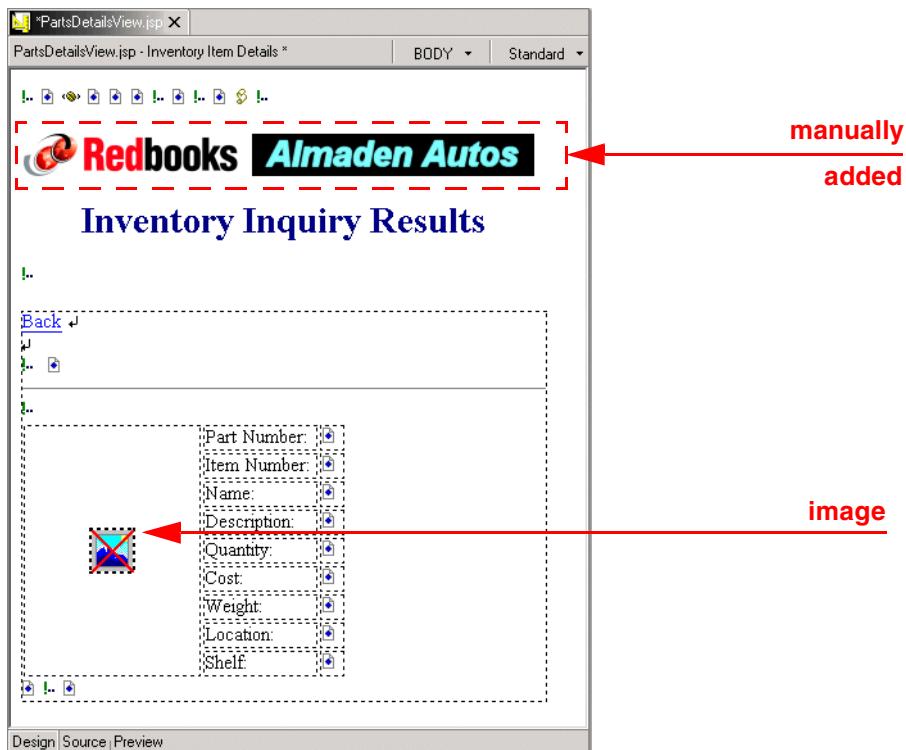


Figure 3-18 Tailored details page

Do not be concerned by the crossed out image—this is because we did not provide a sample image to be included in the page design. It should appear correctly at runtime.

Close the *PartsDetailsView.jsp* file and save the changes.

Open *PartsInputForm.html* and *PartsMasterView.jsp* and add the Almaden Autos image to those pages as described above.

Final thought

We have now completed developing our simple application. Although the *Database Web Pages* wizard is very convenient for a prototype-style application such as the sample, the number of changes required to make the pages behave in the required manner significantly reduces its value.

We expect most developers will prefer to build their servlet and JSP applications by hand rather than selecting the wizard, potentially using a servlet and JSP framework such as Apache Struts (<http://jakarta.apache.org/struts/>).

Modifications required if using code prior to Version 4.0.2

As previously mentioned if the Application Developer is prior to Version 4.0.2, you will have to change the attribute ID for the part number stored in the session data. Prior to V4.0.2 Application Developer generated this name with a numeric suffix, for example, **partNo8**.

Generated attribute name

The part number selected by the user is the key information used throughout the application. The *Database Web Pages wizard* generates the attribute name for the part number key as **partNox**, where x is a number that varies over multiple invocations of the wizard.

The selected part number is stored in the HTTP session and is used by all servlets as the key for database retrieval. Throughout the rest of the book we will be modifying this application and will access the part number stored in the session data through the attribute name generated by the wizard.

All sample code provided with the book uses the attribute name **partNo** to access the part number in the session data. We have to verify the name of the attribute the wizard generated for us (see “Input form” on page 69). Open the *PartsInputForm.html* file and look for the lines generated for the part number (Figure 3-19).

```
<TH>
    Part Number:
</TH>
<TD>
    <INPUT NAME="partNo8" TYPE="text" SIZE="20" MAXLENGTH="40" VALUE="">
</TD>
```

Figure 3-19 Attribute name generated for the part number

If the generated attribute name is not partno, you have two options:

- ▶ Modify the generated code
- ▶ Use the value generated here throughout the rest of the book

We recommend that you modify the value generated by the wizard. To do this, use the *Search* facility to search all JSP, HTML and Java files for the field name (Figure 3-20).

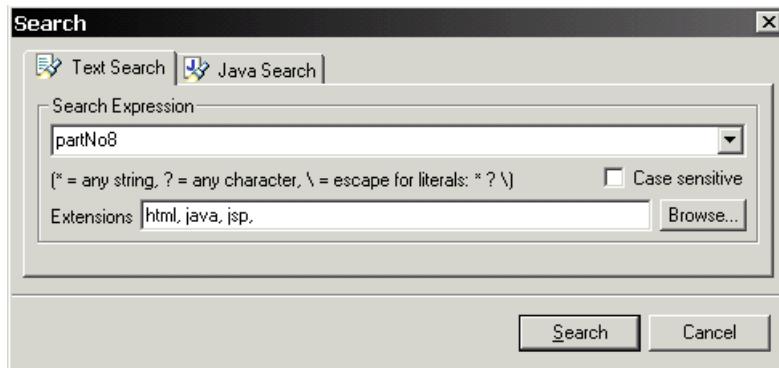


Figure 3-20 Search for references to the session attribute name

The search will find many matches in five files (Figure 3-21).

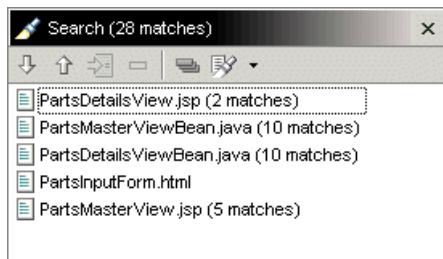


Figure 3-21 Session attribute name search results

Open the files and change all hard-coded references of the value partNoX to partNo.

Deploying and testing the Web application

Once we have completed development, the next logical step is to deploy it to a local server for unit testing. We will be using the same server to deploy all of the sample Web applications in the redbook.

While this is not the most realistic distributed application simulation, it significantly reduces the resources that are required later when multiple applications are communicating using Web services.

The following sections describe how to configure and run the WebSphere Application Server inside Application Developer:

- ▶ Creating a server project

- ▶ Creating a server configuration and instance
- ▶ Associating the project with a server
- ▶ Adding a JDBC data source
- ▶ Executing the application

Creating a server project

Before we can create a server instance, we must first define a server project. Select the *File -> New -> Project* menu and then *Server Project* from the *Server* category. Click *Next*.

In the *Create a new server project* dialog, enter *ITSOWSADServer* as the project name, and click *Finish*.

Creating a server instance and configuration

Application Developer should now be open in the server perspective, with the new *ITSOWSADServer* project selected. This is a very useful perspective, and we recommend leaving it open whenever you are testing J2EE applications.

Select the *ITSOWSADServer* item in the Navigator view then use the *File -> New -> Server Instance and Configuration* menu (or click the  icon). The wizard shown in Figure 3-22 opens.

In the *Create server configuration and instance* wizard, type *ITSOWSADWebSphere* as the server name. Expand *WebSphere Servers*, and select *WebSphere v4.0 Test Environment* as the instance type. Click *Finish*.

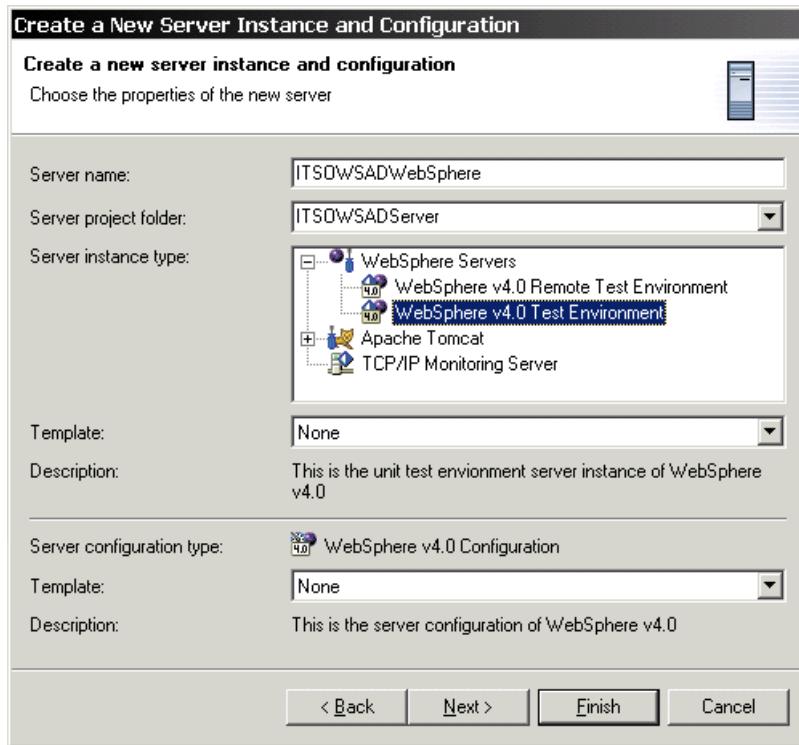


Figure 3-22 Creating a new server instance and configuration

Unlike VisualAge for Java, Application Developer has the option to deploy to and test with both local and remote instance of the WebSphere application server, and additionally, Apache Tomcat. For more explanation about the different options, see “Server project” on page 47.

Associating the project with a server configuration

The next step in our deployment is to associate the EAR project with the server configuration:

- ▶ From the server configuration view in the server perspective, select the server configuration *ITSOWSADWebSphere*.
- ▶ Use the *Add Project -> ITSOAlmaEAR* context menu (Figure 3-23).

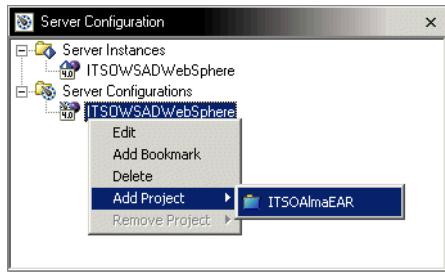


Figure 3-23 Associating the EAR project with the server configuration

The project should now appear under the server configuration in the view. We will be using this approach again later to associate other Web applications with the same server configuration and instance. This allows us to test multiple J2EE applications simultaneously on a single instance of the application server.

Adding the JDBC data source

When we ran through the Database Web Pages wizard, we specified a JDBC 2.0 data source of `jdbc/ITSOWSAD`. This data source must be defined in the application server configuration before our application will work:

- ▶ In the server perspective, select the *ITSOwsADWebSphere* server configuration from the server configuration view (where we associated the EAR project) and select *Open* from the context menu to open the editor.
- ▶ Switch to the *Data source* tab (Figure 3-24).

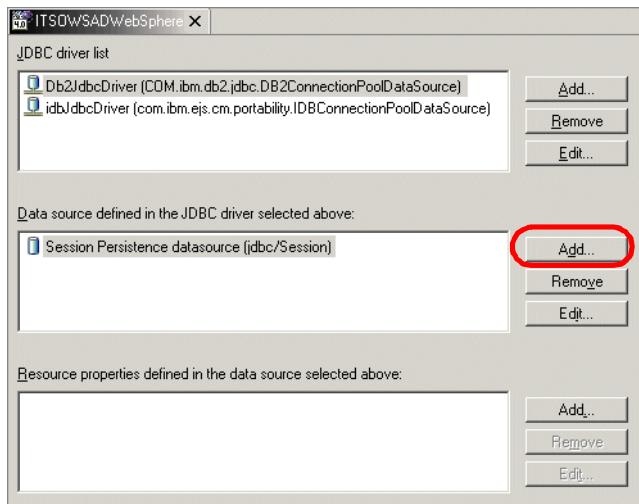


Figure 3-24 Data source definition for a server configuration

DB2 JDBC 2.0 drivers

If you have done development with previous releases of WebSphere and VisualAge for Java, it is likely that you will have used either the COM.ibm.db2.jdbc.app.DB2Driver or COM.ibm.db2.jdbc.net.DB2Driver classes for JDBC 1.0 access. These are no longer the recommended mechanism to create JDBC connections.

There are now two JDBC 2.0 drivers supplied with the latest release of DB2. The first is COM.ibm.db2.jdbc.DB2ConnectionPoolDataSource, which we will be using here, and is suitable for most applications. A second driver is also supplied which supports distributed transactions across both JDBC and JMS using JTA - COM.ibm.db2.jdbc.DB2XDataSource. Unfortunately, a detailed discussion of this support is outside the scope of this redbook.

To add a new DB2 JDBC data source, select the Db2JdbcDriver item from the list, click *Add* next to the data source list, and complete the dialog box (Figure 3-25).

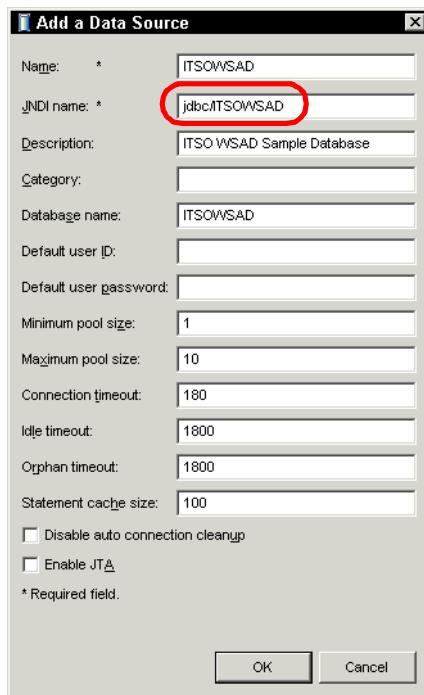


Figure 3-25 JDBC data source definition

Click *OK* and close the configuration editor, and save the changes.

We have now performed all of the configuration steps required to create a realistic WebSphere environment to test our application. The final step is to start the server and test the site.

Executing the Web application

To start the application do the following:

- ▶ Select the first page of the database application, *PartsInputForm.html*, in the *webApplication* folder of the *ITSOA1maWeb* project, and select the *Run on Server* option from its context menu.
- ▶ In one step, this initiates the publishing process, starts the server, and launches a browser on the correct URL.

Note: This action starts the server in debug mode and enables breakpoints in servlets and JSPs. To start the server in normal mode, select the *ITSOWSADWebSphere* server in the Servers view and select *Start*. Then select an HTML file and *Run on Server*. The application executes slower in debug mode.

- ▶ When WebSphere is launching, the Console view should appear and display messages relating to its execution.
- ▶ After the start-up has completed, the embedded browser should be open at <http://localhost:8080/ITSOA1ma/PartsInputForm.html>.

Embedded browser behavior:

When Application Developer launches its Microsoft Internet Explorer plug-in the content area inside the browser does not automatically receive focus. When launching an application, switching between perspectives or switching between views, always click first on the blank area of the displayed page before selecting a link or field element. This returns focus and ensures that your Web application behaves as expected.

- ▶ Figure 3-26 shows the input form for our parts inventory inquiry application as it appears in the embedded browser.

Note: You can also use an external browser with the same URL to connect to the internal server.

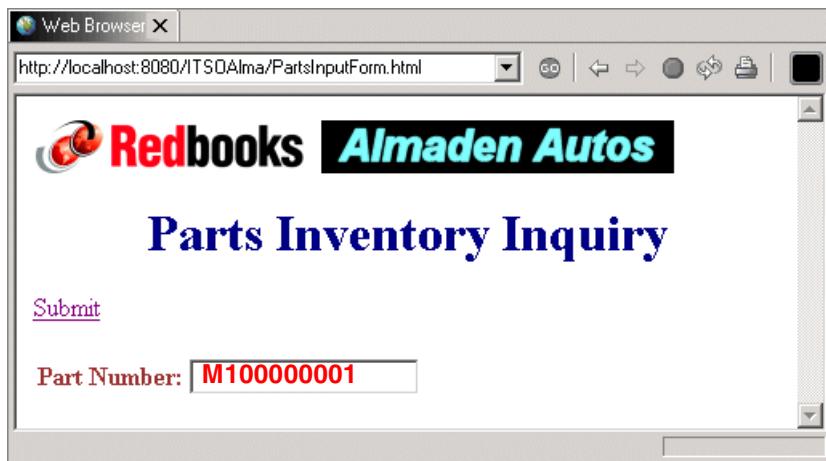


Figure 3-26 Parts inventory inquiry application in the internal WebSphere server

- ▶ To test if our query works, enter a part number of M100000001 and click the *Submit* link.
- ▶ The *PartsControllerServlet* servlet gets control and calls the *PartsMasterView* JSP, which uses the *PartsMasterViewBean* to execute the SQL query. The JSP displays the result table. If the database query is correct, the results should be displayed, as shown in Figure 3-27.

Select	Part Number	Name	Quantity	Cost
<input checked="" type="checkbox"/>	M100000001	CR-MIRROR-L-01	10	89.99

Figure 3-27 Results table for test query

- ▶ Because there is only one record that matches our query, select the row and click the *Details* link to invoke the *PartsControllerServlets* servlet, which dispatches the request to the *PartsDetailsView* JSP, which uses the *PartsDetailsViewBean* to retrieve the data.
- ▶ The details page should display results similar to Figure 3-28.

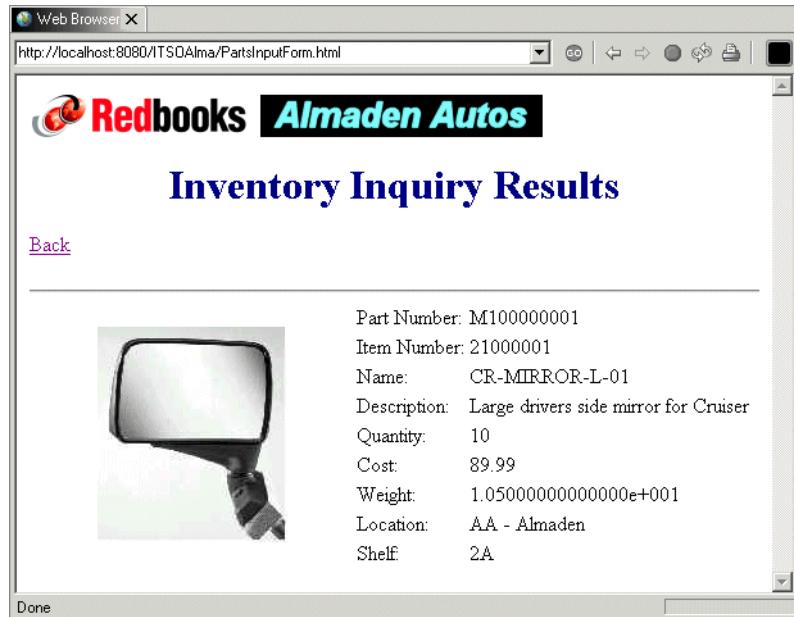


Figure 3-28 Details page for test query

Congratulations! You have just completed developing and deploying your first Web application to the WebSphere V4.0 test environment.

Summary

During this chapter we focused on building a simple application to retrieve information from DB2 with JDBC, using both servlets and JSPs. We also demonstrated how to publish the application to the WebSphere test environment.

In the next chapter, we discuss the XML development capabilities of Application Developer and how to create an XML schema from a relational table.

Quiz: To test your knowledge of the Web development features of Application Developer, try to answer the following five questions. The answers are provided in Appendix C, “Quiz answers” on page 559.

1. Which file contains the information which links Web and EJB projects to the EAR file?
2. To add an external parameter to an SQL query, which of the following options should be selected in the expression builder?
 - a.) Function
 - b.) Constant
 - c.) Subquery
3. Where is the information stored which links the servlets and JSPs together which are generated from the Database Web Pages wizard?
4. Where is the Web application context root stored?
5. What are the class names for the two JDBC 2.0 DB2 drivers?



XML support in Application Developer

In this chapter, we introduce XML and the XML development capabilities of WebSphere Studio Application Developer. In particular, the chapter provides:

- ▶ A brief overview of the key XML concepts a developer of Web applications and Web services is typically confronted with:
 - Basic XML concepts including XML namespaces
 - XML schema definitions (XSD)
 - XML processing with parsers and XSL
- ▶ An introduction to the following Application Developer XML tools:
 - RDB to XML mapping editor
 - XML editor and XML schema editor
 - XML parsers
 - XSL processor

An XML primer

This section gives a short overview to the XML concepts developers of Web services have to be familiar with:

- ▶ XML basics—tags, well-formedness, DTDs, validation
- ▶ XML namespace concept
- ▶ XML schema (XSD)
- ▶ XML processing: parsers, XSL style sheets

Feel free to skip this section if you already are *XMLified*.

Background

XML stands for *eXtensible Markup Language*. XML is developed through the World Wide Web Consortium W3C. It is a meta language, meaning a language for defining other languages. XML by itself does not define any tags; it provides a facility to define custom tags and the structural relationships between them.

XML is a markup language for the exchange of structured documents. It extends the content with some indication of what role that content plays. Hence, XML represents the structure of the content as well as the content itself.

XML is derived from SGML, the Standard Generalized Markup Language, whose predecessor GML was invented by IBM in the 1960s for describing documents in a device-independent fashion. XML is a subset of SGML and compatible with it. Initial focus is on serving structured documents over the Web.

W3C issued XML 1.0 on February 10th, 1998. The second edition of it was published on October 9th, 2000.

Relationship to HTML

Actually, HTML and XML cannot be compared. While XML is a meta language, HTML is a well-defined language. HTML has a fixed set of tags. The meaning of the HTML tags is defined in the W3C standards specifications (or the implementation of a particular browser, whichever came first).

HTML markup is well suited for describing the visual appearance of a human-readable document. HTML defines many presentation instructions for optimizing the display in a browser. With HTML markup, however, the structure inherent to the data is lost; HTML does not maintain the integrity of data type information, the containment hierarchy, or other relationships between entities. XML provides these features.

XML base concepts

Let us start with an example (Figure 4-1).

```
<?xml version="1.0" ?> ← Processing Instruction
<welcome> ← Root Element
  <message> ← Child element
    Welcome to the ITSO SG2462
  </message>
  <book-info>
    <book-title>
      Web services & TBM WSAD ← Character data
    </book-title>
    <date type="firstEdition" >
      <day>23</day>
      <month>08</month>
      <year>2001</year> ← End tag
    </date>
    <date type="lastEdition" >
      <day>23</day>
      <month>08</month>
      <year>2001</year>
    </date>
    <email>redbook@us.ibm.com</email>
    <filename src="sg246292.pdf" /> ← Empty element
  </book-info>
</welcome>
```

Figure 4-1 XML file

We will explain the elements in this XML file in detail in the next sections. The file spawns the following content structure (Figure 4-2):

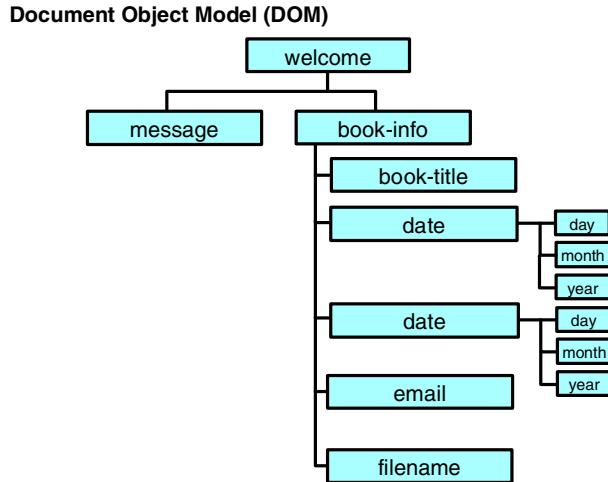


Figure 4-2 XML document structure

XML markup

XML documents are text-based and composed of *markup* and *content*.

- ▶ Markup instructs XML processors about how to treat the content, and how it is organized.
- ▶ Content is the character data you would see on a printed or displayed page.

There are six kinds of markups that can occur in an XML document:

Elements

Elements are the most common form of markup. Elements identify the content they surround:

```
<Date>22.11.2000</Date>
```

Elements begin with a *start tag* `<Date>` and end with an *end tag* `</Date>`.

Non-empty elements contain child elements or character data. Empty elements have no content and can be written in one of two forms: `<empty-element></empty-element>` or `<empty-element/>`.

Attributes

Attributes are name-value pairs that occur inside element tags after the element name, such as the `Price` element has a `currency` attribute:

```
<Price currency="DEM">24.95</Price>
```

All attribute values must be quoted with " " or ' '. They specify the characteristics of an element.

Entity references

Entity references insert reserved characters or arbitrary unicode, refer to repeated or varying text, or include the content of external files:

```
&apos; <!-- Apostroph -->  
&#x211E; <!-- German umlaut -->
```

Entity references begin with the ampersand and end with a semicolon.

The XML specification predefines five reserved entity references `<`, `>`, `&`, `"`, and `'` representing the characters <, >, &, “, and ‘. Custom entities can be declared.

Comments

Comments are not part of the textual content of an XML document. They begin with `<!--` and end with `-->`:

```
<!-- This is a comment. -->
```

An XML processor is not required to pass comments along to an application.

Processing instructions (PIs)

PIs are an escape hatch to provide information to an application. Data that follows the PI target is for the application that recognizes the target:

```
<?PiTarget PiData?>
```

PIs are not textually part of the XML document, but the XML processor is required to pass them to an application. The targets XML and xml are reserved for future XML standardization efforts.

CDATA sections

CDATA sections instruct the parser to ignore most markup characters. They start with <! [CDATA and end with]]>.

```
<! [CDATA[ if( &b <= 3 ) { ... }; ]]>
```

This CDATA section wraps some source code. & and < do not have to be entity references as elsewhere in the document.

Well-formed XML documents

Although XML does not have any predefined tags, there are certain rules that each XML document has to follow in order to be *well-formed*. By definition, if a document is not well-formed, it is not XML.

- ▶ Tags cannot be inferred; they must be explicit.
- ▶ A document must contain one root element.
- ▶ All beginning tags and ending tags match up.
- ▶ Empty tags must end with />.
- ▶ Tags must nest correctly.
- ▶ An entity cannot contain a reference to itself.
- ▶ Entities must be declared before they are used.
- ▶ Element names are case sensitive.
- ▶ All attribute values must be enclosed in single or double quotes.
- ▶ No attribute may appear more than once on the same start-tag or empty-tag.

Note: Similar rules exist for HTML; the XHTML specification defines them. However, HTML browsers also accept HTML documents that are not well formed.

Document type definitions (DTDs)

Well-formed documents offer many of the advantages of XML such as extensibility and adding structure to a set of data. However, exchanging information through XML documents requires additional functionality: it must be possible to build a common grammar for a set of documents and to check compliance automatically.

Document type definitions (DTDs) address these needs. A DTD defines:

- ▶ The allowed sequence and nesting of tags
- ▶ Elements and their attributes including multiplicity
- ▶ Attribute values and their types and defaults
- ▶ Entities and notation

DTDs express meta-information about a document's content. With a DTD, an XML parser can verify the DTD compliance of an XML document. A DTD can be used in one or more XML documents.

Document types can be declared externally or internally:

```
<!DOCTYPE rootElement SYSTEM "filename.dtd">
<!DOCTYPE rootElement [ ...(declarations)... ]>
```

A DTD identifies the root element of the document and may contain additional declarations. It must be the first item in the document after PIs and comments. A mix of external and internal DTD elements is possible.

Here is an excerpt of the DTD of our example from Figure 4-1 on page 93:

```
<!ELEMENT welcome (message,book-info*)>
<!ELEMENT message (#PCDATA)>
<!ELEMENT book-info (...)>
```

Because DTDs are superseded by XML schemas, we do not go into more detail here.

Validation

A well-formed document is *valid* only if it contains a proper document type declaration and if the document obeys the constraints of that declaration.

All valid documents are well-formed; not all well-formed documents are valid.

XML namespaces

Applications associate the content of a document with its element names (tags). Applications might process two documents with elements using the same tag. This tag may have a different meaning in each of the two documents. Hence, the *namespace* concept is introduced in order to eliminate naming collisions.

XML namespaces are defined by a W3C recommendation, dating January 14, 1999. As we will see in part two of this book, the standards defining the Web services oriented architecture make heavy use of this powerful concept.

Tag names should be globally unique, but for performance reasons they also should be short; in order to resolve this conflict, the W3C namespace recommendation defines an attribute `xmlns` which can amend any XML element. If it is present in an element, it identifies the namespace for this element.

The `xmlns` attribute has the following syntax:

```
xmlns=localQualifier:"globallyUniqueName"
```

The globally unique name uses URI syntax, but it is *not* a real URI that can be accessed with a browser through HTTP. Predefined global names exist, for example for the data types defined in XML schema (see below), and in the SOAP standard. (See Part 3 of this book.)

In the following customer element definition, an accounting namespace is defined in order to be able to distinguish the element tags from those appearing in customer records created by other business applications:

```
<acct:customer xmlns:acct="http://www.my.com/acct-REV10">
  <acct:name>Corporation</acct:name>
  <acct:order acct:ref="5566"/>
  <acct:status>invoice</acct:status>
</acct:customer>
```

The *namespace definition* in the first line assigns the global name `http://www.my.com/acct-REV10` to the local qualifier `acct`. This qualifier is used on the element names such as `name` in order to attach them to the namespace.

A second application, for example a fulfillment system, can assign a different namespace to its customer elements:

```
<ful:customer xmlns:ful="http://www.your.com/ful">
  <ful:name>Corporation</ful:name>
  <ful:order ful:ref="A98756"/>
  <ful:status>shipped</ful:status>
</ful:customer>
```

An application processing both data structures is now able to treat the accounting and the fulfillment data differently.

There is a default namespace. It is set if no local name is assigned in the namespace definition:

```
<acct:customer xmlns="http://www.my.com/acct-REV10"
    xmlns:acct="http://www.my.com/acct-REV10">
    <name>Corporation</name>
    <order acct:ref="5566"/>
    <status>invoice</status>
</acct:customer>
```

In this example, all tags in the customer record are qualified to reside in the namespace `http://www.my.com/acct-REV10`. No explicit prefix is needed because the default namespace is used. Note that the default namespace applies to any attributes definitions.

XML schema

So far, we have only introduced DTDs as a notation for meta information about XML documents. DTDs have drawbacks; for example, they use a different syntax than the other definitions in an XML file. They also lack data typing capabilities as well.

XML schemas (XSDs) bring to XML the rich data descriptions that are common to other business systems. A schema allows to precisely define cardinality constraints and enforce data type compliance. XSDs are defined by a W3C recommendation dating May 2, 2001.

XSDs are XML documents themselves: they may be managed by XML authoring tools, or through any XML processor (and can be validated as well). In other words, there is an XML schema for XML schema.

Its worth noting that XSDs can be imported into XML files. At processing time, the imported schemas must be accessible (for example using HTTP).

Elements of a schema definition

The following artifacts can be defined in an XML schema:

Declarations	Declarations enable elements and attributes with specific names and types to appear in document instances. Declarations contain definitions.
Definitions	Definitions create new types, either simple and complex ones. They can include facades constraining the defined types.

Simple types	Simple types cannot have element content. They cannot carry attributes.
Complex types	Complex types have elements in their content and may carry attributes.

As a first example, we use the `simpleType` element to define and name a new simple type:

```

<?xml version="1.0"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://www.ibm.com"
        xmlns:TestSchema="http://www.ibm.com">

    <simpleType name="ZipCodeType">
        <restriction base="integer">
            <minInclusive value="10000"/>
            <maxInclusive value="99999"/>
        </restriction>
    </simpleType>

    <!-- element definitions skipped -->
</schema>
```

The restriction element indicates the existing base type and specifies two *facets* `minInclusive` and `maxInclusive`. These facets constrain the permitted value range of the new type.

Note that we have defined the default namespace for our schema as the standard XML schema namespace `http://www.w3.org/2001/XMLSchema`; we have also defined our own, schema specific namespace `http://www.ibm.com`.

The second part of our example shows several element declarations and two complex type definitions. The resulting schema is suited to represent a list of address records:

```

<element name="address">
    <complexType>
        <sequence minOccurs="1" maxOccurs="1">
            <element ref="TestSchema:street"/>
            <element ref="TestSchema:zipCode"/>
            <element ref="TestSchema:city"/>
        </sequence>
    </complexType>
</element>
<element name="street" type="string"/>
<element name="zipCode" type="TestSchema:ZipCodeType"/>
<element name="city" type="string"/>
```

```

<element name="addressList">
    <complexType>
        <sequence minOccurs="0" maxOccurs="unbounded">
            <element ref="TestSchema:address"/>
        </sequence>
    </complexType>
</element>

```

sequence is a reserved XSD keyword, defining the inner structure of a complex type (its *content model*). The other two XSD content models are choice and all.

The type attribute is optional; it contains a reference to the type definition, which is either defined in an XSD file such as ours (example: TestSchema:ZipCodeType) or a predefined standard data type (example: string in the default XSD namespace).

Assuming that the XSD defined above is saved as C:\temp\TestSchema.xsd, a sample XML file that validates against this schema is:

```

<?xml version="1.0"?>
<x:addressList xmlns:x="http://www.ibm.com"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.ibm.com file:///C:/temp/TestSchema.xsd">
    <x:address>
        <x:street>x:Vangerowstrasse</x:street>
        <x:zipCode>69115</x:zipCode>
        <x:city>x:Heidelberg</x:city>
    </x:address>
    <x:address>
        <x:street>x:Bernal Road</x:street>
        <x:zipCode>90375</x:zipCode>
        <x:city>x:San Jose</x:city>
    </x:address>
</x:addressList>

```

Note that the XSD file has to be accessible through HTTP. In the example, file:///C:/temp/TestSchema.xsd is a true URL pointing to a location. Namespaces such as http://www.ibm.com on the other hand just use the URI notation to be globally unique. Assuming that noProtocol://www.ibm.com is globally unique, it would be a valid namespace name as well.

The syntax of the schema location attribute defined by XML standards is:

```
xsi:schemaLocation="targetnamespaceURI locationURI">
```

This is a quite tricky notation because it is not obvious that two different entities with a similar syntax, but different semantics appear in the same attribute. Even worse, the locationURI often is not a full network address, but a simple local file name.

Target namespace

Several of the XSD elements introduced above can define a *target namespace*, which is either absent or a namespace name.

The target namespace serves to identify the namespace within which the association between the element and its name exists. In the case of declarations, this association determines the namespace of the elements in XML files conforming to the schema. An XML file importing a schema must reference its target namespace in the schemaLocation attribute. Any mismatches between the target and the actual namespace of an element are reported as schema validation errors.

In our example, the target namespace is `http://www.ibm.com`; it is defined in the XSD file and referenced twice in the XML file. Any mismatch between these three occurrences of the namespace lead to validation errors.

In this section, we only introduced the most important XML schema concepts; there are many more features. The XSD specification consists of three parts: *primer*, *structures*, and *data types*. There are numerous tutorials available on the Internet. A good starting point is <http://www.w3.org/XML/Schema>.

Processing XML

XML is a way to store and describe data. We also want programs to act on such data. There are two ways to do so:

- ▶ An *XML parser* can produce a representation of the XML data that can be accessed through language APIs (for example, Java).
- ▶ An *XML processor* can apply an XSL style sheet and transform the source XML document to any text format (another XML file or an HTML, for example).

Figure 4-3 gives an architectural overview.

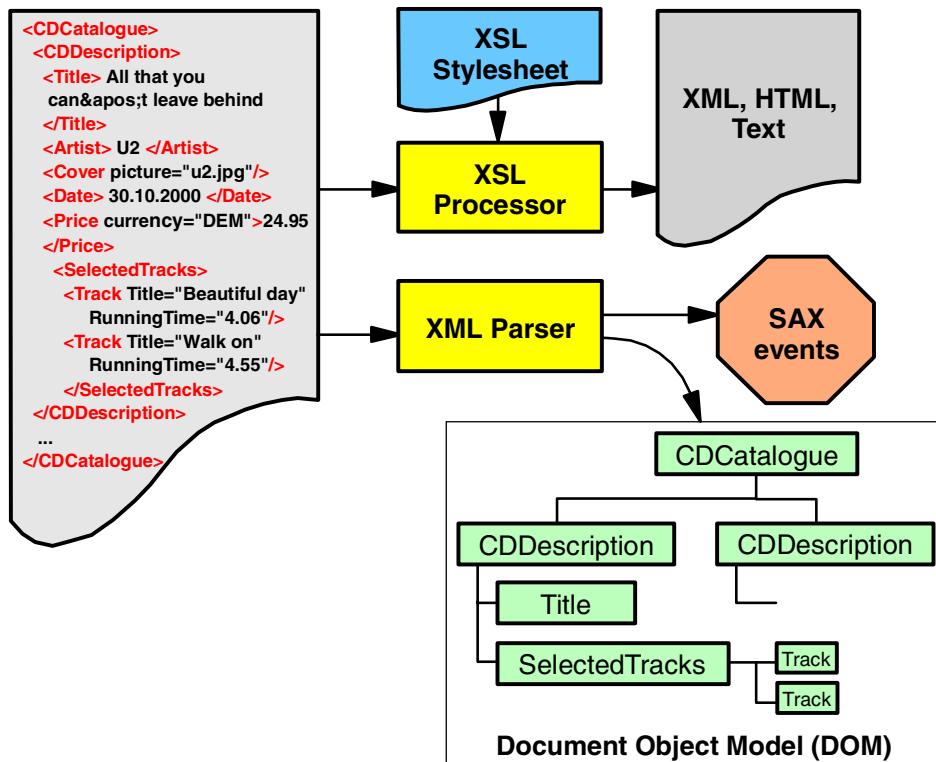


Figure 4-3 XML processing

XML parser

Two parser APIs are available: the document object model (*DOM*) and the simple API for XML (*SAX*).

DOM

DOM is based on an object model: After parsing, the memory contains a tree of DOM objects offering information about both the structure and contents of the XML document. DOM provides a lot of in-memory data manipulation commands operating on the DOM tree.

A drawback of DOM is its huge memory consumption for large XML documents. It can stress the JVM memory management system because it creates many small short-lived objects.

DOM is best suited for in-memory navigation and manipulation on the structure and contents of XML documents.

The most important classes in the DOM API, as defined in the org.w3c.dom package, are org.w3c.dom, Document, Node, Element, and Text. Take a look the following small programming example:

```
String filename = "xmlFile.xml";
DocumentBuilderFactory buildFactory = DocumentBuilderFactory.newInstance();
buildFactory.setNamespaceAware( true );
DocumentBuilder xmlParser = buildFactory.newDocumentBuilder();
Document xmlDoc= xmlParser.parse(filename);
Node n = xmlDoc.getLastChild().getFirstChild(); // navigate the DOM tree
n.setNodeValue("Everybody's happy."); // modify a node
```

SAX

SAX on the other hand is event driven. The parser passes once through the document and generates events that correspond to different features found in the parsed XML document.

By responding to this stream of SAX events in Java code, you can write programs driven by XML-based data. SAX allows to map XML data into an application specific object model in just one pass.

The SAX programming model is often considered to be difficult because it is not possible to navigate on the data. SAX is well suited for direct, one pass mappings to specific problem-domain object models.

JDOM

JDOM provides a means of accessing an XML document within Java through a tree structure, just as DOM. Unlike DOM, however, JDOM was specifically designed for Java, and promises to provide a more intuitive, easier to use API than DOM.

JDOM is an open source project initiated by Brett McLaughlin and Jason Hunter. It has just become a JSR (JSR-102); its implementation currently is at beta level. For more information, refer to <http://www.jdom.org/index.html>.

Parser implementations

There are various parser implementations. For example, Xerces is an Apache open source project based on an early IBM parser implementation that was called *XML4J*.

XML processor

The *eXtensible style sheet language (XSL)* allows true separation of content and presentation style. XSL itself is defined in XML as well. It consists of three parts:

- XPath** *XML path language*, a language for referencing specific parts of an XML document.
- XSL-T** *XSL transformations*, a language for describing how to transform one XML document format into another.
XSL-T is a mix of a rules-based pattern matcher, a template apply mechanism and an imperative programming language.
- XSL-FO** A set of *formatting objects* and formatting properties for precisely specifying layout (not yet in widespread use).

There is an Apache open source implementation of XSL called *Xalan*; it is based on an early XSL implementation from Lotus.

A more detailed discussion of XSL exceeds the scope of this document.

Summary

XML represents information and not just format. It makes data portable.

XML processing and transformation provides capabilities for applying different semantics or style to the data as required by a specific application domain.

XML schemas and DTDs already exists for most business communities. More such domain specific language catalogs are currently being defined by various industry consortium and standards bodies.

XML standards are expanding and gaining rapid support. The use of XML is still growing rapidly. For example, XML is one of the technical foundations for the Web services.

In this section, we have presented a high level overview of the XML and related technologies. For further study in XML, see “Related publications” on page 577.

XML tool support in Application Developer

In the following sections we demonstrate how to use these XML tools:

RDB to XML mapping editor The RDB to XML mapping editor automatically converts SQL query results into an XML file and, even more importantly, creates an XML schema describing the structure of this XML file.

XML editor With the XML editor, you can inspect, modify, and validate XML files. It comprises a *Design* and a *Source* view.

XML schema editor The XML schema editor allows to browse and update XML schema definitions (XSDs).

XML parser Application Developer ships with a version of Apache Xerces.

XML to XML mapping editor This editor gives you a graphical user interface that can be used to construct XSL-based XML to XML conversions.

XSL transformation tools Application Developer includes a version of Apache Xalan, an XSL style sheet processor.

XSL trace editor The XSL trace editor can be used to execute an XSL transformation step by step.

DTD editor The DTD editor allows to create and browse document type definitions (DTDs). DTDs can be converted into XSDs and vice versa.

Other XML utilities are also available, but not used in this book. These include:

XML generation tools There are several tools that allow you to generate various artifacts either from an XSD or a DTD:

- ▶ JavaBeans
- ▶ Table definitions for relational databases
- ▶ Sample XML files

Refer to the help perspective for additional information about these tools.

Solution outline for example

As described in Chapter 1, “Auto parts sample application” on page 3, Mighty Motors is the vehicle manufacturer supplying cars and spare parts to Almaden Autos.

As an international company with many different IT systems, Mighty Motors has decided to leverage the power of XML as a portable and flexible means of representing important business data such as parts and inventory items (Figure 4-4).

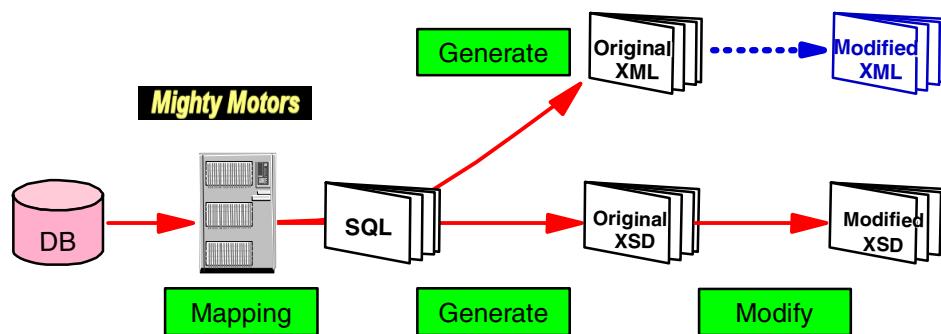


Figure 4-4 XML processing by Mighty Motors

Acting as developers of the inventory management system of Mighty Motors, we use the RDB to XML mapping editor to execute a database query and convert the results into XML format. During this step, the mapping editor generates an XML schema describing a list of InventoryItems.

As a second step, we launch the XML editor in order to browse and validate the generated XML file containing the XML representation of the query results. Using the XML schema editor, we then inspect and modify the XSD file containing the XML schema referenced in the XML file.

We finally show how to process an XML file adhering to the generated XML schema. We do this by using the XML to XML mapping tool to create XSL that represents the mapping, and then apply the generated XSL to the original XML. We map the original XSD to the modified XSD to create the XSL (Figure 4-5).

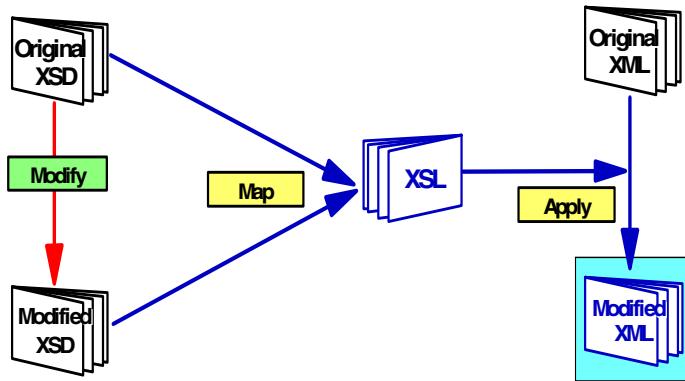


Figure 4-5 XML processing by Mighty Motors extended

Note that at development time we are not really interested in the data. What we want to achieve is to reverse engineer an already existing data model and transform it into an XSD. The XML file merely serves as an intermediate step and as a sample.

In the second half of the book (Chapter 10, “Static Web services” on page 321, in particular), we will use this schema to return inquiry results from the service provider to the service requestor.

Class diagram

A high-level UML class diagram is contained in Figure 4-6. It should be read from the bottom to the top. For example, the XSD and XML file are created by the RDB to XML mapping editor, which executes the SQL statement.

Note: The diagram only shows the first phase, RDB to XML conversion, including XML to HTML conversion. It does not include all the XSDs that are used to create the XSL file for XML to XML conversion.

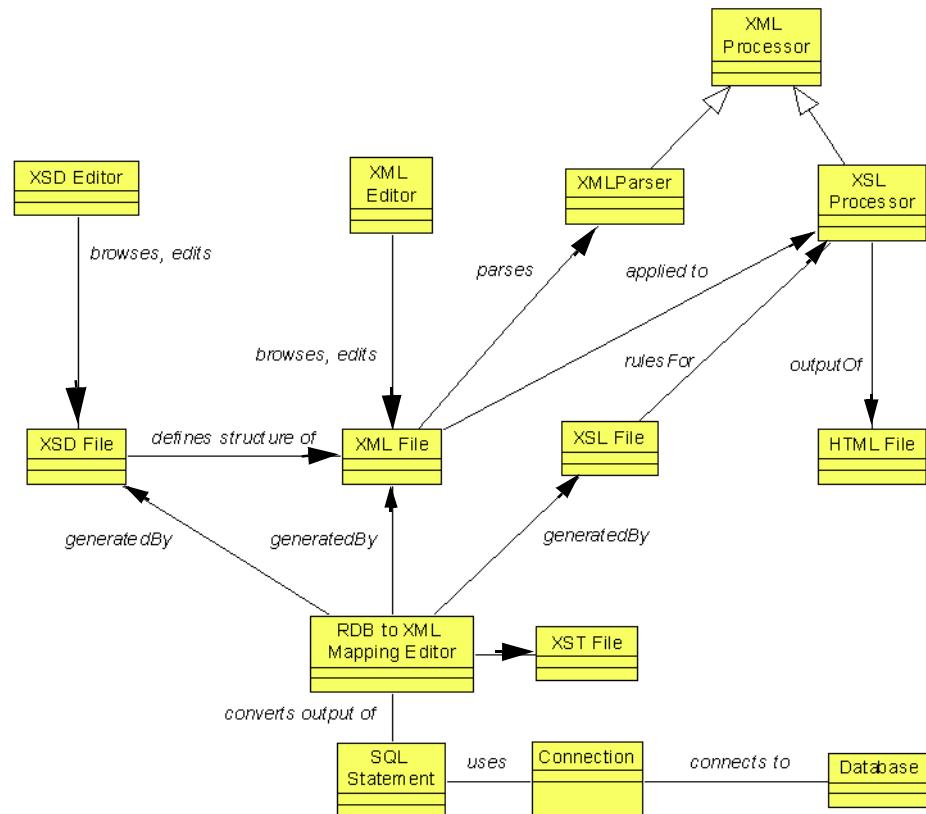


Figure 4-6 Class diagram for ITSOMightyXML

Sequence diagram

Figure 4-7 shows the interaction diagram for the sample scenario of mapping SQL output to XML and to HTML.

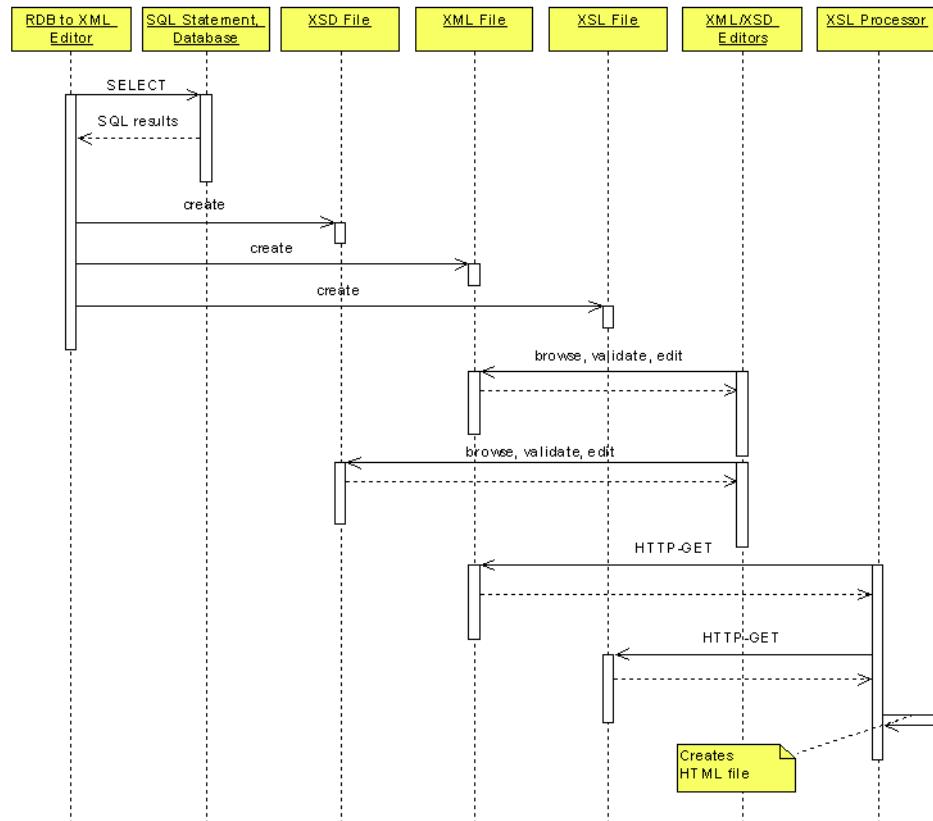


Figure 4-7 Sequence diagram for RDB to XML mapping and conversion

Preparing for development

This example uses the already existing database schema for parts and inventory items from “Web development with Application Developer” on page 55.

Before we can start to work with the XML tools, we have to create a new project, a database connection, and an SQL statement in the data perspective of Application Developer. Then we execute this statement and let the XML from SQL Query wizard convert its results into XML.

In “Web development with Application Developer” on page 55 we have already created a database connection and an SQL statement while working with the Database Web Pages wizard. The steps we have to perform now are a subset of the steps explained there; therefore, we do not go into details here.

First, create a project for the XML work:

- ▶ Create a new Java project ITSOMightyXML and set its source folder to source and build output folder to classes:
 - Select *File -> New Project -> Java Project* and enter ITSOMightyXML.
 - On the *Source* tab of the *Java Setting* window:
 - Select the *Use source folders contained in the project*
 - Click the *Create new folder* button
 - Enter source as the folder name and click *OK*
 - Change the *Build output folder* to /ITSOMightyXML/classes
 - Click *Finish*
- ▶ Create a folder in the ITSOMightyXML project called SQLXML.

Now we create the SQL statement using the *SQL Statement wizard*. This wizard is the same used by the *Database Web Pages wizard* in “*Generating the basic Web application*” on page 63.

In the data perspective:

- ▶ Select *File -> New -> Other -> Data -> SQL Statement* to launch the SQL statement editor.
- ▶ Create a new Select statement called InquireParts in the SQLXML folder of the ITSOMightyXML project.
 - Connect to the database and import the database model into the ITSOMightyXML project's SQLXML folder.

Note: Make sure to *directly* connect to the database from the SQL statement creation dialog; do not use an existing data base model. This would require creating a connection instance and importing the connection elements into the local project. We will explore this when we create EJBs in Chapter 5, “EJB development with Application Developer” on page 133.

- Define the connection MightyConn to the ITSOWSAD database. For details on this definition see Figure 3-8 on page 65.
- Select the two tables ITS0.MMInventory and ITS0.MMParts. Include all columns of MMInventory, except for the partnumber column, and all columns of MMParts. Define a *Join* between the partnumber columns of the tables.
- Click *Next*.

- The statement should look like:

```
SELECT
    ITSO.MMINVENTORY.ITEMNUMBER,
    ITSO.MMINVENTORY.QUANTITY,
    ITSO.MMINVENTORY.COST,
    ITSO.MMINVENTORY.SHELF,
    ITSO.MMINVENTORY.LOCATION,
    ITSO.MMPARTS.PARTNUMBER,
    ITSO.MMPARTS.NAME,
    ITSO.MMPARTS.DESCRIPTION,
    ITSO.MMPARTS.WEIGHT,
    ITSO.MMPARTS.IMAGE_URL
FROM
    ITSO.MMINVENTORY, ITSO.MMPARTS
WHERE
    ITSO.MMINVENTORY.PARTNUMBER = ITSO.MMPARTS.PARTNUMBER
```

Click *Finish* to generated the underlying files. In the Navigator view you should be able to see an SQL statement entry `ITSOWSAD_InquireParts.sqx` in the `SQLXML` folder.

Note that even more files have been generated that end with `xmi`. They contain meta information about the connection, database, and tables we have used. Do not modify any of these files.

Generating an XML file and an XML schema from SQL

Now we are ready to launch the XML from SQL Query wizard:

- Open the XML or the data perspective.
- Open the SQL statement from the Navigator by double-clicking.
- Select the SQL statement `InquireParts` in the Outline view (depending on the current perspective—Data or XML—this view typically appears on the upper left or right corner of the Application Developer window) and *Generate new XML* from the context menu.
- The XML from SQL dialog opens (Figure 4-8).

Note: You can also select *New -> XML -> XML from SQL Query* and navigate to the SQL statement to open this dialog.

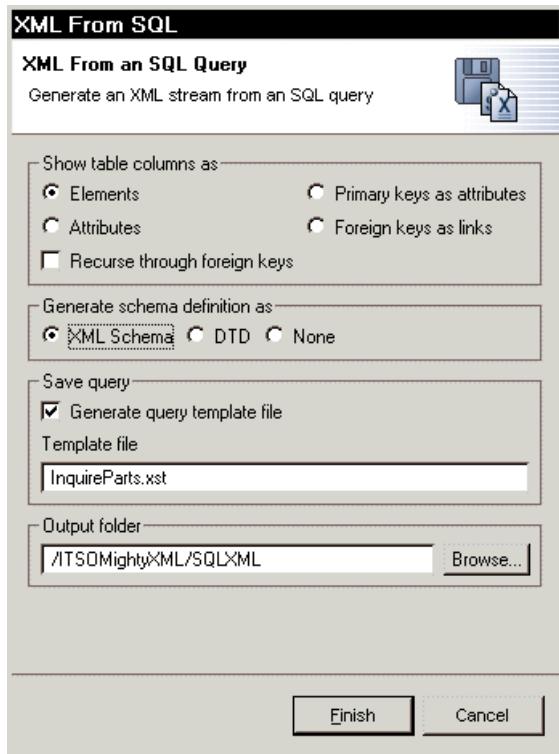


Figure 4-8 XML from SQL Query wizard

This panel allows you to: configure the XML representation of the table columns (which can be useful for complex queries); know how the meta information about the file should be stored (XSD or DTD); and whether a query template should be stored for later use.

- ▶ Make sure that the output folder is /ITSOMightyXML/SQLXML
- ▶ Click *Finish* to generate the files, then close the SQL statement editor.

Five additional files of name InquireParts.xxx are generated, with .xxx being one of the following filename extensions:

- .xml The InquireParts.xml file contains the query results in XML format.
- .xsd The InquireParts.xsd file holds the XML schema used by the query result file.
- .xsl An XSL style sheet that describes a simple XML to HTML conversion for the contents of the .xml file.
- .xst This file contains a template with meta information about the executed database query.

- .html** This file contains an HTML representation of the SQL query results. This representation (a simple table in this case) is the output of the XSL based XML to HTML conversion. (Open the file to see the results.)

Browsing and editing XML and XSD files

In the next sections, we demonstrate the browsing and editing capabilities of the XML editor and the XML schema editor. The output of the XML from SQL Query wizard, the .xml and .xsd files, serve as our starting point:

- ▶ Copy the InquireParts.xml and the InquireParts.xsd files from the /ITSOMightyXML/SQLXML folder to the base /ITSOMightyXML folder. (Select the files and *Copy* from the context, then select the destination folder.)
- ▶ In the /ITSOMightyXML folder, rename the InquireParts.xml file to InquirePartsOrg.xml.

Will will now explore the .xml and .xsd files in the base /ITSOMightyXML folder. The original files will be used later in transforming the XML files.

Editing an XML file

Double-click on the InquirePartsOrg.xml file in the Navigator view to open the XML editor. The XML file contains the results of the database query. Expand the document structure in the Design view (Figure 4-9).

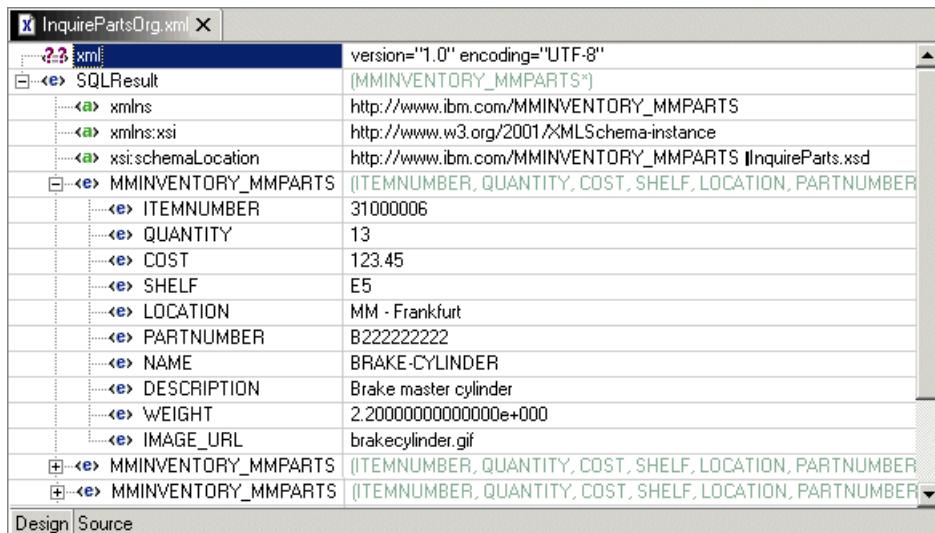


Figure 4-9 XML editor (Design view)

Open the source tab. The Source view of the XML editor shows the plain file representation of the query results (Figure 4-10).

```
<?xml version="1.0" encoding="UTF-8"?>
<SQLResult xmlns="http://www.ibm.com/MMINVENTORY_MMPARTS"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.ibm.com/MMINVENTORY_MMPARTS
    InquireParts.xsd">
    <MMINVENTORY_MMPARTS>
        <ITEMNUMBER>31000006</ITEMNUMBER>
        <QUANTITY>13</QUANTITY>
        <COST>123.45</COST>
        <SHELF>E5</SHELF>
        <LOCATION>MM - Frankfurt</LOCATION>
        <PARTNUMBER>B22222222</PARTNUMBER>
        <NAME>BRAKE-CYLINDER</NAME>
        <DESCRIPTION>Brake master cylinder</DESCRIPTION>
        <WEIGHT>2.2000000000000e+000</WEIGHT>
        <IMAGE_URL>brakecylinder.gif</IMAGE_URL>
    </MMINVENTORY_MMPARTS>
    <MMINVENTORY_MMPARTS>
        .....
    </MMINVENTORY_MMPARTS>
</SQLResult>
```

Figure 4-10 XML representation of SQL query results

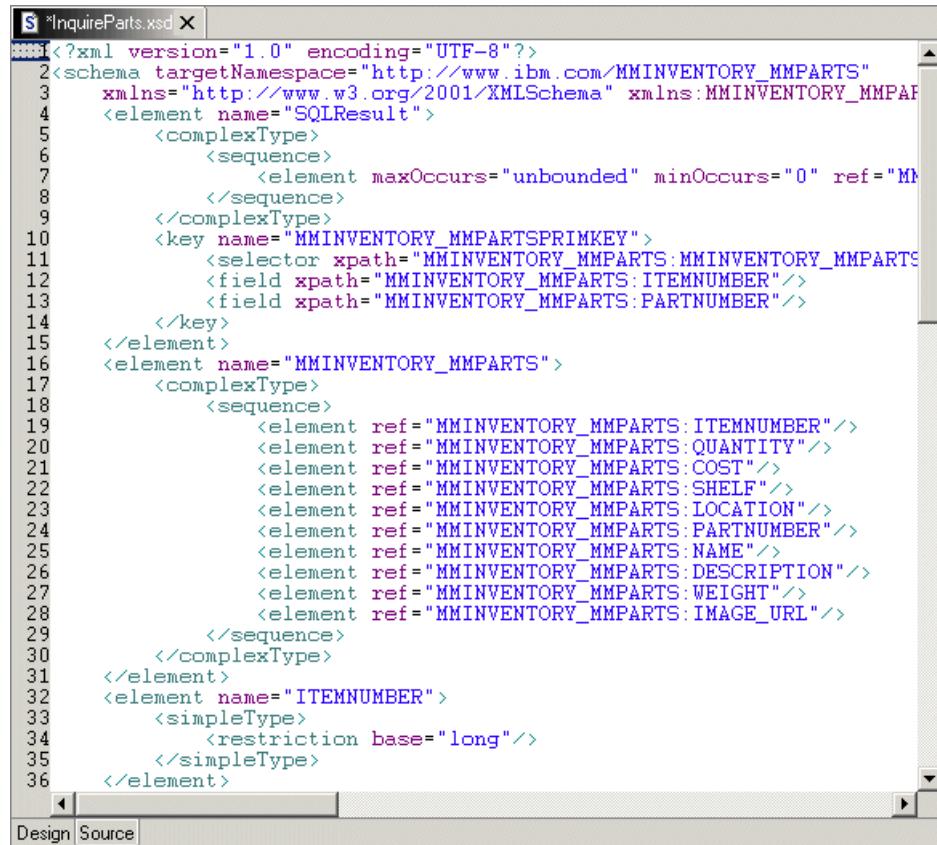
As you can see, the XML file imports the XSD schema. Note that in this case http://www.ibm.com/MMINVENTORY_MMPARTS is the target namespace, and InquireParts.xsd is the location of the XSD file (refer to “XML schema” on page 98 for a discussion on the syntax of the schema location attribute).

Edit an XSD file

Next, we take a look at the schema.

Edit the /ITSOMightyXML/InquireParts.xsd file from the Navigator and select the Source view (Figure 4-11).

As you can see, the query results are presented as a sequence of record-like complex type elements. These elements contain simple types representing the attributes.



The screenshot shows an XSD editor window with the title bar "S *InquireParts.xsd". The main area contains the XML schema code for "InquireParts.xsd". The code defines an XML schema with elements like "SQLResult", "MMINVENTORY_MMPARTS", and "ITEMNUMBER". It includes complex types for sequences of items and specific item details like quantity, cost, and description. A key element "MMINVENTORY_MMPARTSPRIMKEY" is defined with two fields: "ITEMNUMBER" and "PARTNUMBER". The "ITEMNUMBER" element has a simple type restriction of "long". At the bottom of the editor, there are tabs for "Design" and "Source", with "Source" currently selected.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema targetNamespace="http://www.ibm.com/MMINVENTORY_MMPARTS"
  xmlns="http://www.w3.org/2001/XMLSchema" xmlns:MMINVENTORY_MMPARTS="http://www.ibm.com/MMINVENTORY_MMPARTS">
  <element name="SQLResult">
    <complexType>
      <sequence>
        <element maxOccurs="unbounded" minOccurs="0" ref="MMINVENTORY_MMPARTS" />
      </sequence>
    </complexType>
  </element>
  <key name="MMINVENTORY_MMPARTSPRIMKEY">
    <selector xpath="MMINVENTORY_MMPARTS:MMINVENTORY_MMPARTS" />
    <field xpath="MMINVENTORY_MMPARTS:ITEMNUMBER" />
    <field xpath="MMINVENTORY_MMPARTS:PARTNUMBER" />
  </key>
  <element name="MMINVENTORY_MMPARTS">
    <complexType>
      <sequence>
        <element ref="MMINVENTORY_MMPARTS:ITEMNUMBER" />
        <element ref="MMINVENTORY_MMPARTS:QUANTITY" />
        <element ref="MMINVENTORY_MMPARTS:COST" />
        <element ref="MMINVENTORY_MMPARTS:SHELF" />
        <element ref="MMINVENTORY_MMPARTS:LOCATION" />
        <element ref="MMINVENTORY_MMPARTS:PARTNUMBER" />
        <element ref="MMINVENTORY_MMPARTS:NAME" />
        <element ref="MMINVENTORY_MMPARTS:DESCRIPTION" />
        <element ref="MMINVENTORY_MMPARTS:WEIGHT" />
        <element ref="MMINVENTORY_MMPARTS:IMAGE_URL" />
      </sequence>
    </complexType>
  </element>
  <element name="ITEMNUMBER">
    <simpleType>
      <restriction base="long" />
    </simpleType>
  </element>
</schema>
```

Figure 4-11 XSD editor (Source view original)

Validate the XML file

Switch to the editor of the InquireParts0rg.xml file and click the *Validation* icon  in the toolbar to verify that the XML document is valid (conforming to the meta information in the .xsd file, that is).

Editing XML schema and data files

The XML namespace and the first XML tag have names that are misleading if the XSD is a different context, such as to describing the parameters of Web services (see Chapter 10, “Static Web services” on page 321).

Using the XSD editor, we therefore update the schema and enter more appropriate names. Open the XML schema editor for the .xsd file in the Navigator. In the Outline view and expand the root InquireParts element (Figure 4-12).

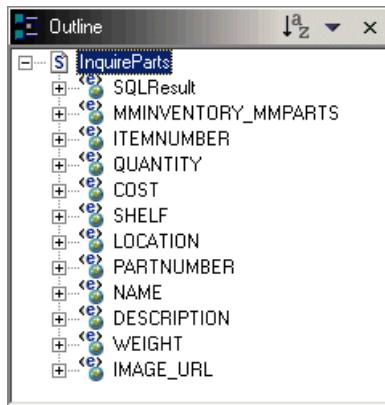


Figure 4-12 XSD editor (Outline view)

Switch to the Design view:

- ▶ Change the *Namespace Prefix* and the *Target Namespace* (Figure 4-13).
from: MMINVENTORY_MMPARTS
to : InquireResults
from: http://www.ibm.com/MMINVENTORY_MMPARTS
to : http://www.redbooks.ibm.com/ITSOVSAD/schemas/InquireResults
- ▶ Click *Apply* after updating the two fields.

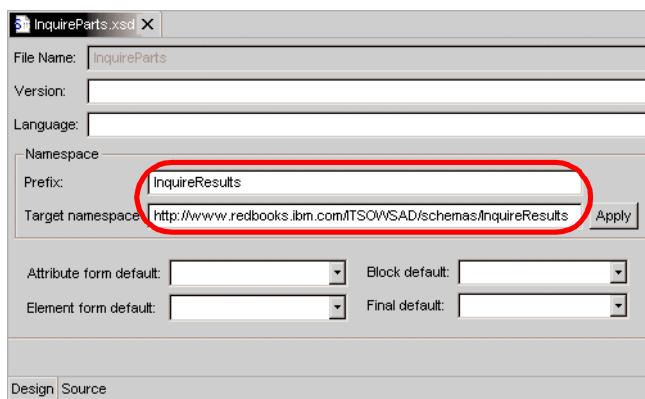


Figure 4-13 XSD editor (Design view)

Back in the Outline view:

- ▶ Select the SQLResult element.
Note that the content of the Design view changes; it now displays the properties of the selected element.
- ▶ Change the element name from SQLResult to InquirePartsResult (Figure 4-14).

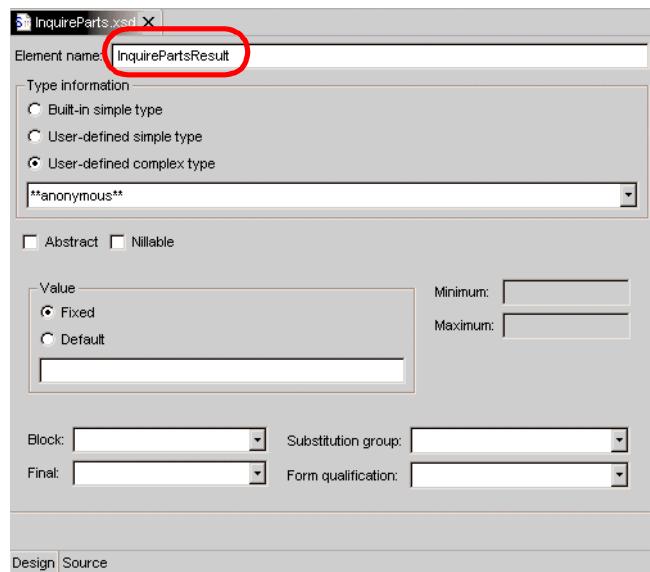


Figure 4-14 XSD editor (Design view of an element)

Change the other element names as well:

- ▶ Select each element in the Outline view and switch to the Design view.
- ▶ Change the name of the selected elements to use mixed case notation. Also change the Type Information from *User-defined simple type* to *Built-in simple type*. Use the names and data types shown in Table 4-1.

Table 4-1 Schema modifications

Old name	New name	Data type
MMINVENTORY_MMPARTS	Part	(remains the same)
ITEMNUMBER	ItemNumber	long
PARTNUMBER	PartNumber	string
QUANTITY	Quantity	integer
COST	Cost	decimal
SHELF	Shelf	string
LOCATION	Location	string
NAME	Name	string
DESCRIPTION	Description	string
WEIGHT	Weight	double
IMAGE_URL	Image_URL	string

After having changed the element names, it is necessary to change the key definition in the XSD:

- ▶ Expand InquirePartsResult
 - ▶ Select the key MMPARTS_MMINVENTORYPRIMKEY
 - ▶ Remove the two *fields*
 - MMINVENTORY_MMPARTS:ITEMNUMBER
 - MMINVENTORY_MMPARTS:PARTNUMBER
 - ▶ Add the correct two fields
 - InquireResults:ItemNumber
 - InquireResults:PartNumber
 - ▶ Change the *Selector* from
 - MMINVENTORY_MMPARTS:MMINVENTORY_MMPARTS to
 - InquireResults:Part
 - ▶ Save the XSD
- Make sure you click on a field other than the Selector. If the *Selector* field still has the focus the change will not be saved.

Switch to the Source view and compare your changes with the file shown in Figure 4-15. Note you can edit the XSD in the source view and you may make changes there.

```
1<?xml version="1.0" encoding="UTF-8"?>
2<schema targetNamespace="http://www.redbooks.ibm.com/ITSOwsAD/schema"
3    xmlns="http://www.w3.org/2001/XMLSchema"
4    xmlns:InquireResults="http://www.redbooks.ibm.com/ITSOwsAD/schema"
5    element name="InquirePartsResult">
6        <complexType>
7            <sequence>
8                <element maxOccurs="unbounded" minOccurs="0" ref="InquireResults:Part"/>
9            </sequence>
10       </complexType>
11       <key name="MMINVENTORY_MMPARTSPRIMKEY">
12           <selector xpath="InquireResults:Part"/>
13           <field xpath="InquireResults:ItemNumber"/></field>
14           <field xpath="InquireResults:PartNumber"/></field>
15       </key>
16   </element>
17   <element name="Part">
18       <complexType>
19           <sequence>
20               <element ref="InquireResults:ItemNumber"/>
21               <element ref="InquireResults:Quantity"/>
22               <element ref="InquireResults:Cost"/>
23               <element ref="InquireResults:Shelf"/>
24               <element ref="InquireResults:Location"/>
25               <element ref="InquireResults:PartNumber"/>
26               <element ref="InquireResults:Name"/>
27               <element ref="InquireResults:Description"/>
28               <element ref="InquireResults:Weight"/>
29               <element ref="InquireResults:Image_Url"/>
30           </sequence>
31       </complexType>
32   </element>
33   <element name="ItemNumber" type="long">
34   </element>
35   <element name="Quantity" type="integer">
```

Figure 4-15 XSD editor (Source view modified)

Validation

Try to verify the InquireParts0rg.xml file again; it is no longer valid because the schema has changed, and plenty of errors are reported in the Tasks view:

- ▶ To validate the original XML file against the original XSD, change the schema pointer in the XML file from InquireParts.xsd to SQLXML/InquireParts.xsd:

```
<SQLResult xmlns="http://www.ibm.com/MMINVENTORY_MMPARTS"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.ibm.com/MMINVENTORY_MMPARTS
    SQLXML/InquireParts.xsd">
```

- ▶ As soon as you save the change, the validation runs and the errors disappear.

Modifying the XML to match the new schema

The errors in the Tasks view are a result of the fact that we modified the schema referenced in the XML generated from RDB. As a result, the XML no longer matches the schema definition. To resolve the errors, you have to modify the XML file so that schema and data file match again.

There are two options for modifying the XML file:

- ▶ Editing the XML directly
- ▶ Creating an XML to XML mapping

Editing the XML directly

You can modify the XML file by hand using the Design or the Source view:

- ▶ Update the two namespace references to use the new namespace <http://www.redbooks.ibm.com/ITSOWSAD/schemas/InquireResults>. The two namespace references are located at the beginning of the file.
- ▶ Change the tag name for the root element from SQLResult to InquirePartsResult.
- ▶ Update all other tag names as well so that they use the new names you have previously defined in the XSD editor (from MMINVENTORY_MMPARTS to Parts, and so forth).

Creating an XML to XML mapping

Often manually modifying the XML is not feasible. In this case you can use the *XML to XML* mapping tool to create an XSL to apply to the original XML. In this section we use the XML to XML mapping to create an XSL style sheet that can be applied to the original XML to transform it into new XML.

To create an XML to XML mapping we:

- ▶ Create an XML to XML mapping from the XSDs
- ▶ Generate an XSL script from the XML to XML mapping
- ▶ Apply the XSL script to the original XML

Create the XML to XML mapping

We now create a mapping between the source, InquireParts.xsd (in /TSOMightyXML/SQLXML), and the target, InquireParts.xsd (in /ITSOMightyXML).

In the XML perspective, select the *XML to XML* mapping icon  to invoke the *XML to XML* mapping tool (Figure 4-16).



Figure 4-16 XML to XML mapping tool

- ▶ Select the folder ITSMightyXML.
- ▶ Change the filename to InquirePartsMap.xmx.
- ▶ Click *Next*.

Now we have to select the source and target XSDs (Figure 4-17).

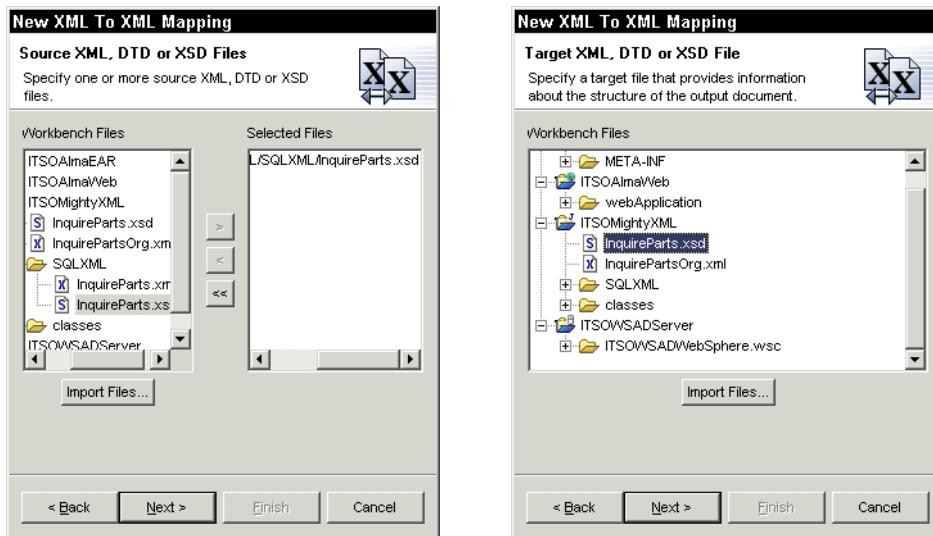


Figure 4-17 XML to XML mapping: source and target XSDs

- ▶ For the source select InquireParts.xsd in the /ITSOMightyXML/SQLXML folder and click *Next*.
- ▶ For the target select InquireParts.xsd in the /ITSOMightyXML folder and click *Next*.
- ▶ In the *Root Element* panel (Figure 4-18) verify that the *Target Root Element* (from the modified DTD) is InquirePartsResult and the *Source Root Element* (from the original DTD) is SQLResult.

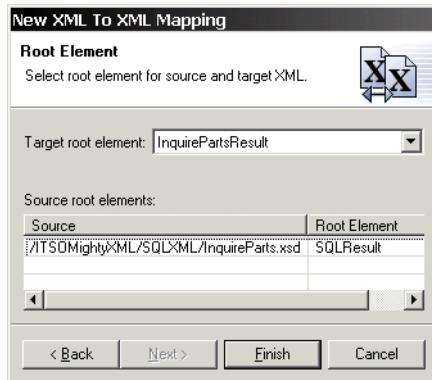


Figure 4-18 XML to XML mapping: root element mapping

- ▶ Click *Finish*.

The mapping produces the new file we named InquirePartsMap.xmx and it opens in the mapping editor (Figure 4-19).

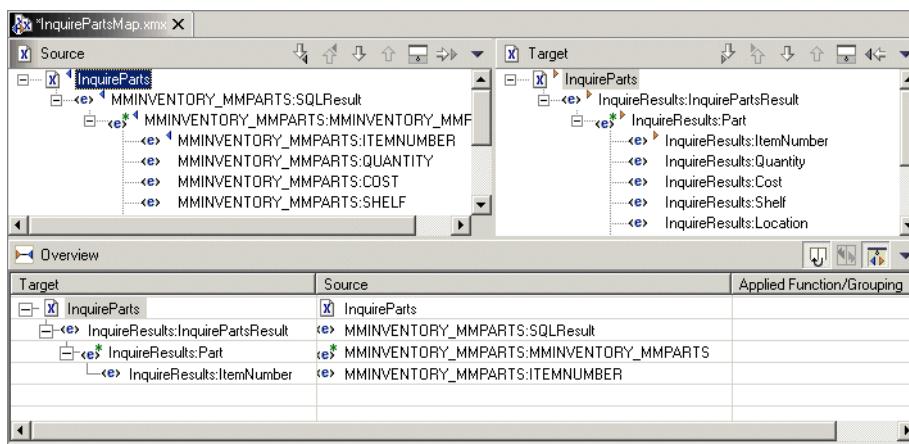


Figure 4-19 XML to XML mapping editor

- ▶ Map the elements from the source to the target by dragging the element from the source panel, and dropping it on the target element in the target panel. Create the following mappings:

Source Element	Target Element
SQLResult	InquirePartsResult
MMINVENTORY_MMPARTS	Part
ITEMNUMBER	ItemNumber
QUANTITY	Quantity
COST	Cost
SHELF	Shelf
LOCATION	Location
PARTNUMBER	PartNumber
NAME	Name
DESCRIPTION	Description
WEIGHT	Weight
IMAGE_URL	Image_URL

As the mappings are created the source and target icons change from  to  and . The mapping is also displayed in the *Outline* panel.

- ▶ Save the mapping.
- ▶ Generate the XSLT script for the mapping by selecting the *Generate XSLT script for Mapping* icon  (Figure 4-20).



Figure 4-20 XSLT generation from an XML to XML mapping

- ▶ Select the /ITSO Mighty XML folder, set the *filename* to InquirePartsMap.xsl, and click *Finish*.

The XSL opens in the XSL editor (Figure 4-21).

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:transform xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="1.0"
    xmlns:xalan="http://xml.apache.org/xslt"
    xmlns:InquireResults="http://www.redbooks.ibm.com/ITSOVSAD/schemas/InquireResults"
    xmlns:MMINVENTORY_MMPARTS="http://www.ibm.com/MMINVENTORY_MMPARTS"
    exclude-result-prefixes="MMINVENTORY_MMPARTS">
<xsl:output method="xml" encoding="UTF-8" indent="yes" xalan:indent-amount="2"/>
<xsl:strip-space elements="*"/>
<!--=====
<!-- This file contains an XSLT transformation stylesheet which      -->
<!-- constructs a result tree from a number of XML sources by filtering   -->
<!-- reordering and adding arbitrary structure. This file is           -->
<!-- automatically generated by the XML Mapper tool from IBM WebSphere   -->
<!-- Studio Workbench.                                                 -->
<!--=====
<!--               The Root Element                                     -->
<!-- The "Root Element" section specifies which template will be     -->
<!-- invoked first thus determining the root element of the result tree. -->
<!--=====
<xsl:template match="/">
    <xsl:apply-templates select="/MMINVENTORY_MMPARTS:SQLResult"/>
</xsl:template>
<!--=====
<!--               Remaining Templates                                -->
<!-- The remaining section defines the template rules. The last template -->
<!-- rule is a generic identity transformation used for moving complete -->
<!-- tree fragments from an input source to the result tree.          -->
<!--=====
<!-- Composed element template -->
<xsl:template match="MMINVENTORY_MMPARTS:MMINVENTORY_MMPARTS">
    <InquireResults:Part>
        <InquireResults:ItemNumber>
            <xsl:value-of select="MMINVENTORY_MMPARTS:ITEMNUMBER/text()"/>
        </InquireResults:ItemNumber>
    ...
    <!-- Composed element template -->
    <xsl:template match="MMINVENTORY_MMPARTS:SQLResult">
        <InquireResults:InquirePartsResult>
            <xsl:apply-templates select="MMINVENTORY_MMPARTS:MMINVENTORY_MMPARTS"/>
        </InquireResults:InquirePartsResult>
    </xsl:template>
    <!-- Identity transformation template -->
    <xsl:template match="*|@*|comment()|processing-instruction()|text()">
        <xsl:copy>
            <xsl:apply-templates select="*|@*|comment()|processing-instruction()|text()"/>
        </xsl:copy>
    </xsl:template>
</xsl:transform>
```

Figure 4-21 XSL generated from XML to XML mapping

XML processing

As introduced in “An XML primer” on page 92, there are two ways of processing XML documents using XSL style sheets and parsers.

Transforming XML to XML with a style sheet

The final step involves applying the XSL to the original XML file.

- ▶ In the Navigator view, select the original XML file, `InquirePartOrg.xml` and the XSL we just created, `InquirePartMap.xsl` (both in `/ITS0MightyXML`).
- ▶ Apply the XSL to the original XML by selecting *Apply XSL -> As XML* from the context menu.
- ▶ The XSL trace editor opens and displays the original XML file, the XSL style sheet, and the result XML file (Figure 4-22).

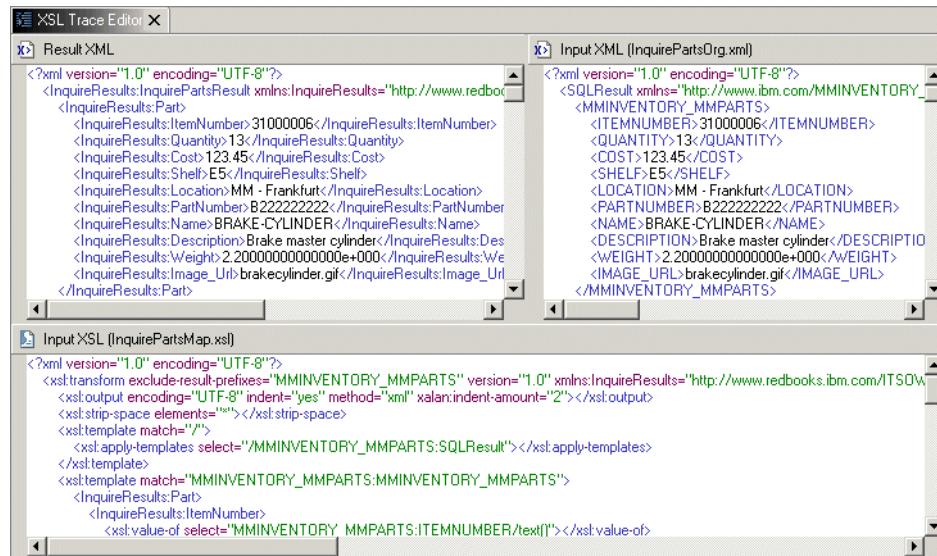


Figure 4-22 XSL trace editor: XML to XML

XSL trace editor

In the XSL trace editor you can trace through the transformation using the trace icons , , and .

Save the file (*File -> Save XSL Trace Editor As*) as `InquireParts.xml` in the `/IST0MightyXML` folder.

Modified XML

In either case, editing by hand or using the XML to XML mapping, the resulting XML file should look as shown in Figure 4-23.

```
<?xml version="1.0" encoding="UTF-8"?>
<InquireResults:InquirePartsResult
  xmlns:InquireResults="http://www.redbooks.ibm.com/ITSOVSAD/schemas/InquireResults">
  <InquireResults:Part>
    <InquireResults:ItemNumber>31000006</InquireResults:ItemNumber>
    <InquireResults:Quantity>13</InquireResults:Quantity>
    <InquireResults:Cost>123.45</InquireResults:Cost>
    <InquireResults:Shelf>E5</InquireResults:Shelf>
    <InquireResults:Location>MM - Frankfurt</InquireResults:Location>
    <InquireResults:PartNumber>B22222222</InquireResults:PartNumber>
    <InquireResults:Name>BRAKE-CYLINDER</InquireResults:Name>
    <InquireResults:Description>Brake master cylinder
      </InquireResults:Description>
    <InquireResults:Weight>2.20000000000e+000</InquireResults:Weight>
    <InquireResults:Image_Url>brakecylinder.gif</InquireResults:Image_Url>
  </InquireResults:Part>
  <InquireResults:Part>
  .....
  </InquireResults:Part>
</InquireResults:InquirePartsResult>
```

Figure 4-23 Sample XML file matching the modified schema

The XSL transformation appears to remove the namespace definitions for the XSI and schema location, therefore we have to add those back in order to make the newly generated XML validate properly.

Change the root element to look as follows by adding the underlined portion:

```
<InquireResults:InquirePartsResult
  xmlns:InquireResults="http://www.redbooks.ibm.com/ITSOVSAD/schemas/InquireResults"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-Instance"
    xsi:schemaLocation="http://www.redbooks.ibm.com/ITSOVSAD/schemas/InquireResults
    InquireParts.xsd" >
```

If you validate the InquireParts.xml file after you have made these changes, the validation returns *Document is valid* because the data file now conforms to the updated schema.

Transforming XML to HTML with a style sheet

When we created the XML from the SQL statement in “Generating an XML file and an XML schema from SQL” on page 111, XSL was also created to transform the XML into HTML.

Edit the generated XSL by opening the InquireParts.xsl file in the /ITSOMightyXML/XML folder. Notice how this XSL is different from the XSL generated in “Modified XML” on page 126. This XSL transforms the original XML file into an HTML file.

- ▶ In the Navigator, select both the InquireParts.xsl and the InquireParts.xml files in the /ITSOMighty/XML folder.
- ▶ Choose *Apply XSL -> As HTML* from the context menu.

The XSL trace editor opens (Figure 4-24).

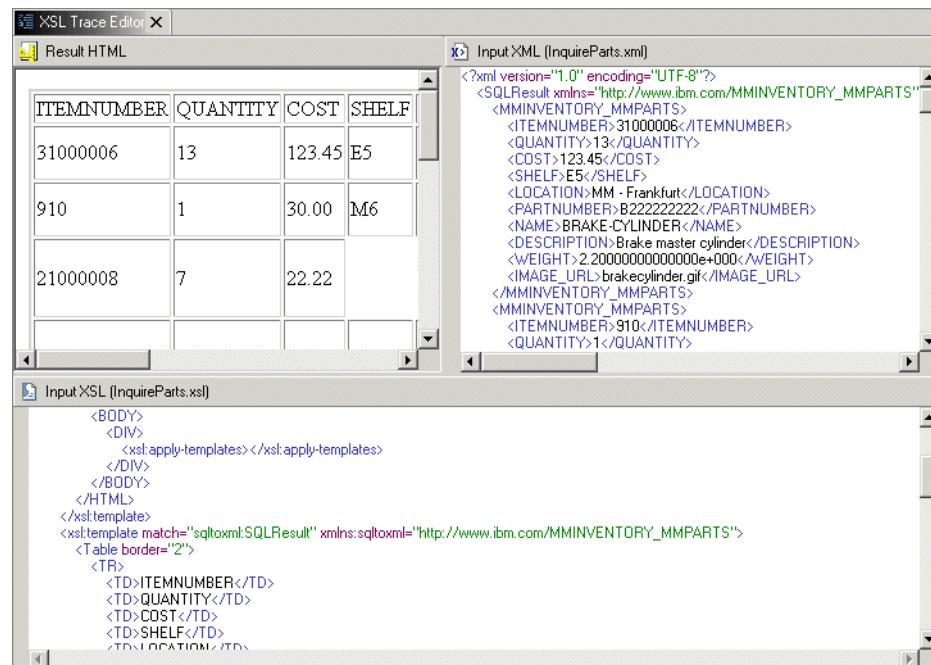


Figure 4-24 XSL trace editor: XML to HTML

In “Static Web services” on page 321 we will show how to programmatically invoke an XSL transformation.

Parsing an XML file

Next, we parse the XML file that contains the inventory item elements using the DOM API. We use the Xerces parser that comes with Application Developer.

We have to add the Xerces JAR file to the build path of the ITSOMightyXML project:

- ▶ Select the project and *Properties* (context), then *Java Build Path -> Libraries* -> *Add Variable* -> *Browse* and select the WAS_XERCES variable. Click *OK* to close the dialog and the *Properties*.
- ▶ Create a package called `itso.wsad.mighty.xml` under the source folder and import the file `XMLParserTest.java` from the `sampcode\wsadxml` directory (see “Using the Web material” on page 570).

Figure 4-25 shows how to invoke the parser from a Java program.

```
package itso.wsad.mighty.xml;
import org.w3c.dom.*;
import javax.xml.parsers.*;

public class XMLParserTest {

    public static void main(String[] args) {
        Element returnResult = null;

        try {
            String filename = "InquireParts.xml";
            DocumentBuilderFactory bf = DocumentBuilderFactory.newInstance();
            bf.setNamespaceAware(true);
            DocumentBuilder xmlParser = bf.newDocumentBuilder();
            Document xmlDoc = xmlParser.parse(filename);
            String result = domWriter(xmlDoc.getDocumentElement(),
                new StringBuffer());
            System.out.println("Content of XML file is: " + result);
        }
        catch(Exception e) {
            System.err.println("XML exception has occurred:");
            e.printStackTrace(System.err);
        }
    }
    // domWriter method definition skipped
}
```

Figure 4-25 Parsing an XML document

The parser instance is obtained from the DocumentBuilderFactory; the name of the XML file to be parsed is passed as a parameter to the parse method. Once the DOM tree is created, it is passed to a helper method called domWriter for further processing.

The domWriter method implementation (Figure 4-26) demonstrates how to recursively traverse the DOM tree generated by the XML parser, access the nodes of the tree, and perform different operations depending on the node type.

```
public static java.lang.String domWriter(Node node, StringBuffer buffer) {
    if ( node == null )
        return "";

    int type = node.getNodeType();
    switch( type ) {
        case Node.DOCUMENT_NODE: {
            buffer.append("<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n");
            domWriter(((org.w3c.dom.Document)node).getDocumentElement(),
                      buffer);
            break;
        }
        case Node.ELEMENT_NODE: {
            buffer.append("<" + node.getnodeName());
            NamedNodeMap attrs = node.getAttributes();
            int length = ( attrs != null ) ? attrs.getLength() : 0;
            for(int i = 0; i < length; i++) {
                Attr attr = (org.w3c.dom.Attr) attrs.item(i);
                buffer.append(" " + attr.getnodeName() + "=\"");
                buffer.append(attr.getNodeValue() + "\"");
            }
            buffer.append(">");
            NodeList children = node.getChildNodes();
            if( children != null) {
                int len = children.getLength();
                for( int i = 0; i < len; i++) {
                    domWriter(children.item(i),buffer);
                }
            }
            buffer.append("</" + node.getnodeName() + ">");
            break;
        }
        case Node.CDATA_SECTION_NODE:
        case Node.TEXT_NODE:{ 
            buffer.append(node.getNodeValue());
            break;
        }
    }
    return buffer.toString();
}
```

Figure 4-26 Traversing an XML DOM tree

Notes:

- ▶ The InquireParts.xml file that we use in this example is the one we created in the previous sections. You have to copy it from the project directory in the workspace, to the Application Developer installation directory.

Restriction: In the Application Developer, any files to be opened by a program have to be put into the directory in which Application Developer was installed.

- ▶ In earlier version of the Java XML APIs, parser instances were created directly. The constructors that allow this are now deprecated and should not be used anymore; instead the DocumentBuilderFactory is now used.
- ▶ The domWriter method simply prints the XML elements and their values to standard output. For example, the code for Element nodes starts the recursion by invoking domWriter for all child nodes; the value of Text nodes is printed to the result buffer.

Running the XML parser application

To run the XMLParserTest application switch to the Java perspective and:

- ▶ Select the XMLParserTest class.
- ▶ Run the class by selecting the  icon.
- ▶ Select the *Java Application launcher* if prompted.
- ▶ The debug perspective opens and the output appears in the console:

```
Content of XML file is: <InquireResults:InquirePartsResult  
    xmlns:InquireResults="http://www.redbooks.ibm.com/ITSOVSAD/schemas/InquireResults"  
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-Instance"  
    xsi:schemaLocation="http://www.redbooks.ibm.com/ITSOVSAD/schemas/InquireResults  
                        InquireParts.xsd">  
    <InquireResults:Part>  
        <InquireResults:ItemNumber>31000006</InquireResults:ItemNumber>  
        <InquireResults:Quantity>13</InquireResults:Quantity>  
        <InquireResults:Cost>123.45</InquireResults:Cost>  
        <InquireResults:Shelf>E5</InquireResults:Shelf>  
        <InquireResults:Location>MM - Frankfurt</InquireResults:Location>  
        <InquireResults:PartNumber>B22222222</InquireResults:PartNumber>  
        <InquireResults:Name>BRAKE-CYLINDER</InquireResults:Name>  
        <InquireResults:Description>Brake master cylinder  
            </InquireResults:Description>  
        <InquireResults:Weight>2.200000000000e+000</InquireResults:Weight>  
        <InquireResults:Image_Url>brakecylinder.gif</InquireResults:Image_Url>  
    </InquireResults:Part>  
    ....  
</InquireResults:InquirePartsResult>
```

Creating an XML file from a Java program

There are two alternatives how XML documents can be created:

- ▶ The DOM API allows you to create the document tree manually in memory.
- ▶ The document can be written to a file or string buffer.

XML DOM tree creation

See Chapter 10, “Static Web services” on page 321 for an example.

XML file

Creating an XML file from with a Java program is no different than writing any other text file, hence, you can use any of the Java I/O and/or string APIs. You can also directly construct the DOM tree using the constructors, and create methods of the Java XML API.

Here are some hints that can help you to simplify XML output development activities:

- ▶ Manually create and validate a sample file before starting to code.
- ▶ Match the (potentially recursive) tree structure of the XML document by organizing your code into multiple methods:
 - You might consider implementing a public `toXML` method returning an `org.w3c.dom.Element` for all your Java classes with data members that need to be serialized using XML.
 - The conversion of Java artifacts from and to XML is often implemented as a general purpose utility.
- ▶ Validate the output of your program with the XML tools such as XML editor, XML schema editor, and DTD editor.

Outlook

In Chapter 5, “EJB development with Application Developer” on page 133, we develop a session bean that returns part inventory data in JavaBeans, and in Chapter 10, “Static Web services” on page 321 we convert those results to an XML DOM tree using the DOM API and the XSD schema we defined in this chapter.

Quiz: To test your knowledge of the XMI and XML tools, try to answer the following five questions. The answers are provided in Appendix C, “Quiz answers” on page 559.

1. Name at least four different types of artifacts that can legally appear in an XML document.
2. Outline an XML schema for the XML file listed in “XML file” on page 93.
3. What is the difference between DTD validation and XSD validation?
4. Name at least four WSAS XML tools. Is there an XML project? Is there an XML perspective?
5. In which namespace resides the schemaLocation attribute, and which local name is typically defined for it (hint: take a look at Figure 4-10 on page 114)?
(*) In Figure 4-23 on page 126, how would the schema location attribute look like if the XSD file would be moved from the local file system to a Web server, assuming that the new location of the file is
`http://localhost:8080/ITSOMighty/allInventoryItems.xsd?`



EJB development with Application Developer

This chapter provides the reader with an opportunity to learn about the EJB development environment by building the second stage of the auto parts sample application—the Mighty Motors parts inventory system.

The concepts that we will be covering include:

- ▶ Creating EJB projects
- ▶ Building container-managed persistence (CMP) entity EJBs
- ▶ EJB associations
- ▶ Creating session bean facades with XML interfaces
- ▶ EJB clients
- ▶ Deployment to the WebSphere test environment

Solution outline for auto parts sample Stage 2a

An overview of the sample application can be found in Chapter 1, “Auto parts sample application” on page 3. This section will discuss the design in more detail before we progress to building the application.

Being a successful international vehicle manufacturer, Mighty Motors has much more complex requirements for its parts inventory system than the Almaden Autos dealership. In order to facilitate component reuse and distribution together with more powerful transactional capabilities, the manufacturer has decided to develop their application using Enterprise JavaBean (EJB) technologies.

The first phase in the development of this J2EE application is to create the data access layer and some simple business logic to enable queries. As the client to this application will eventually be a Web service, we will not focus on the client and presentation layer of the application at this time. An outline of the application is shown in Figure 5-1.

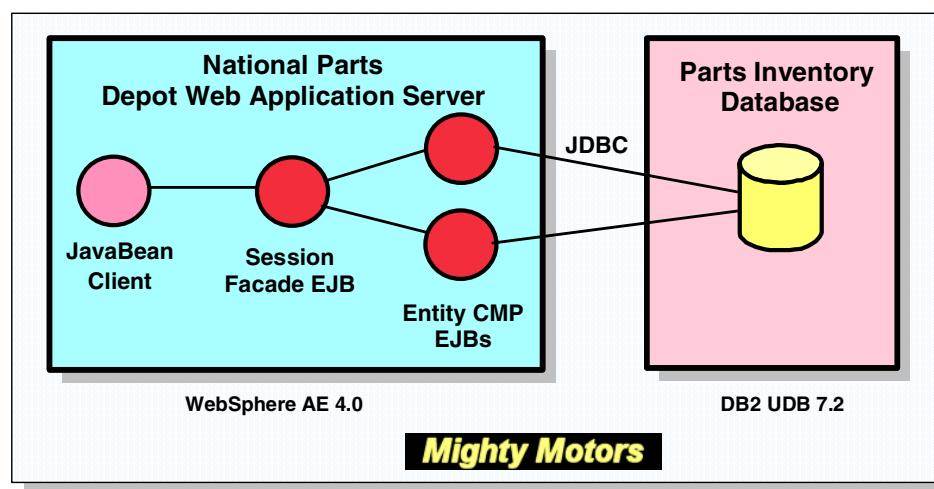


Figure 5-1 Basic design for Mighty Motors application

In Chapter 10, “Static Web services” on page 321, we will develop the EJB client (JavaBean) and create a Web service wrapper using Application Developer.

The intention of this chapter is not to provide an overview of the EJB programming model, but to give the reader an opportunity to understand how such components can be developed using Application Developer.

Class diagrams

Our application comprises of two container-managed entity EJBs (CMPS), representing the two tables in the underlying DB2 database, a session facade EJB, and an JavaBean wrapper class. As each EJB is represented by a number of classes, we have broken down the single class diagram into three figures. The class diagram for the inventory item entity EJB is shown in Figure 5-2.

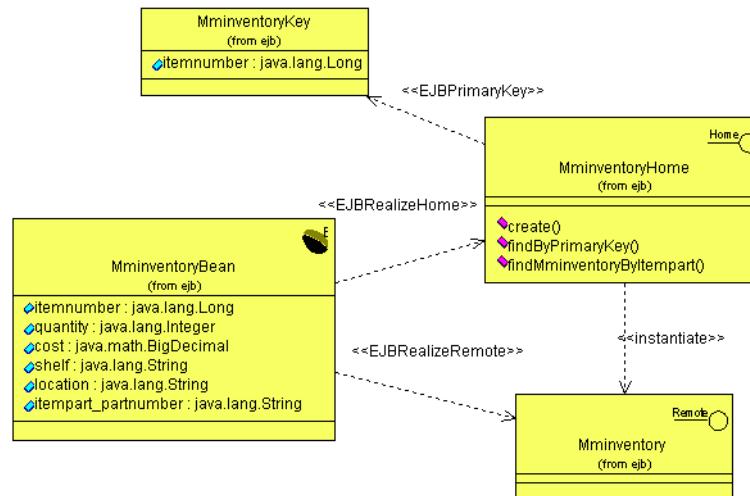


Figure 5-2 Class diagram fragment for the inventory item entity EJB

The class diagram for the part entity EJB is shown in Figure 5-3.

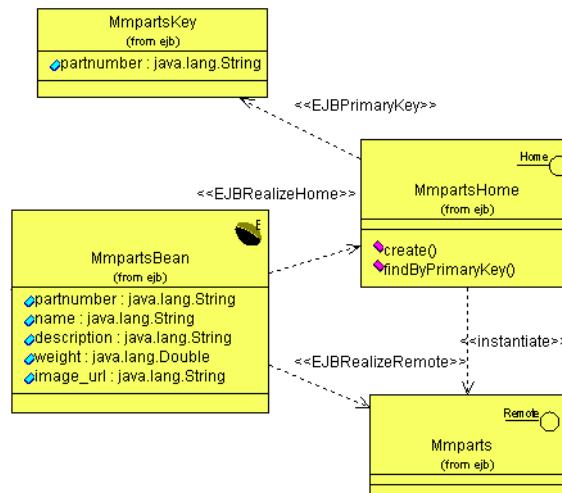


Figure 5-3 Class diagram fragment for the part entity EJB

Finally, the class diagram for the session facade and the JavaBean (Figure 5-4).

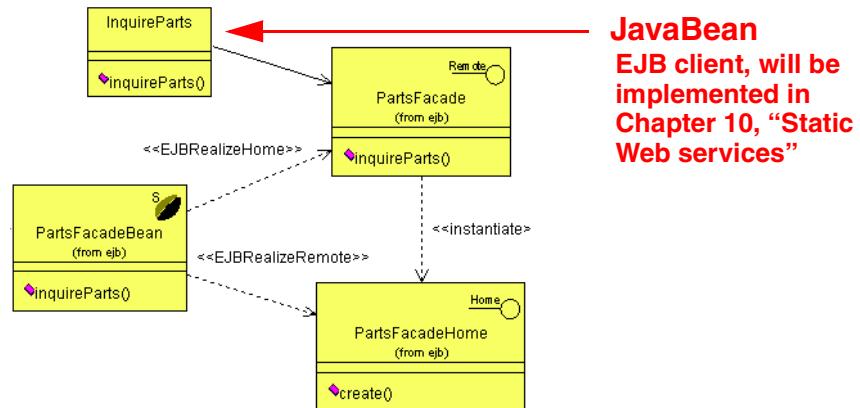


Figure 5-4 Class diagram fragment for the session entity facade EJB

In addition to the information illustrated in the class diagram fragments, there is an EJB relationship between the two entities for parts and inventory, and also a relationship between the session facade and the parts EJB.

Sequence diagram

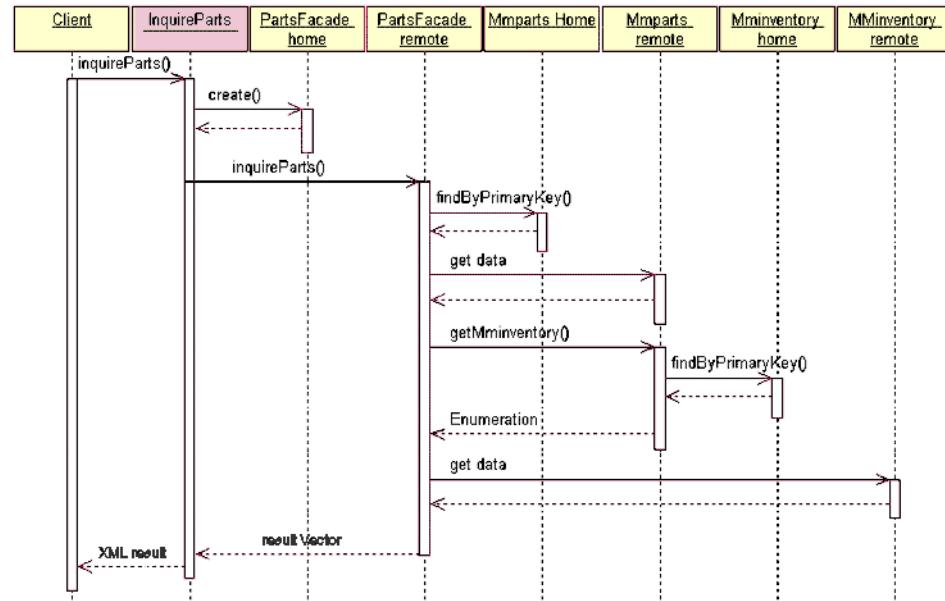


Figure 5-5 Sequence diagram for Stage 2a of the auto parts example

Preparing for development

Before the EJB components are developed, we must first perform some configuration steps. In this section, we:

- ▶ Create a new project
- ▶ Discuss the EJB mapping approaches

Project configuration

Create an EJB project called `ITSO MightyEJB` as described in Chapter 2, “Application Developer overview” on page 17, and associate it with a new EAR project called `ITSO MightyEAR`.

Switching to the J2EE perspective, the Mighty Motors and Almaden Autos projects should appear as shown in Figure 5-6.

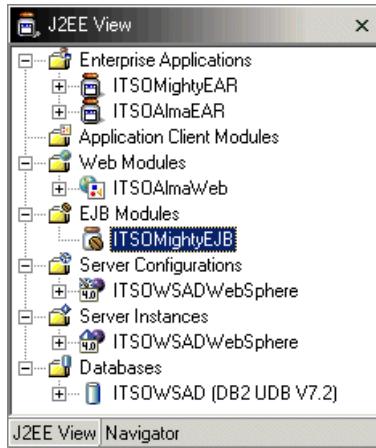


Figure 5-6 J2EE view after EJB and EAR project creation

EJB mapping approaches review

All of the EJB to relational mapping capabilities previously available in VisualAge for Java are now also available in Application Developer, although some techniques require a slightly different approach. As before, there are three different mechanisms:

Top down

The developer creates the CMP entity beans in the application and the database schema and mapping is generated from this definition.

Bottom up	An existing database schema exists for the application, which is imported into the development environment and used to generate the CMP entity beans and the mappings.
Meet in the middle	This scenario is useful when there is a requirement to map a series of new or existing entity beans into an existing database schema.

For simplicity, we are using a schema similar to the Almaden Autos parts inventory system in the ITSOWSAD database, but with table names MMPARTS and MMINVENTORY. This schema was created and populated when executing the DDL during setup. We will use the schema for a bottom-up approach.

Building the entity EJBs

We develop the entity EJBs from the existing parts and inventory tables (MMPARTS and MMINVENTORY). This will include the following steps:

- ▶ Connecting to the database
- ▶ Performing a bottom-up mapping from the schema to create the entity beans
- ▶ Investigating the generated files created by the wizard
- ▶ Viewing the EJB to RDB mapping in the mapping editor
- ▶ Creating a datasource for the CMP EJBs
- ▶ Validating the project and generating the deployed code

Creating the entity EJBs

To perform a bottom up mapping from our schema:

- ▶ Close the data perspective as we no longer require it, and switch to the J2EE perspective.
- ▶ Under the *EJB modules* category, select the ITSOMightyEJB project and select *Generate -> EJB to RDB Mapping* from its context menu.

The mapping wizard opens (Figure 5-7). Because we have no EJBs currently defined in our project, *Top Down* and *Meet In The Middle* are disabled at this time.

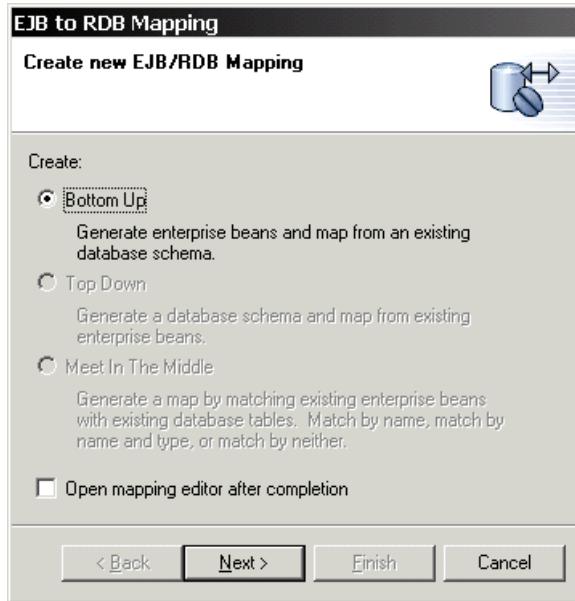


Figure 5-7 EJB to RDB Mapping wizard for ITSOMightyEJB module

- ▶ Select *Bottom Up*, and deselect the *Open mapping editor after completion* checkbox. We will look at the generated mapping later.
- ▶ Click *Next*.

Define the database connection on the *Database Connection* window.

- ▶ Complete the database connection as shown in Figure 5-8.
- ▶ Click *Next*.

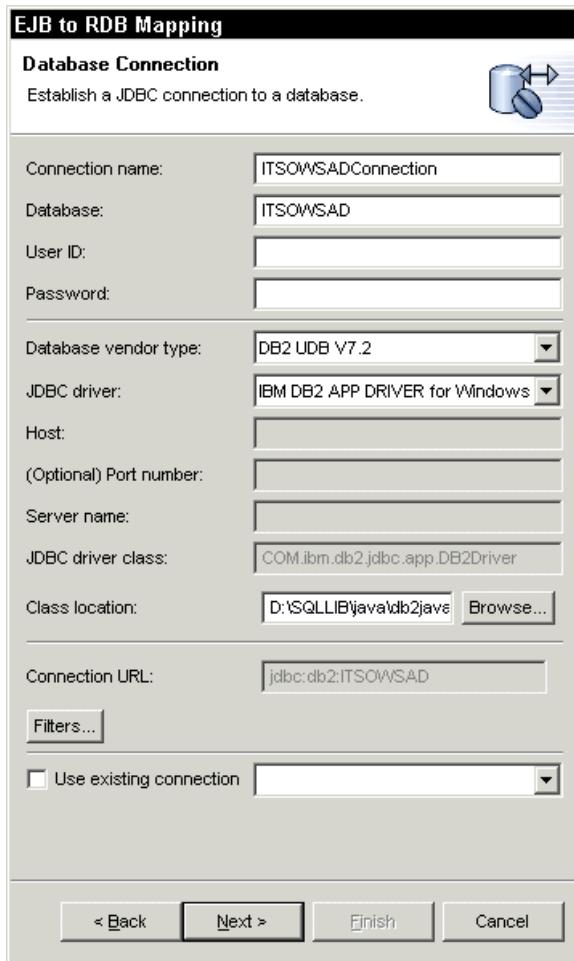


Figure 5-8 EJB to RDB Mapping: database connection definition

Next we select the Mighty Motors tables our EJBs will be accessing. On the Selective Database Import panel:

- ▶ Select the tables prefixed with **MM** (for Mighty Motors) as seen in Figure 5-9.
- ▶ Click *Next*.

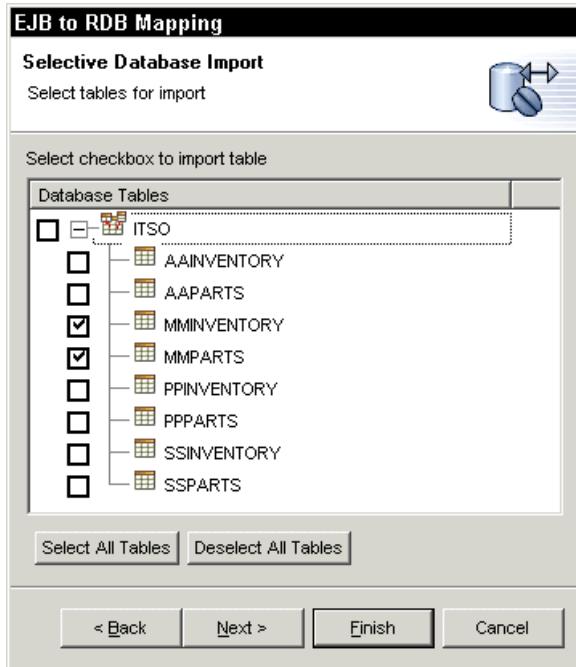


Figure 5-9 EJB to RDB mapping: selective import

- ▶ On the next panel of the wizard we have the option to add a prefix the names of any generated EJBs. By leaving this field blank, the wizard will use the table names defined in the schema.
- ▶ Enter a package name of `itso.wsad.mighty.ejb`.
- ▶ Click *Finish* (be patient, the generation and validation takes some time).

Investigating the generated files

Again, it is worthwhile spending a few moments reviewing the imported database definitions and the generated types from the *EJB to RDB Mapping* wizard.

Table definitions

By expanding the ITSOADSAD database entry under *Databases* in the J2EE view we can see the imported table definitions (Figure 5-10).

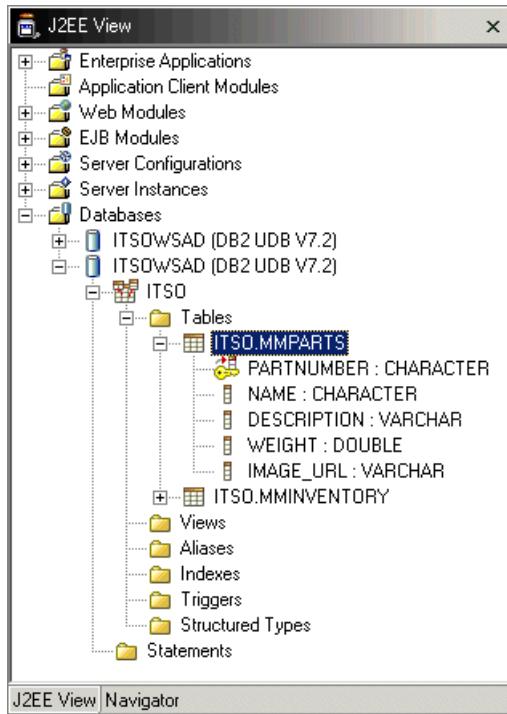


Figure 5-10 Imported tables in the ITSOWSAD database

We now have a local copy of the database definition and we can view the table definitions off-line by double-clicking on them and launching the table editor.

It is important to note that modifying these definitions, such as deleting tables, does not impact the physical database schema. If we make changes here the only caution we must use is not to generate DDL from the modified project and execute it against the database. This is the only way of accidentally overwriting the existing schema with the new one.

XMI files

It is also interesting at this time to open the *Navigator* view on our project. This allows us to view how Application Developer has stored the metadata that represents our schema (Figure 5-11).

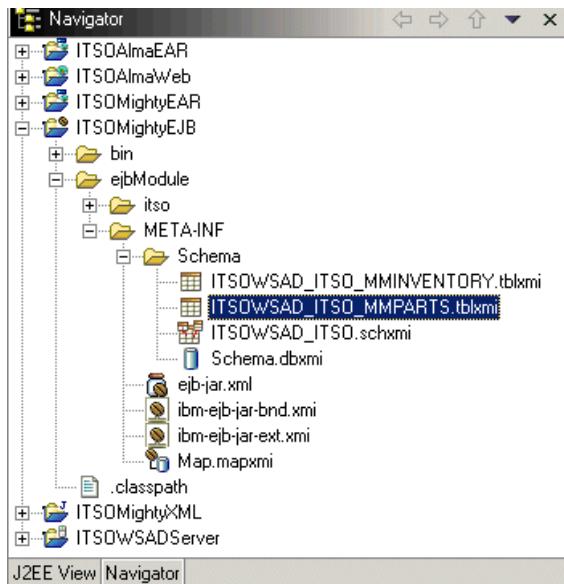


Figure 5-11 XMI files stored in the project for an imported database schema

It is not recommended to edit these files, however their XMI format enables us to store the definitions in our configuration management tool later, as detailed in Chapter 7, “Working in a team” on page 207. XMI is also described in detail in Chapter 2, “Application Developer overview” on page 17.

Application Developer creates a separate XMI file for each table in the schema, with the file extension .tb1xmi, together with one for the schema itself, with a .schxmi extension, and one for the database (Schema.dbxmi). Each of these files has its own dedicated editor. These files are also copied into the bin directory structure for inclusion in the EJB JAR file, although they should not be modified directly from that location.

By default, the bottom-up EJB mapping technique creates a CMP entity EJB for each table in the imported schema metadata. Note that because we only selected the Mighty Motors tables (MM prefix), those are the only ones that appear in the J2EE and Navigator views.

EJB definitions

By expanding the EJB module we can see from the J2EE and Navigator views that the wizard has created two entity beans: Mmparts and Mminventory (Figure 5-12).

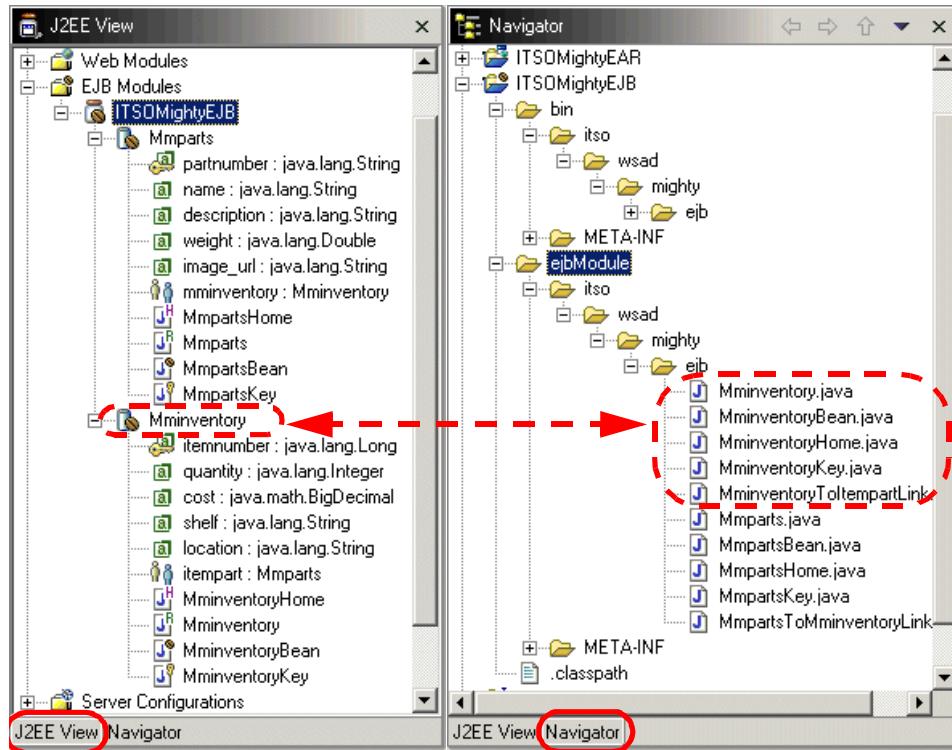


Figure 5-12 J2EE and Navigator views of generated types

- ▶ Application Developer capitalizes only the first letter of the table name—this is not ideal, but we would have to dedicate an entire chapter to refactoring this code if we wanted to change them.
- ▶ For each entity, the key, container managed fields, relationships, and important classes are displayed. Note that the data types used to represent the data are the complex types, such as `java.lang.Long`, instead of the simple alternatives (`long`). This will give us more flexibility when performing data type conversions later.

EJB classes review

Four classes are used to compose a single entity EJB. These are as follows:

- | | |
|--------------------|---|
| EJBNameHome | The home interface. This is a factory that is used to create and find instances of the EJB. |
| EJBName | The remote interface. This class determines which methods can be remotely invoked. |

EJBNameBean	The implementation class. This is where the logic is defined for the methods in the EJB.
EJBNameKey	The key class. Optional when performing top-down persistence with simple keys, this is used to represent the unique key for the entity.

Each of these classes appear in the *Navigator* view as .java files in the package folder defined in the mapping wizard. Two additional classes represent the relationship between parts and inventory EJBs.

Generated EJB metadata

Figure 5-11 on page 143 also shows that three additional XMI files are generated by the mapping wizard into the META-INF folder of the EJB project:

ejb-jar.xml	The deployment descriptor for the EJB module
ibm-ejb-jar-bnd.xmi	WebSphere bindings for the EJB module
ibm-ejb-jar-ext.xmi	WebSphere extensions for the EJB module (both binding files are described in more detail in “EJB project” on page 44)
Map.mapxmi	The mapping between the EJB and the database schema

This behavior is very different to that of VisualAge for Java Enterprise, where this metadata was stored internally inside the repository, which prevented developers from easily building EJBs in a team environment with an external configuration management tool.

It is also worth opening the containing EAR project (double-click on the ITSOMightyEAR application in the J2EE view or the application.xml file in the Navigator view) to see that this new EJB module is included in the EAR file.

Generated mapping

Open the generated mapping by double-clicking the Map.mapxmi file in the Navigator view, or by selecting *Open With -> Mapping Editor* on the EJB module in the J2EE view (Figure 5-13).

This editor opens automatically when you select *Open mapping editor after completion* in the wizard (Figure 5-7 on page 139).

The top half of the editor shows the EJB and table definitions, and the bottom half shows how the contained fields and columns are mapped to each other. Also note the Outline view and the Properties view when selecting items.

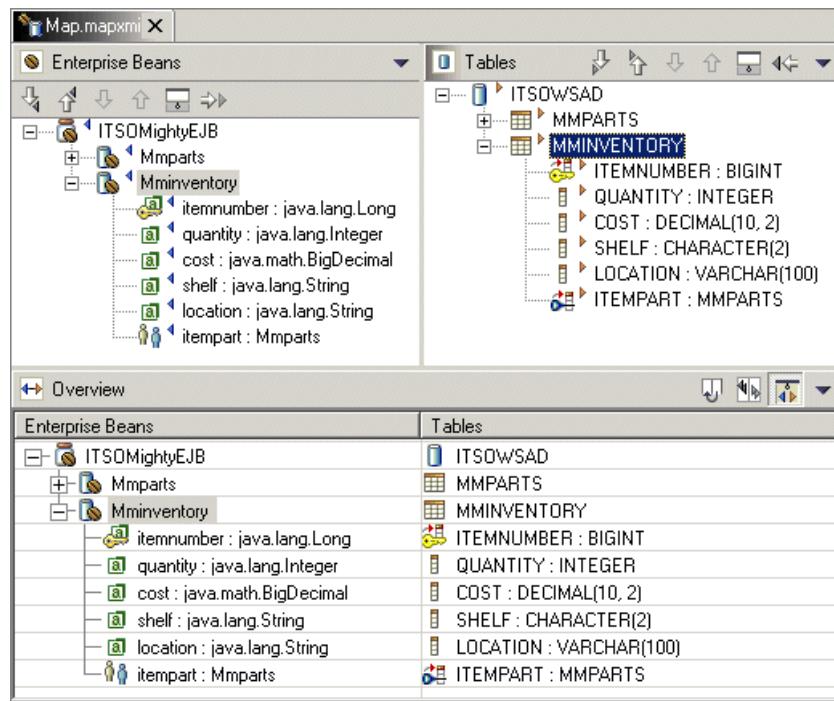


Figure 5-13 EJB to RDB mapping editor

The icons on each pane of the editor provide useful functions. Those used in the EJBs and Tables panes are shown in Figure 5-14, while Figure 5-15 explains the icons in the mapping pane. It is worth spending some time familiarizing yourself with the features of these icons, because they are useful when navigating around the mappings in the future.

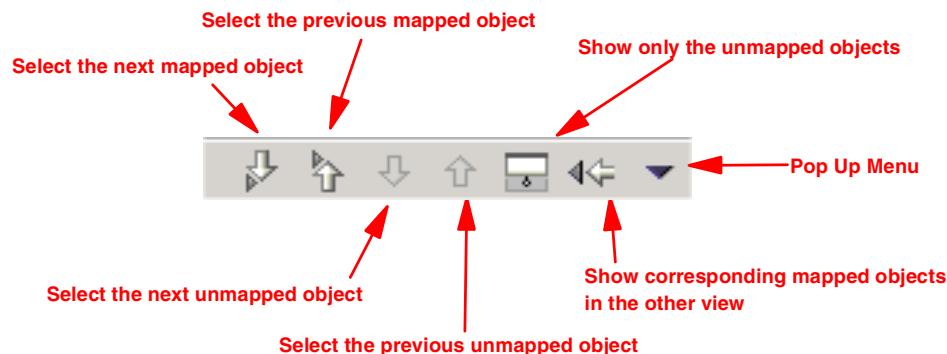


Figure 5-14 Icons used in EJB and Tables panes of EJB to RDB mapping editor

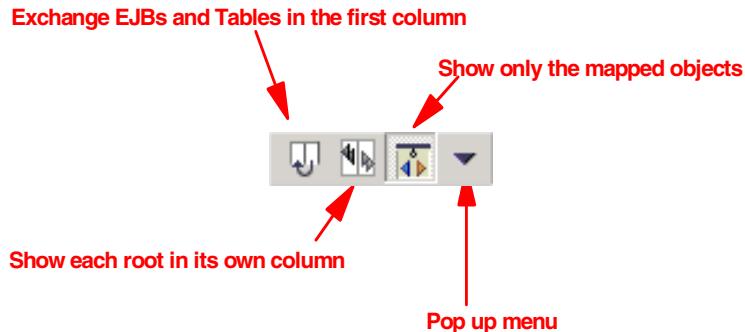


Figure 5-15 Icons used in mapping pane of EJB to RDB mapping editor

In the lower panel, select one of the columns in a table on the right hand side, as shown in Figure 5-16. Note that a drop-down list appears that allows you to select which column is mapped to the corresponding field in the EJB.

Enterprise Beans	Tables
ITSOMightyEJB	ITSOWSAD
Mmparts	MMPARTS
Minventory	MMINVENTORY
Itemnumber : java.lang.Long	ITEMNUMBER : BIGINT
quantity : java.lang.Integer	QUANTITY : INTEGER
cost : java.math.BigDecimal	COST : DECIMAL(10, 2)
shelf : java.lang.String	SHELF : CHARACTER(2)
location : java.lang.String	LOCATION : VARCHAR(100)
Itempart : Mmparts	

Figure 5-16 Mapping a field to a specific column using the mapping editor

Close the mapping editor, but do not save any changes made to the map.

Adding getter methods for the key fields

The Application Developer does not generate a getter method for the key field. We want to get the key field for display purposes so we have to add the *get* method for the key field ourselves.

There are two ways to add the getters for the key field:

- ▶ Have Application Developer generate the getter and setter for the field for the implementation class (select the key field in the Outline view and *Generate Getter and Setter*), then delete the setter method.
- ▶ Manually add the getters to the implementation class.

Open the implementation classes (`MminventoryBean` and `MmpartBean`) and add these methods:

- Getter method for `Mminventory`'s `itemnumber` field:

```
/**  
 * Gets the itemnumber  
 * @return Returns a java.lang.Long  
 */  
public java.lang.Long getItemnumber() {  
    return itemnumber;  
}
```

- Getter method for `MmParts`' `partnumber` field:

```
/**  
 * Gets the partnumber  
 * @return Returns a java.lang.String  
 */  
public java.lang.String getPartnumber() {  
    return partnumber;  
}
```

Add the getter methods to the remote interface by selecting a getter method (`getItemnumber` or `getPartnumber`) in the Outline view and *Enterprise Bean -> Promote to Remote Interface* from the context menu (Figure 5-17).

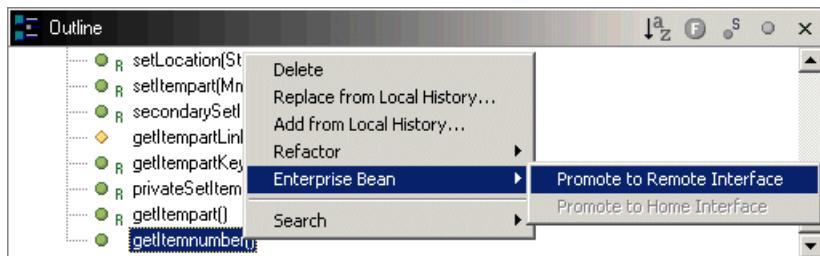


Figure 5-17 EJB key field getter - Promote to Remote Interface

Save the modified bean classes (`MminventoryBean` and `MmpartBean`).

Defining getter methods as read-only

When using bottom-up mapping, Application Developer does not by default mark all getter methods in our EJB as being read-only. Only the relationship getters are set to read-only.

This is an essential technique to obtaining reasonable performance from entity EJBs as it prevents the application server from writing the contents of the EJB back to the data store after each get method is called. To do this:

- ▶ Open the EJB extension editor by selecting the *ITSOMightyEJB* module in the J2EE perspective and *Open With -> EJB Extension Editor* from the context menu.
- ▶ Switch to the *Methods* tab.
- ▶ Expand the *Mminventory* item in the list, and its *Remote methods* element.
- ▶ Select each *get* method that retrieves an entity attribute in turn (multiple selection does not work) and change the access intent from *<None>* to *READ* (Figure 5-18).
- ▶ Repeat for *Mmparts*.
- ▶ Save the changes.

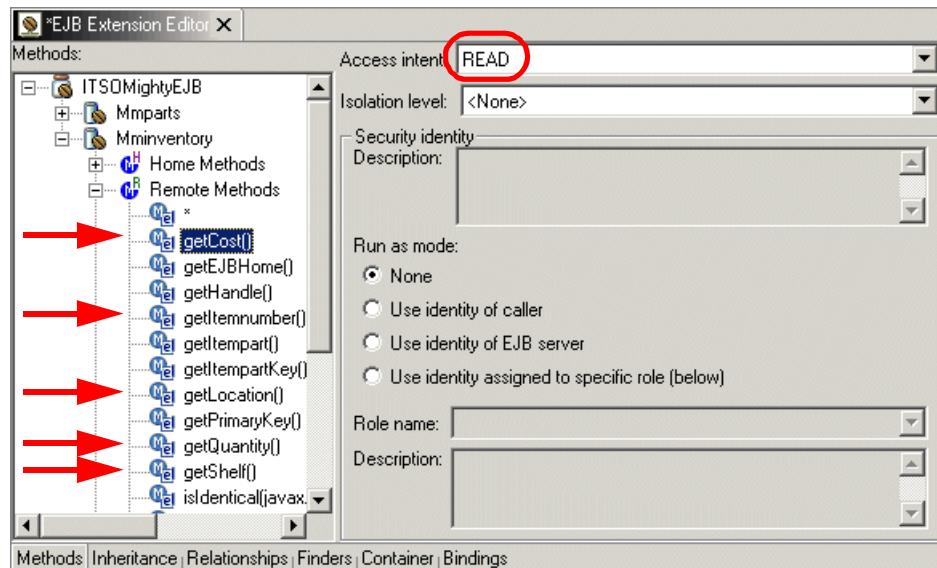


Figure 5-18 Setting EJB methods as read-only

Binding the EJBs to a JDBC data source

Each EJB module must be assigned to a JDBC data source reference in order for its CMP entities to persist correctly. To perform this assignment:

- ▶ Select the *ITSOMightyEJB* module in the *EJB Extension Editor* (if it is not already open).

- ▶ Select the *Bindings* tab and complete the *Datasource* information for the ITSOMightyEJB module (Figure 5-19). The specification of *jdbc/ITSOWSAD* matches the data source we used for the Web application (Figure 3-25 on page 85). Use the user ID and password used when installing DB2 or empty fields.

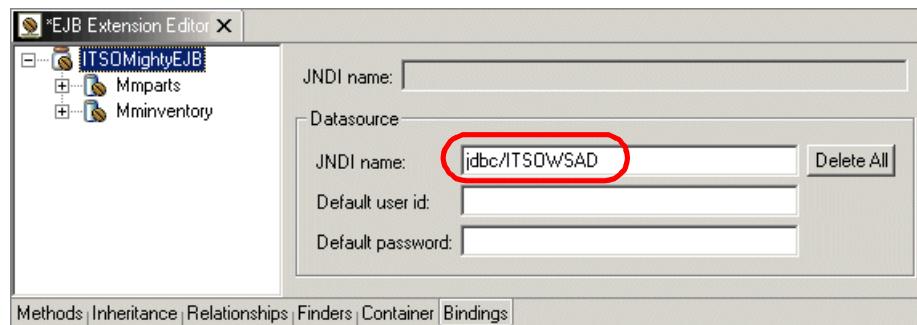


Figure 5-19 Defining the JDBC data source for the EJBs

- ▶ Save the changes and close the editor.

Validating the project

Before any deployed code is generated, we recommend that you validate the EJB project first. The validation process for an EJB project tests that the code is compliant with the Sun Enterprise JavaBeans specification, v1.1, Final release December 17, 1999. Completing this step greatly enhances the chances of the deployed code generation being successful.

Validators are specified in the properties of each project. To examine these properties:

- ▶ Switch to the *Navigator* view in the J2EE perspective and open the *Properties* dialog for the ITSOMightyEJB project.
- ▶ Select the *Validation* entry, and the validators are displayed (Figure 5-20).

There are five validators specified: *DTD Validator*, *EJB Validator*, *Map Validator*, *XML Schema Validator*, and *XML Validator*. These are provided by Application Developer, although this is a defined WebSphere Studio Workbench extension point, which can be used to add further alternatives.

Note there is an option to execute the validation automatically on each save to the contained resources. Although this option is convenient and selected by default, it increases the save time for each change, so we recommend leaving it as a manual process.

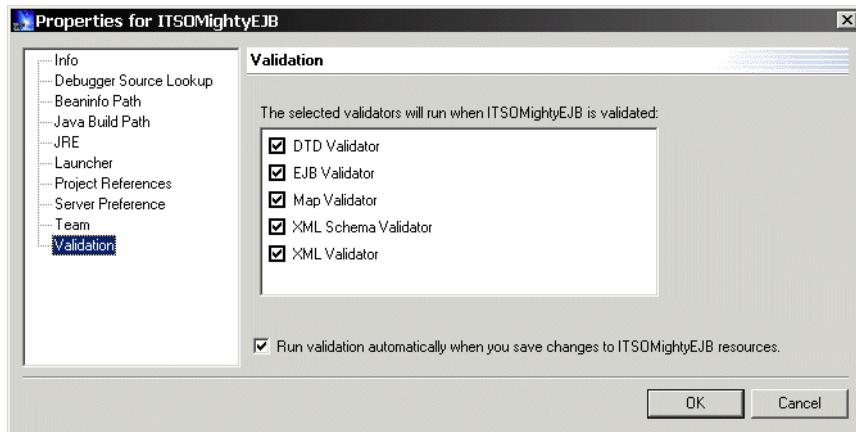


Figure 5-20 Validation options for an EJB project

To validate the project:

- ▶ Close the *Properties* dialog and return to the Navigator view.
- ▶ Select the ITSOMightyEJB project and *Run Validation* from its context menu.

When the validation has completed, no entries should appear in the task list.

Generating the deployed code

After having validated the EJB code against the specification, we can generate the deployed code to support execution in WebSphere Version 4.0:

- ▶ Switch back to the *J2EE* view, select the ITSOMightyEJB EJB module and *Generate -> Deploy and RMIC Code*.
- ▶ Ensure both entity EJBs are selected and click *Finish* (Figure 5-21).

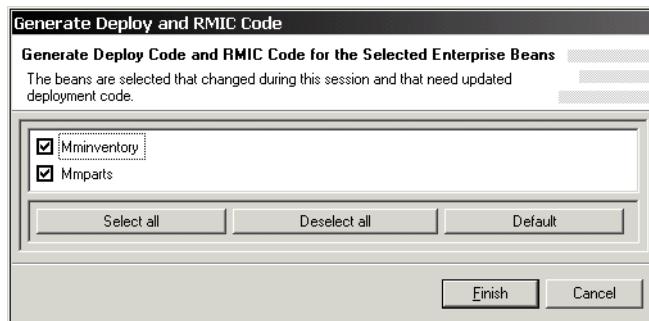


Figure 5-21 Generating the deployed code for the EJB module

This option launches the command line ejbdeploy tool which ships with WebSphere to generate the deployed code. Optionally, this tool can be launched from the command line if you would like to perform a batch deployment on a regular basis from a stream in a configuration management system.

After switching back to the *Navigator* view expand the `itso.wsad.mighty.ejb` package in the `ejbModule` folder to see the deployed code.

We have now completed developing the two entity EJBs using bottom up mapping and generated the deployed code. The next step is to test them in the local WebSphere instance.

Testing the entity EJBs

During this chapter, we will iteratively test the components we develop. This allows us to identify any potential problems early in the development phase. The following section describes deploying and testing the two entities we have just created, `Mmpart` and `Mminventory`, by performing the following:

- ▶ Associating the project with a server configuration
- ▶ Running the EJB test client
- ▶ Invoking an instance of each EJB and testing their methods

Associating the EAR project with a server configuration

As we discussed in the previous chapter, we will be using a single application server instance and configuration, `ITSOWebSphere`, to deploy all of the enterprise application samples developed in this book.

To add the `ITSOMightyEAR` project to the configuration, switch to the server perspective, and in the *Server Configuration* view select the `ITSOWebSphere` server configuration, and click on the *Add Project -> ITSOMightyEAR* (Figure 5-22).

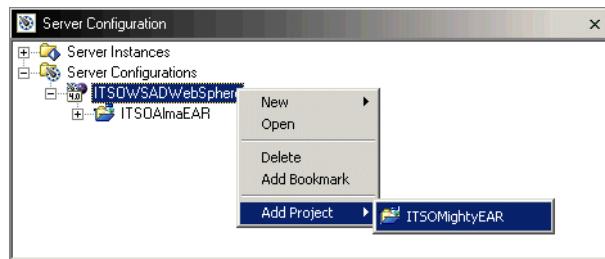


Figure 5-22 Adding the EAR project to the server configuration

Executing the EJB test client

Application Developer provides a first-class browser-based EJB test client that enables us to connect to our EJB and test its methods. It is significantly more functional than the test client incorporated in VisualAge for Java.

Note: The EJB test client is also called the universal test client (UTC) because it can be used to test Web services and other applications as well.

There are two approaches to launching the test client:

- ▶ The first is to click on the *Run on Server* menu for the EJB module we would like to test in the J2EE perspective *J2EE* view (or in the Navigator view of the server perspective). This starts the server in debug mode and opens the test client.
- ▶ The second approach, which we will be following in this section, is to stay in the server perspective, start the server manually, and then launch the test client.

Start the server

To start the server manually:

- ▶ In the server perspective, *Server Configuration* view, double-click the ITSOWSADWebSphere configuration. On the EJB page, check that the *Enable EJB test client* checkbox is enabled. Close the editor.
- ▶ In the *Servers* view select the ITSOWSADWebSphere server instance and start the server using the start icon or *Start* from its context menu (Figure 5-23).

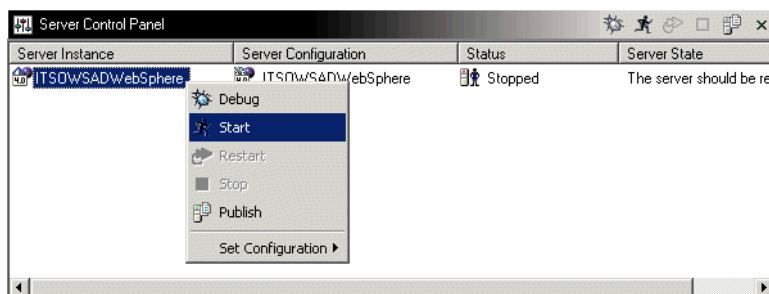


Figure 5-23 Starting the WebSphere server instance

- ▶ Wait until the server has completed starting (indicated either by the “Server open for e-business” message in the *Console* view or a *Started* status in the *Server Control Panel* view).

Start the EJB test client

Select the ITSOMightyEJB project in the Navigator view and *Run on Server* from the context menu.

Alternatively, you can click on the *Open Web Browser* icon  in the toolbar and enter the following URL in the location field:

`http://localhost:8080/UTC`

The browser opens with the EJB test client home page (Figure 5-24).

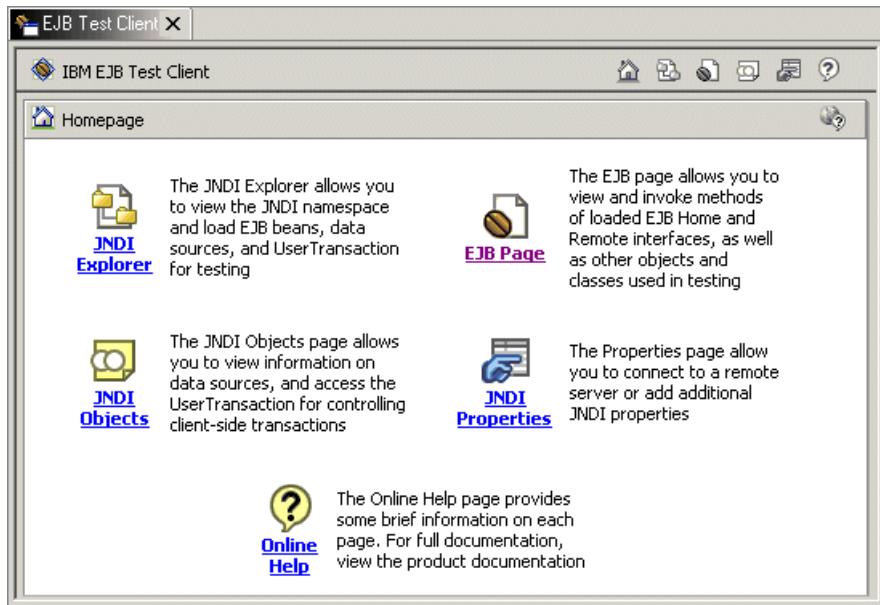


Figure 5-24 Home page for the EJB test client

Testing the entity EJBs

Already we can see some exciting options. Most of the testing is done using the JNDI Explorer to locate the EJBs and the EJB Page to instantiate EJBs and run their methods.

Our first test should be to see if our EJBs have been added to the JNDI namespace:

- ▶ Click on the *JNDI Explorer* link. Expand the folders to see the JNDI names (Figure 5-25).

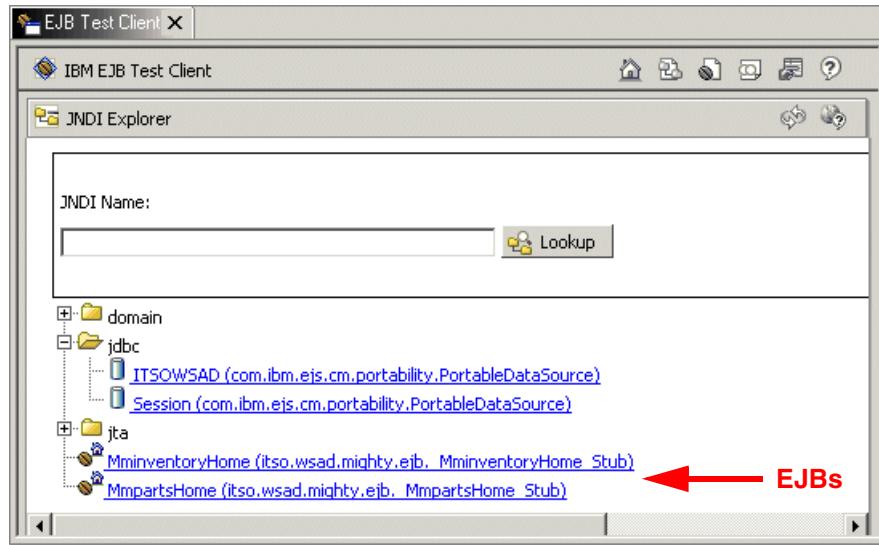


Figure 5-25 JNDI explorer in the EJB test client to browse the JNDI namespace

There are two entries—*MminventoryHome* and *MmpartsHome*—in the namespace. The reason for these names is that we have not defined a JNDI entry for the EJBs in the EJB extension editor and WebSphere takes the name of the home class as a default. We will change this later.

If you fail to see the entries for the sample EJBs, return to the EJB extension editor and ensure that the datasource has been defined correctly for the EJB module.

To test the parts entity:

- ▶ Click on the *MmpartsHome* link to open the EJB page of the test client for the selected item.
- ▶ Expand the EJB references tree. The methods available on the EJBs home interface (*create* and *findByPrimaryKey*) are visible.
- ▶ Select the *findByPrimaryKey* link.
- ▶ On the Parameters frame, select the Constructors drop-down and *MmpartsKey(String)* to create a key object (Figure 5-26).

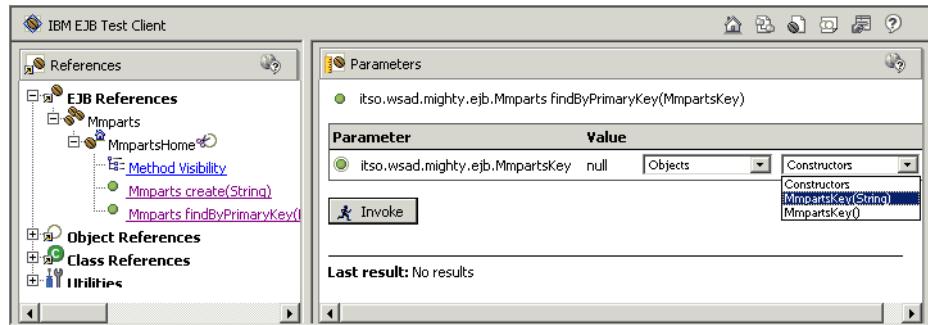


Figure 5-26 Creating the parameter for the *findByPrimaryKey* method

- ▶ This link invokes a page which prompts for the key constructor parameter. Enter a valid part number of M10000003 and click on *Invoke and Return* (Figure 5-27).

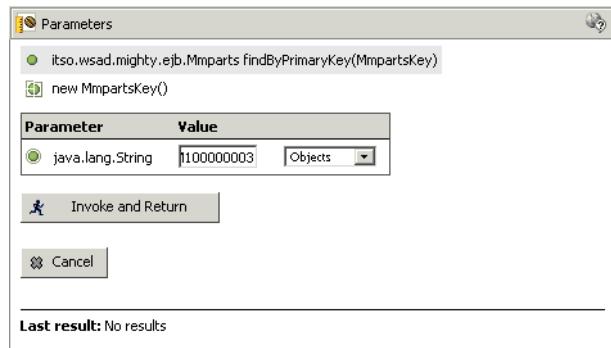


Figure 5-27 Defining the key constructor parameter

- ▶ Back in the Parameters panel, click on the *Invoke* button to complete the call to the *findByPrimaryKey* method.
- ▶ In the results panel you should see an instance of Mmparts returned from the server. Click on the *Work with Object* button.
- ▶ This action adds the remote Mmparts object matching our part number to the EJB references tree. Expanding the tree for this element displays the methods available on the EJBs remote interface (Figure 5-28).

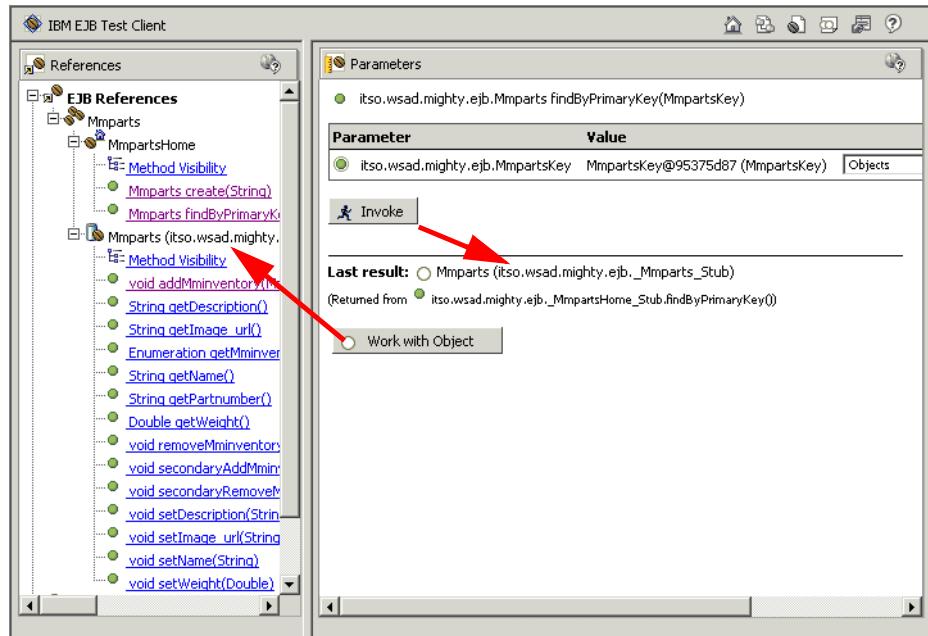


Figure 5-28 Result remote object

- ▶ Click on the *String getDescription* link and on the Parameters pane, click on the *Invoke* button. The results panel displays the description for the part (Figure 5-29).

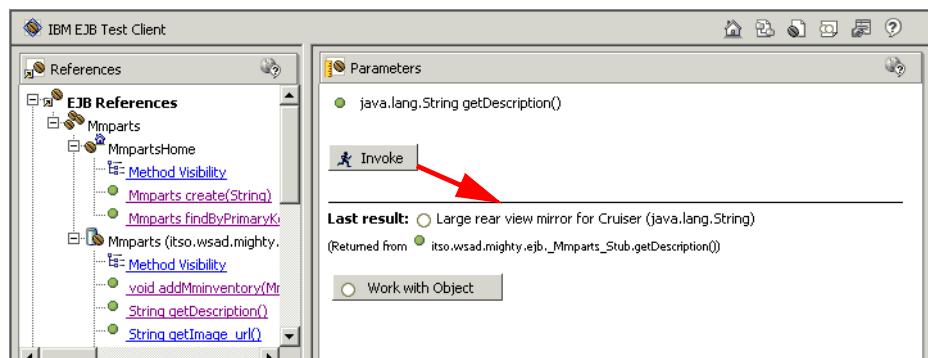


Figure 5-29 Results from the *getDescription* method call on the remote interface

Repeat the test for the *MminventoryHome* entry in the JNDI namespace with an inventory number of 21000003. Invoking the returned remote interface method *getLocation* should return a location of MM - San Francisco.

You can also test the setter methods and change some of the data stored in the underlying tables.

We have now tested our entity EJBs using the EJB container in the WebSphere test environment. Being able to perform this level of unit testing is very beneficial because it reduces the amount of debugging we might have to perform at a later stage. Next, we must develop the business logic in our application.

Before progressing, stop the browser and the ITSOWSADWebSphere server.

Developing the session EJB facade

A well known best practice for EJB programming is to wrap all calls to entity EJBs by a session EJB that represents the business process. This is known as the session entity facade pattern, and is described in more detail in Chapter 13 of the ITSO redbook *EJB Development with VisualAge for Java*, SG24-6144.

The following section will describe how to build a session entity facade to the two entity EJBs created earlier in this chapter. The session bean will contain only two methods to find all the matching inventory item instances for a given part number. One method returns the inventory items in a vector, the other in an array, to enabling us to explore Web services later.

To build the facade, we will perform the following:

- ▶ Create a new session EJB.
- ▶ Complete its implementation.
- ▶ Modify its remote interface to expose the methods containing the business logic.

Creating the session EJB

To create a session EJB in the ITSOMightyEJB project:

- ▶ Switch to the J2EE perspective J2EE view, select the ITSOMightyEJB module and *New -> Enterprise Bean* from the context menu.
- ▶ On the first dialog of the *Create an Enterprise Bean* wizard, complete fields as shown in Figure 5-30.
 - Set the Enterprise Bean type to *Session bean*.
 - Set the *Bean name* to *PartsFacade*.
 - Set the *EJB Project* as *ITSOMightyEJB*.
 - Leave the *Source folder* as *ejbModule*.
 - Set the *Default package* to *itso.wsad.mighty.ejb*.

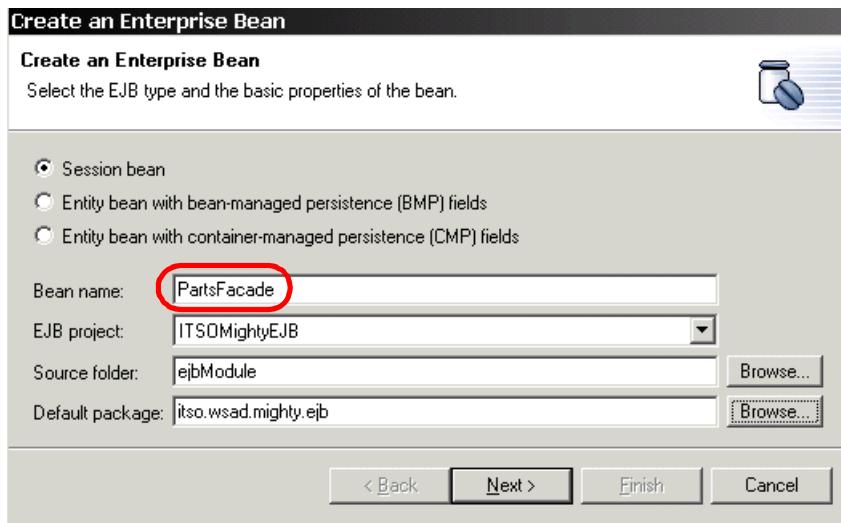


Figure 5-30 Create Enterprise Bean for the session entity facade

- ▶ Click *Next* and take the defaults specified in the *Enterprise Bean Details* dialog (Figure 5-31).
- ▶ Click *Finish*.

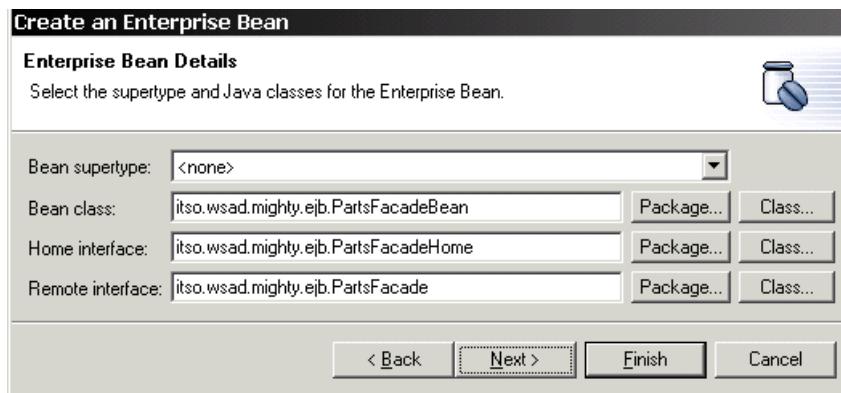


Figure 5-31 Session bean details

When we return to the *J2EE* view, we see a *PartsFacade* entry under the *ITSOMightyEJB* module containing entries for the home, remote interface, and implementation class of the new session bean.

Completing the session entity facade

We now have to add the methods that contain the business logic. Our business logic involves accessing the previously defined entity beans and requesting the inventory items for a given part number.

The EJB 1.1 specification requires that EJBs return serializable results. Therefore, we return the part inventory as a Vector of serializable JavaBeans. So first, we will create the serializable JavaBean class called PartInventory. The code is available in sampcode\wsadejb\PartInventory.java.

Defining the PartInventory bean

To create the PartInventory bean:

- ▶ In the Navigator view select the ITSOMightyEJB project and *New -> Other -> Java -> Java Package*, and enter itso.wsad.mighty.bean as package name (under the ejbModule folder).
- ▶ Select the bean folder and *New -> Other -> Java -> Java Class*. Complete the *Java Class* wizard as seen in Figure 5-32. Click on *Add* for Extended interfaces and start typing Seri... to locate the java.io.Serializable interface. Select *Constructors from superclass* and click *Finish*.

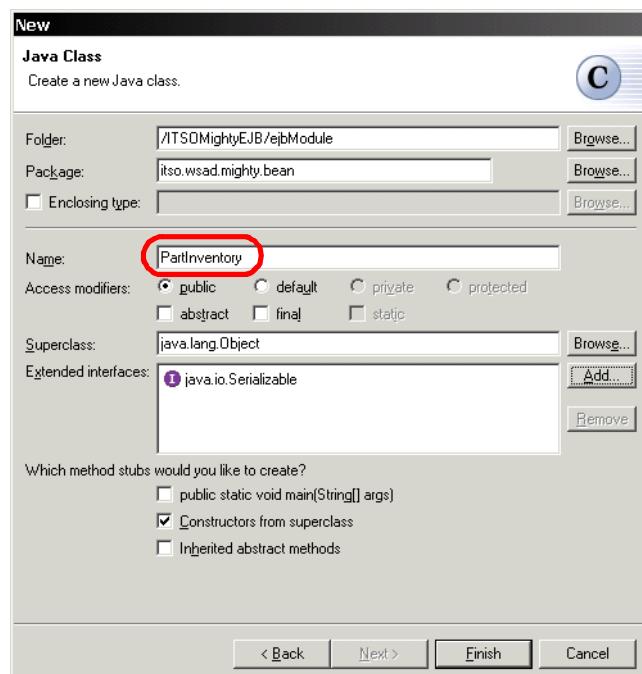


Figure 5-32 Java Class wizard: PartInventory bean

The PartInventory bean stores the attributes of the Mmparts and Mminventory entity beans. Therefore, we define fields for all the attributes:

```
protected java.lang.String      partNumber;
protected java.lang.String      name;
protected java.lang.String      description;
protected double                weight;
protected java.lang.String      imageUrl;
protected java.lang.Long       itemNumber;
protected int                  quantity;
protected java.math.BigDecimal cost;
protected java.lang.String      shelf;
protected java.lang.String      location;
```

- ▶ To generate the getters and setters for each field, select each field in the Outline view and *Generate Getter and Setter* from the context menu.
- ▶ Add a `toString` method for testing:

```
public String toString() {
    return("Part="+partNumber+"/"+name+"/"+weight+
          "/item/"+itemNumber+"/"+quantity+"/"+cost+"/"+shelf+""/+location);
}
```

- ▶ Save and close the bean.

Completing the business logic in the session bean

We now need to complete the implementation of the session bean. We will add:

- ▶ A `partsHome` field to store the home of the Mmparts entity bean
- ▶ A `getPartsHome` method to retrieve the home
- ▶ The business logic methods `inquireParts` and `inquirePartsArray`

The session bean code is available in `sampcode\wsadejb\PartsFacadeBean.java`. To complete the implementation of the session bean, perform these steps:

- ▶ Open the PartsFacadeBean implementation class of the session EJB in the Java editor. Add the following import statements:

```
import javax.naming.*;
import javax.ejb.*;
import java.util.*;
import itso.wsad.mighty.bean.PartInventory;
```
- ▶ Add an instance variables to the class:

```
// Home cache
private MmpartsHome partsHome;
```
- ▶ Now add the helper method, `getPartsHome` to ensure that we have only a single instance of the parts EJB home in the session bean (Figure 5-33).

```

/**
 * getPartsHome - share home of part
 */
private MmpartsHome getPartsHome() {
    if (partsHome == null) {
        try {
            InitialContext ic = new InitialContext();
            Object objref = ic.lookup("java:comp/env/ejb/Mmparts");
            partsHome = (MmpartsHome)javax.rmi.PortableRemoteObject.narrow(
                objref, MmpartsHome.class);
        } catch (NamingException ex) {
            ex.printStackTrace();
            throw new EJBException("Error looking up MmpartsHome: "+
                ex.getMessage());
        }
    }
    return partsHome;
}

```

Figure 5-33 Source code for getPartsHome (helper method)

getPartsHome

This method is responsible for creating a singleton of the home interface of the parts entity EJB for all invocations on the session. It uses a local reference to look up the home in the naming service using the J2EE `java:comp/env` approach. We will map this local JNDI name to a global JNDI name in the deployment descriptor for the EJB module later.

Now we must add the business logic to get the parts in the inventory based on a part number and return a vector of PartInventory beans.

We add two methods to the PartsFacadeBean:

inquireParts

This method retrieves the part entity bean and all the inventory entity beans related to it. For each inventory item, a PartInventory bean is created with all the attribute values and the bean is added to the result vector.

inquirePartsArray

This method converts the result vector of the inquireParts method into an array. This enables us later to explore how Web services handle vectors and arrays. (This method is not needed for the auto parts application.)

Figure 5-34 shows the inquireParts method and Figure 5-35 shows the inquirePartsArray method.

```

/**
 * inquireParts - find matching parts and return as Vector of beans
 */
public Vector inquireParts(String partNumber) {
    Vector      resultVector = new Vector();
    PartInventory  partInv;
    Mmparts      part = null;
    Mminventory   inv  = null;
    try {
        part = getPartsHome().findByPrimaryKey(
            new MmpartsKey(partNumber));
        Enumeration items = part.getMminventory();
        while (items.hasMoreElements()) {
            inv = (Mminventory)javax.rmi.PortableRemoteObject.narrow
                (items.nextElement(), Mminventory.class);
            partInv = new itso.wsad.mighty.bean.PartInventory();
            partInv.setPartNumber ( partNumber );
            partInv.setName      ( part.getName() );
            partInv.setDescription( part.getDescription() );
            partInv.setWeight     ( part.getWeight().doubleValue() );
            partInv.setImageUrl  ( part.getImage_url() );
            partInv.setItemNumber ( inv.getItemnumber() );
            partInv.setQuantity   ( inv.getQuantity().intValue() );
            partInv.setCost       ( inv.getCost() );
            partInv.setShelf      ( inv.getShelf() );
            partInv.setLocation   ( inv.getLocation() );
            resultVector.addElement(partInv);
        }
    } catch (FinderException ex) {
        System.out.println("Cannot find part: "+partNumber);
        ex.printStackTrace();
    } catch (java.rmi.RemoteException ex) {
        ex.printStackTrace();
    }
    if (resultVector.size() == 0) return null;
    else return resultVector;
}

```

Figure 5-34 PartsFacade business logic: inquireParts

```

/**
 * inquirePartsArray - find matching parts and return as array of beans
 */
public PartInventory[] inquirePartsArray(String partNumber) {
    Vector resultVector = inquireParts(partNumber);
    if (resultVector == null) return null;
    int noResults = resultVector.size();
    PartInventory[] result = new PartInventory[noResults];
    for (int i=0; i < noResults; i++) {
        result[i] = (PartInventory)resultVector.elementAt(i);
    }
    return result;
}

```

Figure 5-35 PartsFacade business logic: inquirePartsArray

The majority of the coding for this class is complete. Save all changes, but leave the editor open.

Promoting the new methods to the EJB remote interface

In order for the Web service client to call the *inquireParts* method in the session EJB, we must first promote it to the remote interface:

- ▶ While still in the PartsFacadeBean editor, select the *inquireParts* and *inquirePartsArray* methods in the *Outline* view.
- ▶ Select *Enterprise Bean -> Promote to Remote Interface* from the context menu.
- ▶ Close the PartsFacadeBean editor and save the changes.

There will now be a compiler errors in the PartsFacade remote class because *PartInventory* and *Vector* cannot be resolved. To correct this, add these import statements:

```

import itso.wsad.mighty.bean.PartInventory;
import java.util.Vector;

```

Save and close the PartsFacade class.

Defining the EJB references

The session EJB uses a local JNDI reference to look up the home for the entity EJB using the following code:

```
ic.lookup("java:comp/env/ejb/Mmparts")
```

This may be unfamiliar to developers who have previously worked with version 3.5 of WebSphere and VisualAge for Java. The notation `java:comp/env` defines that the lookup is to a local reference of `ejb/Mmparts` instead of an entry in the global JNDI namespace.

Each of the EJBs therefore has two names, which are mapped using a reference definition in the deployment descriptor, `ejb-jar.xml`.

To complete the example, we must edit the deployment descriptor in the ITS0MightyEJB project, and add the reference for the PartsFacade session bean to the Mmparts entity bean:

- ▶ In the J2EE perspective *J2EE* view, select the ITS0MightyEJB EJB module and *Open With- > EJB Editor* from the context menu. Switch to the *EJB References* tab.
- ▶ Two references are already there for the association between part and inventory entity beans.
- ▶ Add a new reference for the PartsFacade bean by selecting the record in the table and clicking the *New* button.
- ▶ Add the `ejb/Mmparts` reference to the PartsFacade bean as shown in Figure 5-36. The entry is identical to the entry under the `Mminventory` bean.
- ▶ Save the changes and close the editor.

EJB Bean	Type	Home	Remote	Link
Mmparts				
ejb/Mminventory	Entity	itso.wsad.mighty.ejb.MminventoryH	itso.wsad.mighty.ejb.Mminven	Mminven
Mminventory				
ejb/Mmparts	Entity	itso.wsad.mighty.ejb.MmpartsHome	itso.wsad.mighty.ejb.Mmparts	Mmparts
PartsFacade				
ejb/Mmparts	Entity	itso.wsad.mighty.ejb.MmpartsHome	itso.wsad.mighty.ejb.Mmparts	Mmparts

A red arrow points to the last row of the table, labeled "new entry".

Figure 5-36 EJB references defined for the application

We must also edit the binding details in the EJB extension editor:

- ▶ Select the ITS0MightyEJB EJB module and *Open With- > EJB Extension Editor* from the context menu.

- ▶ Switch to the *Binding* tab. The EJB references are displayed (Figure 5-37).

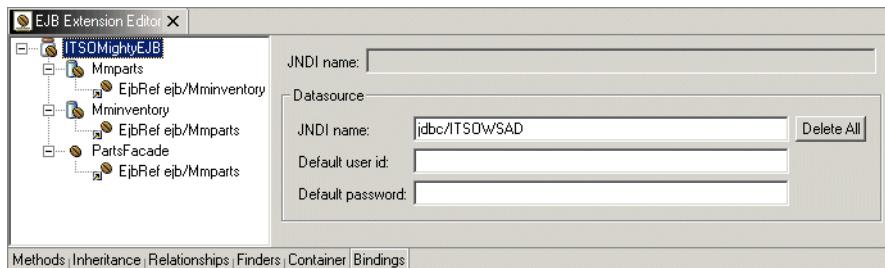


Figure 5-37 Binding details for the Mighty Motors EJBs

- ▶ Select each EJB and each reference and set the JNDI name as shown in Table 5-1 (do not touch the Datasource JNDI name).

Table 5-1 EJB bindings defined for Mighty Motors

Element	JNDI name
Mmparts	itso/wsad/Mmparts
EjbRef ejb/Mminventory	itso/wsad/Mminventory
Mminventory	itso/wsad/Mminventory
EjbRef ejb/Mmparts	itso/wsad/Mmparts
PartsFacade	itso/wsad/PartsFacade
EjbRef ejb/Mmparts	itso/wsad/Mmparts

- ▶ Save the changes to the EJB extensions and close the editor

We have now correctly defined EJB references and full JNDI names for each EJB in the Mighty Motors system.

Validating the project

Again, before any deployed code is generated, we recommend that you validate the EJB project first as described in “Validating the project” on page 150. This step greatly enhances the chances of the deployed code generation being successful.

Generating the deployed code

We now generate the deployed code to support execution in WebSphere 4.0 for the *PartsFacade* session EJB:

- ▶ In the *J2EE* view, select the *ITSO Mighty EJB* EJB module and *Generate -> Deploy and RMIC Code* from the context menu.
- ▶ We have previously generated the deployed code for the entity EJBs so we only have to generate the deployed code for the session EJB. After selecting the session bean, click *Finish* in the deployment dialog.

In the *Navigator* view we can see the deployed code for the EJBs in the *itso.wsad.mighty.ejb* package in the *ejbModule* folder.

We have now completed developing the session EJB and generated the deployed code. The next step is to test the session bean in the local WebSphere instance.

Testing the session EJB

To launch the EJB test client:

- ▶ First ensure that the *ITSOWSADWebSphere* server is stopped. Restart the server for the session bean and new JNDI entries to be added.
- ▶ Start the EJB test client by selecting *Run on Server* from the context menu of the *ITSO Mighty EJB* EJB module.
- ▶ Repeat the steps performed previously in “Testing the entity EJBs” on page 152 to find the entries in the JNDI Explorer. Note that the entries are now under *itso -> wsad*.
- ▶ Select the *PartsFacade* session bean and execute the *create* method in the home interface to instantiate a session bean.
- ▶ Click on *Work with Object* to add the instance to the remote EJB references.
- ▶ Invoke the *inquireParts* method with a part number of M10000003.
- ▶ A Vector should be returned as the result. Note how the content of the Vector is displayed using the *toString* method of the *PartInventory* bean (Figure 5-38).
- ▶ Click *Work with all Contained Objects*.
- ▶ Two *PartInventory* objects are created under the *Object References* section of the tree.
- ▶ Expand the first object and invoke the *getLocation* method; it should return a string containing MM - San Francisco.
- ▶ You can also test the *inquirePartsArray* method of the session bean.
- ▶ Close the browser and stop the server.

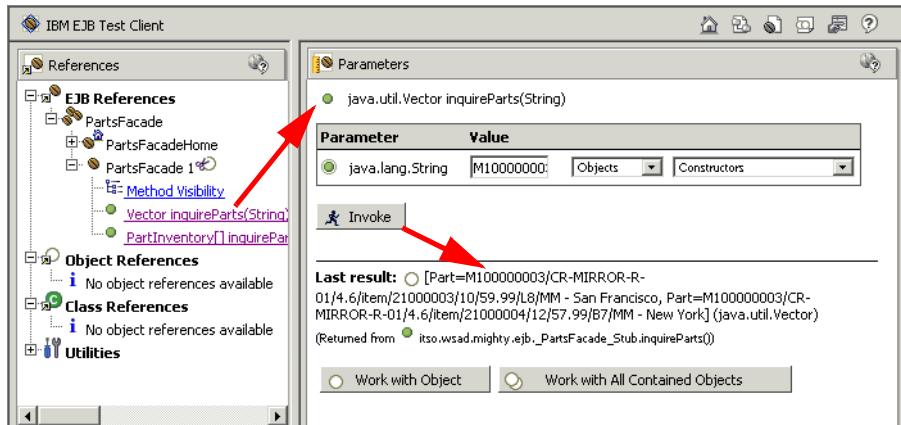


Figure 5-38 Testing the session bean in the EJB test client

Summary

The EJB application for Mighty Motors is now complete and ready for us to implement as a Web service. We have explored the EJB to RDBMS mapping capabilities, generated a session facade containing the business logic and tested the application with the local test environment.

The next chapter discusses how we can now take this application and deploy it from Application Developer into a stand-alone WebSphere environment.

Quiz: To test your knowledge of the EJB development features of Application Developer, try to answer the following five questions. The answers are provided in Appendix C, “Quiz answers” on page 559.

1. What are the three EJB to RDBMS mapping approaches supported by Application Developer?
2. Which file contains the generated metadata for the mapping?
3. How do you bind an EJB to a given data source?
4. What are the names of the two supplied project validators?
5. In which files are the EJB references defined?



Deployment of Web and EJB applications to WebSphere

In this chapter we deploy the first two stages of our sample application that we developed in the previous chapters to WebSphere Application Server 4.0. We also describe how to deploy the EJB test client.

We describe the deployment to

- ▶ WebSphere Application Server Advanced Edition 4.0 Single Server (AEs)
- ▶ WebSphere Application Server Advanced Edition 4.0 Full Version (AE)
- ▶ WebSphere Application Server Advanced Edition 4.0 Single Server from within Application Developer (remote unit test)

Differences between WebSphere AEs and AE

Before discussing the deployment to WebSphere AEs and WebSphere AE let us first summarize the main differences between those two available editions of WebSphere Application Server 4.0. The differences are listed in Table 6-1.

Table 6-1 Differences between WebSphere AEs and AE

Advanced Edition, Single Server	Advanced Edition
Runs on one machine	Runs on multiple, distributed machines
One application server	Multiple application servers
No WLM support	WLM support
XML file based configuration	Relational DB Repository + XML files
Local OS security	Local OS, LDAP, Custom security
Browser-based administration console	Java stand-alone administration console
J2C not supported	J2C supported
Enterprise Extensions not supported	Enterprise Extensions supported
No Merant JDBC drivers	Merant JDBC drivers included

It is also important to note that because there is only one application server in WebSphere AEs, the administration server is contained in the WebSphere AEs application server—that is, there is no separate administration server.

Administration of the single application server in WebSphere AEs is done through a specified port of the application server itself.

Because WebSphere Advanced is intended for multiple application servers, it contains a separate administration server through which administration of the multiple application servers is done.

Deployment to WebSphere AEs

In this section we discuss the deployment of the first two stages of our sample application to WebSphere AEs.

Exporting the EAR files

We will deploy the Alma Autos and Mighty Motors enterprise applications we developed in the previous chapters. Before we can deploy these sample applications, we have to package them in an enterprise archive file (EAR) file.

To create an EAR file is a simple process:

- ▶ From any perspective select *File -> Export -> EAR file* and click *Next*.
- ▶ Select the ITS0AlmaEAR (or ITS0MightyEAR) project and specify an output directory on the file system for the EAR file as shown in Figure 6-1.

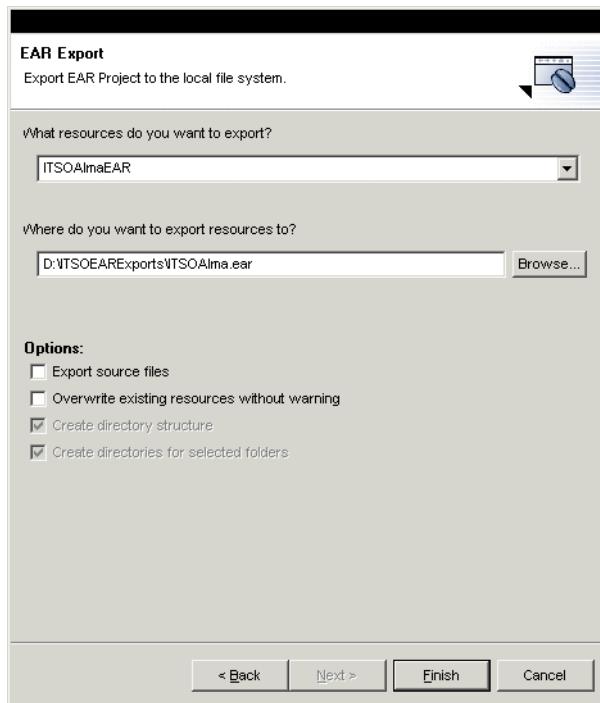


Figure 6-1 Exporting an EAR file

Now we have EAR files of the two projects, we can install in WebSphere Application Server 4.0:

ITS0Alma.ear
ITS0Mighty.ear

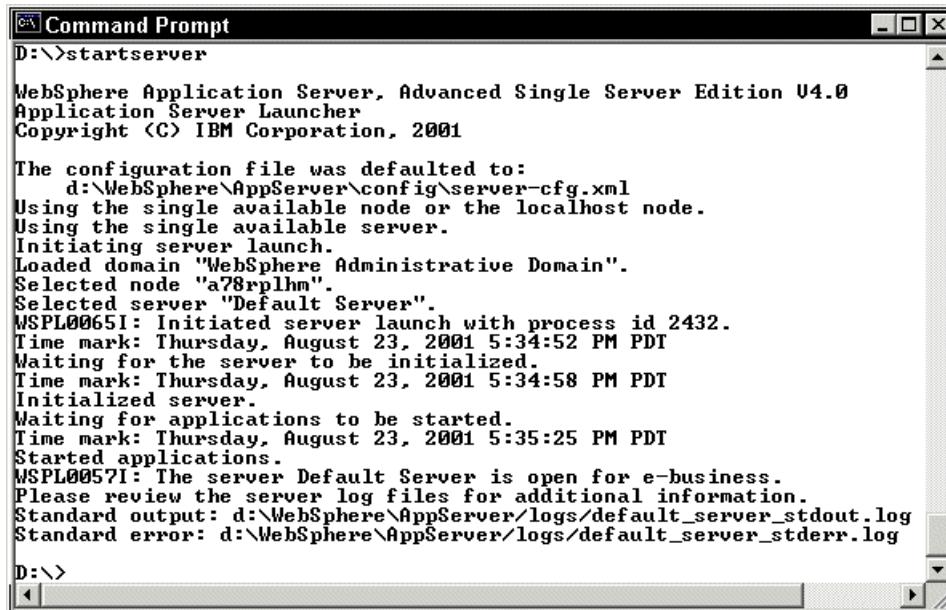
Starting the AEs server

To deploy the applications we will use the Administrator's Console, but first we have to start the Application Server:

- ▶ Select *Start -> Programs -> IBM WebSphere -> Application Server V4.0 AES -> Start Application Server*

or

- ▶ Open a *Command Prompt* and type **startserver**. *Startserver.bat* is located in **WAS_ROOT\bin** where **WAS_ROOT** is the directory where you installed WebSphere, such as **d:\WebSphere\AppServer**. The Console output is shown in Figure 6-2.



The screenshot shows a Windows Command Prompt window titled "Command Prompt". The command entered is "D:\>startserver". The output text is as follows:

```
WebSphere Application Server, Advanced Single Server Edition V4.0
Application Server Launcher
Copyright (C) IBM Corporation, 2001

The configuration file was defaulted to:
  d:\WebSphere\AppServer\config\server-cfg.xml
Using the single available node or the localhost node.
Using the single available server.
Initiating server launch.
Loaded domain "WebSphere Administrative Domain".
Selected node "a78rplhm".
Selected server "Default Server".
WSPL0065I: Initiated server launch with process id 2432.
Time mark: Thursday, August 23, 2001 5:34:52 PM PDT
Waiting for the server to be initialized.
Time mark: Thursday, August 23, 2001 5:34:58 PM PDT
Initialized server.
Waiting for applications to be started.
Time mark: Thursday, August 23, 2001 5:35:25 PM PDT
Started applications.
WSPL0057I: The server Default Server is open for e-business.
Please review the server log files for additional information.
Standard output: d:\WebSphere\AppServer\logs\default_server_stdout.log
Standard error: d:\WebSphere\AppServer\logs\default_server_stderr.log
```

Figure 6-2 Output when starting AEs

During the start up process the server will often pause on the line:

Waiting for applications to be started.

The server is completely started when you see the following lines displayed:

```
WSPL0057I: The server Default Server is open for e-business.
Please review the server log files for additional information.
Standard output: D:\WebSphere\AppServer\logs\default_server_stdout.log
Standard error: D:\WebSphere\AppServer\logs\default_server_stderr.log
```

You can also watch the progress of starting AEs by opening a Windows Task Manager and sorting the processes on memory usage (*Mem Usage* column) as shown in Figure 6-3. When the *java.exe* process, which is the application server process, has zero CPU usage, the server is started.

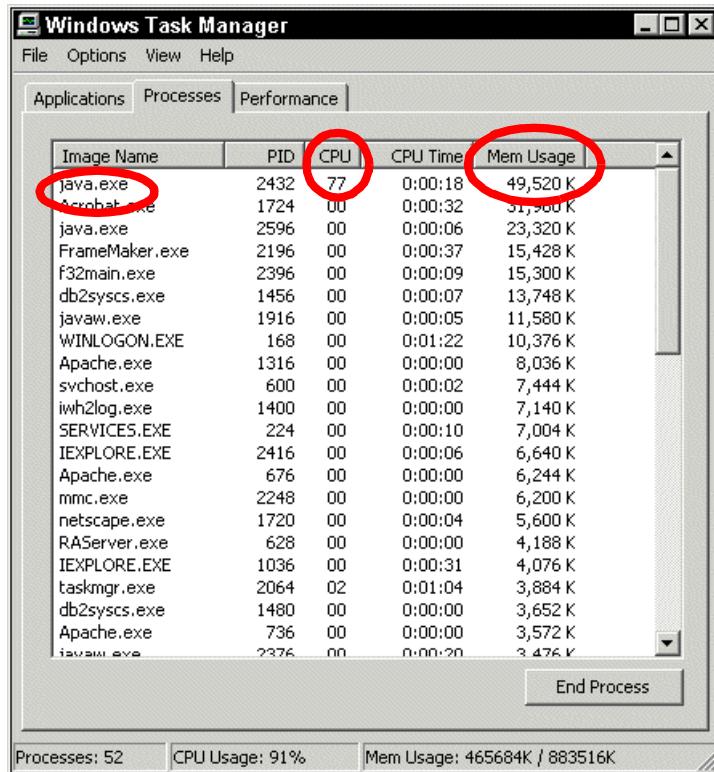


Figure 6-3 Task manager when starting AEs

Starting the Administrator's Console

Once you have started the Application Server you can start the Administrator's Console by either

- ▶ Select *Start -> Programs -> IBM WebSphere -> Application Server V4.0 AES -> Administrator's Console*

or

- ▶ Open a browser and enter <http://localhost:9090/admin/>

You are prompted for a user ID. Enter any user ID and click *Submit*. After you are logged in, your browser displays the Welcome panel (Figure 6-4).

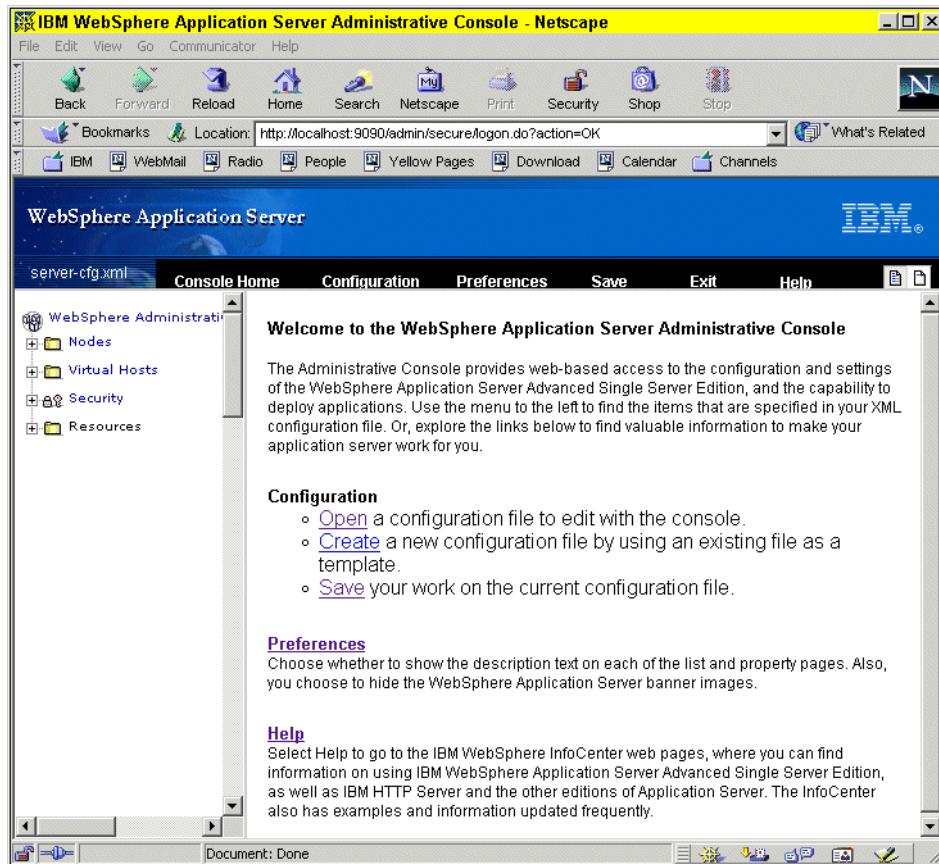


Figure 6-4 WebSphere AEs Administrator's Console welcome screen

Creating the JDBC driver and data source

Both of the enterprise applications we are installing use a JDBC data source. Therefore, the first thing we must do before installing our enterprise applications is to create the JDBC data source used by the enterprise applications. This is in fact the same data source we defined for the Application Developer internal server (see “Adding the JDBC data source” on page 84) but now in WebSphere AEs.

To create the datasource, first configure the JDBC driver:

- ▶ In the Administrator’s Console on the left menu panel, expand *Resources* and *JDBC Drivers* and click on *DB2JdbcDriver*.

- ▶ Specify the *Server Class Path* for the JDBC Driver as shown in Figure 6-5. The *Server Class Path* is the location of the db2java.zip file. Click *OK*.

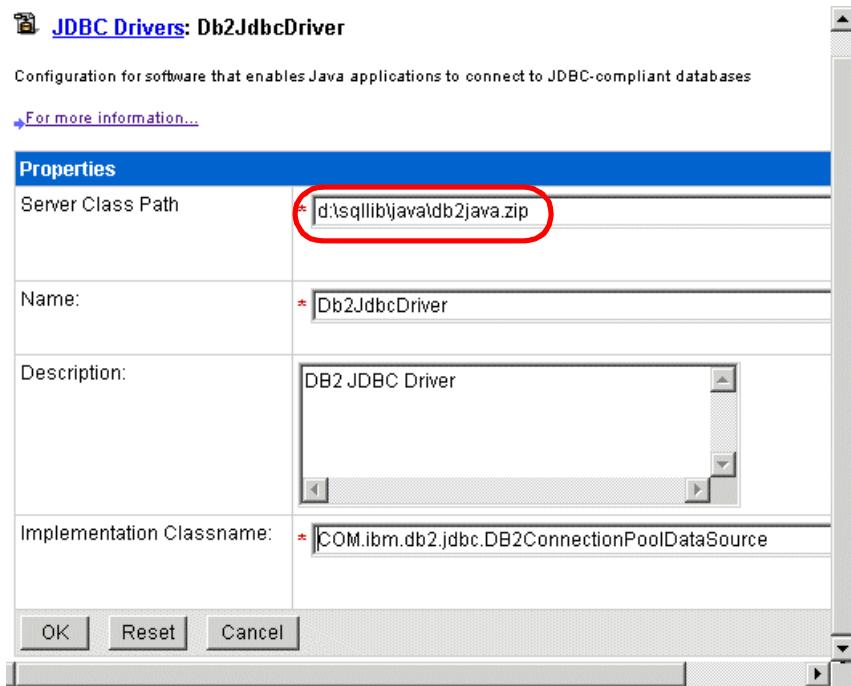


Figure 6-5 DB2 JDBC driver specification in AEs

Next configure the data source.

- ▶ Expand *Db2JdbcDriver* on the left panel and click on *Data Sources*. Click on *New* on the right panel. Complete the form with these values:

<i>Name</i>	ITSO DataSource
<i>JNDI Name</i>	jdbc/ITSOWSAD
<i>Description</i>	ITSO DataSource
<i>Database Name</i>	ITSOWSAD
<i>Default User ID</i>	db2admin (or empty)
<i>Default Password</i>	db2admin (or empty)

Specify the user ID and password that were used to install DB2.

- ▶ Click *OK*, and the list of data sources is now displayed (Figure 6-6).

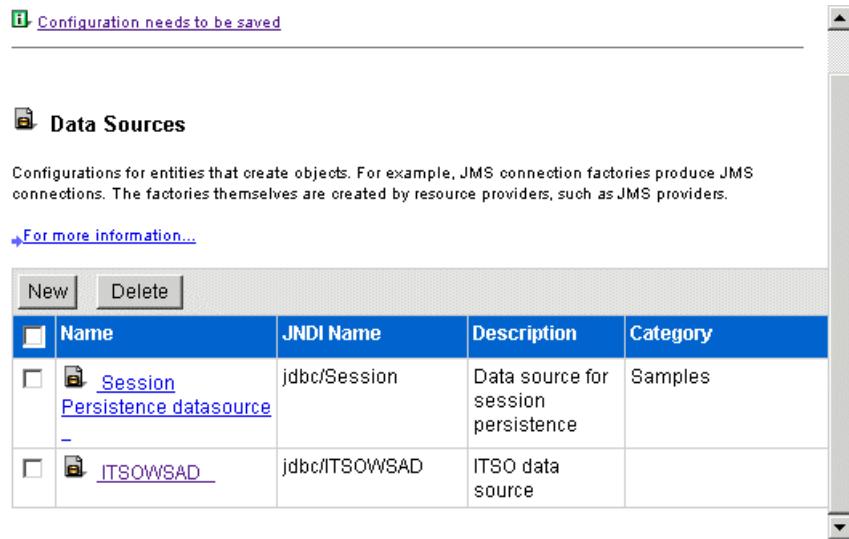


Figure 6-6 Data source for ITSOWSAD database

Installing the Almaden Autos enterprise application

Now we are ready to install the first enterprise application, Almaden Autos. We install the EAR file, `ITSOAlma.ear`, we exported in “Exporting the EAR files” on page 171.

To install the Almaden Autos enterprise application:

- ▶ Expand *Nodes* and *YourNodeName* on the left panel of the Administrator's Console.
- ▶ Click on *Enterprise Applications* on the left panel and on the right panel click on *Install*. The Application Installation wizard starts.
- ▶ Under *Specify the Application or Module located on this machine to upload and install* click on *Browse* (next to *Path*) and locate the path were you exported `ITSOAlma.ear` and click *OK* (Figure 6-7).
- ▶ Click on *Next* to continue the wizard. (Make sure not to choose the *Next* button at the bottom of the window.)

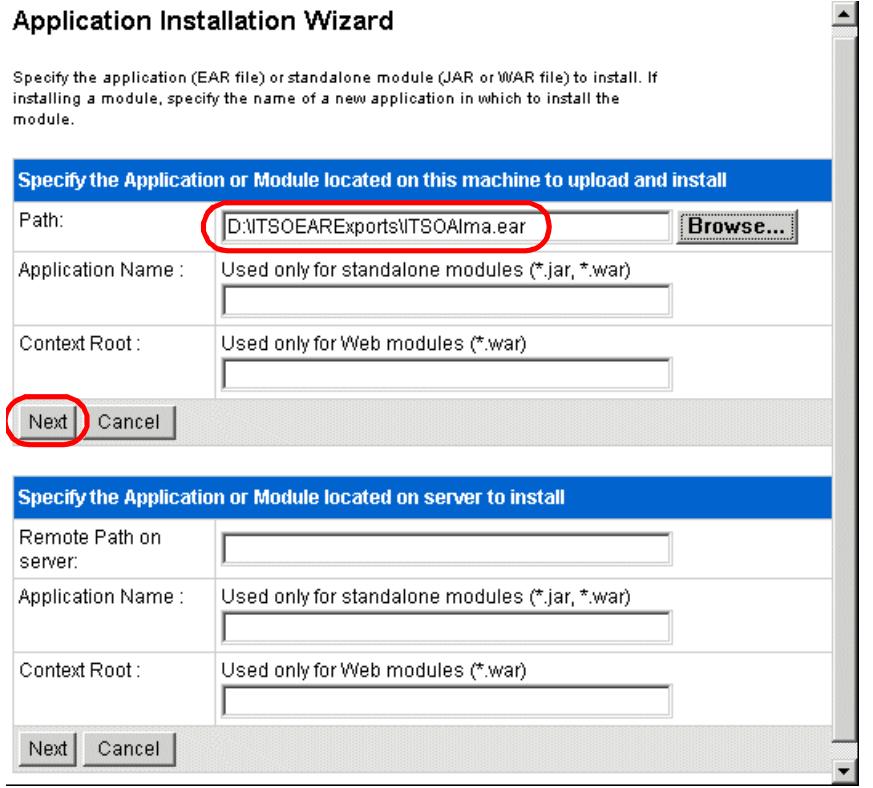


Figure 6-7 WebSphere AEs application installation wizard: EAR file

- On the next panel of the wizard, leave the *Virtual Host Name* as *default_host* and the *Precompile JSPs* option as Yes and click *Next* (Figure 6-8).

Application Installation Wizard

Each Web module in your application must be mapped to a virtual host and may have JSPs precompiled. For each Web module below, enter a virtual host name and select precompiled JSP option.

Specifying Virtual Host names and Precompiled JSP option for Web Modules		
Web Module Name	Virtual Host Name	Precompile JSPs
ITSOAlmaWeb	default_host	Yes

Figure 6-8 WebSphere AEs application installation wizard: virtual host

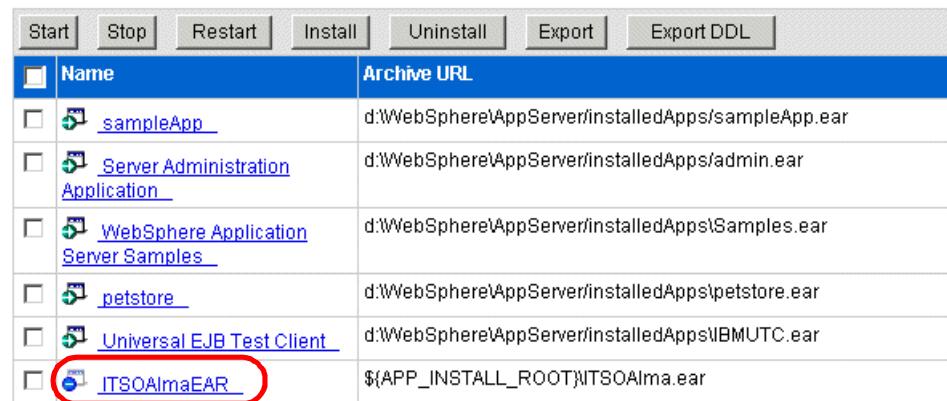
- On the next panel (*Confirm the following*) click on *Finish* to deploy the application.

- After deployment, the list of enterprise applications should contain the ITSOAlmaEAR enterprise application as shown in Figure 6-9. Note that the application is not active (it has not been started).

Enterprise Applications

The J2EE applications (EAR files) installed on the application server

[For more information...](#)



<input type="checkbox"/>	Name	Archive URL
<input type="checkbox"/>	sampleApp	d:\WebSphere\AppServer\installedApps\sampleApp.ear
<input type="checkbox"/>	Server Administration Application	d:\WebSphere\AppServer\installedApps\admin.ear
<input type="checkbox"/>	WebSphere Application Server Samples	d:\WebSphere\AppServer\installedApps\Samples.ear
<input type="checkbox"/>	petstore	d:\WebSphere\AppServer\installedApps\petstore.ear
<input type="checkbox"/>	Universal EJB Test Client	d:\WebSphere\AppServer\installedApps\IBMUTC.ear
<input type="checkbox"/>	ITSOAlmaEAR	\$(APP_INSTALL_ROOT)\ITSOAlma.ear

Figure 6-9 WebSphere AEs list of enterprise applications

Installing the Mighty Motors enterprise application

We now install the Mighty Motors enterprise application. Mighty Motors contains EJBs so we must install the EAR file as we did for Almaden Autos and also configure the EJB references, data source, and mappings. Most of this was defined when we built the enterprise application, so we are mainly verifying the information in this process.

To install the Mighty Motors enterprise application (ITSOMighty.ear) we exported in “Exporting the EAR files” on page 171:

- Again click on *Install* to start the Application Installation wizard.
- Click on *Browse* and specify the path where you exported ITSOMighty.ear and click *Next*.
- On the *Binding Enterprise Beans to JNDI Names* panel, verify the JNDI bindings (Figure 6-10).

These JNDI names were defined in “Defining the EJB references” on page 165. Leave the bindings as they are and click on *Next*.

Application Installation Wizard

Each enterprise bean in your application or module must be bound to a JNDI name. For each enterprise bean below, enter a Name.

Binding Enterprise Beans to JNDI Names	
Enterprise Bean JNDI Names	JNDI Name
ITSO Mighty EJB.jar:itso.wsad.mighty.ejb.MminventoryBean	itso/wsad/Mminventory
ITSO Mighty EJB.jar:itso.wsad.mighty.ejb.MmpartsBean	itso/wsad/Mmparts
ITSO Mighty EJB.jar:itso.wsad.mighty.ejb.PartsFacadeBean	itso/wsad/PartsFacade

[Back](#) [Next](#) [Cancel](#)

Figure 6-10 AEs application installation: binding enterprise beans to JNDI names

- The next page of the wizard is the *Mapping EJB references to Enterprise beans* page (Figure 6-11).

These references were defined and explained in “Defining the EJB references” on page 165. Leave the values as they are and click *Next*.

Application Installation Wizard

Each EJB Reference in your application or module must be mapped to an enterprise bean. For each EJB reference below, enter a name of an enterprise bean.

Mapping EJB References to Enterprise Beans		
Archives	References	Enterprise Bean JNDI Name
ITSO Mighty EJB.jar:itso.wsad.mighty.ejb.MminventoryBean	ejb/Mmparts	itso/wsad/Mmparts
ITSO Mighty EJB.jar:itso.wsad.mighty.ejb.MmpartsBean	ejb/Mminventory	itso/wsad/Mminventory
ITSO Mighty EJB.jar:itso.wsad.mighty.ejb.PartsFacadeBean	ejb/Mmparts	itso/wsad/Mmparts

[Back](#) [Next](#) [Cancel](#)

Figure 6-11 AEs application installation: mapping EJB references to JNDI names

- In the next panel of the wizard we can specify the *Database* and *Data Source* settings. Set the *Schema Name(qualifier)* field to *ITSO* and leave the other values as they are and click *Next* (Figure 6-12).

The CMP data sources are left blank because we use the default data source defined for the EJB container (*jdbc/ITSOWSAD*).

Application Installation Wizard

Each CMP data source reference in your application must be bound to a JNDI name. For each resource reference below, enter Name, User ID, and Password.

Database settings		
Database Type	DB2UDBWIN_V72	
Schema Name(Qualifier)	ITSO	
Mapping EJB Jar Default Data Source References to JNDI Names		
EJB Jar Name	Default Datasource JNDI Name	User ID
ITSO Mighty EJB.jar	jdbc/ITSOADSAD	db2admin
Mapping CMP Data Source References to JNDI Names		
Enterprise Bean	JNDI Name	User ID
ITSO Mighty EJB.jar:Mminventory		
ITSO Mighty EJB.jar:Mmparts		

[Back](#) [Next](#) [Cancel](#)

Figure 6-12 AEs application installation: mapping EJB data source

- On the next panel of the wizard, uncheck the *Re-Deploy option* (Figure 6-13). This option generates the deployable EJB code, but we already generated the code in Application Developer (see “Generating the deployed code” on page 151).

Application Installation Wizard

The following EJB JARs have already been deployed. However, it is recommended that you let WebSphere re-deploy the JAR to avoid potential problems. If you wish to have WebSphere re-deploy the EJB JARs, leave the checkboxes selected.

EJB Deploy	
Re-Deploy option	EJB JAR
<input checked="" type="checkbox"/>	ITSO Mighty EJB.jar

[Back](#) [Next](#) [Cancel](#)

Figure 6-13 AEs application installation: EJB deploy option

- On the next panel (*Confirm the following*), click on *Finish*.

The list of enterprise applications should now contain both the Almaden Autos and the Mighty Motors enterprise applications.

Saving the configuration

After installing enterprise applications it is a good moment to save the configuration:

- ▶ Click on the link *Configuration needs to be saved*, or click on *Save* in the black menu bar (Figure 6-14).

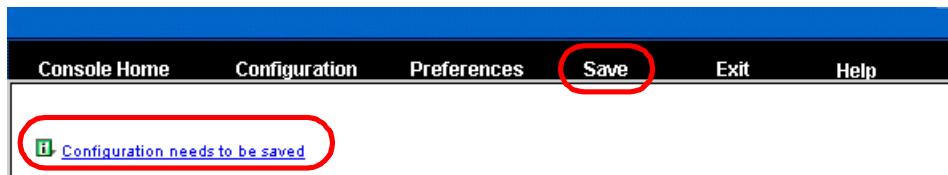


Figure 6-14 WebSphere AEs: save configuration

- ▶ In the *Save Configuration* panel click *OK* to save the configuration in `server-cfg.xml` or specify a file of your own preference to save the configuration (Figure 6-15).

Save Configuration

You made changes to the configuration file: `d:\WebSphere\AppServer\config\server-cfg.xml`. You can either:

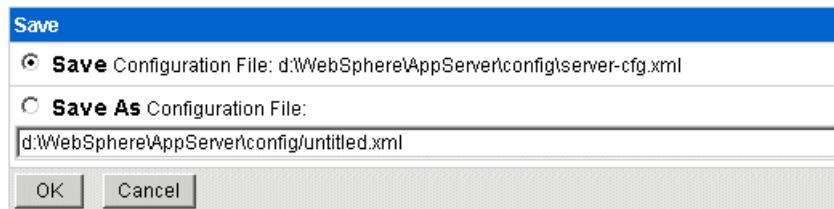


Figure 6-15 WebSphere AEs: save configuration file specification

Important: If you specify another file than the default `server-cfg.xml`, you have to specify the `-configFile YourConfigFile.xml` option with the `startserver` and `stopserver` commands.

After saving the configuration, we can log out of the current session by clicking on *Exit* in the black menu bar, and we can close the browser window.

Stopping the AEs server

Most of the time the server has to be stopped and started again after installing enterprise applications. Also if we use the command line install program, SEAppInstall, we have to stop AEs first.

To stop the Application Server open a command window and enter **stopserver**. The Console output when stopping AEs is shown Figure 6-16.

```
C:\>stopserver

WebSphere Application Server, Advanced Single Server Edition V4.0
WebSphere Application Server, Advanced Developer Edition V4.0
WebSphere Application Server, Advanced Edition V4.0
Runtime Utility Program
Copyright (C) IBM Corporation, 1997-2001

MSRU0025I: Loading configuration from file.
MSRU0028I: Using the specified configuration file:
d:\WebSphere\AppServer\config\server-cfg.xml
MSRU0029I: The diagnostic host name read as localhost.
MSRU0030I: The diagnostic port was read as 7000.
Stopping server.
The server was successfully stopped.

C:\>
```

Figure 6-16 Stopping the Application Server

Deploying an enterprise application with SEAppInstall

Enterprise applications can be installed using the Administrator's Console (as we did for the Almaden Autos and Mighty Motors application), or we can use a command line tool, SEAppInstall.

To illustrate the command line tool, we deploy the EJB test client included in Application Developer to WebSphere AEs. Installing the EJB test client enables us to test deployed EJBs in AEs using the same GUI as in Application Developer.

Note that we could have used SEAppInstall for our sample applications because we predefined all EJB bindings in Application Developer and we did not have to change any of the deployment options.

Deploying the EJB test client

In this section we deploy the EJB test client. This application can be very helpful to deploy enterprise applications that contain EJBs, such as the Mighty Motors enterprise application. It enables you to browse the JNDI namespace and test the deployed EJBs.

We used the EJB test client when developing the EJBs in “Executing the EJB test client” on page 153.

We will use the SEAppInstall command line tool provided with WebSphere AEs to deploy the EJB Test Client. This is a tool for installing an application into a server configuration file and preparing it to run within an application server. You can also use it to uninstall an application from a server, and to export an installed application containing configuration information.

SEAppInstall command

SEAppInstall has the following options:

```
-install  
-uninstall  
-export  
-list  
-extract DDL  
-validate
```

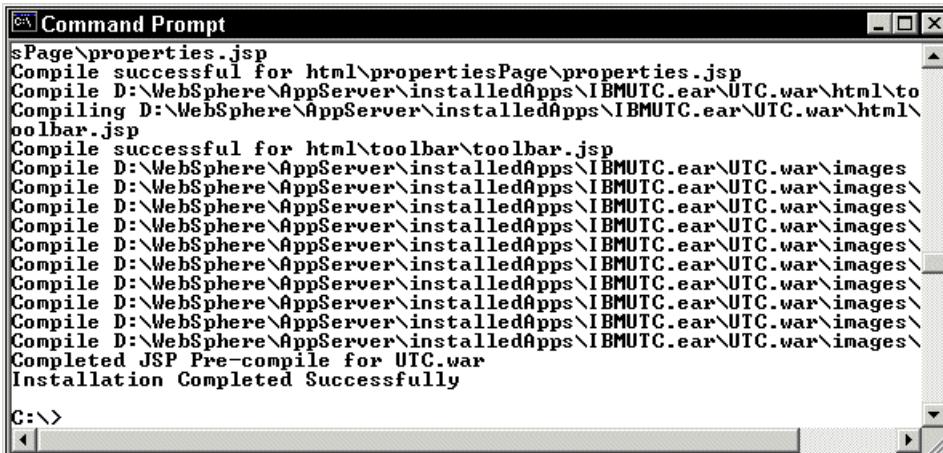
To view the full syntax of these options, run SEAppInstall without any parameters (SEAppInstall.bat is in the WAS_ROOT/BIN directory).

To deploy the EJB test client, complete these steps:

- ▶ Copy the entire directory named **IBMUTC.ear** which can be found under `WSAD_ROOT\plugins\com.ibm.etools.websphere.tools\IBMUTC` to the directory `WAS_ROOT\installableApps`. Note that in Version 4.0.2 you can also use the **IBMUTC.ear file** instead of the directory.
- ▶ Open a command window and enter:

```
SEAppInstall -install WAS_ROOT\installableApps\IBMUTC.ear  
              -expandDir WAS_ROOT\installedApps\IBMUTC.ear  
              -ejbdeploy false -interactive false
```

If you used a different configuration file you have to add the `-configFile` option to the command.
- ▶ Ensure that you get an *Installation Completed Successfully* message as shown in Figure 6-17. Ignore any messages indicating that you have not assigned security roles.



```
C:\ Command Prompt
sPage\properties.jsp
Compile successful for html\propertiesPage\properties.jsp
Compile D:\WebSphere\AppServer\installedApps\IBMUTC.ear\UTC.war\html\to
Compiling D:\WebSphere\AppServer\installedApps\IBMUTC.ear\UTC.war\html\
toolbar.jsp
Compile successful for html\toolbar\toolbar.jsp
Compile D:\WebSphere\AppServer\installedApps\IBMUTC.ear\UTC.war\images\
Completed JSP Pre-compile for UTC.war
Installation Completed Successfully

C:\>
```

Figure 6-17 Installation of EJB test client

Enabling the EJB test client

By default an enterprise application cannot see other enterprise applications. However, the EJB test client must have access to other applications to be able to invoke their EJBs.

We can change the visibility of enterprise applications in WebSphere AEs:

- ▶ Start AEs (startserver)—as described in “Starting the AEs server” on page 172. Use the -configFile option if you are using a different configuration file.
- ▶ Start the Administrator’s Console (see “Starting the Administrator’s Console” on page 173).
- ▶ In the Administrator’s console, expand *Nodes*, *YourNodeName*, *Application Servers*. Click on *Default Server*.
- ▶ In the right panel set the *Module Visibility* to *Compatibility* as shown in Figure 6-18. Click *OK*.
- ▶ Save the configuration and exit the Administrator’s console.
- ▶ Stop AEs (stopserver).

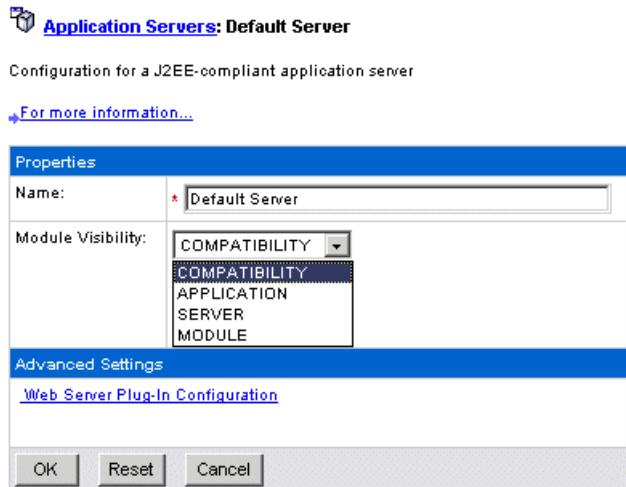


Figure 6-18 Changing the visibility mode to compatibility

Verifying the enterprise applications

Start the application server and make sure all the newly installed applications are running:

- ▶ Start AEs and the Administrator's Console.
- ▶ Check to see if all the applications we installed are running. Expand *nodes*, *YourNodeName*, and click on *Enterprise Applications*.
- ▶ You should see a green arrow indicating that the applications are running for *ITSOAlmaEAR*, *ITSOMightyEAR* and *Universal EJB Test Client* as shown in Figure 6-19.
- ▶ If one of the applications we have installed is not started, mark the checkbox next to it and click *Start*.

Probably this will not start the application because some specification must be wrong—otherwise the application would have started already. Check the log files in *WAS_ROOT\logs* for error messages.

Enterprise Applications

The J2EE applications (EAR files) installed on the application server

[For more information...](#)

<input type="checkbox"/>	Name	Archive URL
<input type="checkbox"/>	sampleApp	d:\WebSphere\AppServer\installedApps\sampleApp.ear
<input type="checkbox"/>	Server Administration Application	d:\WebSphere\AppServer\installedApps\admin.ear
<input type="checkbox"/>	WebSphere Application Server Samples	d:\WebSphere\AppServer\installedApps\Samples.ear
<input type="checkbox"/>	petstore	d:\WebSphere\AppServer\installedApps\petstore.ear
<input type="checkbox"/>	ITSOAlmaEAR	d:\WebSphere\AppServer\installedApps\ITSOAlma.ear
<input type="checkbox"/>	ITSO Mighty EAR	\$(APP_INSTALL_ROOT)\ITSO Mighty.ear
<input type="checkbox"/>	Universal EJB Test Client	\$(APP_INSTALL_ROOT)\IBMUTC.ear

Figure 6-19 All installed applications running

Regenerating the plug-in

WebSphere AE and AEs use a plug-in to get requests routed from the IBM HTTP Server to the application server.

What is a Web server plug-in?

When you install WebSphere Application Server AE or AEs a plug-in into the Web server (in our case the IBM HTTP Server) is also installed.

This plug-in enables the Web server to talk with the WebSphere Application Server. In order to enable the HTTP transport between the plug-in and the application server, every WebSphere Application Server Web container has a built-in HTTP server—by default, the built-in HTTP server uses port 9080.

Figure 6-20 shows the setup for the HTTP server and the application server.

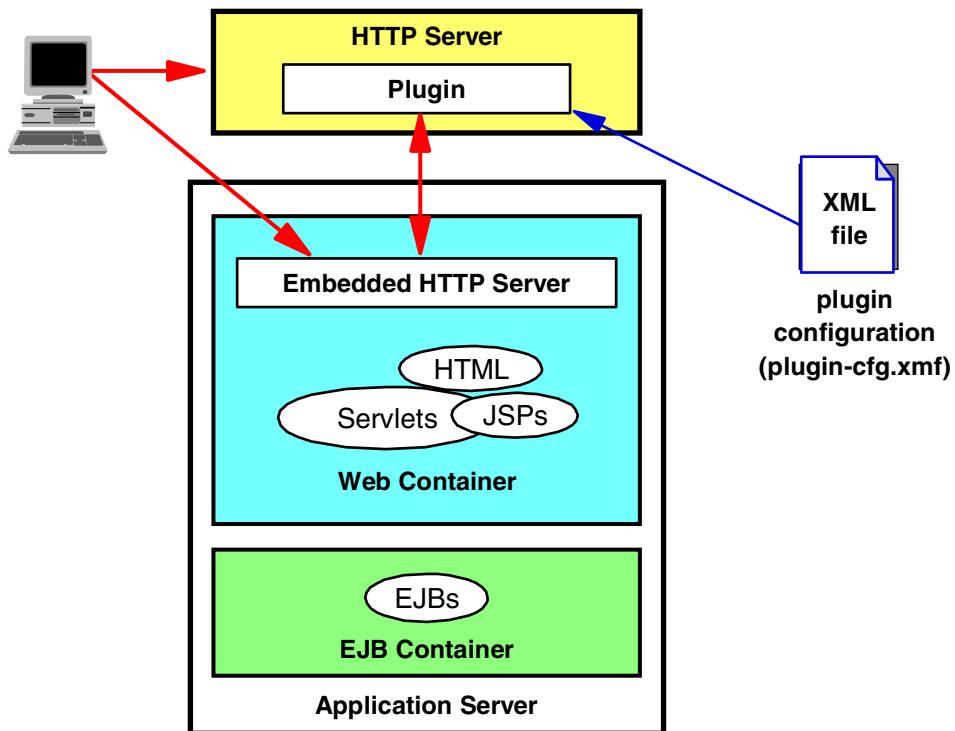


Figure 6-20 Web server plug-in

A plug-in has a plug-in configuration which tells the HTTP server which requests it should forward to the application server. For example, the Web server forwards the requests it gets for servlets and JSPs.

The WebSphere plug-in configuration is saved in the `plugin-cfg.xml` file, which you can find in the `\WAS_ROOT\config` directory. When you install a new Web application you have to update the plug-in configuration so that the Web server becomes aware of the new servlet or JSP paths which come with your newly installed Web Application. If you do not regenerate the plug-in and try to invoke a new servlet, it results in a *404 File Not Found* Error.

Important: In WebSphere Application Server version 3.5.x, the plug-in communicated with the application server either through the OSE transport or through the Servlet Redirector, based on the IOP protocol. The OSE protocol has been deprecated in 4.0 and the Servlet Redirector has been completely removed. HTTP is now the recommended option and it provides better performance.

Manually triggering the plug-in regeneration

We have installed our two sample applications and the EJB Test Client so we have to regenerate the plug-in for the IBM HTTP Server:

- ▶ Start AEs and the Administrator's Console.
- ▶ Expand *Nodes*, *YourNodeName*, click on *Application Servers*.
- ▶ In the right panel, click on the *Default Server* link.
- ▶ At the bottom of the *Application Servers: Default Server* page, click on the link *Web Server Plug-in Configuration*.
- ▶ Click the *Generate* button.
- ▶ To make sure that the IBM HTTP Server picks up the new plug-in configuration immediately for testing the new applications, restart the HTTP Server. (You can restart the HTTP Server through the Windows *Services* panel or through the *Programs* menu.)

After you have generated the plug-in configuration, you can open the *plugin-cfg.xml* file located in the *WAS_ROOT/config* directory. It should contain the context roots of the installed applications (Figure 6-21).

```
<?xml version="1.0"?>
<Config>
    .....
    <UriGroup Name="default_host_URIs">
        ...
        <Uri Name="/UTC"/>
        <Uri Name="/ITS0Mighty"/>
        <Uri Name="/ITS0Alma"/>
        ...
    </UriGroup>
    .....
</config>
```

Figure 6-21 Plugin-cfg.xml

Testing the deployed applications in AEs

In this section we test the Almaden Autos and Mighty Motors enterprise applications we deployed. The Almaden Autos enterprise application contains a Web module that can be used for testing the application, whereas the Mighty Motors enterprise application just contains an EJB module, therefore we will use the EJB test client to test it.

We can test our enterprise applications with an external Web server such as IBM HTTP Server, or we can use the embedded HTTP server in WebSphere AEs.

Important: Before you can test the enterprise applications you must make sure that you have created the ITSOWSAD database on the machine where WebSphere AEs is running (see “Creating the ITSOWSAD sample database” on page 61).

Testing using the embedded HTTP server

When we test the installed applications with the embedded HTTP server we send the requests directly to AEs by specifying the port 9080.

Testing Almaden Autos

To test the Almaden Autos enterprise application:

- ▶ Open a browser and enter:

<code>http://localhost:9080/ITSOAlma/PartsInputForm.html</code>	own machine
<code>http://host-name:9080/ITSOAlma/PartsInputForm.html</code>	other machine
- ▶ The first Web page is the *Parts Inventory Inquiry* page. Enter a part number such as M100000001 and click *Submit*.
- ▶ You should get the *Inventory Inquiry Results* for the part number.

Testing Mighty Motors

To test the Mighty Motors enterprise application and more specifically the EJB module which it contains, we use the EJB Test Client.

- ▶ Open a browser and enter `http://host-name:9080/UTC`. The browser displays the EJB test client welcome panel (Figure 5-24 on page 154).
- ▶ Use the JNDI explorer to access the EJBs as described in detail in “Executing the EJB test client” on page 153.

Testing using the IBM HTTP Server (AEs)

The embedded Web Server is usually only used during the development stage. In normal production environments, an external Web server such as IBM HTTP Server is used. The only difference between testing against the embedded HTTP server and an external HTTP server, is the port specification in the URL. The external Web server listens on the default port 80 so we can omit it in the URL:

```
http://host-name/ITSOAlma/PartsInputForm.html  
http://host-name/UTC
```

Deployment to WebSphere AE

In this section we discuss the deployment of the first two stages of the sample application, Almaden Autos and Mighty Motors, to WebSphere Application Server Advanced Edition 4.0. We will deploy the same EAR files we used in “Deployment to WebSphere AEs” on page 170.

Starting the Admin Server

WebSphere Advanced does have a separate Administration Server, and it must be running in order to launch the Java stand-alone Administrator’s Console.

The Admin Server can be started from the Programs menu (*Start -> Program Files -> IBM WebSphere -> Application Server V4.0 -> Start Admin Server*) or from the Windows Services panel (start the *IBM WS AdminServer 4.0* service).

You can watch the progress using the Windows Task Manager, which is the same as what we described in “Starting the AEs server” on page 172.

Starting the Administrator’s Console

Start the Administrator’s Console using *Start -> Program Files -> IBM WebSphere -> Application Server V4.0 -> Administrator’s console* (Figure 6-22).

If you want to launch the Administrator’s Console from a remote machine open a command window and enter **adminClient hostname**, where *hostname* is the host where the Admin Server is running (either the full hostname or the IP address).

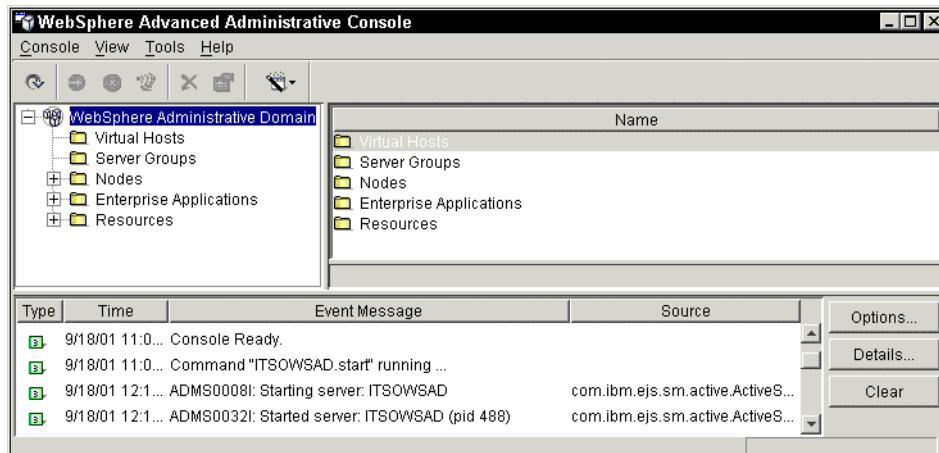


Figure 6-22 WebSphere AE Administrator’s Console

Creating the JDBC driver and data source

We have to verify the JDBC driver and create the data source for the enterprise applications in WebSphere AE:

- ▶ In the Administrator's Console, expand *Resources*, *JDBC Drivers*.
- ▶ Click on *Sample DB Driver*.
- ▶ On the right panel, *General* tab make sure the *Implementation class* is `COM.ibm.db2.jdbc.DB2ConnectionPoolDataSource`.
- ▶ On the *Nodes* tab the *Classpath* variable should point to the `db2java.zip` file where you installed DB2.
- ▶ Expand *Sample DB Driver*.
- ▶ Right-click on *Data Sources* and select *New*.
- ▶ Complete the *Data Source properties* dialog as shown in Figure 6-23. Use the user ID and password that were used to install DB2.
- ▶ Click *OK*.

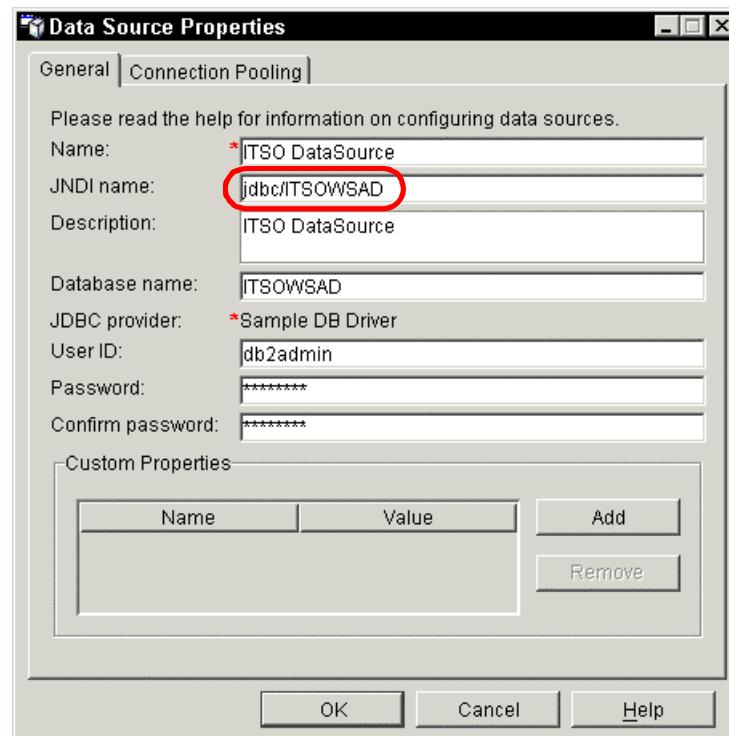


Figure 6-23 Adding a data source in WebSphere AE

Creating an application server

In WebSphere Application Server AE, we can create multiple application servers. This was not possible in WebSphere AEs. To create a new application server to run the sample applications:

- ▶ In the Administrator's Console expand *Nodes*, *YourNodeName*.
- ▶ Right-click on *Application Servers* and click *New*.
- ▶ In the *General* tab of the dialog specify ITSOWSAD for the *Application Server name* property (Figure 6-24).
- ▶ Switch to the *File* tab and specify dedicated file names for the *Standard output* and *Standard error* properties (Figure 6-25).
- ▶ Click *OK* to create the application server.

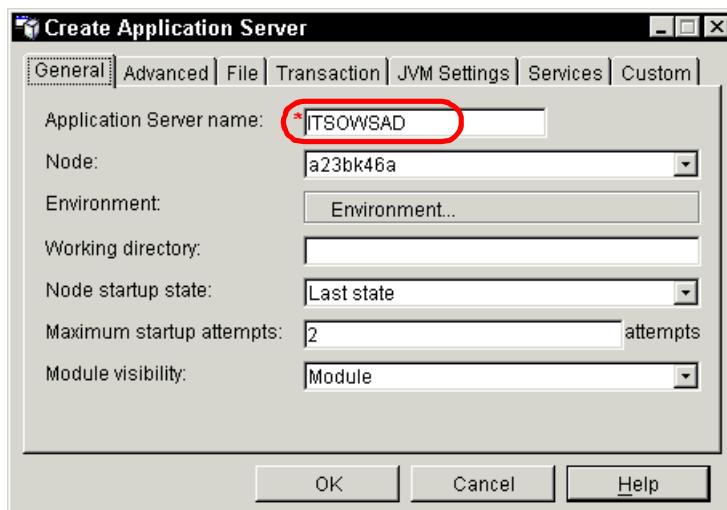


Figure 6-24 Create an application server in AE

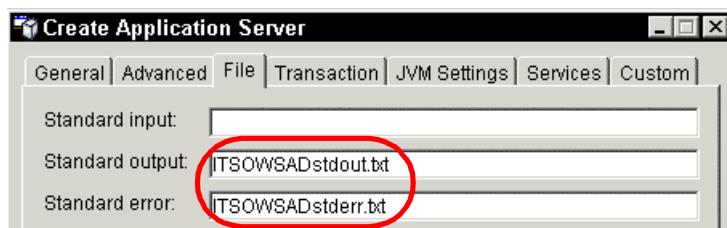


Figure 6-25 Specifying dedicated output file names

Generate the plug-in

When we deployed our sample applications to WebSphere AEs, we manually generated the plug-in for the Web server after having installed the applications (see “Regenerating the plug-in” on page 186).

In WebSphere AE we can specify a property to automatically generate the plug-in when making modifications to the configuration:

- ▶ In the Administrator’s Console under *Application Servers*, select *ITSOWSAD*, the application server we just created.
- ▶ In the right panel click on *Custom* tab. You see the *Automatic Generation of Plug-in* property, which has the *Enabled* property set to *false*.
- ▶ Select the *Service* and click *Edit*.
- ▶ Mark the *Enabled* checkbox and click *OK* (Figure 6-26).

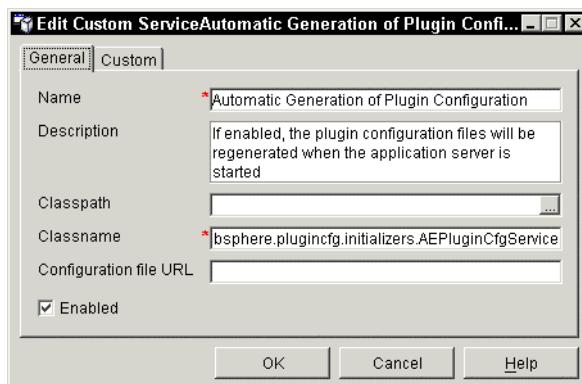


Figure 6-26 Enable plug-in generation

Important: In a production environment the *Automatic Generation of Plug-in* property should be turned off because it decreases performance.

Module visibility

Next we have to change the *Module visibility* property to *Compatibility*, so that the EJB test client will be able to see the other applications:

- ▶ Select *Application Servers*, *ITSOWSAD*, on the *General* tab of Figure 6-24, change the *Module visibility* property to *Compatibility*.
- ▶ Click *Apply* to enable all the changes we made.

Installing the Almaden Autos enterprise application

Now we are ready to install the Almaden Autos enterprise application:

- ▶ In the Administrator's Console, in the toolbar click on the far right hand icon, and select *Install enterprise application* (Figure 6-27).



Figure 6-27 *Installing an enterprise application*

- ▶ In the *Specifying the Application or Module* panel of the wizard, specify the path for `ITSOAlma.ear` and set the *Application name* property (Figure 6-28).

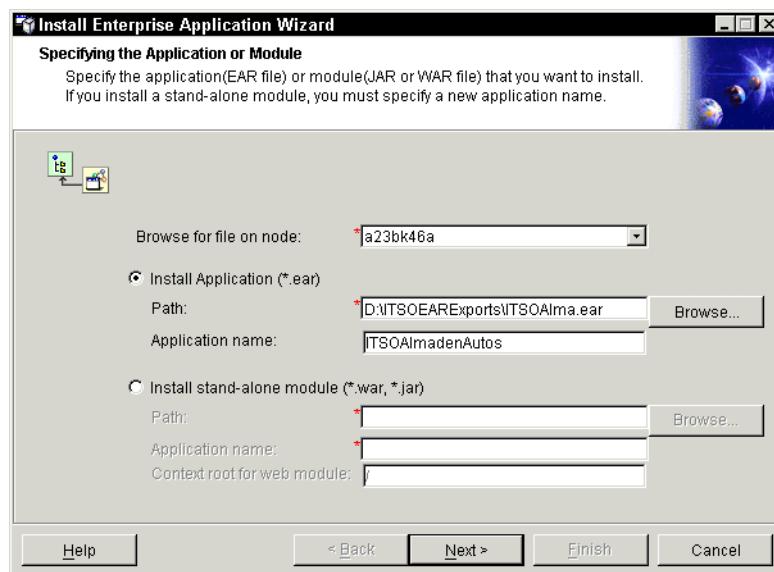


Figure 6-28 *AE application installation: specifying the EAR file*

The application name is the name that will appear in the list of enterprise applications in the Administrator's Console. Do not confuse this with the context root (`/ITSOAlma`) which is defined in the deployment descriptor.

- ▶ Click *Next* until you get the *Selecting Virtual Host for Web Modules* panel, and select *default_host* as the *virtual host*.
- ▶ Click *Next* and select *ITSOWSAD* as the *Application Server* on the *Selecting Application Servers* panel (Figure 6-29).

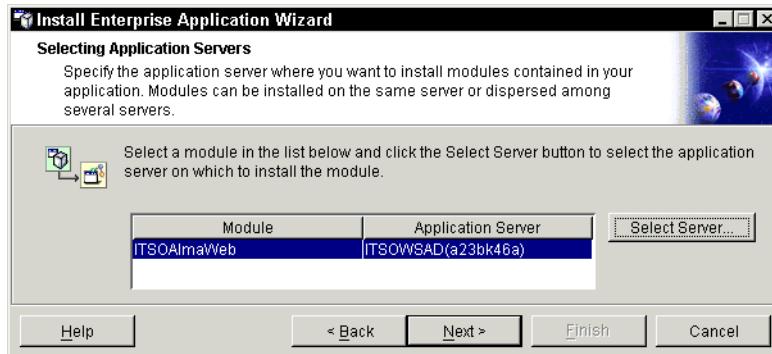


Figure 6-29 AE application installation: select application server

- ▶ Click *Next* and then *Finish* in the *Completing the Application Installation Wizard* panel.

Installing the Mighty Motors enterprise application

The process for installing the Mighty Motors enterprise application is the same, with some extra panels for setting the EJB JNDI names and data source:

- ▶ Start Enterprise Application Wizard as we did in “Installing the Almaden Autos enterprise application” on page 176.
- ▶ Specify the path for *ITSOMighty.ear* and set the *Application name* to *ITSOMightyMotors*.
- ▶ Click *Next* until you get the *Binding Enterprise Beans to JNDI Names* panel (Figure 6-30). These JNDI names were defined in “Defining the EJB references” on page 165. Click *Next*.

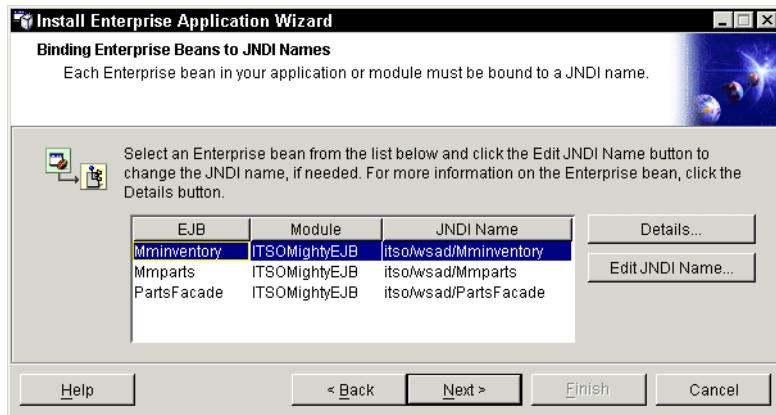


Figure 6-30 AE application installation: binding EJBs to JNDI names

- ▶ The *Mapping EJB References to Enterprise Beans* panel maps the EJB references we defined to the JNDI names of the Enterprise Beans. These references were defined and explained in “Defining the EJB references” on page 165. Leave the values as they were extracted from the EJB deployment descriptor inside the EAR file.
- ▶ Click *Next* until you get the *Specifying the Default Datasource for EJB Modules* panel.

The default data source JNDI name (`jdbc/ITSOWSAD`) matches the JNDI name we gave to the data source that we defined in “Creating the JDBC driver and data source” on page 191. Click *Next*.

- ▶ On the *Specifying DataSources for individual CMP Beans* panel no data source is specified for the individual CMPs because they use the default data source specified for the EJB module.
- ▶ Click *Next* until you get the *Selecting the Application Servers* panel. Click on the *ITSOMightyEJB* module and select the *ITSOWSAD* application server. Click *Next*.
- ▶ In the *Completing the Application Installation Wizard* panel click *Finish*.
- ▶ Select *No* when prompted to redeploy the code.

You should now see the two enterprise applications (*ITSOAlmaEAR* and *ITSOMightyEAR*) in the list of enterprise applications.

Deploying the EJB test client

To test the deployed EJBs in WebSphere AE, we deploy the EJB test client:

- ▶ Copy the entire directory named IBMUTC.ear, which can be found under WSAD_ROOT\plugins\com.ibm.etools.websphere.tools\IBMUTC to the directory WAS_ROOT\installableApps where WSAD_ROOT is the WSAD installation directory (such as d:\WSAD) and WAS_ROOT is the WebSphere AE installation directory (such as d:\WebSphere\AppServer). Note that in Version 4.0.2 you can also use the **IBMUTC.ear file** instead of the directory.
- ▶ In the Administrator's Console, start the *Enterprise Application Installation* wizard.
- ▶ On the *Specifying the Application or Module* page of the wizard, specify the path where you copied the IBMUTC.ear directory set the *Application name* property to *IBMUTC* (Figure 6-31). Click *Next*.

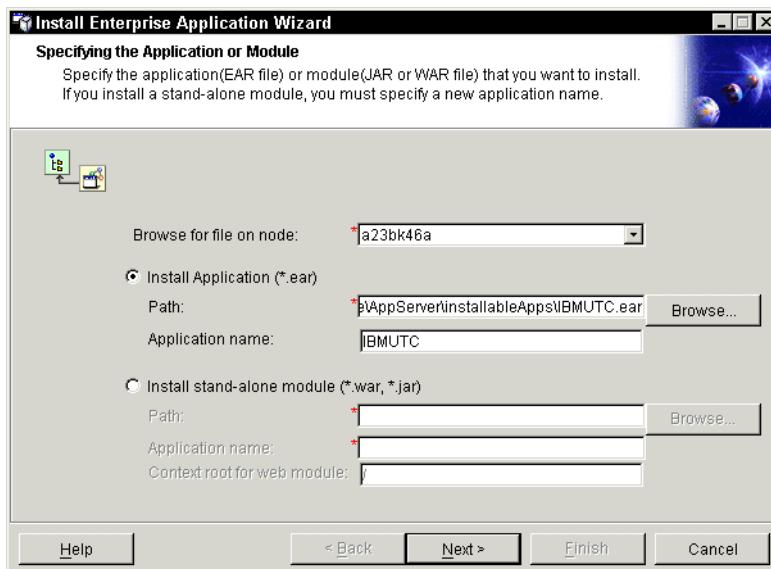


Figure 6-31 AE application installation: EJB test client

- ▶ Click *Next* until you get the *Selecting Virtual Hosts for Web Modules* page. Make sure *default_host* is selected as the *Virtual Host* for the *UTC* Web Module. Click *Next*.
- ▶ On *Selecting the Application Servers* page, click on the *UTC* module and select the *ITSOWSAD* application server.
- ▶ Click *Next* and in the *Completing the Application Installation Wizard* click *Finish*.

Starting the WebSphere AE application server

To start the newly defined ITSOWSAD application server, in which we installed the three enterprise applications, use the Administrator's Console:

- ▶ Expand *Nodes*, *YourNodeName*, *Application Servers* and right-click on *ITSOWSAD* and click *Start* as shown in Figure 6-32.

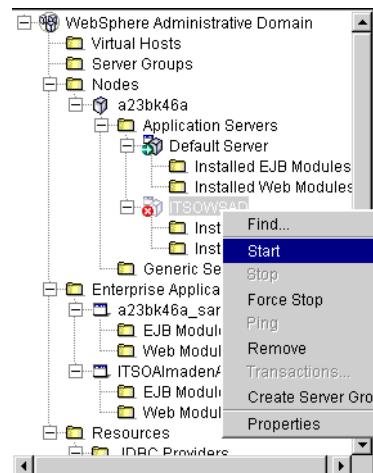


Figure 6-32 AE: starting the application server

- ▶ When you start the application server, the Web server plug-in is regenerated because we enabled the *Automatic Generation of Plug-in* property. To make sure the Web server picks up the new plug-in configuration immediately, you can stop and restart it.

Testing the deployed applications in WebSphere AE

In this section we test if the deployment of the Almaden Autos and Mighty Motors enterprise applications was successful. The Almaden Autos enterprise application contains a Web module that can be used for testing the application, whereas the ITSOMighty enterprise application just contains an EJB module, therefore we use the EJB test client. We can test our enterprise applications with an external Web server such as IBM HTTP Server or we can use the embedded HTTP server in WebSphere AE.

Important: Before you can test the enterprise applications you must create the *ITSOWSAD* database on the WebSphere machine (see “Creating the ITSOWSAD sample database” on page 61).

Testing using the embedded HTTP server

When we test with the embedded HTTP server we send our request directly to the Embedded HTTP Server by specifying the port number it listens on.

Embedded HTTP server within WebSphere AE

With WebSphere AE things are a bit more complicated than with AEs. We can have multiple application servers, each having an embedded HTTP server running within a Web container on a different port. Each time you create an application server WebSphere automatically increments the port number for the associated embedded HTTP server. To see the port number:

- ▶ In the Administrator's Console expand the *Application Servers*.
- ▶ Select *ITSOWSAD* and in the right panel click on the *Services* tab.
- ▶ Select *Web Container Service* and click *Edit properties*.
- ▶ The *Web Container Service* panel pops up. Select the *Transport* tab (Figure 6-33). Under *HTTP Transports* you will see the port used by the embedded HTTP server for the selected application server.

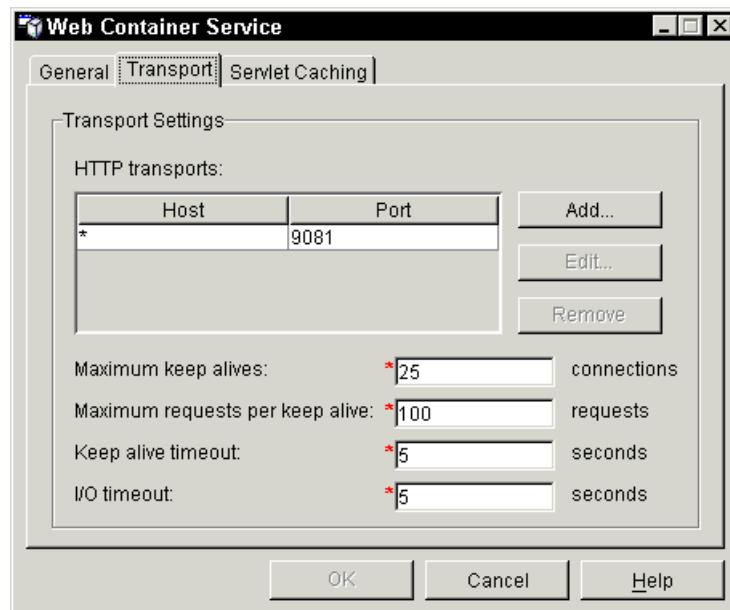


Figure 6-33 HTTP Transport settings

If we want to use the embedded HTTP server to test the enterprise applications, we have to use that port in the URL.

We also have to define a virtual host alias for the `default_host`:

- ▶ In the Administrator's Console select *Virtual Hosts*.
- ▶ Click on `default_host` and click on the *Add* button next to the list of aliases, and add `localhost:9081` as an alias. You might have another port number than 9081 depending on the number of application servers defined on your node. Use the port number defined in Figure 6-33.
- ▶ You have to restart the Admin Server before you can start testing the applications.

To test the Almaden Autos enterprise application open a browser and enter:

`http://localhost:9081/ITSOAlma/PartsInputForm.html`

To run the EJB test client open a browser and enter:

`http://localhost:9081/UTC`

Attention: You may have to activate the virtual host alias ***:9081** in the Administrative Console (select *Virtual Hosts*).

Testing using the IBM HTTP Server (AE)

This is the same as testing on WebSphere AEs (see “Testing using the IBM HTTP Server (AEs)” on page 189):

`http://host-name/ITSOAlma/PartsInputForm.html`
`http://host-name/UTC`

Remote unit testing from Application Developer

In this section we describe how to use Application Developer for a remote unit test of the Almaden Autos enterprise application.

You must have installed WebSphere Application Server 4.0 Single Server (AEs) locally or on a remote machine, and you have IBM Agent Controller running on that machine.

The IBM Agent Controller comes with Application Developer. It is a process that runs on the remote machine and which enables client applications to start new host processes. The IBM Agent Controller will start the application server on the same or remote machine (Figure 6-34).

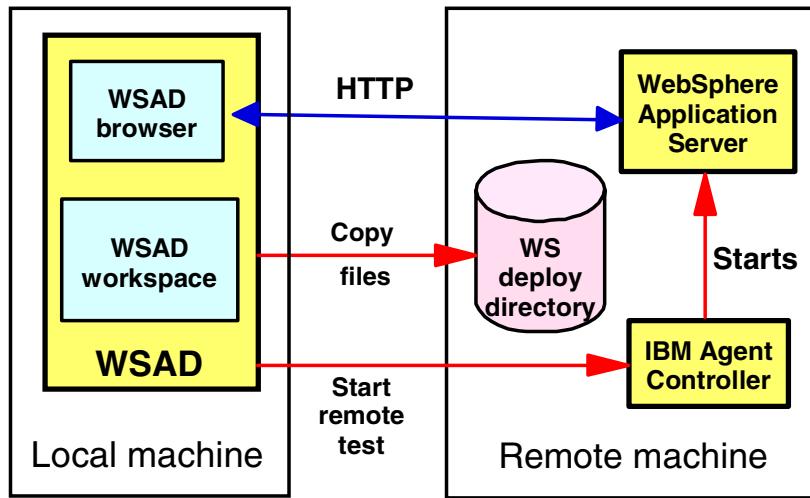


Figure 6-34 Remote unit test

Attention: If you already have the Agent Controller installed on a remote test machine, and you are installing WebSphere Application Server on the same machine, you must edit the settings in the server configuration file after you install WebSphere Application Server:

- ▶ Open the `serviceconfig.xml` file that is located in the `x:\Agent_Controller\config` directory (where `x:\Agent_controller` is the installation path of the Agent Controller).
- ▶ In the file, search for the Application executable=`"wteRemote.exe"` tag.
- ▶ Change all occurrences of `path="%WAS_HOME%"` within the Application executable tag to `path="WAS_ROOT"` (where `WAS_ROOT` is the installation path of WebSphere Application Server).
- ▶ Save the changes and close the file.

In order to test from Application Developer against a remote WebSphere instance, we have to define and configure a remote server instance:

- ▶ Open the server perspective.
- ▶ Select *File -> New -> Server Instance and Configuration*.
- ▶ Specify a *Server name* (ITSOWSADRemote) and the *Server project* folder where you want to store the server instance and configuration (Figure 6-35).
- ▶ Under *Server instance type*, expand *WebSphere Servers* and select *WebSphere v.4.0 Remote Test Environment*. Click *Next*.

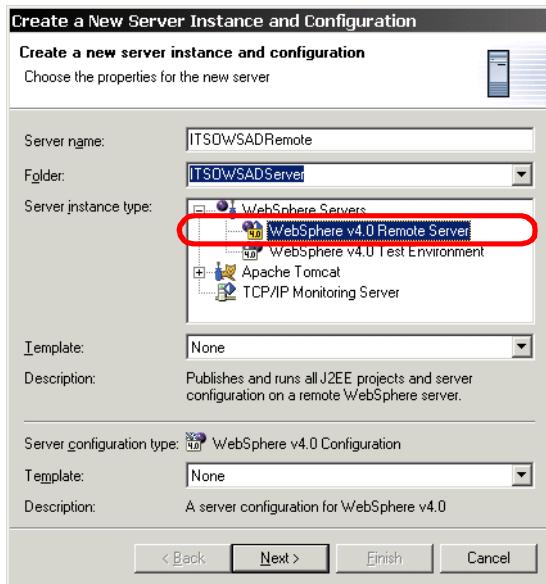


Figure 6-35 Specify remote server instance

- ▶ On the WebSphere Remote Server Instance Settings panel (Figure 6-36):
 - Specify the *Host Address* (either the IP address or the full hostname, localhost does not work).
 - Specify the *WebSphere Installation Directory* (d:\WebSphere\AppServer). Click *Next*.

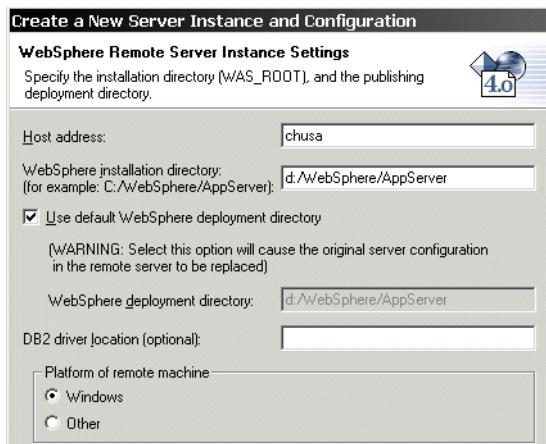


Figure 6-36 Remote server instance settings

- On the *Create or Select a Remote File Transfer Instance* panel (Figure 6-37), you can choose either to use the *Copy file transfer mechanism* or to use *FTP*. If you want to use *FTP*, an *FTP server* must be running on the remote machine. In this example we use the *Copy File Transfer Mechanism*. Click *Next*.

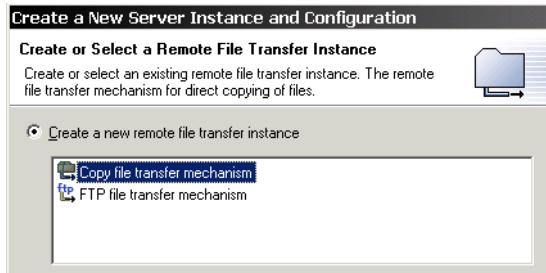


Figure 6-37 Remote file transfer instance

- In the next panel specify the *remote file transfer name* and the *remote target directory* (Figure 6-38).
 - The *remote file transfer name* is a name you assign to the remote file transfer instance so that you can use it in another server configuration.
 - The *remote target directory* seen by the current machine, where you want to publish your Web application. For example, if you mapped the WebSphere installation drive as drive **k**, then you would specify **k:\WebSphere\AppServer** as the target directory. Click *Next*.

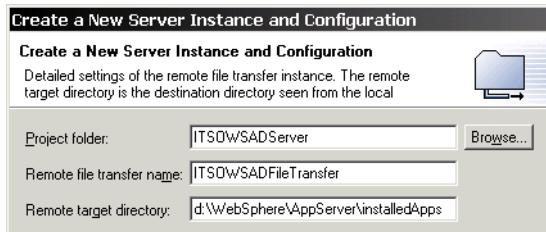


Figure 6-38 File transfer instance configuration

- Select a *Server port* or leave the default of 8080.
- Click *Finish*.

We have configured the remote server instance and the remote server appears in the Servers view. Now we need to configure the data source for that instance.

JDBC driver and data source

To configure the data source for the remote server instance:

- ▶ In the server perspective, double-click the ITS0WSADRemote server configuration.
- ▶ On the *Datasource Tab* (Figure 6-39), edit the *Db2JdbcDriver* and make sure it has the correct *Classpath*. Save and close the editor.

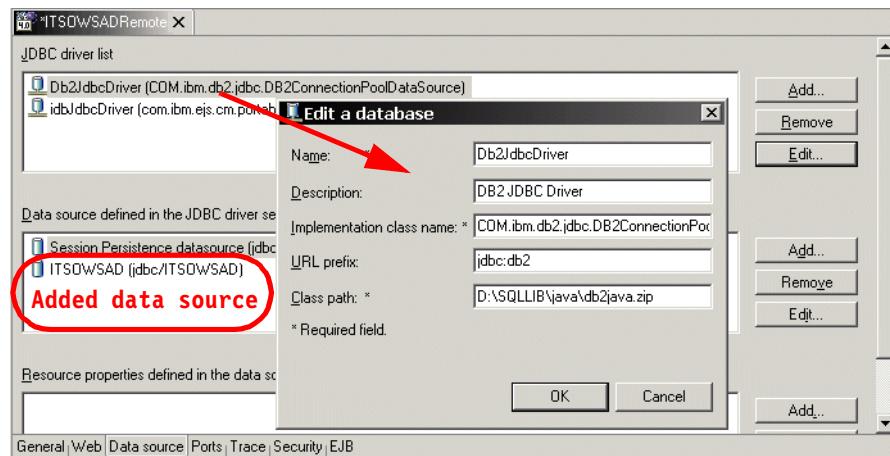


Figure 6-39 JDBC driver configuration

- ▶ Click on *Add* to add a data source with JNDI name *jdbc/ITS0WSAD* and database name *ITS0WSAD*. (See Figure 3-25 on page 85.)

Assign a project to the remote server

Add the Almaden Autos enterprise application to the remote server instance:

- ▶ In the server perspective expand the server configuration, select the ITS0WSADRemote server, and *Add Project* from the context menu. Select the ITS0A1maEAR project.

Testing Almaden Autos on the remote server instance

Make sure the remote server instance is the default server for the Almaden Autos project.

- ▶ In the Web perspective open the *Properties* of the ITS0A1maWeb project.
- ▶ Make sure that the remote server instance you defined is selected as the *preferred server instance* for the *Server Launcher* property.

Run the application:

- ▶ Start the remote server. (Be patient.)
- ▶ Expand the ITSOA1maWeb project and under *webApplication* right-click *PartsInputForm.html*, and select *Run on server*.
- ▶ The browser is launched. Now you can test the Web application on an application server on a remote machine.

Summary

In this chapter, we have explored the many different deployment options. We have deployed the two previously developed enterprise applications, Almaden Autos and Mighty Motors, to WebSphere Advanced Edition Single Server (AEs) and WebSphere Advanced Edition (AE). We tested against both servers using the embedded HTTP server the IBM HTTP Server as an external HTTP Server.

Finally, we configured a remote server instance in Application Developer for testing against a remote server.

Quiz: To test your knowledge of the Web and EJB deployment features of Application Developer, try to answer the following five questions. The answers are provided in Appendix C, “Quiz answers” on page 559.

1. What are the two different ways to install an enterprise application in AEs?
2. Name at least two big differences between AE and AEs.
3. What protocol is used to communicate between the Web server plug-in and the application server?
4. On what port is the embedded Web server listening for AEs and for AE?
5. In what file is the Web server plug-in configuration stored?



Working in a team

This chapter provides an introduction to the team support provided in the beta code of Application Developer, and some considerations for developing J2EE applications in a shared environment.

The following topics are described:

- ▶ IDE architecture and team terminology
- ▶ Repository types and differences
- ▶ CVS overview and configuration
- ▶ Using the team perspective
- ▶ Releasing, versioning and streams
- ▶ Comparing and merging artifacts
- ▶ Limitations in the current release
- ▶ Best practices and administration tasks

Refer to Appendix B, “State transitions for resources in Application Developer” on page 545 for additional information on team support.

Attention: The GA version of the product can be configured with either CVS or ClearCase LT at installation time. **We only cover CVS in this book.**

Team overview

Developers migrating to Application Developer from previous IDEs from IBM, such as VisualAge for Java and WebSphere Studio classic, will notice that one of the most significant changes is related to how the IDE supports team development. This section describes the new extensible architecture provided with Application Developer.

An extensible architecture

Figure 7-1 provides a very simple illustration of how the development environment provides the team capabilities.

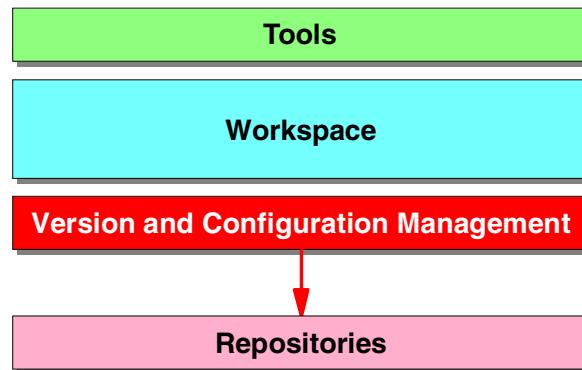


Figure 7-1 Version and configuration management plug-in architecture

The team support component of the WebSphere Studio Workbench platform adds version and configuration management (VCM) capabilities to projects in the workspace, and provides the Workbench with all the necessary views for presenting team features to the user.

The team support model centers on *repositories* that store version-managed resources on shared servers. These resources may be any artifact stored in the workbench, such as Java files, HTML pages, documents and generated metadata. The Workbench platform is designed to support a range of existing repository types, although IBM does not provide one of its own. A single workspace can access different types of repositories simultaneously both over local and wide area networks.

Vendors of software configuration management (SCM) products can develop to the VCM extension point in the workbench and provide a connector between their repository client and the workspace. The current implementations of the workbench only provide support for the open source version control system known as CVS (Concurrent Versions System) although both Rational and Merant have made press announcements that they will be providing Workbench support for ClearCase and PVCS respectively.

Differences between CVS and ClearCase

The illustration in Figure 7-2 shows the main differences in functionality between CVS and the two available versions of Rational's product - ClearCase LT and ClearCase.

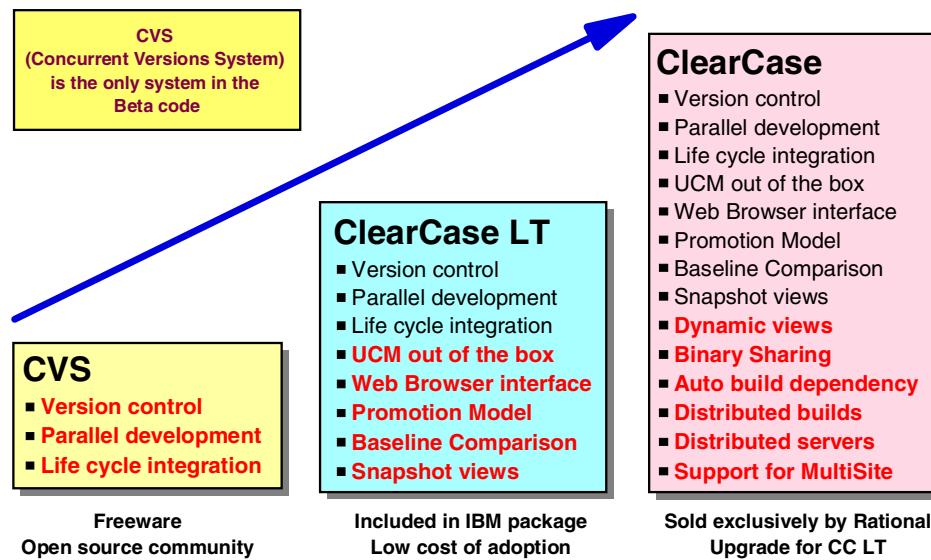


Figure 7-2 Differences between CVS and the two ClearCase versions

ClearCase LT is a limited release of ClearCase, which Rational has recently been shipping with their latest suite releases, and is ideally suited for small to medium sized development teams. If customers require a fully distributed enterprise-scale repository, then the full ClearCase product is the recommended Rational solution.

Application Developer will be fully compatible with both versions, although there is a limitation that there is no interoperability between the two Rational products (a ClearCase LT server cannot access a ClearCase repository).

IBM is including both CVS and ClearCase LT into the packaging for the various editions of WebSphere Studio, and upgrades to the full version of ClearCase will be sold by Rational directly.

Because the ClearCase support is not currently available in the beta code, the remains of this chapter will focus on working with CVS. CVS is available on a number of platforms including Windows, Linux, AIX, HP-UX, and Solaris.

Workspace

The workspace is maintained by the IDE and contains a snapshot of all of the files that it knows about that reside in the file system and also contains information relating to the last time they were modified by a developer using the tool. If you edit a file using an external editor or add a file to the file system, you must always use the *Refresh from Local* menu option to reflect those changes in the Navigator and editors in the IDE.

Important: If you have previously been developing in VisualAge for Java, one of the most important features you will need to know is that *file deletions from the workspace are permanent* as there is no local repository. Even if you are working alone, we still recommend that you install a local CVS instance and regularly release changes to it.

Local history

Application Developer stores a history of locally made changes to files that still exist in the file system for a period that is specified in the Workbench preferences (Figure 7-3), however deleted files cannot be recovered from the local history.

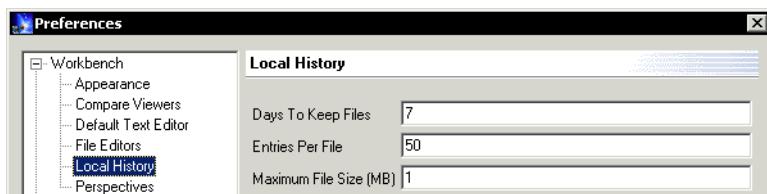


Figure 7-3 Local history preferences

The files are stored in a non-editable format in the following directory:

```
workspace/.metadata/.plugins/org.eclipse.core.resources/.history
```

To view the history of a file, simply select *Compare with* or *Replace with -> Local History* from the files context menu.

Configuring multiple workspaces

As there is a memory overhead associated with having many open projects in the workspace at the same time, a developer may wish to configure multiple workspaces for the different development projects they are working on. This reduces the amount of clutter experienced in the *Navigator* view, and also minimises the number of entries in the *Server control panel* when trying to find a server configuration or instance.

Each workspace is created as a folder in the file system where the source code will reside. To start the IDE with a different workspace, simply execute:

```
wsappdev -data myworkspacedir
```

Where *myworkspacedir* is the absolute path to the workspace folder to use. If it is the first time the workspace has been used, the IDE displays the *Welcome* page and the default view specified when installing.

If there is large amounts of memory available on the developer workstation, it is possible to execute multiple instances of the IDE simultaneously against different workspaces. If a workspace is not explicitly defined when starting the IDE, it uses the default workspace provided in the installation directory.

This technique is useful when working on two releases of a project at the same time—such as building a new release and performing maintenance on an old one. Each development stream can have its own workspace and copies of the files in the file system.

Executing Application Developer with the binaries on the server

A common requirement for large VisualAge for Java installations was how to install the executables for the IDE on a server (such as Windows 2000 Server or Novell Netware) but store the workspace on a local drive or a mapped home directory.

Using the technique we have just discussed, this is also possible using Application Developer. Assuming that all of the organizations applications are installed on the p: drive, and each developer has their own source files on a mapped directory q:, simply start the IDE using:

```
p:/wsad/wsappdev -data q:/myworkspacedir
```

As before, it is recommended that a very fast and reliable local network is available to ensure that the response times in the IDE are acceptable.

Team terminology introduction

Normally, each team member does all of the work in their own individual workbench, isolated from others. Eventually they will want to share their work and get their team-mates changes. They do this through *streams*.

Streams

A stream maintains a team's shared configuration of one or more related projects and their folders and files. A team of developers share a stream that reflects their ongoing work and all their changes integrated to date. In effect, a stream is a shared workspace that resides in a repository.

Each team member works in a separate workspace and makes changes to private copies of the resources. Other team members do not immediately see these changes. At convenient times, a developer can synchronize their workspace with the stream:

- ▶ As a team member produces new work, they share this work with the rest of the team by *releasing* those changes to the stream.
- ▶ Similarly, when a team member wishes to get the latest work available, they *catch-up* to the changes made to the stream by others.

Both synchronization operations can be done selectively on resource sub-trees with an opportunity to preview incoming and outgoing changes.

Optimistic concurrency model

Version control systems provide two important features required for working in a team:

- ▶ A history of the work submitted by the team
- ▶ A way to coordinate and integrate this work

Maintaining history is important so that one can compare the current work against previous efforts, or revert to older work that is better. Coordinating the work is critical so that there is one definition of the current project state, containing the integrated work of the team. This coordination is provided through the stream model.

An optimistic model is one where any member of the team can make changes to any resource they have access to. Because two team members can release to the stream changes to the same resource, conflicts can occur and must be dealt with. In Application Developer, conflicting incoming and outgoing changes are detected automatically. This model is termed *optimistic* because it is assumed that conflicts are rare.

This model supports groups of highly collaborative developers who work on a common base of files and who frequently share their changes with each other. It also supports developers who work off-line for long periods, connecting to their repository only occasionally to synchronize with the stream. This is illustrated in Figure 7-4.

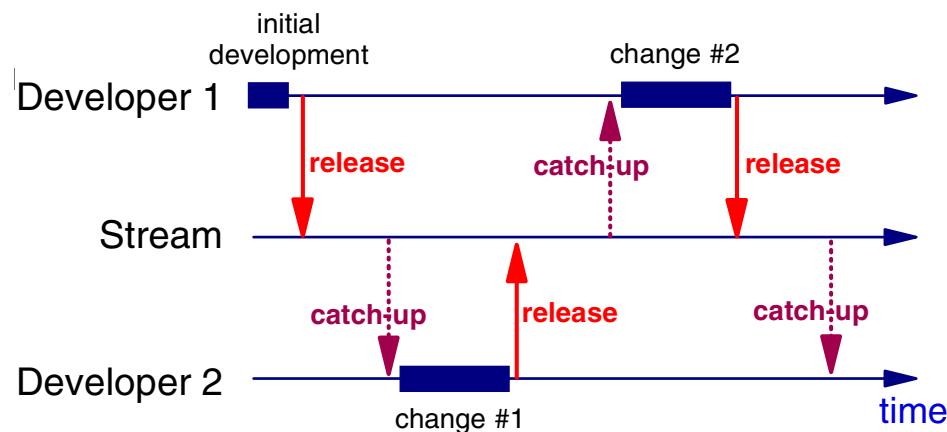


Figure 7-4 Two developers doing sequential development using a stream

Branching

A repository typically may have any number of streams, which are distinguished by name. In a CVS repository, for instance, a stream maps to the main trunk (known as *HEAD*) or to a branch.

Streams act as lightweight containers for building product releases as safe places to store a developer's personal work and early prototype versions outside the team's main development stream. When the same project appears in different streams, the resources evolve independently; changes released in one stream have no effect on other streams until they are merged.

Versions

Repositories also support the notion of finalized resource *versions*. A version of a file resource captures the content of the file. A version of a project captures the configuration of folders and files and their specific versions (in other words, a *baseline*).

In a CVS repository, a project version corresponds to a distinctive tag assigned to a set of file versions. New versions of changed files are created automatically when those changes are released to a stream. New project versions may be created as a snapshot of either the current configuration of a project in a stream, or the current configuration of a project in the workspace.

In the team environment, resources other than projects (such as files or folders) cannot be explicitly versioned. However, they are implicitly versioned when they are released to the stream.

A base version of a resource contains what was released on the stream at the time of the most recent synchronization. Therefore, the base version of a resource in the workbench denotes its corresponding state in the repository. This remains true even after modifying the resource locally: the base version denotes what the resource looked like before you started modifying it. A new local history for a resource is created the first time you modify the base version.

Terminology matrix

Table 7-1 attempts to map the terminology used between Application Developer, native CVS, and Rational ClearCase.

Table 7-1 Terminology matrix for team support

Application Developer	CVS	ClearCase
Workspace	File system	Work area
Repository	Repository	VOB
Stream	Branch (tag)	Stream and Project
Project	Folder	View
Resource	File	Element
Release	Revision	Check in
Catch up	Update	Compare with
Version	Commit (tag)	Version

Team perspective

The next section of this chapter provides a walk-through of the team perspective by connecting to an existing CVS repository and adding a project to the workspace.

Perspective overview

Open the team perspective. The default views are as follows:

Navigator	The standard navigator showing all files in the workspace, except those defined by the assigned filters. The default filter removes all <i>.class</i> files from the view.
Properties	Shows properties of the currently selected file in the workspace, including the time it was last modified and its size. If an item in a repository is selected, this view shows the artifacts author, version number, and comments.
Repositories	Shows all repository connections and their contained streams and versions
Synchronize	Used to perform release and catch-up operations, and view differences between files
Tasks	Standard task list
Resource History	Responsible for displaying the version history of the selected item when requested

Connecting to a CVS repository

As we have not yet installed a CVS repository locally, we will connect to one of the most popular repositories on the Internet—`cvs.apache.org`—to explore some of the team features.

`cvs.apache.org` is a publicly run CVS server that is used around the world by developers contributing to the open source Apache projects, such as the Apache Web server, the Jakarta Java tools, and the XML utilities and parsers.

To be able to perform this step, you must have a live Internet connection, or be working behind a corporate proxy, or socks server.

Select the *Repositories* view and click on the *File -> New -> Other* menu in the Workbench. From the *CVS* category select *Repository Location*. Click *Next*. Complete the repository location wizard (Figure 7-5).

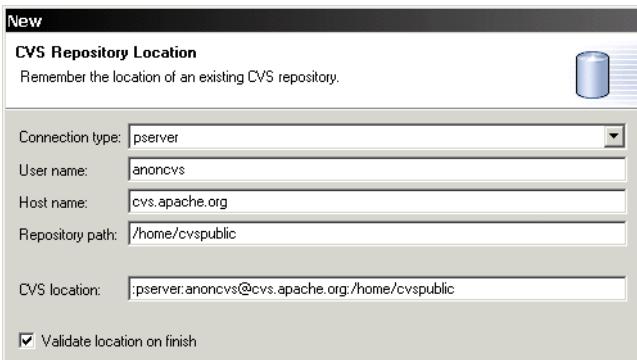


Figure 7-5 Defining the repository location for the Apache CVS server

Here are brief descriptions of the CVS terms on this panel of the wizard:

Connection type	The protocol you want to use to connect to the CVS server. The default selection is pserver, which represents the <i>password server protocol</i> , that is used by most public repositories. Other CVS alternatives include the secure shell <i>ssh</i> , an explanation of which is outside the scope of this document.
User name	The user ID you wish to use to log on to the CVS server. Most public servers have an anonymous user ID published on their Web site, which has read-only access to the server.
Host name	The machine name of the server you wish to connect to.
Repository path	The fully qualified path to the location of the repository on the server.
CVS location	The complete location used by Application Developer to connect to the repository. This is dynamically built as you complete the previous fields.
Validate on finish	This tests the repository connection before returning to the IDE, which we recommend to keep enabled as a first connectivity test.

Click *Finish*. If Application Developer is able to successfully connect to the CVS server, you are prompted for a password for the anoncvs user ID. Enter anoncvs as the corresponding password.

Once the user ID and password have been validated, we should be returned to the *Repositories* view with a new entry for the Apache server.

Viewing the projects in the HEAD stream

Two entries should appear under the repository connection—*Project Versions* and *Streams* (Figure 7-6). Because we are logged in as an anonymous user, we only have the ability to read the HEAD stream, with no access to the project versions or other streams. We will deal with how to access these later.

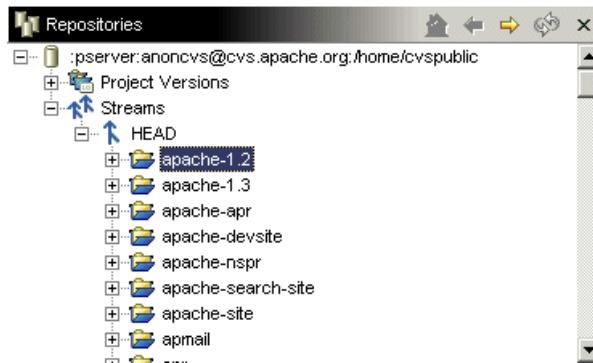


Figure 7-6 Projects in the HEAD stream of the CVS server

Let us assume that we want to retrieve the very latest source code for the Jakarta JMeter project (a load tester for Web applications), which includes a fix that we need to get it working with our application.

Expand the *HEAD* stream, scroll down the list of Apache projects to the *jakarta-jmeter* entry. By expanding this element in the tree, we can see all of the elements contained inside and their corresponding version numbers.

Browsing the resource history

Select the *README* file in the stream. In the *Properties* view you see the name of the last author who modified the source, and the comment that was added the last time the code was released by a developer into the stream.

Right-click on the file and select *Show in Resource History*. The history of changes appears in the *Resource History* view with entries similar to those shown in Figure 7-7.

To open a specific version of the *README* file to see its contents, select an entry from the history, such as version 1.1, and select *Open* from the context menu for that row in the table. An editor opens on the resource in the team perspective. Select *Open* again on another version, such as 1.6. A second editor opens. Currently, none of these files have been added to the workspace. This is very similar to features of the *Repository Explorer* that was in VisualAge for Java.

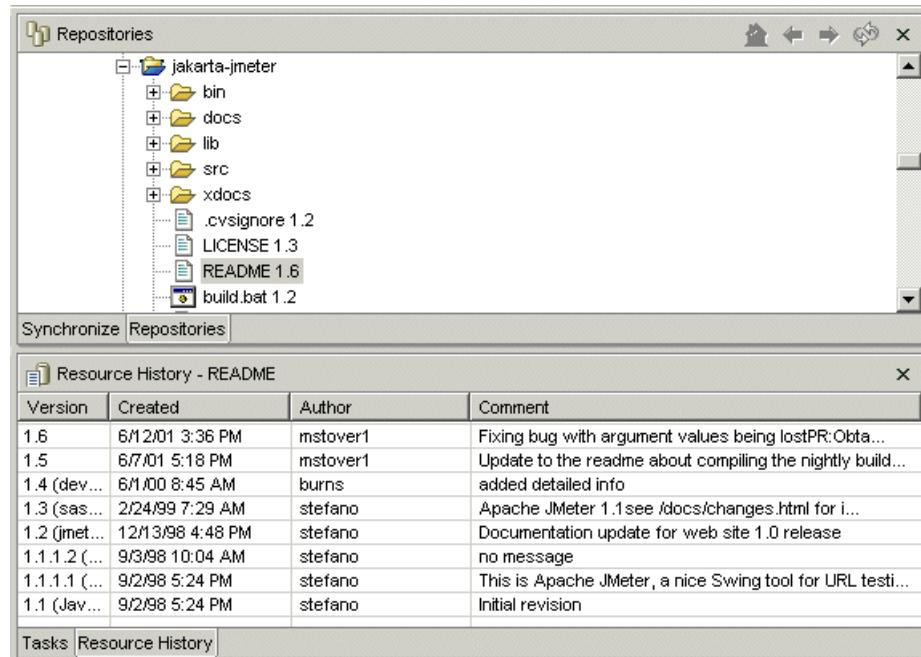


Figure 7-7 Resource history view for JMeter project

Comparing two versions of a file in the repository

Let us see how much the *README* file has changed between version 1.1 and 1.6. While holding down the *Control* key, select both versions in the *Resource History* view, then select *Compare* from the context menu. The *Text Compare* view is now opened in the perspective (Figure 7-8).

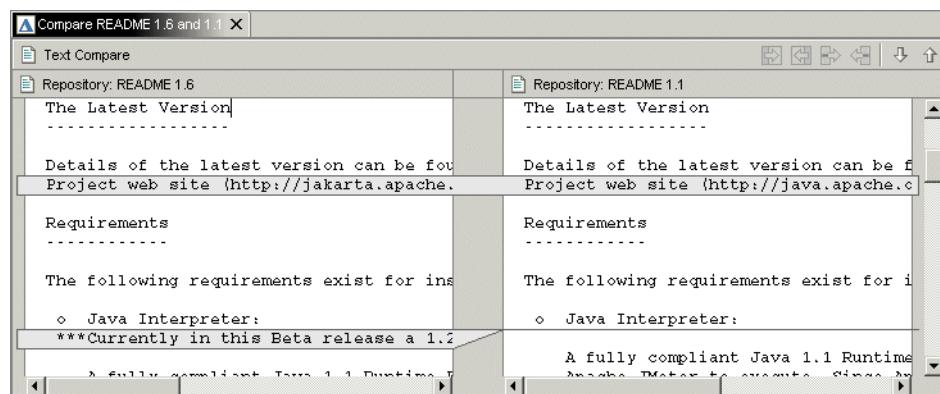


Figure 7-8 Comparing two versions of the *README* file

When scrolling through the document, the changes between each release are highlighted with insertion points shown for new sections. The arrows in the top right hand side of this view allow jumping between changes.

Additionally, there is a toolbar button in the main workbench, which allows whitespace sensitivity in the comparison to be turned on or off. This is especially useful when comparing text files.

Close the comparison window for the README. Note that the right hand side of this perspective is now split into three views—one of which is empty. Do not be tempted to move the repositories view to remove the empty space, as future editors will not be visible.

Next, we compare two versions of a Java source file, to demonstrate that the comparison view behaves differently for various file types.

Select Assertion.java from the org.apache.jmeter.assertions package in the src folder. Open this file in the *Resource History* view and compare versions 1.1 and 1.3 (Figure 7-9).

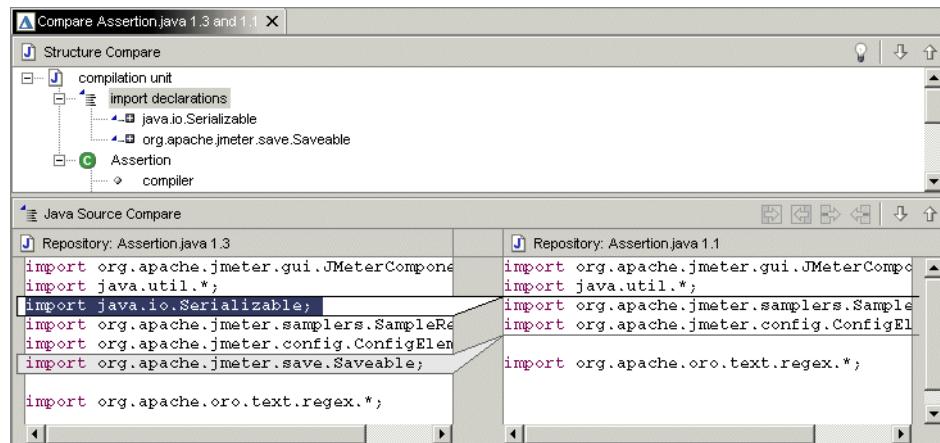


Figure 7-9 Comparing two versions of the Assertion.java file

Notice that when comparing two Java source files, a *Structure Compare* panel is also included in the comparison view. This shows an outline of the changes to the file. Select *import declarations*, and you will see all changes to the import statements in the file. Select *java.io.Serializable* and you will see that this does not appear in version 1.1 (this is also shown by the + icon in the comparison structure). By clicking on the *compiler* element, you can see that this class variable has changed from being declared as static to transient static.

The structure compare view also has arrow icons to navigate between change scope. These work in conjunction with the icons in the bottom panel of the perspective, which selects individual changes within that scope. Also included in this toolbar is the icon, which enables intelligent guessing when comparing changes. When finished, close the comparison view.

Adding a project to the workspace

Let us assume that we now would like to import all the source code from the Apache JMeter project into Application Developer and compile the source. This process is described in “In repository stream -> Assigned in workspace (4)” on page 549.

Navigate back to the *jakarta-jmeter* entry under the *HEAD* stream in the *Repositories* view and select *Add to Workspace* from its context menu. After a brief delay, while the files are downloading from the Internet, the project should now appear in the Navigator. These files have been copied into a *jakarta-jmeter* folder under the current workspace folder of the Application Developer (the default is the workspace directory in the installation directory). Note that there is currently no option to specify the directory you would like for importing this source into, other than the default workspace folder.

Open the properties dialogue of the new project. Switch to the *Team* category, and we can see that the project is associated with the Apache CVS server and its *HEAD* stream (Figure 7-10). If we wish to amend the stream or repository the IDE uses to release any changes, we can do so here by clicking on the *Change* button.



Figure 7-10 The repository and stream assigned to the imported project.

Before we look into the files copied in more detail, open the Resource History again on the README file. Notice that the 1.6 version is now marked with an asterisk (*). This illustrates that this is the base version currently loaded into the workspace.

Changing project types

Unfortunately, the team environment creates by default a *Simple* project type locally in the file system, which means that we cannot compile the source code. We must now switch from the Simple project to a *Java* project. There are no operations provided by Application Developer to support changing project types, so we must do this manually by changing the metadata used to store this information on the file system.

Open the file `.prj` file stored in the installation directory and change the contents as shown in Figure 7-11 (the lines to add are in bold).

```
workspace/.metadata/.plugins/org.eclipse.core.resources/.projects/jakarta-jmeter/.prj
```

```
<?xml version="1.0"?>
<projectDescription>
    <name>jakarta-jmeter</name>
    <version>0.0 000</version>
    <comment></comment>
    <projects>
    </projects>
    <buildSpec>
        <buildCommand>
            <name>org.eclipse.jdt.core.javabuilder</name>
            <arguments>
            </arguments>
        </buildCommand>
    </buildSpec>
    <natures>
        <nature>org.eclipse.jdt.core.javanature</nature>
    </natures>
    </projectDescription>
```

Figure 7-11 Modified project file which stores the project type

To refresh the value stored in the IDE, select the project in the navigator and select *Close Project* and then *Open Project* from its context menu. On reopening the project, Application Developer reads the `.prj` file again and changes the type of project to Java. This is noticeable by the small letter J shown in the folder icon. Once it has detected this change, it then attempts to compile the resources contained inside.

Configuring the project

The JMeter project does not compile cleanly the first time. To fix these errors switch to the Java perspective and take these steps:

- ▶ Open the properties editor for the project and add the JAR files in the imported lib folder as external JARs in the build path (multiple selections can be made when adding JAR files).
- ▶ Also add the rt.jar from the jre/lib folder in the installation directory by selecting *Add Variable* and *Browse* to locate the JRE_LIB variable.
- ▶ In the *Source* tab of the *Java Build Path* category, select *Use source folders contained in the project*, click the *Add Existing Folders* button, and select the src folder and click *OK*.

When the properties dialog is closed, the project is rebuilt. Although a few remaining deprecation warnings and some optional JavaMail errors appear, the application should now be ready for execution.

Important: The JMeter project will not be used in the redbook. Therefore delete the project in the Packages view to clear all the error messages.

Installing a local CVS repository

The examples to date have been focused on working with an existing CVS system as an end user. The next step is to install CVS locally and configure the IDE to connect to it.

Downloading and installing CVS

The easiest way to get hold of CVS is to download the binaries from <http://www.cvsnt.org>. The version we will be using is version 1.10.8, which is downloadable as a pre-compiled executable from the homepage. The file name is cvs-1.10.8NT.exe.zip. If you want to download CVS and execute it on a Unix system, similar downloads are available from <http://www.cvshome.org>.

When the code has been downloaded, unzip it into a directory named CVSNT on the local hard disk.

Creating a new repository

Open an MS-DOS prompt and change to the CVSNT subdirectory created in the previous step. Issue this command to create the CVS repository:

```
cvs -d :local:x:/CVSRepo init
```

Where x: is the drive letter where you want to create the repository.

Creating a Windows service

To configure CVS as a Windows service, execute this command:

```
ntservice -i x:/CVSRepo
```

A CVS NT service appears in the Windows services window (Figure 7-12). Start the CVS service. If there are problems starting the service, check the Windows event viewer to see the details of the errors.

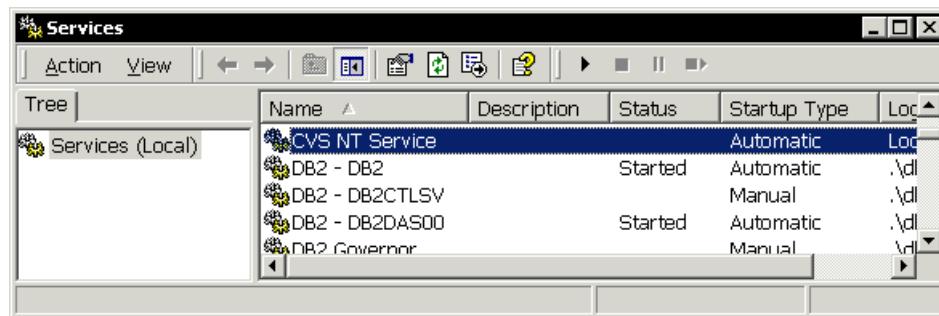


Figure 7-12 CVS Windows service

Creating new users

In this chapter, to properly understand team development we need at least two developers for our application. With the pserver protocol, the CVS Windows port uses the user IDs of the operating system to authenticate with the CVS server.

Create two new Windows users—developer1 and developer2. Set both passwords to ibmitso.

To properly simulate two users working simultaneously, we will use the multiple workspace approach discussed earlier in the chapter.

Suggested naming conventions: Resource versioning is an automatic process which cannot be changed. All changes start with version 1.1 and increments during each release by a minor number.

There are a number of limitations on the names you can use for projects versions. Specifically the CVS interface prevents the use of periods (.) and suggests using either dashes (-) or underscores (_) instead. Through this book we will be creating versions using incrementing numbers, with minor revisions separated by underscores, such as v1_1, v1_2.

Team development simulation

In this section, we will use two separate workspaces to simulate branching and merging changes in a team environment.

Configuration

Create two local workspace directories using the following commands from a Windows command prompt:

```
md x:/ITSOWSAD/developer1_ws  
md x:/ITSOWSAD/developer2_ws
```

Where x: is the local data drive defined in previous chapters. Then start Application Developer using these commands:

```
y:/wsad/wsappdev -data x:/ITSOWSAD/developer1_ws  
y:/wsad/wsappdev -data x:/ITSOWSAD/developer2_ws
```

Where y: is the drive used to install Application Developer.

Two instances of the IDE are now executing. Close the default perspective in each instance to preserve memory and open the team perspective. Switch to the *Repositories* view.

In each instance of the IDE, create a connection to the local CVS repository installed in the previous chapter using the appropriate developers user ID and password, as described in “Connecting to a CVS repository” on page 215.

Because we will be demonstrating the team capabilities using Java classes, also open the Java perspective in each instance.

Sequential development scenario

The simplest scenario to demonstrate is that of sequential development. This involves only one developer working on a file at the same time (Figure 7-13).

Both developers are working on the same development stream with a class named TeamTest1.java in the itso.wsad.team package.

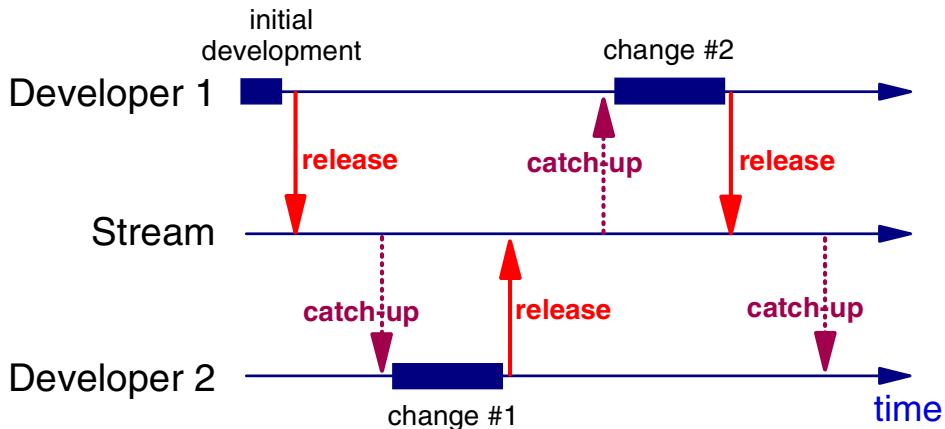


Figure 7-13 Sequential development scenario

Developer 1: Initial code release

Select the workspace for Developer 1 and complete these tasks:

- ▶ Create a new Java project named `ITS0WSADTeam`. Notice that the default location is now under the `developer1_ws` directory.
- ▶ Create a new package named `itso.wsad.team` under this project.
- ▶ Create a new Java class named `TeamTest1` inside this package. Accept all the default options.
- ▶ Close the editor that is open on `TestTest1`. Switch to the team perspective.
- ▶ Open the *Properties* dialog for the `ITS0WSADTeam` project. Switch to the *Team* category and click the *Change* button.
- ▶ Select the *HEAD* stream of the local repository connection and click *OK*.
- ▶ Click *OK* again to close the *Property* dialog. The project is now assigned to a stream.
- ▶ From the projects context menu, select *Team -> Synchronize with Stream*. This opens the *Release Mode* in the *Synchronize* view.
- ▶ Select the `ITS0WSADTeam` project entry in the *Structure Compare* panel and select *Release* from its context menu.
- ▶ Enter a comment of `Initial development complete` and click *OK*.

The first step of the development has now been completed. Next, Developer 2 must connect to the repository and retrieve the latest contents of the *HEAD* stream into the workspace.

Developer 2: Retrieve code and change

Switching to the Developer 2 workspace and complete these tasks:

- ▶ Switch to the team perspective.
- ▶ Refresh the *Repositories* view. Expand the *Stream* and *HEAD* elements and the ITSOWSADTeam is displayed.
- ▶ Select the project entry and from its context menu select *Add to Workspace*.
- ▶ Switch back to the Java perspective. Open the TeamTest1 class for editing.
- ▶ Add a new method with the following signature:

```
public void change1() { }
```
- ▶ Save the changes and close the editor.
- ▶ From the context menu for the item, select *Compare With -> Base version*. This allows us to view the changes made since the file was copied from the repository. Click *Done* when finished.
- ▶ Select the ITSOWSADTeam project. From its context menu select *Team -> Synchronize with Stream*. This opens the team perspective in *Release Mode*.
- ▶ In the *Structure Compare* panel, note only the *TeamTest1* file is shown in its hierarchy. Select the file and from its context menu click on *Release*.
- ▶ Provide a comment of *Change 1* complete and click *OK*. No more entries should appear in the *Synchronize* view.

Developer 1: Catch-up and change

Developer 1 now makes a modification to the code. Switch back to that instance of the IDE and complete these tasks:

- ▶ Select the *ITSOWSADTeam* project. From its context menu select *Team -> Synchronize with stream*.
- ▶ The *Synchronize* view should open in *Catch Up Mode*. *TeamTest1* should be selected with an incoming change shown.
- ▶ From the context menu of *TeamTest1*, select the *Catch Up* menu.
- ▶ Switch back to the Java perspective. Open the *TeamTest1* class in the Java editor and add a second method with the following signature, save and close:

```
public void change2() { }
```
- ▶ To release the changes to the stream again, select the *ITSOWSADTeam* projects *Team -> Synchronize with stream* menu.
- ▶ *TeamTest1* is again displayed with an outgoing change. Select the *ITSOWSADTeam* project in the *Structure Compare* view and select *Release*.
- ▶ Enter a comment of *Change 2* complete and click *OK*.

Developer 2: Catch-up

Finally, refresh the source in the workspace of Developer 2:

- ▶ From the Java perspective, select the *ITSOWSADTeam* project. From its context menu select *Replace with -> Stream contents*.
- ▶ Open an editor on the TeamTest1 class and notice both changes have been included in the file.
- ▶ To see a list of changes, select the *Team -> Show in Resource History* menu and the resource history window opens (Figure 7-14).

Version	Created	Author	Comment
*1.3	03/09/01 21:01	developer1	Change 2 complete
1.2	03/09/01 20:50	developer2	Change 1 complete
1.1	03/09/01 20:38	developer1	Initial development complete

Figure 7-14 Resource history for sequential development scenario

Note the version numbering used here. Each time the file was released by a developer it increments from 1.1 to 1.2 then to 1.3. An asterisk is displayed next to the 1.3 version because this is now the base version for the class that is in the workspace of Developer 2. From the context menu of each of these elements in the table, the *Add to workspace* menu item is available.

Revert to previous version

Assuming you want to revert back to the previous version:

- ▶ Select the 1.2 version in the *Resource History* view. Select *Add to Workspace*.
- ▶ When prompted if you want to overwrite, select *Yes*.
- ▶ Change 2 should now be removed in the editor. To refresh the Resource History select *Team -> Show in Resource History* again. The view show an asterisk marking 1.2 as the base version.
- ▶ Revert back to the latest changes by selecting the 1.3 version and adding it back to the workspace.

Parallel development in a single stream scenario

After having demonstrated how to perform simple sequential development using the Application Developer team capabilities, the next step is to understand how to manage parallel development by two developers in the same stream. This scenario assumes that you have already completed the sequential scenario described in the previous section.

Developers 1 and 2 are both currently working on the TeamTest1 class in the HEAD stream and both currently have version 1.3 as the base version in their workspace. Let us now assume that they both make changes to the file simultaneously without letting each other know. This is illustrated in Figure 7-15.

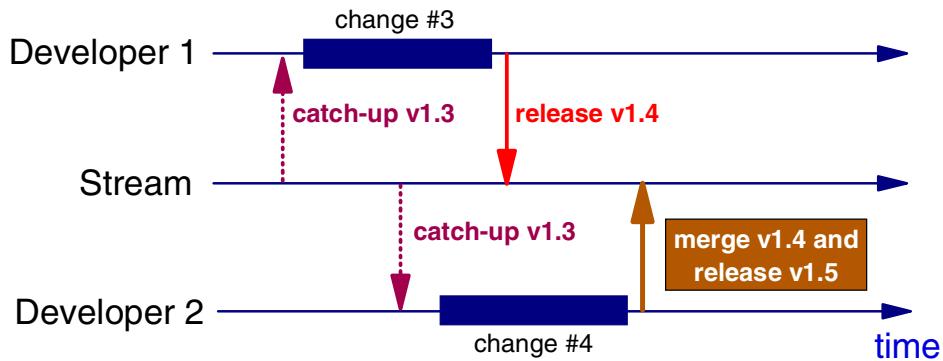


Figure 7-15 Parallel development in a single stream scenario

Developer 1: Change

Switch to the workspace for Developer 1 and complete these tasks:

- ▶ Open the Java perspective and edit the TeamTest1 class. Add a new method:

```
public void change3() { }
```
- ▶ Save the changes and close the editor.
- ▶ Release the changes to the stream, adding a comment of Change 3 complete. By opening the *Resource History* view, we can see this was stored as version 1.4 in the repository.

Developer 2: Change and merge

Our second developer is unaware of the new version 1.4 and completes change 4 on the local copy whose base version is still 1.3. Switch to the workspace of Developer 2 and perform this update:

- ▶ Open the Java perspective and edit the TeamTest1 class. Add a new method:

```
public void change 4() { }
```
- ▶ Save the changes and close the editor.
- ▶ Synchronize the ITSOWSADTeam with the *HEAD* stream. The *Synchronize* view should open in *Catch Up Mode*. TeamTest1 is shown with a double-headed arrow icon (Figure 7-16). This means there have been changes made to the stream, which conflict with the version Developer 2 wants to release.

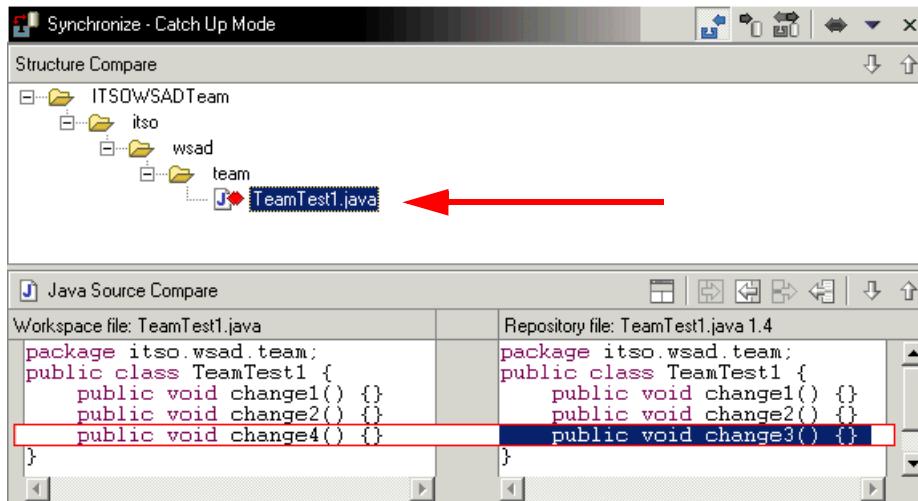


Figure 7-16 Conflicts shown in the Synchronize view

- ▶ To view what has happened, select the TeamTest1.java file from the *Navigator* view and launch the *Resource History* on the resource.
- ▶ We can see from the resource history that Developer 1 has made a change to the file since our base version.
- ▶ Next, from the *Structure Compare* panel in the *Synchronize* view, double-click on the TeamTest1.java file. This opens the Java structure comparison panel (Figure 7-17).

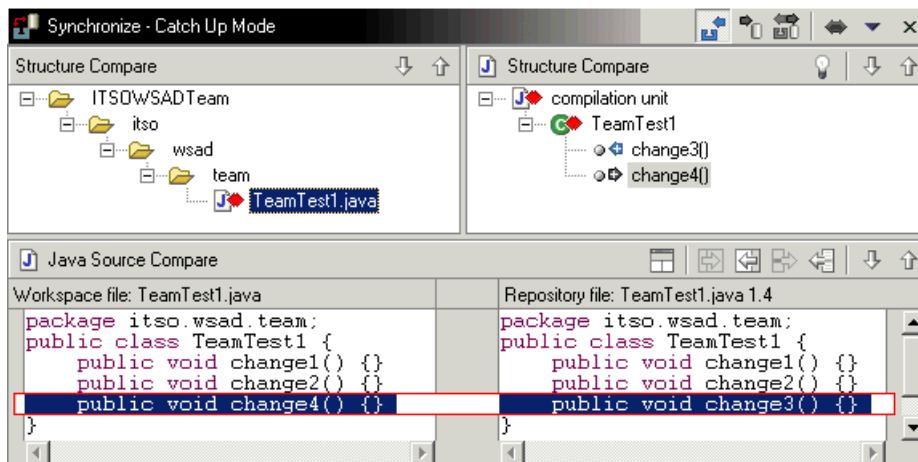


Figure 7-17 Displaying the Java structure comparison panel

- ▶ From this view, we can get a much better understanding of what has happened. *change3()* is shown with a left-arrow containing a + sign and *change4()* is shown with a right-arrow also containing a + sign.
- ▶ There are a number of icons in the Java sources compare panel (Figure 7-18). First click on *Control visibility of ancestor pane*.

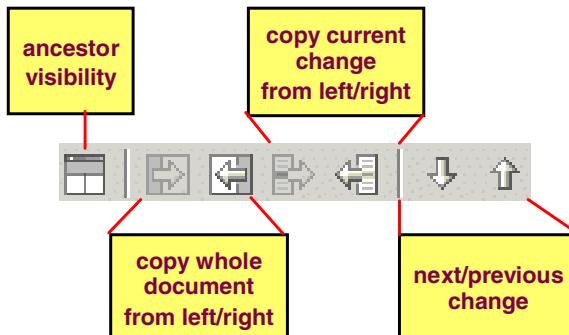


Figure 7-18 Icons in the Java sources compare panel

- ▶ This action opens a fifth panel in the *Synchronize* view. This window shows the source of the common ancestor of the two files—in this case version 1.3, which is the base version of the current code in the workspace (Figure 7-19).

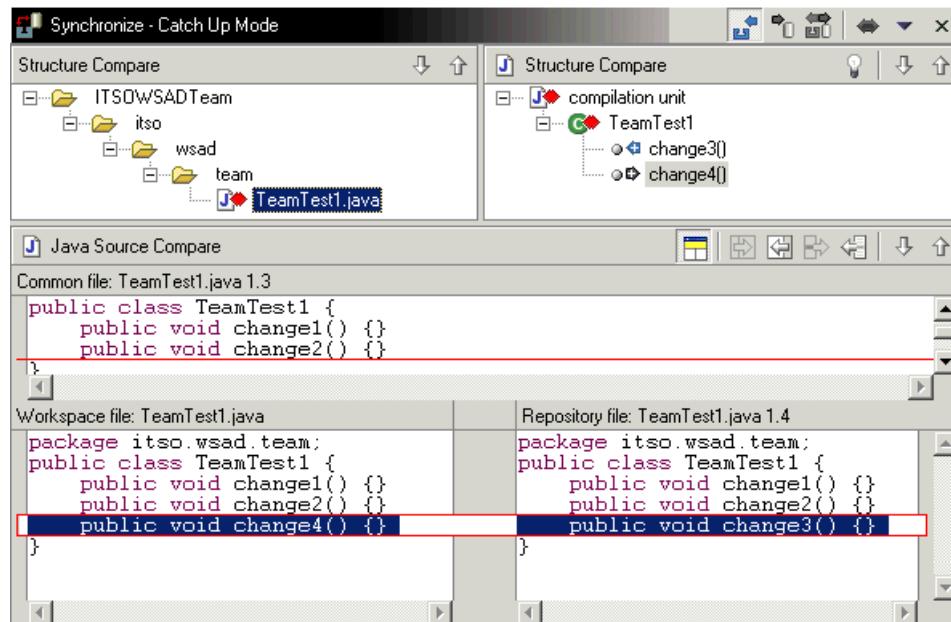


Figure 7-19 Displaying the common ancestor during comparison

- ▶ In this situation, merging the changes is relatively easy—we must add the *change3()* method into our workspace copy before releasing:
 - Select the *change3()* item in the Java structure comparison panel in the top right of this view.
 - Click on the *Copy current change from right to left* icon.
 - Right-click on the workspace entry in the bottom-left view and select *Save*.
- ▶ Save the workspace edition of the file. The *Catch Up Mode* in the Synchronize view should now be empty. Switch to the *Release Mode*.
- ▶ TeamTest1 should now show that it contains both changes. Select the file and from its context menu select *Release*.
- ▶ Enter a comment of Change 4 complete and click *OK*.
- ▶ Open the resource history for the file and notice the new version is 1.5.

So what would have happened if we had not merged the changes? By releasing the file, the current top of the stream would have been replaced with a version that does not include the *change3()* method, and this version would have been used for all future development. Streams do not support development branches—this must be done by creating a second stream.

Branching using multiple streams scenario

While building and maintaining applications, situations often occur where two developers wish to intentionally create a branch in the development process. Some examples might include prototyping a new approach or performing maintenance on an old release. In such circumstances, it is undesirable to release changes back into the HEAD stream immediately. This is the scenario we will focus on in the next section.

Figure 7-20 illustrates a scenario where our two developers branch the development of the application:

- ▶ Ensure both developers currently have version 1.5 loaded into their workspace.
- ▶ First, a project version is created from the current contents of the stream.
- ▶ Developer 2 then starts adding new functionality into the application while Developer 1 supports and maintains the version in production.

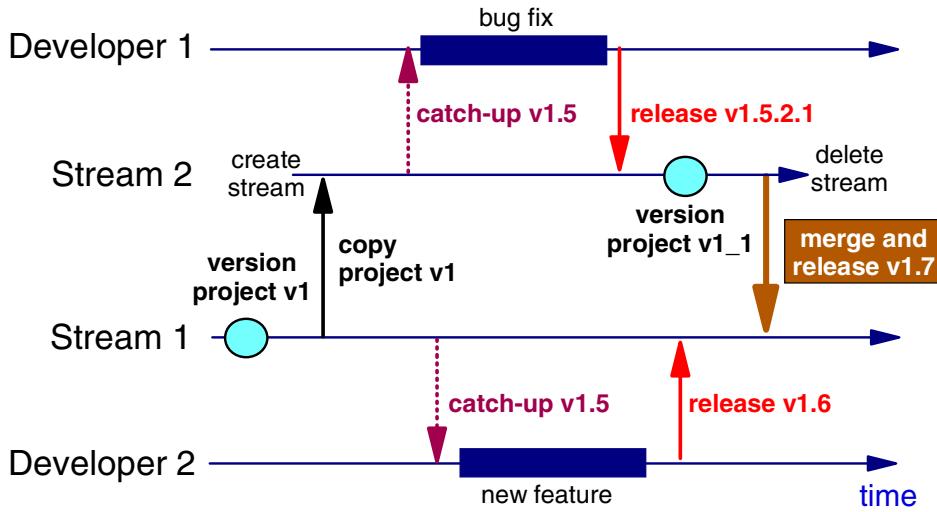


Figure 7-20 Parallel development using multiple streams scenario

Developer 1: Create new maintenance stream

Switch to the IDE for Developer 1 and complete these steps:

- ▶ From the team perspective, open the *Repositories* view. Select the *ITSOWSADTeam* in the *HEAD* stream and click on *Version from Stream* from its context menu.
- ▶ Leave the selection on *Automatic* and click *OK*. The project gets version *v1*.
- ▶ Expand the project versions element in the tree. The *ITSOWSADTeam* element should have one entry contained inside it—*ITSOWSADTeam v1*. Expand the folders of the java package—notice that it contains *TeamTest1.java* version 1.5.

Restriction: It is only possible to create a new stream from a version of a project—not from the current elements at the top of another stream. This ensures that there is always a baseline of the project we can revert to if we decide to remove all of the new streams contents.

- ▶ Next, create a new stream for the local repository using the *File -> New -> Other* menu and selecting *Stream* from the *CVS* category. Enter *Maintenance* as the Stream name. Click *Finish*.
- ▶ From the *Repositories* view, select the new stream. Currently no projects are assigned to this stream. Select *Copy version to stream* from the context menu.

- ▶ In the Choose version dialog, select the *ITSOWSADTeam v1* project and click *OK*.
- ▶ Refresh the *Repositories* view. The project should now appear in the new stream.
- ▶ Select the project entry in the new stream and click on its *Add to workspace* menu item. If prompted to overwrite the existing files, select *Yes*. The workspace project is now assigned to the new stream.
- ▶ Perform the bug fix. In this case, add a new method in *TeamTest1* with the following signature:


```
public static void main bugfix1() { }
```
- ▶ Save the changes and close the editor.
- ▶ Release the changes to *TeamTest1* into the *Maintenance* stream.
- ▶ Add a comment of *Bug fix 1 complete*. Click *OK*. By viewing the *Resource History* for the file, you will see it has been added to the repository as version 1.5.2.1.

Developer 2: Add feature to old stream

Next, switch to the Developer 2 workspace and complete these tasks:

- ▶ Edit the *TeamTest1* class and add a new method whose signature is:


```
public void newfeature1() { }
```
- ▶ Save all changes and close the editor.
- ▶ Synchronize the project with the *HEAD* stream. Release the file with a comment of *New feature 1 added*. Click *OK*. This should now have created version 1.6 in the repository.
- ▶ Version the project as *v1_1* using the *Team -> Version from Workspace* menu from the projects context menu.

Our final step is to merge the changes that were released into the *Maintenance* stream into the *HEAD* stream, so that the bug fix is included in the next release, which also contains the new feature.

Developer 2: Merge streams

In the Developer 2 workspace, which still contains the project that is assigned to the *HEAD* stream, complete these tasks:

- ▶ Select the *ITSOWSADTeam* project. Click on the *Team -> Merge* menu from its context menu.
- ▶ In the first pane of the Merge wizard, select the local repository and the *ITSOWSADTeam* project. Click *Next*.

- ▶ Select a Merge start point of version 1 of the *ITSOWSADTeam* project. Click *Next*.
- ▶ Select the end point as version 1_1 of the *ITSOWSADTeam* project. Click *Finish*.
- ▶ The merge editor open as shown in Figure 7-21.

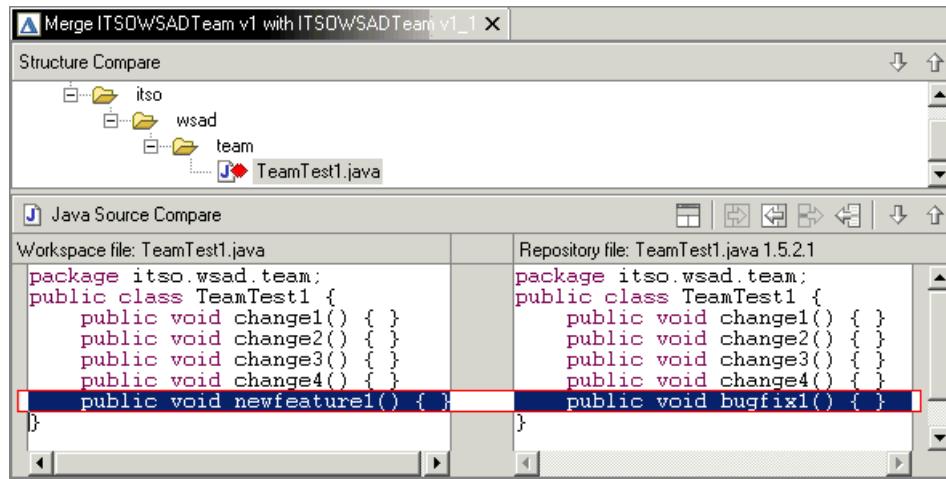


Figure 7-21 Merging the contents of two streams.

- ▶ A conflict is shown between the two streams: *newfeature1()* is in the *HEAD* stream and *bugfix1()* is in the *Maintenance* stream.
- ▶ The merge features do not work correctly in this view, so manually add the *bugfix1()* method to the version in the workspace. Save the changes in the merge editor.
- ▶ Perform a synchronization operation. Release the changes to *TeamTest1* and enter a comment of *Merged changes from maintenance*. Click *OK*.
- ▶ Open the *Resource History* view for the file. A new version 1.7 has been created in the repository containing the merged changes.

Developer 1: Delete maintenance stream

Finally, to clean up, switch back to the Developer 1 view and delete the *Maintenance* stream.

Close both IDEs and reopen Application Developer using the normal workspace to continue development for the next chapter.

Additional team topics

The following sections discuss some advanced topics relating to team development with Application Developer and CVS.

Determining which files are managed

There are two mechanisms you can use to determine which files are managed by the repository.

- ▶ The global ignore facility, provided by the workbench
 - To add a global ignore pattern for the workbench, simply open the Workbench *Window -> Preferences* dialog and select the *Team -> Ignored Resources* category. Click *Add* to add the pattern to ignore.
- ▶ The CVS ignore facility, provided by a file called *.cvignore* in the file system
 - The *.cvignore* file should be added to the workspace directory of the project whose files you wish to ignore. A good example would be to ignore all of the files stored in the *classes* folder of the ITS0MightyWeb project. In this case, create a *.cvignore* file in the *ITS0MightyWeb/webApplication/WEB-INF* directory containing the line:

```
classes
```

Note the ignore file only relates to the files and folders contained in that directory, so you may require many *.cvignore* files throughout the project. To share the ignores with the other developers in the team, simply release the *.cvignore* file into the stream.

A good practice is to only include source files in the project. For example, in the case of EJBs, store the generated types in the repository along with the generated metadata such as *ejb-jar.xml*, but do not store the JAR files or compiled classes.

Backing up the CVS repository

Probably the most important task for a repository administrator is to ensure that the CVS repository is backed up on at least a daily basis. Unlike VisualAge for Java's team repository, which required the administrator to create a copy of the locked repository file before the backup, the CVS repository directory structure can be backed up from its current location.

To restore from a backup, simply restore the files from the backup copy into their previous directory structure. If there are no backup facilities available, create a scheduled batch script that creates a timestamped zip file of the entire CVS directory and copies it to another machine.

Repository management

When working with CVS, it is generally recommended not to archive and remove old versions of projects from the repository. This is one of the major limitations of the product.

There is a feature in the command line tools installed with CVS, invoked by:

```
cvs admin -o
```

This command can remove either a specific version number or a range of versions. However, extreme caution is recommended as such changes cannot be backed out.

Implementing security

In our current installation, all valid users of the Windows server have unlimited access to read and write files in the CVS repository. This is unlikely to be a satisfactory scenario.

The creation of two files is all that is required to implement tighter security for an individual repository. On the server, create two text files called `readers` and `writers` in the root directory of the CVS repository.

These files should contain a new line-separated list of users that you want to grant that level of permission. If no `writers` file is supplied, then the CVS server implicitly assumes that all users have both read and write permission. If only the `writers` file exists, CVS grants all users read permission and only the specified users write permission.

Note that you should not include the same user in both files—only in the file with the higher permission level you want to provide. Putting the same name in both `readers` and `writers` assumes that you want to only give read access.

Unfortunately, there is no facility in CVS to limit which users have capability of versioning projects or creating new streams. Rational ClearCase may be a more suitable alternative to CVS if an organisation has these requirements.

Build scripts

Because CVS has a command line interface, it is easy to develop a batch client that retrieves the latest versions from a particular stream to a directory structure using `cvs update -q`.

A series of command line tools can then be invoked, such as the Java compiler, WebSphere command line tools (WSCP, XMLConfig) or Apache Jakarta Ant scripts to build and deploy an application either to a test server, or even to automate the process of releasing a build.

These topics are outside the scope of this document, but are documented in some detail in the *WebSphere Version 4 Application Development Handbook*, SG24-6134.

Managing class paths

One of the problems we have often ran into when developing this redbook is how to do team development with CVS when each developer has their workspace in a different directory or drive. A number of the files which are included in the repository by default include metadata containing drive and directory names.

While the simplest solution is to ensure everyone is working from an identical workspace path, there are some workarounds for the following files:

.classpath This file contains the class path definition for the project. We have found that it is easier to define a Application Developer class path variable such as WSADHOME and WSHOME to store the root directory of the installation and the current workspace the developer is using on each developer workstation, and then use these to refer to the path of specific JAR files used in the build path.

ibm-application-ext.xmi

This file is used in an EAR project to contain binding information between the EAR project and its contained Web and EJB projects, and uses absolute paths to define them. A symptom of this problem is that the *Run on Server* menu is disabled in the contained projects. To quickly resolve this, simply remove and then add the contained project using the editor for *application.xml*, which then regenerates this file.

server-cfg.xml This is the file likely to cause the most problems, because it contains a number of references to class paths for JDBC drivers, and the WebSphere transaction log file, which may be different for each client. We recommend that if developers have different workspace configurations, do not version Server projects but redefine them manually on each developer desktop with the appropriate class paths and JDBC drivers.

Using CVS macros

CVS provides a feature that enables developers to include macros in their source code to add features such as their user ID or timestamps when the file is released into a stream or project version. This feature is only available during CVS operations with text files. Unfortunately, the Application Developer CVS client implementation treats every resource as a binary file, and these facilities do not work with the beta code (this has not been tested with the final product code).

Watching a file for changes

Another common requirement by previous VisualAge for Java users was to be informed of changes made to a resource by other developers automatically when a release was performed. CVS provides limited support for this mode of operation, however, it must be performed using a command line CVS client.

Once the client has been installed, and a connection established to the repository, enter this command:

```
cvs watch add -a commit files
```

This command enables the default notification mechanism to inform the developer that a commit (release) has been performed on the file, usually through e-mail. To remove watches on the files, enter the command:

```
cvs watch remove -a commit files
```

And finally, to find out who else is watching those files, use:

```
cvs watchers files
```

Other CVS commands

For a comprehensive list of the extensive CVS commands, browse the on-line documentation available at <http://www.cvshome.org>.

Quiz: To test your knowledge of the Team development features of Application Developer, try to answer the following five questions. The answers are provided in Appendix C, “Quiz answers” on page 559.

1. Are file deletions from the workspace permanent?
2. What is a stream?
3. What are the two possible operations when synchronizing with a stream?
4. What is the most common protocol used with CVS servers?
5. How do you define the version number for a released resource?



Part 3

Web services

In part three we show you how to develop and deploy Web services using Application Developer.

We start with an introduction to Web services and the service oriented architecture (SOA). Then we give an overview on the IBM product support for Web services.

Next we demonstrate each of the steps you have to perform when Web service enabling an existing application. We use the wizards and tools available in Application Developer.

The hands-on part is structured into four chapters: static Web services, dynamic Web services, composed Web services, and advanced topics (advanced SOAP, advanced UDDI, WSFL, Web service support in other IBM products). We also cover the deployment process for Web services into WebSphere Version 4.0 Advanced Edition (AE), including the Single Server (AEs).

In the remaining chapters, we discuss architecture and design issues, including runtime aspects such as network topologies. We also give an outlook on how the SOA will evolve.



Web services overview and architecture

Our objective in this chapter is to introduce the technology underlying the Application Developer Web services tools and to define the terminology we will use in the following chapters.

This is a product neutral introduction; the IBM product support for Web services will be introduced in “Product support for Web services” on page 307. We assume that your development environment is Java based, however.

The chapter is structured in the following way:

- ▶ Business rationale and technical motivation for Web services
- ▶ Overview on the service oriented architecture (SOA) and its building blocks (SOAP, WSDL, and UDDI), including a first example
- ▶ Discussion on development approaches
- ▶ A closer look at the SOAP, WSDL, and UDDI specifications

The first three sections should give you sufficient information to start implementing the auto parts sample application. The remaining sections cover SOAP, WSDL, and UDDI in detail.

The SOA is based on XML. We will not introduce XML here; for an introduction to XML including schemas and namespaces, refer to “An XML primer” on page 92.

Motivation

Let us start with an excerpt from the XML Protocol activity charter of the World Wide Web Consortium (W3C):

"Today, the principal use of the World Wide Web is for interactive access to documents and applications. In almost all cases, such access is by human users, typically working through Web browsers, audio players, or other interactive front-end systems. The Web can grow significantly in power and scope if it is extended to support communications between applications, from one program to another."

As the protocol activity charter points out, business to business (B2B) has significantly more potential than business to consumer (B2C), as various reports show.

However, most Web clients still are human beings browsing linked documents, downloading files, or manually initiating transactions. More automated solutions face technical problems such as:

- ▶ Existing integration approaches lack flexibility because they follow a tight coupling approach.
- ▶ Problems occur if firewall boundaries have to be crossed because the security administrators usually only open the HTTP port (port 80).
- ▶ There are software distribution problems when protocols such as RMI/IOP are used. For example, the EJB client JAR file has to be made available on the Web client.

As a solution to these problems, program to program communication over HTTP is a common design pattern; up to now, the message exchange format typically was proprietary and not necessarily XML based. There were no means of formally describing the messages. Moreover, there are no general purpose B2B directories serving systems; the common existing Web directories are designed for the human user (for example, Yahoo, and so forth).

The *Web services* abstraction and the *service oriented architecture* (SOA) address these issues, allowing IT systems to become clients of Web applications.

Technical foundation

Web services combine the power of two ubiquitous technologies: XML, the universal data description language, and the HTTP transport protocol widely supported by browser and Web servers:

$$\text{Web services} = \text{XML} + \text{HTTP}$$

Now what is a Web service anyway? As you might already have observed, *Web service* recently has become one of the highly overloaded terms in the IT industry. In this redbook, we use the following widely accepted definition:

Web services are self-contained, self-describing, modular applications that can be published, located, and invoked across the Web.

Let us further explain the information contained in this definition.

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 a Web server and a servlet engine are required. It is possible to Web service enable an existing application without writing a single line of code.

Web services are self-describing

- ▶ Neither the client nor the server knows or cares about anything besides the format and content of request and response messages (loosely-coupled application integration).
- ▶ The definition of the message format travels with the message; no external metadata repositories or code generation tool are required.

Web services are modular

- ▶ Some introductions to Web services state that the Web services standards framework provides a transport mechanism as well as a component model, which is suited for the macro and micro level of your system design.
- ▶ We do not fully agree: in our view, Web services are a technology for *deploying and providing access* to business functions over the Web; J2EE, CORBA and other standards are technologies for *implementing* these Web services.

Web services can be published, located, and invoked across the Web

- ▶ Some additional standards are required to do so:
 - SOAP, the *Simple Object Access Protocol*, also known as service-oriented architecture protocol, an XML based RPC and messaging protocol
 - WSDL, the *Web Service Description Language*, a descriptive interface and protocol binding language
 - UDDI, *Universal Description, Discovery, and Integration*, a registry mechanism that can be used to lookup Web service descriptions

In this chapter, we introduce each of these technologies.

At this point readers typically react in either one of two entirely different ways: they either agree that Web services provide a highly promising vision, or they believe that this is just yet another distributed computing approach. Let us move further.

Web services are language independent and interoperable

- ▶ Client and server can be implemented in different environments. Existing code does not have to change in order to be Web service enabled.
- ▶ In this redbook, 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 standards based

- ▶ XML and HTTP are the 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 this time.

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. We say *can* because for example you will certainly not bind the accounting system of your company to a new online corporate banking provider you have just dynamically discovered.

Web services are composable

- ▶ 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 build on proven mature technology

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

By now you are hopefully just as excited about this emerging technology as we are. If not, please carry on reading anyway. We hope that you will be at the end of the book. As Web services receive a lot of attention at the moment, chances are you will have to understand and be able to position this technology soon.

Let us now introduce the overall architecture of the Web services technology framework.

Introduction to the service oriented architecture

In this book, we refer to the Web services suite of standards, APIs, and implementations, as *Web services technology framework* or *service oriented architecture (SOA)*.

This is just a short introduction to the key concepts in the SOA. We focus on the aspects that you as a Web services application developer will be confronted with. For more information, refer to:

<http://www-4.ibm.com/software/solutions/webservices/resources.html>

This Web site provides a collection of IBM resources on the topic at hand. For example, you can find an introduction to the SOA in a white paper titled “Web Services Conceptual Architecture (WSCA 1.0)”.

Service roles

Let us first introduce the roles a business and its Web service enabled applications can take in the SOA. Three roles can be identified (Figure 8-1):

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

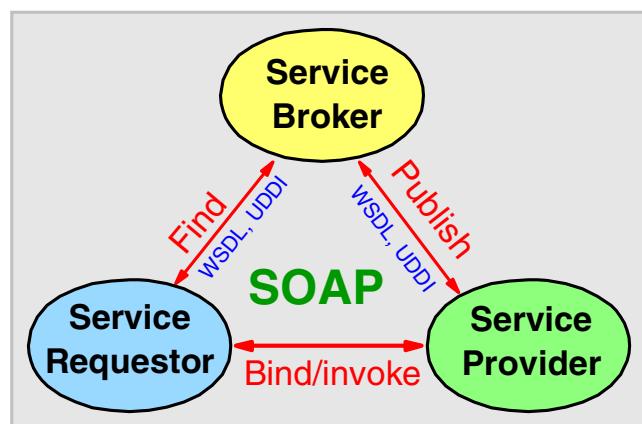


Figure 8-1 Web services roles and operations

The service broker does not have to be a public UDDI registry; as discussed in “Development strategies for provider and requestor” on page 255, there are other alternatives; for example, a direct document exchange link between service provider and requestor.

As we will see in “WSDL primer” on page 277, a WSDL specification consists of two parts, the service interface and the service implementation. Hence, *service interface provider* and *service implementation provider* are the two respective sub-roles for the service provider. The two roles can, but do not have to be taken by the same business.

Next, we investigate the hierarchy of build and runtime technologies in more detail.

SOA stack

Figure 8-2 shows how SOAP, WSDL, and UDDI fit into the overall Web services protocol and standards stack.

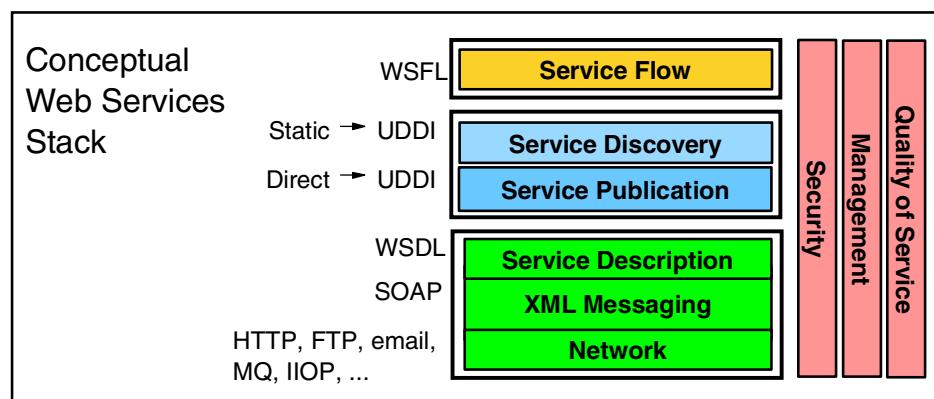


Figure 8-2 Web services protocol stack

- SOAP is a network, transport, and programming language neutral protocol that allows a client to call a remote service. The message format is XML.
- WSDL 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, as well as the parameters and data types of these operations. A WSDL document also contains the service access information.
- UDDI is both a client side API and a SOAP-based server implementation which can be used to store and retrieve information on service providers and Web services.

- ▶ The upper layer of the stack contains the Web services flow language (WSFL). WSFL specifications formally describe workflows that include Web service nodes. This stack layer is not yet as maturely defined as the lower layers, but will evolve over time.

The columns on the right hand side of the diagram contain the cross layer operational building blocks such as security, management, and quality of service. This book mainly focusses on development aspects; we discuss some of the operational aspects in “Architecture and design considerations” on page 513.

Here is a first glance at the relationship between the core elements of the SOA (Figure 8-3):

- ▶ All elements use XML including XML namespaces and XML schemas
- ▶ Service requestor and provider communicate with each other via SOAP
- ▶ UDDI access is performed through SOAP over HTTP
- ▶ WSDL is one alternative to make service interfaces and implementations available in the UDDI registry
- ▶ WSDL is the base for SOAP server deployment and SOAP client generation

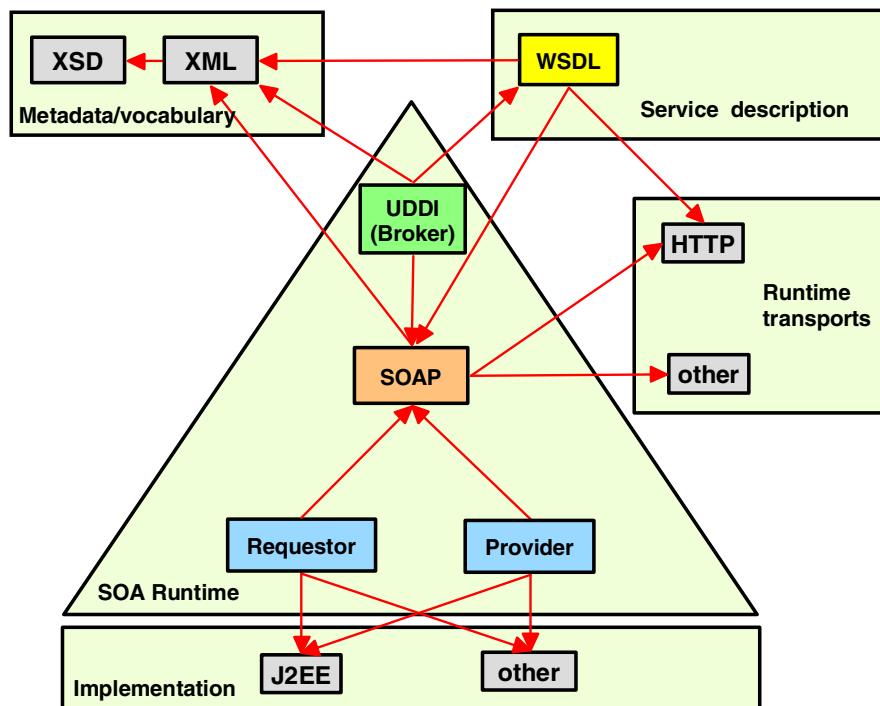


Figure 8-3 Building blocks of the SOA and their relationships

This is a simplified and incomplete view. For example, SOAP is transport neutral and WSDL supports more bindings than the one for SOAP. UDDI does not prerequisite WSDL.

In “The web of Web service standards” on page 304, we revisit the elements of the SOA, providing more insight on the relationships between them.

First examples

Here are some examples for macro and micro-level Web services:

- ▶ A news clipping service, with which library systems can contact Web portals when researching for publications containing certain keywords (system initiated Web crawling).
- ▶ A travel agent application comparing the ticket prices of different airlines.
- ▶ A CD track listing service that audio players and record companies connect to.
- ▶ An Excel/COM based insurance policy calculation sheet that is supposed to support a Web interface.
- ▶ Business information with rich content such as weather reports, news feeds, airline schedules, and stock quotes.
- ▶ Transactional Web services for B2B or B2C (airline reservations, rental car agreements).
- ▶ Business process externalization, providing business linkages at a workflow level, allowing complete integration at a process level.
- ▶ Use your imagination to come up with examples related to your business and project domain.

Let us examine a simple micro level Web service. Assume there is an existing Java based IT system calculating the current exchange rates between foreign currencies. It would be nice to make this functionality available as a Web service that any application with Web access could invoke.

Note: In fact this Web service exists on the Web today. XMethods acts both in service provider and broker role, as we will see in “Existing registries” on page 300. Also note that Graham Glass uses the same example for his “The Web service (r)evolution” series of articles on developerWorks:

<http://www.ibm.com/developerworks/webservices/library/ws-peer4/>

The articles use an outdated version of the WSTK and Tomcat; here we will work with the Application Developer tool and the WebSphere test environment.

Let us start with the Java code the service requestor and the service provider have to implement (Figure 8-4).

```
// ** service provider (service implementation running on SOAP server):
public class Exchange {
    public float getRate(String country1, String country2) {
        System.out.println("getRate(" + country1 + ", " + country2 + ")");
        return 1.45F; // fixed value for now
    }
}

// ** service requestor (any client with SOAP API support):
public class ExchangeClient
{
    public static void main( String[] args ) throws Exception {
        ExchangeProxy exchange = new ExchangeProxy(
            "urn:Exchange", "http://localhost/servlet/rpcrouter");
        float rate = exchange.getRate("USA", "Germany");
        System.out.println( "rate = " + rate );
    }
}
```

Figure 8-4 Java implementation of service requestor and provider

You might be surprised not to see more protocol specific code. As a matter of fact, this is all you have to code when implementing a requestor/provider pair.

On the provider side, the only link between this service implementation and the SOAP server hosting it is a configuration file, and the deployment descriptor, which we will introduce in “SOAP server” on page 270.

On the requestor side, the code using the SOAP API to bind to and invoke the service is encapsulated in the ExchangeProxy class. This client stub class is typically generated by tools, as we will see in “Development steps” on page 254. We will introduce the SOAP RPC client API used by the proxy in “SOAP client API” on page 272. The client is simple. First it passes addressing information: a SOAP server URL (<http://localhost/servlet/rpcrouter>) and a service ID (`urn:Exchange`). Then it invokes a method very much in the same way as if it was executing in the same address space as the server.

The information required by client proxy and deployment descriptor can formally be specified in WSDL. Figure 8-5 contains a snippet of the description of the Exchange service.

```

// ** Excerpt from WSDL specification (interface)
<message name="getRateRequest">
    <part name="country1" type="xsd:string"/>
    <part name="country2" type="xsd:string"/>
</message>
<message name="getRateResponse">
    <part name="result" type="xsd:float"/>
</message>
<portType name="Exchange">
    <operation name="getRate">
        <input name="getRateRequest" message="tns:getRateRequest"/>
        <output name="getRateResponse" message="tns:getRateResponse"/>
    </operation>
</portType>
<binding name="ExchangeBinding" type="tns:Exchange">
    <soap:binding style="rpc"
                  transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="getRate">
        <soap:operation soapAction="urn:Exchange" style="rpc"/>
        <input>
            ...
        </input>
        <output>
            ...
        </output>
    </operation>
</binding>

// ** Excerpt from WSDL specification (implementation)
<service name="ExchangeService">
    <port name="ExchangePort" binding="binding:ExchangeBinding">
        <soap:address>
            location="http://localhost/servlet/rpcrouter"/>
    </port>
</service>

```

Figure 8-5 WSDL specification of a Web service

Note that such a description can be generated by a tool from the Java service implementation, as we will discuss in “Development steps” on page 254.

The Java class is represented as a port type, the method as an operation. The method parameters are translated into message parts with matching data types. A binding and a service definition contain access information, such as a service ID and the URL of the SOAP server hosting the service. Note that all definitions are abstract because WSDL is language independent; in the SOA, the Java mapping is performed by the SOAP runtime.

You can find more information on these modelling concepts in “WSDL primer” on page 277; we will complete the WSDL specification there as well.

Finally, a UDDI registration for this Web service could contain the entities shown in Figure 8-6.

```
// ** Excerpt from the UDDI registration
<businessService serviceKey="81FFBE10-9CE6-11D5-BA0D-0004AC49CC1E"
<name>ExchangeService</name>
<description xml:lang="en">Exchange Rate Service</description>
<bindingTemplates>
<bindingTemplate bindingKey="82031970-9CE6-11D5-BA0D-0004AC49CC1E"
<accessPoint URLType="http">http://localhost/servlet/rpcrouter</accessPoint>
<tModelInstanceDetails>
<tModelInstanceInfo tModelKey="UUID:7CEF0CF0-9CE6-11D5-BA0D-0004AC49CC1E">
<instanceDetails>
<overviewDoc>
<overviewURL>http://localhost/wsdl/Exchange-service.wsdl</overviewURL>
</overviewDoc>
<instanceParms><port name="ExchangePort" binding="ExchangeBinding"/>
</instanceParms>
</instanceDetails>
</tModelInstanceInfo>
</tModelInstanceDetails>
</bindingTemplate>
</bindingTemplates>
</businessService>
```

Figure 8-6 Excerpt from the UDDI registry entry for the Exchange service

- ▶ A business entity, describing the company providing the Exchange service (not displayed in the figure).
- ▶ An informal description of the semantics of the Exchange service (businessService entity).
- ▶ An informal technical description of the syntax of the Exchange service (tModel), including a URL reference to the formal specification in the WSDL interface document.
- ▶ An access locator describing how to access the service, including a URL pointing to the SOAP server hosting the service, and a URL reference to the WSDL implementation document (bindingTemplate).

For an introduction to these entities, refer to “UDDI overview” on page 293.

Implementations of the SOA

Even if this introduction to the SOA is product neutral, let us briefly list the implementation alternatives that are available today.

Java solutions

There is a wide support for Web services in the Java world, both from vendors and the open source community.

Open source

The following open source projects focus on elements of the service oriented architecture:

- ▶ The Apache XML project, with subprojects for Xalan, Xerces, and SOAP:
<http://xml.apache.org/>
- ▶ Apache AXIS, the second generation SOAP implementation:
<http://xml.apache.org/axis/>
- ▶ WSDL4J and UDDI4J, hosted at IBM developerWorks:
<http://www.ibm.com/developerworks/projects/wsdl4j/>
<http://www.ibm.com/developerworks/projects/uddi4j/>

IBM

See “Product support for Web services” on page 307 for an overview of the IBM product support for Web services. All standards and development steps we have introduced are supported.

IBM has played an important role in the standardization effort:

- ▶ XML schema—author of the primer, contributor in all phases
- ▶ SOAP—co-author of the specification, first implementation, chair of XML Protocol working group in W3C, contributed SOAP4J to the Apache open source project
- ▶ WSDL—co-author of specification, first WSDL toolkit on alphaWorks
- ▶ UDDI—co-designer, leader in creation of the UDDI project, host of the UDDI Business Registry and the UDDI Business Test Registry

Non Java solutions

One of the major design goals of the SOA is to be language independent. The two most important non-Java environments are Microsoft’s .NET approach and SOAP Lite, which supports Perl.

Microsoft .NET

The .NET platform is Microsoft's C# based XML Web services platform:

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

SOAP Lite

SOAP::Lite for Perl is a collection of Perl modules, which provide a simple and lightweight interface to the Simple Object Access Protocol (SOAP) both on client and server side:

<http://www.soaplite.com/>

More information

James Snell gives an overview on the SOAP implementation alternatives in the first part of his *Web services insider* series of articles on developerWorks:

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

What is next

Now that we have learned about the core technologies of the Web service SOA, let us take a look at the development process for service providers and requestors.

Developing Web services

We give only a brief overview on development aspects here, covering:

- ▶ General development steps
- ▶ Strategies for the service provider and the service requestor
- ▶ Service life cycle

For a more comprehensive discussion of the topic, see the document "Web Services Development Concepts 1.0" from the Web services development toolkit. It is also available under the following URL:

<http://www-4.ibm.com/software/solutions/webservices/pdf/WSDC.pdf>

Development steps

During the build cycle of any project involving Web services, certain steps have to be performed. Figure 8-7 illustrates them in a pseudo-UML collaboration diagram.

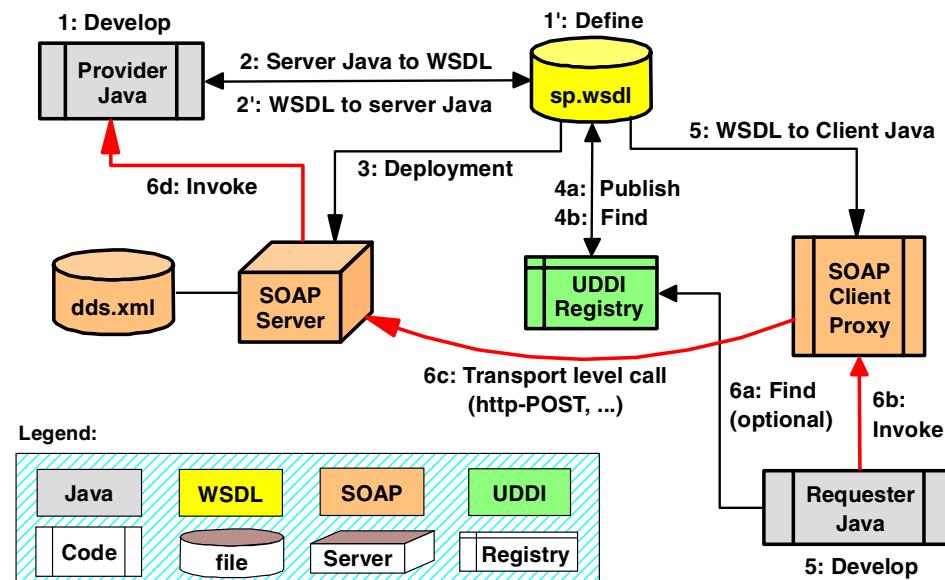


Figure 8-7 Web services development and runtime information flow

Let us walk through the diagram:

- ▶ Service provider side Java development (1)

- ▶ Conversion of existing programming language artifact into WSDL interface and implementation document (2)
- ▶ Alternatively, you can start with the definition of the WSDL specification (1') and generate server side Java from it (2'). We describe the two alternatives in “Development strategies for provider and requestor” on page 255.
- ▶ Generation of server side SOAP deployment descriptor from WSDL specification (3)
- ▶ UDDI export and import (4)
 - Publishing, unpublishing, and update of service registrations in the UDDI registry (4a)
 - Finding service interface registrations in the UDDI registry (4b)
- ▶ Generation of client side SOAP stub and optional generation of Web service test clients (5)
- ▶ Service invocation and execution (6)
 - Dynamic lookup of service providers in the UDDI registry (provider dynamic requestor only, see below for definition) (6a)
 - Service requestor side service invocation (6b)
 - Transport level communication between requestor and provider (6c)
 - Service provider side service invocation (6d)

Even if all specifications are human readable (XML), there is a strong need for tools supporting these development steps—a lot of documents with overlapping content are involved. It would be cumbersome and error prone to define all these files without tools.

“Product support for Web services” on page 307 will introduce the Application Developer product support for these six Web service development steps.

Development strategies for provider and requestor

Let us next investigate how implementations for the three roles defined in the SOA can be developed.

Service provider

A service provider can choose between three different development styles when defining the WSDL and the Java implementation for his/her Web service:

Top down

When following the top down approach, both the server and client side Java code are developed from an existing WSDL specification.

Bottom up	If some server-side Java code already exists, the WSDL specification can be generated from it. The client side Java proxy is still generated from this WSDL document.
Meet in the middle	The meet in the middle (MIM) development style is a combination of the two previous ones. There are two variants:
MIM variant 1	Some server side Java code is already there, its interface however, it is not fully suited in order to be exposed as a Web service. For example, the method signatures might contain unsupported data types. A Java wrapper is developed and used as input to the WSDL generation tools in use.
MIM variant 2	There is an existing WSDL specification for the problem domain; however, its operations, parameters, and data types do not fully match with the envisioned solution architecture. The WSDL is adopted before server side Java is generated from it.

In the near future, we expect most real world projects to follow the meet in the middle approach, with a strong emphasis on its bottom up elements. This is MIM variant 1, starting from and modify existing server side Java and generate WSDL from it.

When developing the auto parts sample application throughout the following chapters, we will start with the bottom up approach. Refer to “Static Web services” on page 321.

We will demonstrate the top down approach later on as well. See “Composed Web services” on page 413.

Service requestor

Web service clients (service requestor implementations, that is) have to import the WSDL interface and implementation specification of the Web service to be invoked into their environment.

Three types of requestors can be identified. They import interface and implementation information at different points in time (build time vs. runtime):

Static service	No public, private or shared UDDI registry is involved; the service requestor obtains service interface and implementation description through a proprietary channel from the service provider (an e-mail, for example), and stores it into a local configuration file.
-----------------------	---

Provider-dynamic	The service requestor obtains the service interface specification from a public, private or shared UDDI registry at build time and generates proxy code for it. The service implementation document identifying the service provider is dynamically discovered at runtime (using the same or another UDDI registry).
Type-dynamic	The service requestor obtains both the service interface specification and the service implementation information from a public, private or shared UDDI registry at runtime. No proxy code is generated; the service requestor directly uses the more generic SOAP APIs to bind to the service provider and invoke the Web service.

Service broker

Since we do not expect that application developers implement UDDI registries themselves, we do not discuss any related development issues in this document.

UDDI registries are provided in two forms:

- ▶ Public registries, such as the IBM UDDI Business Registry and the IBM UDDI Business Test Registry

<http://www.ibm.com/services/uddi/protect/registry.html>

<http://www.ibm.com/services/uddi/testregistry/protect/registry.html>
- ▶ Private registries, such as the IBM WebSphere UDDI Registry (currently preview or beta code), a stand-alone product that can be installed for an intranet solution

<http://www7b.boulder.ibm.com/wsdd/downloads/UDDIregistry.html>

Level of integration between requestor and provider

In a homogeneous environment, client and server (requestor and provider, that is) use the same implementation technology possibly from the same vendor. They might even run in the same network.

In such an environment, runtime optimizations such as performance and security improvements are possible. We expect such additional vendor specific features to become available as the Web services technology evolves.

We do not recommend to enable such features though, because some of the main advantages of the Web service technology such as openness, language independence, and flexibility can no longer be exploited. Rather you should design your solution to loosely couple requestor and provider, allowing heterogeneous systems to communicate with each other.

Service life cycle

During its lifetime, a Web service always is in one of the states shown in Figure 8-8.

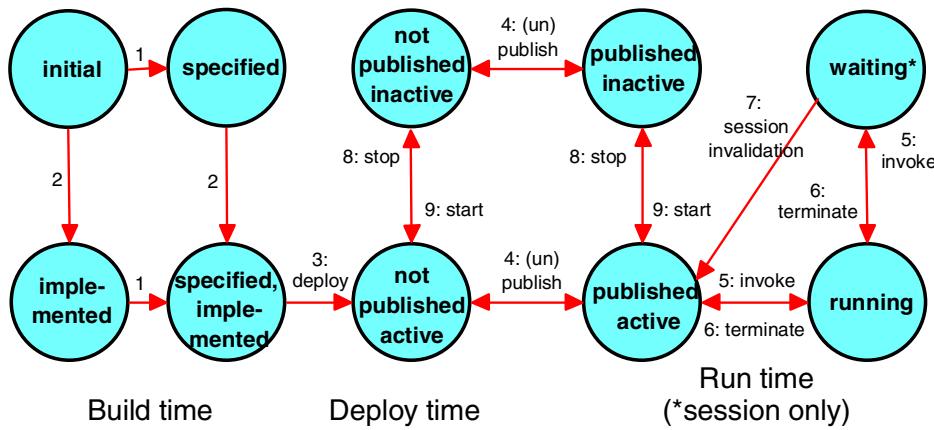


Figure 8-8 Service lifecycle

Specified

The application developers of the service provider are currently in the process of Web service enabling the IT system of the service provider. A WSDL specification is already there.

Implemented

The application developers of the service provider are currently in the process of Web service enabling the IT system of the service provider. A server-side Java implementation of the Web service exists.

Specified, Implemented

Both a WSDL file and a server side implementation of the Web service are available.

Not published, active

The Web service is deployed in the SOAP server.

Published, active

The Web service is both deployed and published in a UDDI registry.

Not published, inactive

The Web service is deployed, but currently not ready for execution. It is not published, either.

Published, inactive

The Web service is published, but currently it is not ready for execution.

Running

A client is using the service. The server is currently processing a service invocation.

Waiting

A client is currently using the service, however, the last invocation has already terminated. The Web service instance waits in the transport session for its next invocation (session scope only). The concept of Web service scope will be introduced in “Server deployment” on page 271.

The state transitions are shown in Table 8-1.

Table 8-1 State transitions in the service lifecycle

ID	Step	From	To
1	WSDL creation	initial or implemented	specified or specified/implemented
2	Java implementation	initial or specified	implemented or specified/implemented
3	Deployment	specified/implemented	not published active
4	UDDI publishing	the two not published states	the two published states
5	Service requestor finds, binds, and invokes service via SOAP	published active	running
6	Service provider implementation terminates	running	to published active if scope is <i>application</i> or <i>request</i> , to waiting otherwise
7	Session invalidation	running	to published active
8	Stop operation (SOAP server admin GUI)	active	inactive
9	Start operation (SOAP server admin GUI)	inactive	active

Note that state transition diagram and table only show the recommended *sunny day scenario*. For instance, it would be possible to already publish from the specified/implemented state; this would not make sense, however, because any clients finding and binding to the service would be returned an error message.

For the sake of simplicity of the diagrams, the undeploy operation is not shown.

What is next

You have two options for how to continue to read this book now: if you want to start implementing immediately and to see Application Developer tool in action, jump to “Static Web services” on page 321 now.

If you prefer to learn more about the SOAP, WDSL, and UDDI concepts before starting to code, proceed with “An introduction to SOAP” on page 261, “WSDL primer” on page 277, and “UDDI overview” on page 293. Even if you skip these sections now, you can always come back and use them as a reference during development.

An introduction to SOAP

The current industry standard for XML messaging is SOAP. IBM, Microsoft and others submitted SOAP to the W3C as the basis of the XML Protocol Working Group.

SOAP has the following characteristics:

- ▶ SOAP is designed to be simple and extensible.
- ▶ All SOAP messages are encoded using XML.
- ▶ SOAP is transport protocol independent. HTTP is one of the supported transports. Hence, SOAP can be run over existing Internet infrastructure.
- ▶ There is no distributed garbage collection. Therefore, call by reference is not supported by SOAP; a SOAP client does not hold any stateful references to remote objects.
- ▶ SOAP is operating system independent and not tied to any programming language or component technology. It is object model neutral.

Due to these characteristics, it does not matter what technology is used to implement the client, as long as the client can issue XML messages. Similarly, the service can be implemented in any language, as long as it can process XML messages.

In this section, we discuss both the W3C SOAP 1.1 specification and the SOAP 2.2 implementation from Apache. Apache SOAP 2.2 implements most of the W3C SOAP 1.1 specification. There are other Java implementations, for example Apache AXIS, as well as non Java ones. See “Architecture and design considerations” on page 513 for more information.

Refer to “An XML primer” on page 92 for an introduction to the XML concepts used within SOAP.

Overview

Historically, SOAP was meant to be a network and transport neutral protocol to carry XML messages around. SOAP over HTTP became the premier way of implementing this protocol, to the point that the latest SOAP specification mandates HTTP support.

However, conceptually there is no limitation for the network protocol that can be utilized. For example, because HTTP is a transport that does not guarantee delivery and is non-transactional, SOAP messages can also be transferred by a messaging software such as MQSeries.

SOAP remote procedure call (RPC) is the latest stage in the evolution of SOAP; the body of a SOAP message contains a call to a remote procedure (expressed in XML) and the parameters to pass in (again, expressed in XML).

The SOAP standard specifies three aspects of XML based message exchange:

Overall message format

- ▶ 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 transport any 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.

Encoding rules

- ▶ *Encoding rules* define a serialization mechanism that can be used to exchange instances of application-defined data types.
- ▶ SOAP defines a programming language independent data type scheme based on XSD, plus encoding rules for all data types defined according to this model.

RPC representation

- ▶ The *RPC representation* is a convention suited to represent remote procedure calls and the related response messages.
- ▶ The usage of this convention is optional. If the RPC 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.

Anatomy of a SOAP message

Figure 8-9 shows an example of a SOAP message transported through HTTP:

- ▶ The HTTP header contains the URL of the SOAP server, which in this case is localhost/servlet/rpcrouter. Relative to this URL, the Web service is identified by urn:Exchange.
- ▶ The HTTP payload of the message is 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 getRate(country1, country2) of a Web service called urn:Exchange 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 both on the client and server side to XML (and vice versa).

```

POST /servlet/rpcrouter HTTP/1.0
Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: 494
SOAPAction: ""

<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <SOAP-ENV:Body>
        <ns1:getRate xmlns:ns1="urn:Exchange"
            SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
            <country1 xsi:type="xsd:string">USA</country1>
            <country2 xsi:type="xsd:string">Germany</country2>
        </ns1:getRate>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 8-9 A SOAP message

Namespaces

SOAP defines the XML namespaces shown in Table 8-2.

Table 8-2 SOAP namespaces

Prefix	Namespace URI	Explanation
SOAP-ENV	<code>http://schemas.xmlsoap.org/soap/envelope/</code>	SOAP envelope
SOAP-ENC	<code>http://schemas.xmlsoap.org/soap/encoding/</code>	SOAP serialization

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:Exchange` is the URN of our Exchange Web service.

The URN is used to route the request when it comes in. In the SOAP server implementation from Apache (see “Apache SOAP implementation” on page 270), dispatching is done based on the namespace URI of the first child element of the <SOAP:Body> element only. Therefore, this namespace must point to the URN of the service to be invoked.

All other addressing information is transport dependent. When using HTTP as transport, for example, the URL of the HTTP request points to the SOAP server instance on the destination host. For the exchange service, the URL can be <http://localhost:8080/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 EJB.

Envelope

The envelope has the following structure:

```
<SOAP-ENV:Envelope .... >
  <SOAP-ENV:Header name="nmtoken"> *
    <SOAP-ENV:HeaderEntry.... />0..*
  </SOAP-ENV:Header>
  <SOAP-ENV:Body name="nmtoken">
    [message payload]
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

We use an XML element oriented pseudo XSD syntax here, with 0 and * indicating cardinalities. In general, a SOAP message is a set of headers plus one body. Note that the message in Figure 8-10 on page 270 does not contain a SOAP header.

The SOAP envelope defines the namespace for structuring messages. The entire SOAP message (headers and body), is wrapped in this envelope; its main payload (the message content) appears in the body contained in this envelope.

Note that the message body uses a service specific namespace, urn:Exchange in our example (Figure 8-9 on page 263). 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 mechanism for adding features to a SOAP message without prior agreement between the communicating parties.

Headers are optional elements in the envelope; they allow to pass control information to the receiving SOAP server and provide extensibility for message structures as well. Moreover, headers serve as hooks for protocol evolution.

There is a predefined header attribute called `SOAP-ENV:mustUnderstand`. If it is present in a header element and set to “1”, the service provider must implement the semantics defined by the element:

```
<ns:qos xmlns:t="someURI" SOAP-ENV:mustUnderstand="1">
    3
</ns: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) level 3 (whatever qos=3 stands for in the actual invocation context). The second header attribute is `SOAP-ENV:actor`, identifying the recipient of the header information.

Another example are multi-hop messages. Headers can be used to address the intermediate actors forwarding multi-hop messages to their target destination. Headers can also carry authentication data, digital signatures, and encryption information, or transactional settings.

Body

In the most simple case, the body of a basic SOAP message comprises:

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

There are three message types:

- ▶ *Request messages* with in parameters invoking an operation on a Web service
- ▶ *Response messages* with an out parameter, used when returning from an invocation
- ▶ *Fault messages* communicating error situation

For all three message types, the same message format is used. The messages can contain almost any XML construct. Document type definitions (DTDs) and processing instructions are illegal, however.

Error handling

SOAP itself predefines one body element, which is the *fault element* used for reporting errors. The fields of the fault element are defined as follows:

- ▶ **Faultcode** is a code that indicates the type of the fault. The valid values are:
 - SOAP-ENV:Client, indicating incorrectly formatted messages
 - SOAP-ENV:Server, for delivery problems
 - SOAP-ENV:VersionMismatch, which can report any invalid namespaces for envelope element
 - SOAP-ENV:MustUnderstand for errors regarding the processing of header content
- ▶ **Faultstring** is a human readable description of the fault.
- ▶ **Faultactor** is an optional field that indicates the URI of the source of the fault.
- ▶ **Detail** is an application specific field that contains detailed information about the fault.

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

Advanced concepts

In this section, we discuss some more advanced SOAP concepts, such as the different message styles as well as the SOAP data model, the available encodings and the corresponding type mappings.

When beginning to work with SOAP, you will probably not immediately touch these concepts. Most of the Web service samples available on the Web do not feature them, either. When you are planning to implement a nontrivial Web service, however, you have to be familiar with them.

Therefore, feel free to skip this section when this is your first contact with Web services, and come back when the design issues in your project require you to change the default settings of the Application Developer wizards, which is quite likely to happen.

We will come back to these concepts in “Web services advanced topics” on page 447.

Communication styles

SOAP supports two different *communication styles*:

Remote procedure call (RPC)

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.

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, however, not yet frequently used communication style.

Messages, parameters, and invocation APIs look different for RPC and document. The decision about which style to use is made at build/deployment time; a different client API and server side router servlet are used.

The main focus of this redbook is on the RPC style. Unless explicitly stated otherwise, all explanations refer to the RPC style; the example given above and the auto parts sample application used throughout the book, use it as well.

Batch processing, applications with configurable parameter lists, form oriented data entry applications, and access to information stores such as patient records, are a few scenarios where message/document oriented communicating might be appropriate.

See “Web services advanced topics” on page 447 and the “*Apache SOAP User’s Guide*” for more information on the document-oriented style.

Data model

The purpose of the SOAP data model is to provide 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, date, and so forth.

Compound types There are two kinds of compound types, *structs* and *arrays*.

Structs Named aggregated types. Each element has a unique name, its *accessor*. An accessor is an XML tag. Structs are conceptually similar to records 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 as found in languages such as C and Java.

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 how data types can be constructed; a project specific XML schema has to define the actual data types.

A SOAP request message such as getRate in Figure 8-9 on page 263 is modelled as a struct containing an accessor for each in and in/out parameter:

```
<ns1:getRate xmlns:ns1="urn:Exchange">
    <country1 xsi:type="xsd:string">USA</country1>
    <country2 xsi:type="xsd:string">Germany</country2>
</ns1:getRate>
```

The accessors in this example are country1 and country2. The accessor names correspond to the names of the parameters, 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 SOAP data model makes self-describing messages possible. No external schema is needed to understand an XML element such as:

```
<country1 xsi:type="xsd:string">USA</country1>
```

SOAP provides a preferred encoding for all data types defined according to this model (see next section on Encodings).

Encodings

In distributed computing environments, *encodings* define how data values defined in the application can be translated to and from 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).

The protocol format for Web services is XML, and we assume that service requestor and provider are developed in Java. Thus, SOAP encodings tell the SOAP runtime environment how to translate from data structures constructed in Java into SOAP XML and vice versa.

The following encodings are defined:

- | | |
|----------------------|---|
| SOAP encoding | The SOAP encoding allows to marshall/unmarshall values of data types from the SOAP data model (see “Data model” on page 267). This encoding is defined in the SOAP 1.1 standard. |
| Literal XML | The literal XML encoding allows to directly convert existing XML DOM tree elements into SOAP message content and vice versa. This encoding style is not defined by the SOAP standard, but the Apache SOAP implementation. |

XMI	XMI stands for <i>XML metadata interchange</i> . This encoding style is defined by the Apache SOAP implementation. We will not work with this encoding in this book.
------------	--

The encoding to be used by the SOAP runtime can be specified at deploy time or runtime. If it is specified at deploy time, it appears in the WSDL specification of the Web service (see “WSDL primer” on page 277). Tools can then analyze this specification and configure the SOAP runtime to use the desired encoding.

At runtime, the SOAP client API (see “SOAP client API” on page 272) 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 (see “Server deployment” on page 271). The settings on the client and the server side have to match.

In “Encodings and type mapping alternatives” on page 448, we discuss this subject in more detail and give a practical example.

At runtime, it must be possible to encode and decode values of all the data types that are being used under a certain encoding scheme. Mappings address this requirement.

Mappings

A *mapping* defines an ternary association between a qualified XML element name, a Java class name, and one of the encodings as introduced above (note that this is the first time that we have to discuss Java specific aspects).

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`.

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 as well as JavaBeans are supported by default; for a list of predefined mappings for the following types and encodings, refer to “Encodings and type mapping alternatives” on page 448.

Non-standard (custom) data types require the introduction of *custom mappings* on the client and on the server side.

Apache SOAP implementation

Let us first investigate the SOAP server and then the SOAP client API.

SOAP server

Apache SOAP 2.2, which comes with Application Developer, provides an implementation for a SOAP server, allowing to deploy Web services and invoke methods on them.

Figure 8-10 gives an overview of the Apache SOAP server components:

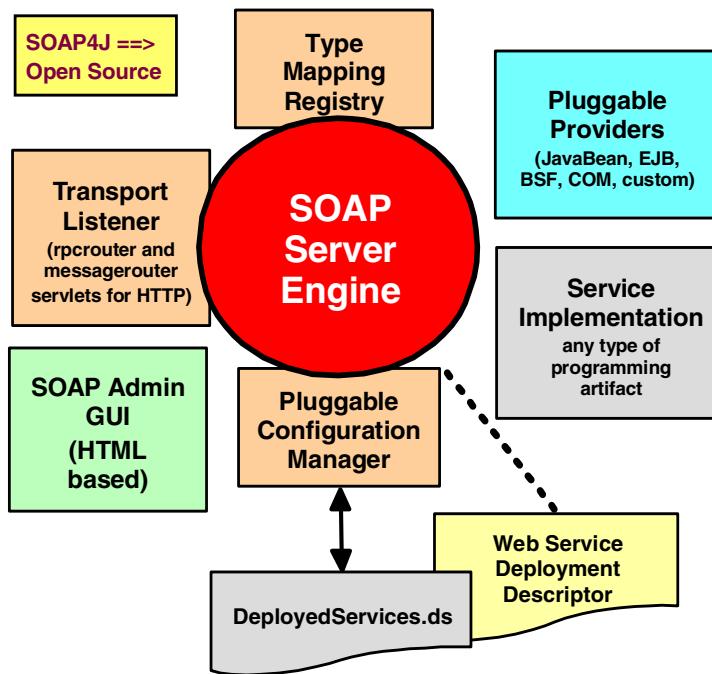


Figure 8-10 Components of Apache SOAP server implementation

For now, the important elements in this architecture are the rpcrouter and messengerrouter servlets, the deployment descriptor (explained below), 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 actually executing the invoked Web service. We do not go into details about the configuration manager, and the admin 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 which is supported by one of the Apache SOAP providers.
- ▶ Ensure that parameters to your method/function are serializable by SOAP, and exist within the SOAP type mapping registry. Develop and register custom mappings otherwise.
- ▶ Create an entry in the Apache SOAP deployment descriptor for your service.

A service implementation implementing the first step for the Exchange Web service is:

```
public class Exchange {  
    public float getRate(String country1, String country2) {  
        System.out.println("getRate(" + country1 + ", " + country2 + ")");  
        return 1.45F; // fixed value for now  
    }  
}
```

This is 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:Exchange" checkMustUnderstands="true">  
    <isd:provider type="java" scope="Request" methods="getRate">  
        <isd:java class="Exchange" static="false"/>  
    </isd:provider>  
</isd:service>  
</root>
```

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

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 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 case HTTP is the transport protocol).

The actual deployment step can either be performed using the admin GUI that comes with Apache SOAP (for more information, refer to “Static Web services” on page 321) or programmatically (see “Advanced SOAP programming” on page 448).

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:

Call	Contains the URN, the SOAP address of the router servlet on the SOAP servers well as the name of the method to be called. A call object contains Parameter instances as data members.
Parameter	Contains 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 (Figure 8-11). The message that travels if this code is executed is the same message we inspected in “Anatomy of a SOAP message” on page 262.

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.

```

public class MySoapClient {
    public static void main(String[] args){
        Call call = new Call();
        call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
        call.setTargetObjectURI ("urn:Exchange");
        call.setMethodName ("getRate");
        Vector params = new Vector();
        Parameter country1Param = new Parameter(
            "country1", String.class, "USA", Constants.NS_URI_SOAP_ENC);
        params.addElement(country1Param);
        Parameter country2Param = new Parameter(
            "country2", String.class, "Germany", Constants.NS_URI_SOAP_ENC);
        params.addElement(country2Param);

        call.setParams(params);
        Response resp = null;
        URL url = new URL ("http://localhost/soap/servlet/rpcrouter");
        resp = call.invoke (url, "urn:Exchange"); // 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 8-11 SOAP client

- ▶ 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 wish 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.

We will cover the SOAP API and the code Application Developer generates on top of it, shown in detail in “Static Web services” on page 321.

SOAP summary

Figure 8-12 contains a high level component model showing the conceptual architecture of both the service provider (SOAP server) and the service requestor (SOAP client).

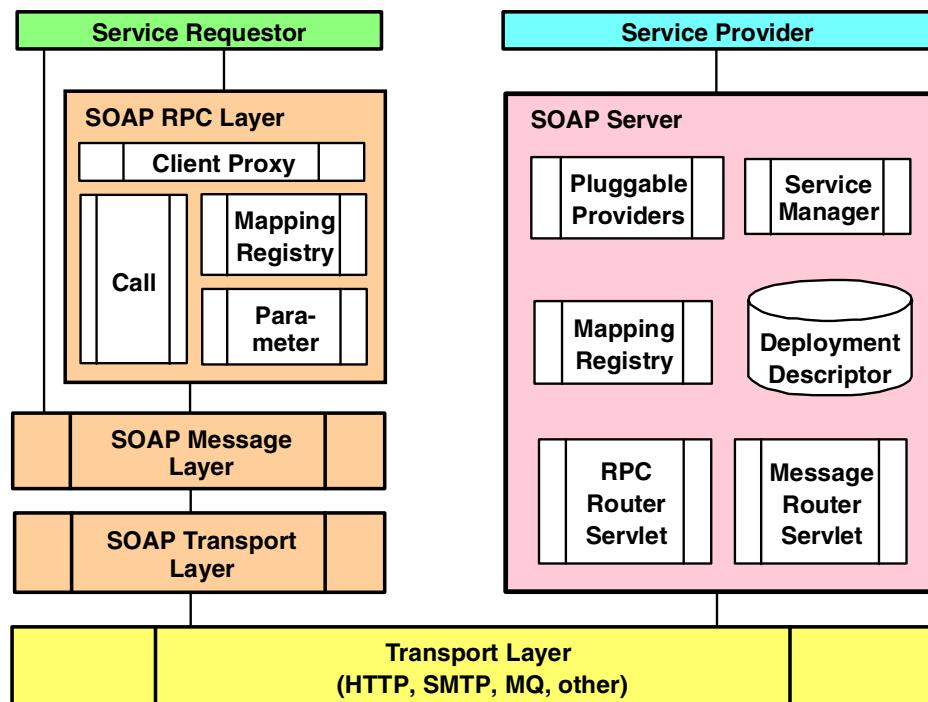


Figure 8-12 High-level SOAP component model

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

Figure 8-13 shows the interactions between client and server as well as the server side processing of a service invocation.

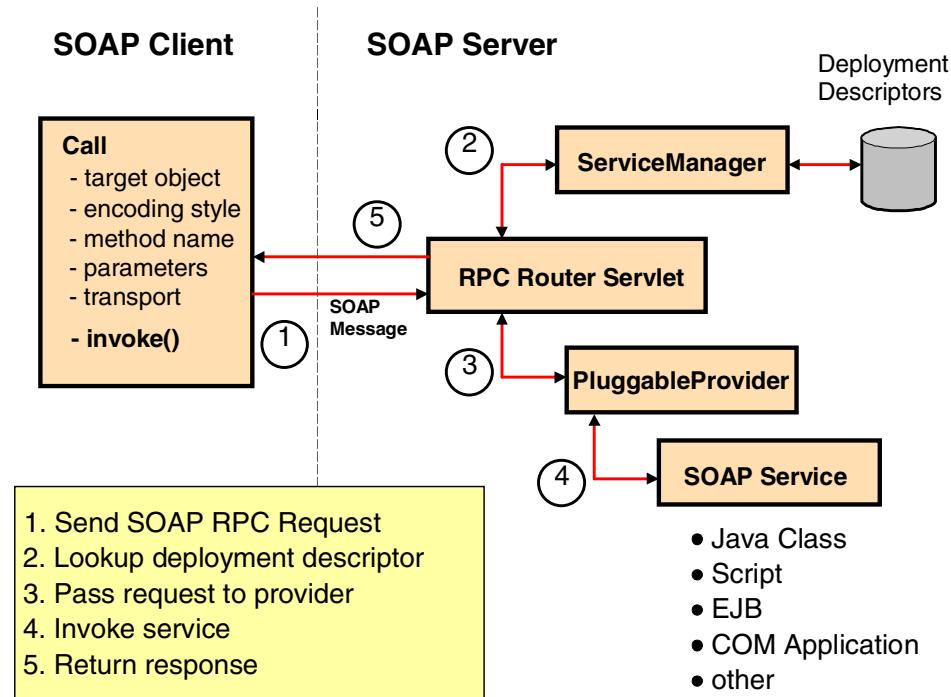


Figure 8-13 SOAP client and server interactions

Outlook

In the future, the SOAP 1.2 (currently work under progress under the project name *XML Protocol*) will follow SOAP 1.1 as the industry standard XML messaging protocol.

Some recent decisions made by W3C XML Protocol Working Group are:

- ▶ SOAP no longer is an acronym, but stands for itself.
- ▶ The use of SOAPActionURI in the HTTP header is likely to become discouraged or deprecated.

There are more open issues. Check <http://www.w3c.org/TR/soap12> for updates.

Regarding the SOAP implementation, the Apache AXIS project is currently implementing SOAP 3.x. Note that AXIS has a different code base than SOAP 2.2.

More information

The SOAP 1.1 specification can be downloaded from:

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

For information on Apache SOAP, check out the user and API documentation as well as the FAQ list:

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

In the near future, there will be a book on SOAP: “Programming Web Services with SOAP”, by James Sell, Ken Mulched, and Paul Kulchenko, O'Reilly & Associates, ISBN 0596000952.

What is next

SOAP enables the publication and invocation of Web services, but SOAP alone is not enough to build a Web services infrastructure.

Without WSDL, a client has no information about the interface of a service the server URL, the URN, the methods, the types and type mappings. Without UDDI, a client has no information about the existence and characteristics of services.

We will introduce these two technologies in the next two sections.

WSDL primer

If we want to find services automatically, we have to have a way to *formally* describe both their invocation interface and their location. The *Web services description language (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; its current status is *submission acknowledged* by the W3C.

Overview

WSDL allows 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. For an introduction to XML refer to “An XML primer” on page 92 in part one of this book.

A valid WSDL document consists of two parts:

1. The *interface* part describing the abstract type interface and its protocol binding.
2. The *implementation* part describing the service access (location) information.

Following the design principle *separation of concerns*, there is a convention to store these two parts in two separate XML files. This is possible because XML files can import each other. In this book, we refer to these two files as:

- ▶ *Service interface file or service interface document* (part 1)
- ▶ *Service implementation file or service implementation document* (part 2).

Service interface document

The interface document contains the following top-level elements:

- Port type** An abstract set of one or more *operations* supported by one or more *ports*. Each operation defines an *in* and an *out message* as well as an optional *fault message*.
- Message** An abstract, typed definition of the data being communicated. A message can have one or more typed *parts*.
- Type** A container for data type definitions using some type system such as XSD.
- 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.

There is a WSDL SOAP binding, which is capable of describing SOAP over HTTP, as well as other transport protocols. Moreover, you can also define Web services that have a direct HTTP interface (any plain servlet, for example). WSDL 1.1 also contains a MIME binding.

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 specification (XSD).

Service implementation document

The service implementation document (*service implementation file*) contains the following elements:

- Service** A collection of related ports
- Port** A single *endpoint*, which is defined as an aggregation of a binding and a network address

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 to interface description languages, such as CORBA IDL, which have language bindings.

Next, we investigate these elements in detail.

Anatomy of a WSDL document

Figure 8-14 shows the elements comprising a WSDL document as well as the various relationships between them:

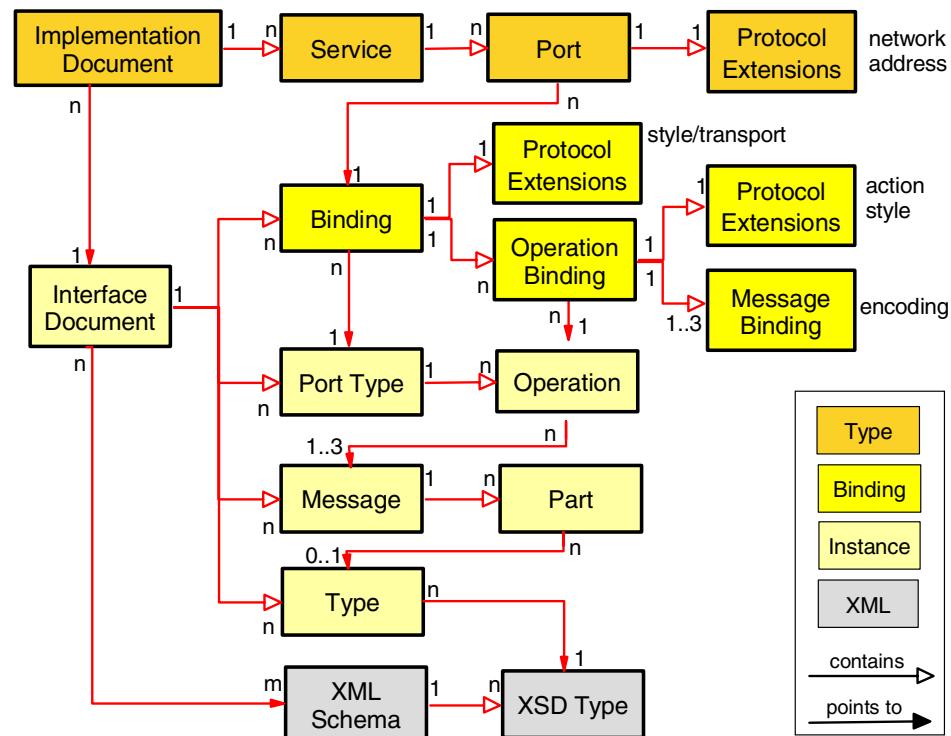


Figure 8-14 WSDL elements and relationships

The diagram should be read in the following way:

- ▶ One service interface document can be imported by several service implementation documents. Hence, there is an $n:1$ pointer relationship between the implementation and the interface document.
- ▶ A service contains one or more port definitions (service endpoints). Hence, there is a $1:n$ containment relationship between the service and the port element.
- ▶ Interpret other connectors in the same way.

The containment relationship shown in the diagram directly map to the XML schema for WSDL.

Example

Let us now give an example of a simple complete and valid WSDL specification. As we will see, even a simple WSDL document contains quite a few elements with various relationships to each other. Try to identify these element in Figure 8-14 while examining the example.

Figure 8-15 contains the service interface file for the Exchange Web service whose SOAP representation we saw in the previous section.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="ExchangeRemoteInterface"
    targetNamespace=
        "http://www.exchange.com/definitions/ExchangeRemoteInterface"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:tns="http://www.exchange.com/definitions/ExchangeRemoteInterface"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">
    <message name="getRateRequest">
        <part name="country1" type="xsd:string"/>
        <part name="country2" type="xsd:string"/>
    </message>
    <message name="getRateResponse">
        <part name="result" type="xsd:float"/>
    </message>
    <portType name="Exchange">
        <operation name="getRate">
            <input name="getRateRequest" message="tns:getRateRequest"/>
            <output name="getRateResponse" message="tns:getRateResponse"/>
        </operation>
    </portType>
    <!-- binding element skipped -->
</definitions>
```

Figure 8-15 WSDL service interface file

This service interface file defines a single port type Exchange with only one operation getRate. The operation makes use of two abstract messages getRateRequest and getRateResponse. There are no type definitions because the messages only use simple directly referenced, XSD types.

Note that the names of port type, messages, and parts are local to the WSDL specification; the mapping to the actual transport protocol names (such as SOAP XML elements) is performed in the binding, the next part of the service interface file (Figure 8-16).

```

<binding name="ExchangeBinding" type="tns:Exchange">
    <soap:binding style="rpc"
        transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="getRate">
        <soap:operation soapAction="urn:Exchange" style="rpc"/>
        <input>
            <soap:body use="encoded"
                encodingStyle="http://schemas.xmlsoap.org/soap/encoding"/>
        </input>
        <output>
            <soap:body use="encoded"
                encodingStyle="http://schemas.xmlsoap.org/soap/encoding"/>
        </output>
    </operation>
</binding>

```

Figure 8-16 SOAP binding in the WSDL service interface file

There is a SOAP binding ExchangeBinding for the port type. The binding describes which runtime protocol (rpc) is supported by the service provider and contains transport protocol configuration information (<http://schemas.xmlsoap.org/soap/http>).

Here, the identifying URN urn:Exchange and the SOAP encoding are specified per operation and message (<http://schemas.xmlsoap.org/soap/encoding/>).

Figure 8-17 shows the corresponding service implementation file.

```

<?xml version="1.0" encoding="UTF-8"?>
<definitions name="ExchangeService"
    targetNamespace="http://localhost/wsdl/Exchange-service.wsdl"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:tns="http://localhost/wsdl/Exchange-service.wsdl"
    xmlns:binding="http://www.exchange.com/definitions/ExchangeRemoteInterface"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">
    <import namespace="http://www.exchange.com/definitions/ExchangeRemoteInterface"
        location="http://localhost/wsdl/Exchange-binding.wsdl"/>
    <service name="ExchangeService">
        <port name="ExchangePort" binding="binding:ExchangeBinding">
            <soap:address
                location="http://localhost/servlet/rpcrouter"/>
        </port>
    </service>
</definitions>

```

Figure 8-17 WSDL service implementation file

The implementation file imports the interface file. The port element ExchangePort contains the SOAP address `http://localhost/servlet/rpcrouter`; this is the URL under which the RPC router servlet of the service provider's SOAP server can be reached.

The port references the binding in the interface file. Note that a qualified name `binding:ExchangeBinding` is used in this case because the interface file uses a different namespace than the implementation file.

Our Exchange Web service conforms to the XML schema defined in the WSDL specification; it is possible to validate it with an XML validator. Note that in order to verify the semantic correctness of a WSDL specification, you would require an additional WSDL validation tool.

Namespaces

WSDL uses the XML namespaces listed in Table 8-3.

Table 8-3 WSDL namespaces

Prefix	Namespace URI	Explanation
wsdl	<code>http://schemas.xmlsoap.org/wsdl/</code>	Namespace for WSDL framework
soap	<code>http://schemas.xmlsoap.org/wsdl/soap/</code>	SOAP binding
http	<code>http://schemas.xmlsoap.org/wsdl/http/</code>	HTTP binding
mime	<code>http://schemas.xmlsoap.org/wsdl/mime/</code>	MIME binding
soapenc	<code>http://schemas.xmlsoap.org/soap/encoding/</code>	Encoding namespace as defined by SOAP 1.1
soapenv	<code>http://schemas.xmlsoap.org/soap/envelope/</code>	Encoding namespace as defined by SOAP 1.1
xsi	<code>http://www.w3.org/2000/10/XMLSchema-instance</code>	Instance namespace as defined by XSD
xsd	<code>http://www.w3.org/2000/10/XMLSchema</code>	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 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.

The next two sections on service implementation document and service interface document explain each of the WSDL specification elements in more detail and discuss advanced modelling issues. Feel free to skip them at this point, or to merely use them as a reference later on.

Service implementation document

The service implementation file contains *services* and contained *ports*.

Service definition

A service definition merely bundles a set of ports 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"> *
    <wsdl:port .... />*
  </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 .... > *
    <wsdl:port name="nmtoken" binding="qname"> *
      <!-- 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.

Our Exchange Web service provides such a SOAP address (see Figure 8-17 on page 281).

Service interface document

The service interface file contains the following elements:

- ▶ *Port types* and contained *operations*
- ▶ *Messages* consisting of message *parts* (parameters)
- ▶ Protocol *bindings* including binding information for the operations

Port type

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" .... /> *
    </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 .... > *
        <wsdl:operation name="nmtoken" parameterOrder="nmtokens">
            <wsdl:input name="nmtoken"? message="qname"/>
            <wsdl:output name="nmtoken"? message="qname"/>
            <wsdl:fault name="nmtoken" message="qname"/>*
        </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, or
- ▶ As two independent transport level operations, which can be required if the transport protocol only supports one way communication.

Messages

A message represents one interaction between service requestor and service provider. If an operation is bidirectional (an RPC call returning a result, for example), two messages are used in order to specify the transmitted on the way to and from the service provider:

```

<definitions .... >
    <message name="nmtoken"> *
        <part name="nmtoken" element="qname"? type="qname"?/> *
    </message>
</definitions>

```

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

Operation and messages are modelled 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.

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 .... />*
    </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 actually XML. For more information on XML and XSD see “An XML primer” on page 92.

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

Bindings

The bindings are the last set of elements in the WSDL interface document. A binding contains:

- ▶ Protocol specific general binding data such as the underlying transport protocol and the communication style for SOAP, for example.
- ▶ 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"> *
    <!-- extensibility element (1) --> *
    <wsdl:operation name="nmtoken"> *
      <!-- extensibility element (2) --> *
      <wsdl:input name="nmtoken"? > ?
        <!-- extensibility element (3) -->
      </wsdl:input>
      <wsdl:output name="nmtoken"? > ?
        <!-- extensibility element (4) --> *
      </wsdl:output>
      <wsdl:fault name="nmtoken"> *
        <!-- extensibility element (5) --> *
      </wsdl:fault>
    </wsdl:operation>
  </wsdl:binding>
</wsdl:definitions>
```

As we have already seen, a port references a binding. Port and binding are modelled 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 use XML namespaces in order to incorporate protocol specific information into the language and protocol independent WSDL specification. We will introduce the extensibility elements supporting SOAP, HTTP, and MIME in “Transport bindings” on page 287.

Rationale for location of binding information

It could be argued whether the binding element belongs to the type or to the implementation document; it is the first transport protocol specific element and can therefore accompany the port. For example, addressing information such as the URN is part of it.

On the other hand, the service provider implementation looks different for each binding, because the encoding information is part of it. A requestor following the provider-dynamic binding development approach (see “Service requestor” on page 256) needs the binding information at build time.

Therefore, the cut between interface and the implementation file is usually made between the binding and the port element. There are no drawbacks because it is possible to define a different binding in case different ports might wish to use different URNs.

Transport bindings

As a last step, we now investigate the WSDL extensibility elements supporting SOAP, HTTP, and MIME transport bindings.

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 8-4 lists the corresponding extension elements.

Table 8-4 SOAP extensibility elements in WSDL

Extension name	Explanation
<soap:binding ...>	binding level, specifies defaults for all operations
transport="uri"?	Binding level, transport is the runtime transport protocol used by SOAP (HTTP, SMTP, MQ., for example)

Extension name	Explanation
style="rpc document"?	The style is one of the two SOAP communication styles introduced in “An introduction to SOAP” on page 261).
<soap:operation ... >	extends operation definition
soapAction="uri"?	URN
style="rpc document"?	see binding level
<soap:body ... >	extends operation definition, specifies how message parts appear inside the SOAP body element.
parts="nmtokens"	optional, allows to externalize message parts
use="literal encoded"	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"?	list of supported message encoding styles
namespace="uri"?	URN of the service
<soap:fault ... >	extends operation definition, contents of fault details element
name="nmtoken"	relates soap:fault to wsdl:fault for operation
use="literal encoded"	see soap:body
encodingStyle="uri-list"?	see soap:body
namespace="uri"?	see soap:body
<soap:address ... >	extends port definition
location="uri"	network address of RPC router
<soap:header ... >	operation level, shaped after <soap:body ...>
<soap:headerfault ... >	operation level, shaped after <soap:body ...>

For an example of extensibility elements, refer to Figure 8-16 on page 281.

Note that the WSDL specification deals with encoding only. The mappings to be used for a specific type under a certain encoding are out of scope; they are part of the SOAP client and server runtime configuration (client API and deployment descriptor, respectively). See “Mappings” on page 269.

More detailed information on the WSDL SOAP binding exceeds the scope of this introduction. Please refer to the WSDL 1.1 specification.

HTTP binding

WSDL includes a binding for HTTP 1.1's GET and POST verbs in order 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 8-5 lists the defined extension elements.

Table 8-5 HTTP extension elements in WSDL

Extension name	Explanation
<http:address location="uri"/>	Extends the port definition. Contains the base URL.
<http:binding verb="nmtoken"/>	The HTTP operation to be performed (nmtoken=GET or POST).
<http:operation location="uri"/>	Extends the operation binding. Specifies the relative URL.
<http:urlEncoded/>	See the WSDL 1.1 specification for explanations.
<http:urlReplacement/>	See the WSDL 1.1 specification for explanations.

MIME extension elements might have to be used as well (see below).

Further information on the WSDL HTTP binding is out of the scope of this introduction. Please refer to the WSDL 1.1 specification for a more complete introduction, as well as examples.

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 8-6 lists the extensions that are available to describe a Web service that can be accessed via MIME.

Table 8-6 MIME extension elements in WSDL

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

Further information on the WSDL MIME binding is out of the scope of this introduction. Please refer to the WSDL 1.1 specification for a more complete introduction as well as examples.

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. The two XML processing techniques are introduced in “An XML primer” on page 92.

Currently, WSDL4J is an open source project available on the IBM developerWorks site:

<http://oss.software.ibm.com/developerworks/projects/wsdl4j/>

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

Figure 8-18 gives an example, first reading in a WSDL specification from a file and then extracting the service and all contained port elements.

```

Service service;
Port port = null;
// Read WSDL document and get definitions element
Definition definition = WSDLReader.readWSDL(null, fileName);

// Get the service elements
Map services = definition.getServices();

// Get an iterator for the list of services
Iterator serviceIterator = services.values().iterator();

boolean bPortFound = false;
while ((serviceIterator.hasNext()) && !(bPortFound)) {
    // Get next service element
    service = (Service) serviceIterator.next();

    // Determine if this service element contains the specified port
    if ((port = service.getPort(portName)) != null)
        bPortFound = true;
}

```

Figure 8-18 WSDL API

WSDL summary

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

WSDL covers two very different aspects of a Web service specification: type and protocol binding information appears in the interface file, and implementation access information in the implementation file (Figure 8-19).

However, the primer 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 (see “Product support for Web services” on page 307).

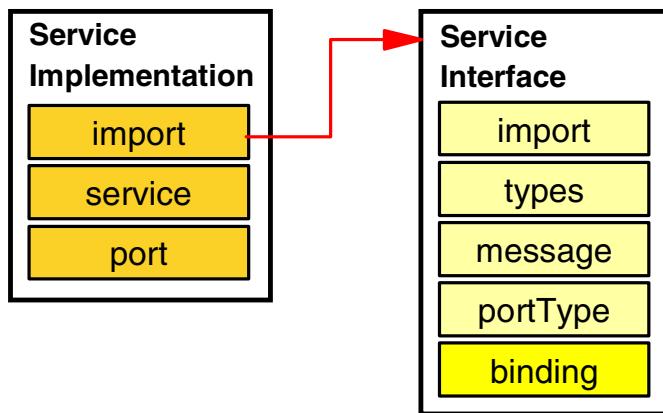


Figure 8-19 WSDL document structure

Outlook

The WSDL is currently in the process of being standardized by the W3C. We are not aware of any plans for a WSDL Version 1.2 at the time of writing.

Basic tool support for WSDL is already available, as we will see in Chapter 9, “Product support for Web services” on page 307. We expect several additional WSDL tools, such as schema validators and consistency checker, to become available in the near future. An example of a recent innovation in this 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. Two articles on developerWorks provide an introduction to WSIF:

<http://www-106.ibm.com/developerworks/webservices/library/ws-wsif.html>

Where to find more information

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

The best sources for more information are the IBM Web services toolkit (WSTK) and the WSDL 1.1 specification. WSDL 1.1 is contained in the IBM WSTK and can also be found at <http://www.w3.org/TR/wsdl>.

What is next

Once a Web service has been specified in WSDL, it can be made publicly available. In the SOA, UDDI is the preferred approach to do so. Therefore, the next section introduces UDDI.

UDDI overview

UDDI stands for universal description, discovery, and integration. UDDI is a *technical* discovery layer. It defines:

- ▶ The structure for a registry of service providers and services
- ▶ The API that can be used to access registries with this structure
- ▶ The organization and project defining this registry structure and its API

UDDI is a search engine for application clients rather than human beings; however, there is a browser interface for human users as well.

UDDI is not a full business oriented discovery or directory service; it operates on the technical level. The core information model is defined in an XML schema that can be found at http://www.uddi.org/schema/uddi_1.xsd

Note that UDDI is an organization currently supported by more than seventy companies, but not a standards body. It works closely with organizations such as the W3C, however.

UDDI registry structure

The following entities comprise the core of the UDDI data model:

Business entity Business entities are the white and yellow pages of the registry. Business entities provide business information about a service provider such as business name, contacts, description, identifiers, and categories.

Business service *The green pages, part 1* provide nontechnical service information. A business service is a descriptive container used to group a set of Web services related to a business process or group of services. Categorization is available on this level. A business service maps to a WSDL service.

Binding template *The green pages, part 2*, containing service access information. These are the technical Web service descriptions relevant for application developers who want to find and invoke a Web service. A binding template points to a service implementation description (for example, a URL). This entity is sometimes also called *access locator*. A binding template maps to a WSDL port.

tModel A tModel is a technical fingerprint, holding metadata about type specifications as well as categorization information. Its attributes are key, name, optional description, and URL. The simplest tModel contains some text characterizing a service.

A tModel *points to* a service interface description (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. This entity is sometimes also called *service type*.

Figure 8-20 displays this data model with the entities introduced above. It also shows their relationships and the link to the WSDL level.

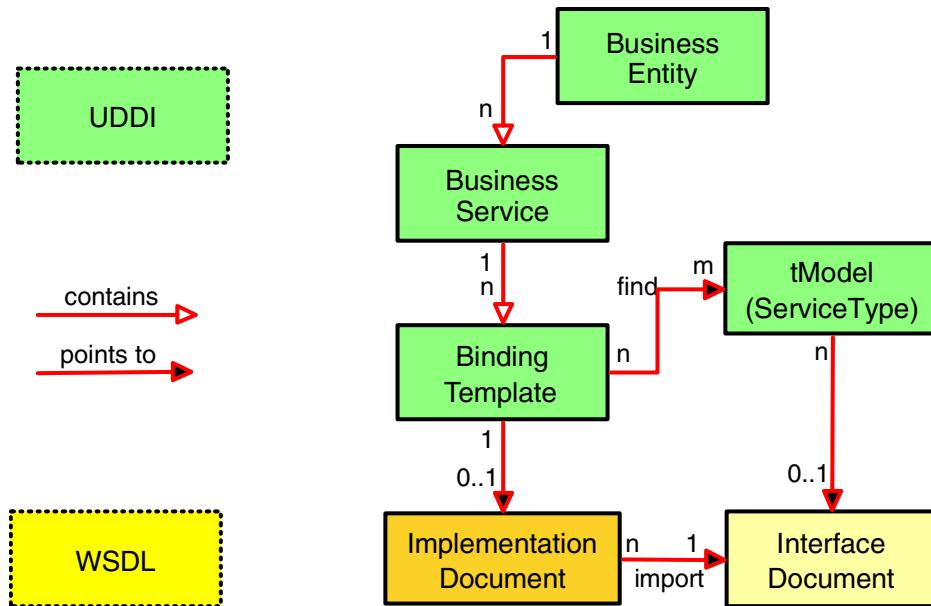


Figure 8-20 UDDI entities and their relationships

The business entity tops the containment hierarchy, containing one or more business service entities and in turn 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 holds a reference to a Web service implementation description file (typically a WSDL implementation file), a service type entity (tModel) a reference to a Web service type specification, which typically is a WSDL interface document. Note that the UDDI registry merely contains a URL pointing to the Web site of the service provider, not the WSDL document itself.

These references are implemented as URLs, therefore, any other specification format can easily be supported as well. UDDI is not bound to WSDL.

There is an m:m relationship between the binding template and the tModel. At first glance, this could surprise you. However, 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. Hence, there can be more than one technical description for one service (a formal specification and a supporting tutorial, for example). 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 oriented architecture.

Example

Let us take a closer look at an example, the UDDI entry for the Exchange service.

A business entity for the provider of the exchange service is shown in Figure 8-21.

```
<?xml version="1.0" encoding="utf-8" ?>
<businessDetail generic="1.0" xmlns="urn:uddi-org:api"
operator="www.ibm.com/services/uddi" truncated="false">
<businessEntity authorizedName="1000000PYG"
operator="www.ibm.com/services/uddi"
businessKey="4C4ABBB0-91B2-11D5-B9C3-0004AC49CC1E">
<discoveryURLs>
<discoveryURL
useType="businessEntity">http://www-3.ibm.com/services/uddi/testregistry/udit
iget?businessKey=4C4ABBB0-91B2-11D5-B9C3-0004AC49CC1E</discoveryURL>
</discoveryURLs>
<name>Exchange service provider</name>
<description xml:lang="en">Sample business entity.</description>
<businessServices>
<!-- businessServices skipped -->
</businessServices>
<categoryBag>
<keyedReference tModelKey="UUID:COB9FE13-179F-413D-8A5B-5004DB8E5BB2"
keyName="Exchange Key" keyValue="52" />
<keyedReference tModelKey="UUID:C1ACF26D-9672-4404-9D70-39B756E62AB4"
keyName="types" keyValue="wsdlSpec" />
</categoryBag>
</businessEntity>
</businessDetail>
```

Figure 8-21 UDDI business entity

The business service contained in this business entity is shown in Figure 8-22.

```
<businessService serviceKey="81FFBE10-9CE6-11D5-BA0D-0004AC49CC1E">
  businessKey="4C4ABBB0-91B2-11D5-B9C3-0004AC49CC1E">
    <name>ExchangeService</name>
    <description xml:lang="en">Exchange Rate Service</description>
    <bindingTemplates>
      <bindingTemplate bindingKey="82031970-9CE6-11D5-BA0D-0004AC49CC1E">
        serviceKey="81FFBE10-9CE6-11D5-BA0D-0004AC49CC1E">
          <description xml:lang="en" />
          <accessPoint URLType="http">http://localhost/servlet/rpcrouter</accessPoint>
          <!-- tModelInstance reference skipped -->
        </bindingTemplate>
      </bindingTemplates>
      <categoryBag>
        <keyedReference tModelKey="UUID:C0B9FE13-179F-413D-8A5B-5004DB8E5BB2">
          keyName="Finance and Insurance" keyValue="52" />
        <keyedReference tModelKey="UUID:C1ACF26D-9672-4404-9D70-39B756E62AB4">
          keyName="types" keyValue="wsdlSpec" />
        </categoryBag>
      </businessService>
```

Figure 8-22 UDDI business service and binding template

The business service ExchangeService contains one binding template. This binding template names the access point of the Web service and references a tModel instance. This tModel instance is shown in Figure 8-23.

```
<tModelInstanceDetails>
  <tModelInstanceInfo tModelKey="UUID:7CEF0CF0-9CE6-11D5-BA0D-0004AC49CC1E">
    <instanceDetails>
      <overviewDoc>
        <overviewURL>http://localhost/wsdl/Exchange-service.wsdl</overviewURL>
      </overviewDoc>
      <instanceParms><port name="ExchangePort" binding="ExchangeBinding"/>
      </instanceParms>
    </instanceDetails>
  </tModelInstanceInfo>
</tModelInstanceDetails>
```

Figure 8-23 UDDI tModel instance

The overviewURL element contains the pointer to the WSDL implementation file. The tModel, which contains the pointer to the WSDL interface file is not shown in the example. It looks similar to the tModel instance shown above.

Namespaces

UDDI defines the XML namespaces listed in Table 8-7.

Table 8-7 UDDI namespaces

Prefix	Namespace URI	Explanation
(none)	urn:uddi-org:api	Target namespace for the UDDI data model
xml	http://www.w3.org/1999/XMLSchema	Schema namespace as defined by XSD

Identification and categorization

In this section, we briefly explain UUID, identification, and categorization. You may have noticed that our Exchange example has already made use of these concepts.

UUID

The *universal unique identifier (UUID)* serves as data type for all key attributes of entities in the UDDI data model. In UDDI version 1, a UUID is a string of length 255.

A UUID is not specified as in parameter of a publish operation, but created by the UDDI server (for example, the client's hardware address can be used).

For example, as you can see in Figure 8-21 on page 295, the UUID of the business entity is 4C4ABBB0-91B2-11D5-B9C3-0004AC49CC1E.

The identifier bag explained next is a different concept; there is one and only one UUID per entity. It is guaranteed to be unique if you are using the UDDI cloud as described in "Advanced UDDI topics" on page 489.

Identifier bags

Business, services, and types can be known under more than one name. Hence, an optional identifier bag can amend a business entity or a tModel.

The identifier bag is modelled as a list of name-value pairs.

Categorization

In order to allow potential service requestors to search for services meeting certain criteria (other than names), UDDI introduces a *categorization* concept. This concept defines taxonomies on the entries of a registry.

Three standard taxonomies are defined in UDDI Version 1:

NAICS	North American Industry Classification System—roughly 1800 industry codes defined by the US government
UN/SPSC (ECMA)	Universal Standards Products and Services—United Nations standards effort, over 10000 numbers for products and services
Location	Geographical taxonomy

The category attributes are implemented as name-value pairs of the business entity instances (the business white pages), the business services (green pages), and the tModels (service types). Any valid taxonomy identifier is allowed.

Our example uses the NAICS taxonomy, assigning the value 52 to the key Exchange Key (see Figure 8-21 on page 295).

There is a need for tool support in order to simplify category assignment and browsing.

UDDI API overview

The UDDI programming API provides publishing and inquiry features.

The inquiry API supports the following operations:

- ▶ Find a business entity through its UUID.
- ▶ Find a business entity through a wild-carded name.
- ▶ Find a business entity through categorization information.
- ▶ Navigation from business entity to contained services—an optional categorization parameter can be specified.
- ▶ Find a service type (tModel).
- ▶ Access to detailed information on the found entities.

There are `find_xxx` and `get_xxx` methods for business, service, binding template, and service type (tModel).

The publishing API has the following functions:

- ▶ Publishing, update, and unpublishing of business entities
- ▶ Publishing, update, and unpublishing of business services and binding templates
- ▶ Publishing, update, and unpublishing of service types (tModels)

- ▶ Methods related to the individual API user—authenticating operations belong to this group of operations

There are `save_xxx` and `delete_xxx` methods for the elements of the UDDI object model (business, service, binding, tModel).

Figure 8-24 gives an example, inquiring the IBM Test Registry for a given business entity.

```
package itso.wsad.wsdynamic;

import java.util.*;
import com.ibm.uddi.client.UDDIProxy;
import com.ibm.uddi.response.*;

public class UddiTest {
    protected UDDIProxy proxy = new UDDIProxy();
    protected String inquiryAPI =
        "http://www-3.ibm.com/services/uddi/testregistry/inquiryapi";
    protected String publishAPI =
        "http://www-3.ibm.com/services/uddi/testregistry/protect/publishapi";
    String providerId = "ITSO";

    public void findServiceProvider() {
        try {
            proxy.setInquiryURL(inquiryAPI);
            proxy.setPublishURL(publishAPI);
            BusinessList bl = proxy.find_business(providerId, null, 0);
            BusinessInfos bis = bl.getBusinessInfos();
            Vector v2 = bis.getBusinessInfoVector();
            for (int i2 = 0; i2 < v2.size(); i2++) {
                BusinessInfo bi = (BusinessInfo)v2.elementAt(i2);
                String bkey = bi.getBusinessKey();
                System.out.println("Business="+bi.getNameString()+" "+bkey);
            }
        } catch (Exception e) { e.printStackTrace(); }
    }
}
```

Figure 8-24 UDDI API

UDDI4J is a Java implementation of the UDDI API from Apache. It requires Apache SOAP 2.1 or higher.

In “Dynamic Web services” on page 379 we show how to implement a dynamic service requestor using this API.

Anatomy of a UDDI registry

Figure 8-25 shows a conceptual component model for the service broker:

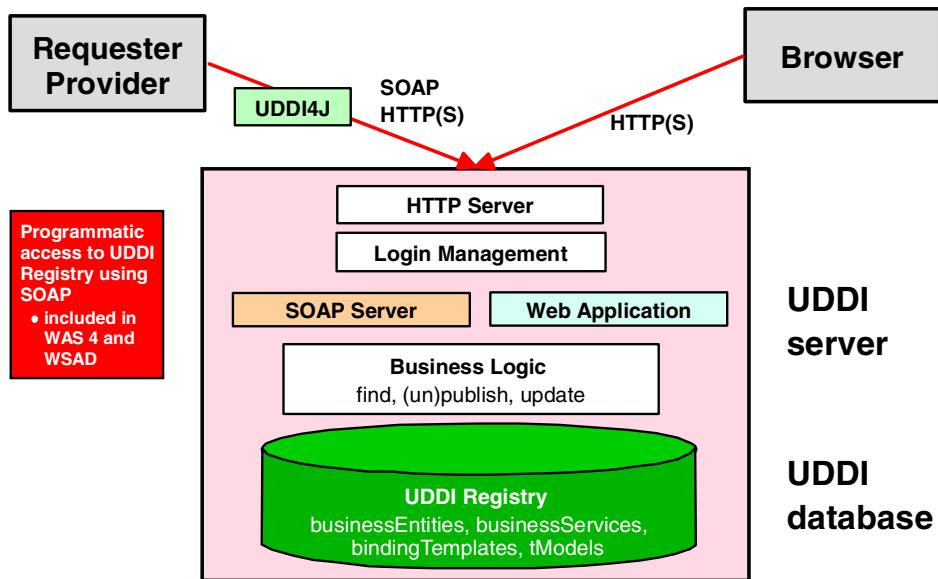


Figure 8-25 High-level component model for UDDI

UDDI consists of the client side API, UDDI4J, and the registry server implementation. The server provides HTTP and SOAP access to the registry that is structured according to the object model described above. Basic HTTP authentication (user ID, password) is required for all write operations such as publish, update, and unpublish.

Existing registries

The following UDDI registries exist as of today:

UDDI Business Registry	A replicated registry maintained by Microsoft, IBM, and Ariba (soon to be replaced by HP). Conceptually, this is just one registry. It is physically distributed, however.
IBM Test Registry	A registry for developers to experiment with the technology and test their services.
Private registries	IBM ships a private UDDI registry implementation. Refer to "Advanced UDDI topics" on page 489 for more information.

Existing Web services can also be discovered using classical, non UDDI Web portals such as Yahoo. Several such portals specializing on Web services have recently been established, for example:

- ▶ XMethods: <http://www.xmethods.net>

Note that XMethods hosts a service provider for our Exchange service. It can be reached under the URL <http://services.xmethods.net:80/soap>. The SOAPAction field is an empty string. The URN of the service is urn:xmethods-CurrencyExchange. Try to connect our SOAP client from Figure 8-11 on page 273 to it.

- ▶ <http://www.webservices.org>

For more information, refer to these Web sites.

Browser access to IBM registries

Both the IBM business and test registry can be reached under

<http://www.ibm.com/services/uddi>

It is worth noting that the browser GUIs for the IBM business and test registries as well as the one that comes with the private UDDI implementation use a slightly different terminology than the UDDI object model exposed to API programmers:

- ▶ Business service instances are called *businesses*
- ▶ Business services are called *services*
- ▶ Binding templates are called *access locators*
- ▶ tModel instances are called *service types*

UDDI summary

As we have seen, UDDI provides a structured means of describing and locating meta information about service providers, service implementations, and service interfaces.

Figure 8-26 summarizes the UDDI entities.

In this introduction, we have described UDDI Version 1. Most existing UDDI sites implement UDDI Version 1 at the time of writing, but the first implementations of UDDI Version 2 are now appearing (UDDI Business Registry V2 Beta at the IBM site <http://www.ibm.com/services/uddi>).

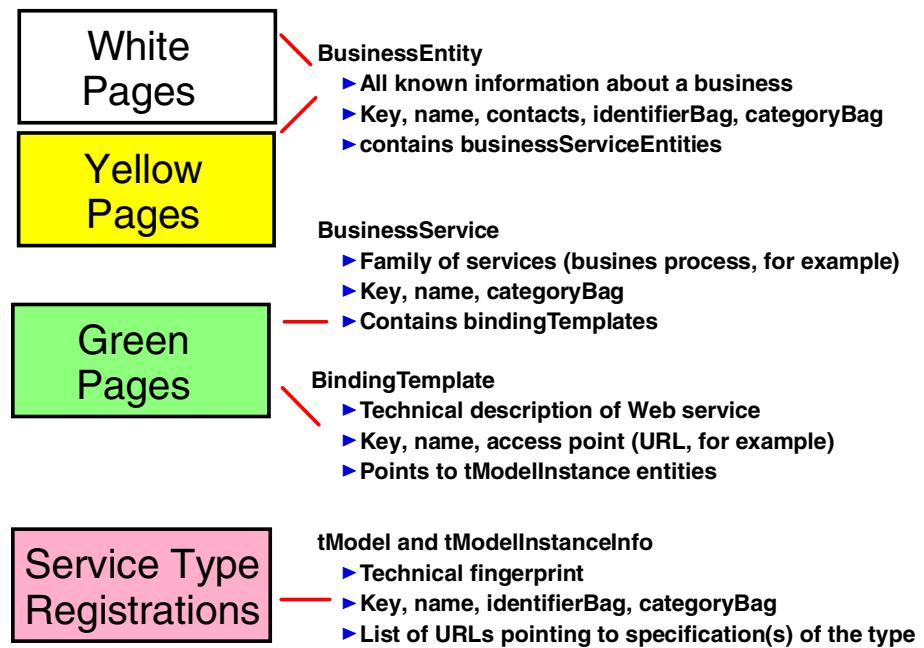


Figure 8-26 UDDI entities

Outlook

There is a road map for UDDI Version 2 and 3. UDDI version 2 provides support for more taxonomies (categorization schemes) and for layered services. The Version 2 specification is already available; beta level version 2 implementations are available in January 2002.

The Version 3 specification is targeted for 2002. It will add workflow support and allow to define custom taxonomies.

IBM and Microsoft have recently defined an additional specification for Web service discovery: the Web services inspection language (WSIL). As opposed to UDDI, WSIL defines a decentralized discovery mechanism, providing an XML format for assisting in the inspection of a Web site for available services. WSIL complements and provides bindings for WSDL and UDDI. The following developerWorks article provides an overview:

<http://www-106.ibm.com/developerworks/webservices/library/ws-wsilover/>

More information

The Web site of the UDDI organization is rich in content; you can find all UDDI specifications there, for example, the data model and API for Version 1 and 2. Moreover, there are technical white papers, a best practices section, and an informative list of frequently asked questions:

<http://www.uddi.org>

“Dynamic Web services” on page 379 has an example for a provider dynamic service requestor; additional information about types of registries and IBM’s private UDDI implementation can be found in “Advanced UDDI topics” on page 489.

Summary

The different standards for comprising the SOA (XML, SOAP, WSDL, UDDI) complement each other nicely without depending unnecessarily on each other. Each of them uses a slightly different terminology, since they are address different aspects, are designed to be extensible, and are not to be bound to each other.

Unfortunately, there is no official or proposed standard defining the SOA itself or a Web service reference architecture. Each vendor provides such a document (see “Web Services Conceptual Architecture (WSCA) Version 1.0” for the IBM document).

Therefore, in this section we attempt to clarify the terminology used by the different architecture elements, as well as the relationships between them.

The web of Web service standards

Figure 8-7, a topology map for the SOA, shows the big picture. All SOA elements and their dependencies upon each other are displayed.

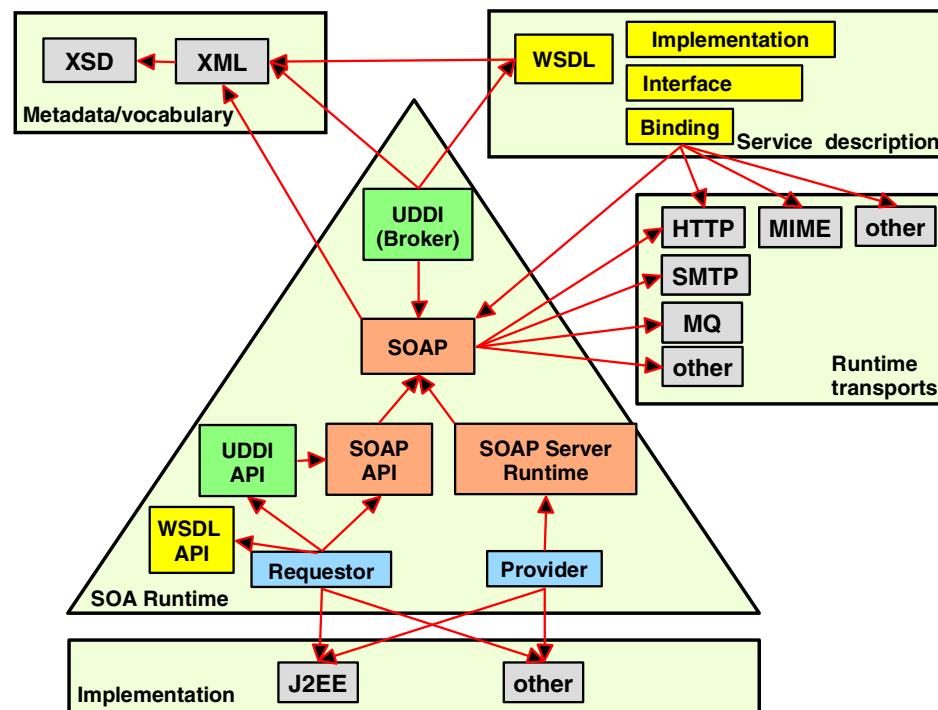


Figure 8-27 Dependency diagram for Web services SOA

Terminology quick reference

Table 8-8 provides a Web services terminology quick reference sheet.

Table 8-8 Terminology quick reference

Artifact	SOA	SOAP	WSDL	UDDI
Web service interface/class	Web service	(generated client proxy class)	Port type and binding (in binding.wsdl)	Service type or tModel (informal specification)
Web service instance	Web service	Service element in dds.xml	Port (in service.wsdl)	Binding template and contained tModelInstance
Group of related service instances		Entire dds.xml (and the two SOAP servlets in the Web application)	Service	Business service
Hosting application (process running the service implementation)	Service provider			Business entity
Client application	Service requestor	SOAP client	/	Individual (API user)
Registry	Service broker	/	/	UDDI site operator
Lookup	Find	/	/	Discovery
Service ID	/	URN	Service name and URN in namespace attribute of SOAP body element	UUID
Method	Bind	Message call	Operation, message	/
Parameter	/	Parameter	Parts	/
Data types	XSD Types	Types	Types	/

The table should be read in the following way: if two terms appear in the same row of the table, they address related concepts. The two entries are not necessarily synonyms that are equivalent to each other.

For example, a tModel provides supporting meta information about a service type interface and points to the formal specification, which can be a WSDL port type in a binding file.

Sources of information

Taking into account that Web services are an emerging technology, the amount of information available on the Web is massive.

Here are links to the addresses we found to be most useful:

For more information on Web sites with information on Web services, see “Referenced Web sites” on page 580.

Next

Now, let us take a look into the IBM tool support for Web services, before we begin developing our auto parts sample application.

Quiz: To test your knowledge of the service oriented architecture, try to answer the following 5 questions. The answers are provided in Appendix C, “Quiz answers” on page 559.

1. Name the service roles and operations comprising the service oriented architecture, as well as at least two development strategies each for the client and the server.
2. Which SOAP, WSDL, and/or UDDI files contain the URN of a Web service?
3. What are the two parts of a WSDL specification? Which of them contains protocol binding information such as the encoding to be used?
4. Is any one of the WSDL files involved in the SOAP server deployment process?

(*) Explain the difference between a UDDI tModel and a WSDL interface file.



Product support for Web services

This chapter provides an overview of the IBM tools that contain support for Web services. When we discuss the tools, we use the terminology introduced in Chapter 8, “Web services overview and architecture” on page 241. We discuss the following tools:

- ▶ WebSphere Studio Application Developer
- ▶ Web services toolkit (WSTK)
- ▶ WebSphere Studio 4.0 (classic version)
- ▶ DB2
- ▶ Versata

WebSphere Studio Application Developer

This section gives an overview about the possibilities of Application Developer in the area of Web services. For exact examples of how to use the different wizards we refer to the following chapters. We use the Web services development diagram that was introduced in Chapter 8, “Web services overview and architecture” on page 241, and which is shown again in Figure 9-1 to describe the functionality of the different wizards.

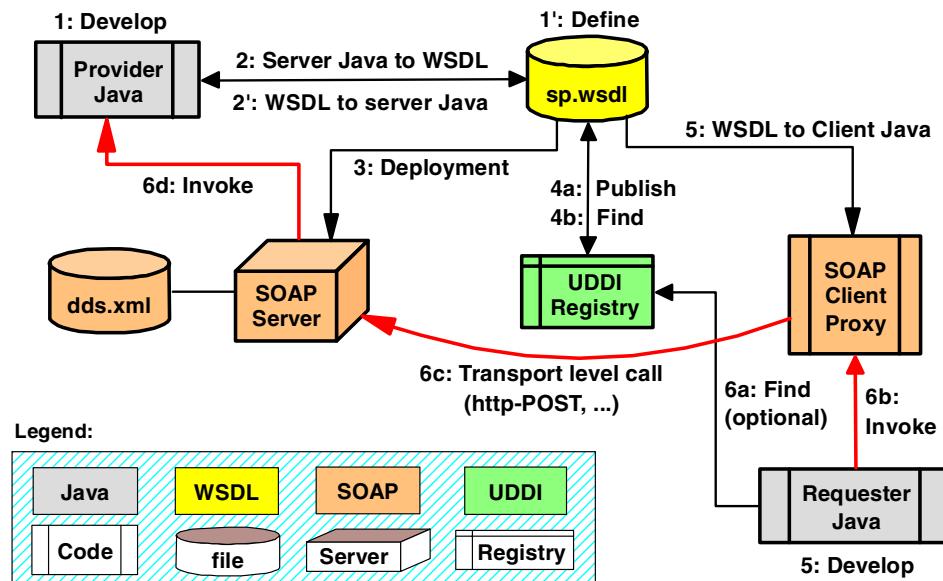


Figure 9-1 Web services development and runtime information flow

Web service wizard

The Web service wizard starts with step two in Figure 9-1:

- ▶ Starting from an existing JavaBean, you specify which methods you want to expose as a Web service and the wizard generates the corresponding WSDL files (both service interface and service implementation files). The Web service wizard supports both bottom up (2) and top down (2') approaches described in “Development strategies for provider and requestor” on page 255.
- ▶ The Web service wizard generates the ISD file. The ISD file is the SOAP deployment descriptor for a particular Web service.

All the ISD files in a project are composed to form the SOAP deployment descriptor, `dds.xml`, for all Web services, step three in Figure 9-1.

- ▶ During the Web service wizard, you have to specify the SOAP encodings. As discussed in “Encodings” on page 268, SOAP encodings tell the SOAP runtime environment how to translate from data structures constructed in Java into SOAP XML and vice versa.

With Application Developer you can either choose literal XML encoding or SOAP encoding.

- ▶ During the Web service wizard, you can specify to generate a Web service proxy class which conforms to step five in Figure 9-1.

A Web service proxy class makes it very easy to invoke a Web service. You just instantiate the class and invoke the desired method on it. The Web service proxy class itself contains the code to set the encoding styles for the input and output parameters and to make a SOAP request (step 6c in Figure 9-1).

- ▶ Another step in the wizard gives you the ability to generate and invoke a sample application.

The sample application is a small Web application consisting of some JSPs which enable you to invoke the different methods of the Web service, and to view the results. The generated JSPs contain the code to instantiate the generated Web service proxy class and to invoke the desired methods on it (Step 6b in Figure 9-1).

The proxy class implementation provides you with an example of how to use the Web service proxy class.

- ▶ The wizard adds two servlets to the Web application:
 - The Apache SOAP RPC router servlet
 - The Apache SOAP message router servlet

These servlets receive the SOAP requests from the SOAP client (step 6c in Figure 9-1), invoke the Web service (for example a method on a JavaBean, step 6d in Figure 9-1) and send back a SOAP response.

You can specify to immediately launch the sample application after the wizard. A default WebSphere 4.0 test environment is started. The SOAP servlets are deployed to the WebSphere 4.0 Test environment (step 3 in Figure 9-1) and started. The JSP test client starts in a browser and you can test the Web service.

To start the Web service wizard:

- ▶ Create a Web project to contain the Web service.
- ▶ Select the Web project and *File -> New -> Other*. Select *Web Services* at the left and *Web Service* at the right of the panel and click *Next* to start the wizard.

The Web service wizard is demonstrated in “Using the Web service wizard” on page 336.

Figure 9-2 shows the input and output of the Web service wizard.

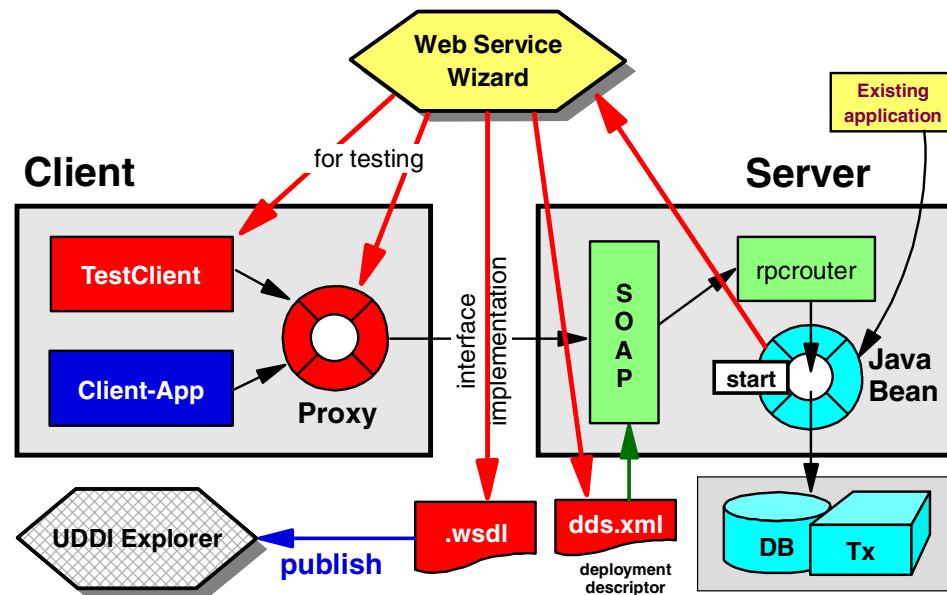


Figure 9-2 Web service wizard

The generated components include:

- ▶ WSDL interface and implementation files
- ▶ SOAP deployment descriptor (dds.xml)
- ▶ Client proxy for testing (JavaBean)
- ▶ Sample test client (a Web application)

Web service client wizard

The Web service client wizard gives the ability to generate a Java proxy class starting from an existing WSDL file (step 5 in Figure 9-1).

The Java proxy class encapsulates the code to make a SOAP request and to call the Web service. In fact, the Web service client wizard is part of the Web service wizard discussed above. It also contains the ability to generate, deploy and run a sample Web application to test your Web service. A demonstration of the Web service client wizard is described in “Web service client proxy” on page 354.

Figure 9-3 shows the input and output of the Web service client wizard.

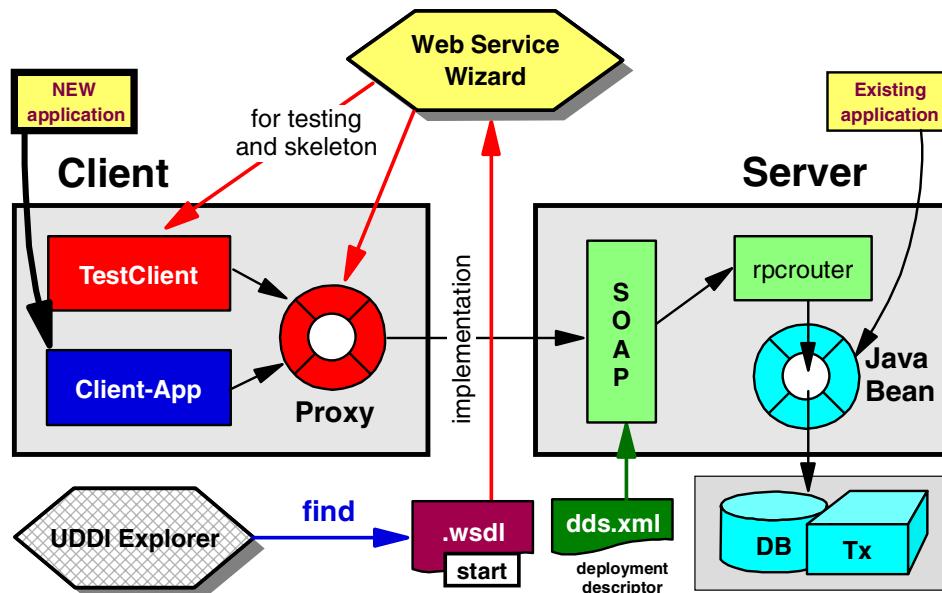


Figure 9-3 Web service client wizard

The main output is the client proxy (JavaBean). The sample test client can help you in implementing the real client application.

Web service skeleton JavaBean wizard

The Web service skeleton JavaBean wizard conforms to the top down development strategy described in “Development strategies for provider and requestor” on page 255.

In this wizard you start from an existing WSDL document and generate a JavaBean. The generated JavaBean contains method definitions that conform to the operations described in your WSDL document. After running this wizard you have to implement the various methods in your skeleton bean to complete your Web service implementation.

As in the Web service wizard, this wizard also allows you to generate a Java proxy class and a sample Web application to test your Web service after you have implemented the skeleton JavaBean. This wizard is demonstrated in “Creating the JavaBean skeleton” on page 419.

Figure 9-3 shows the input and output of the Web service skeleton JavaBean wizard.

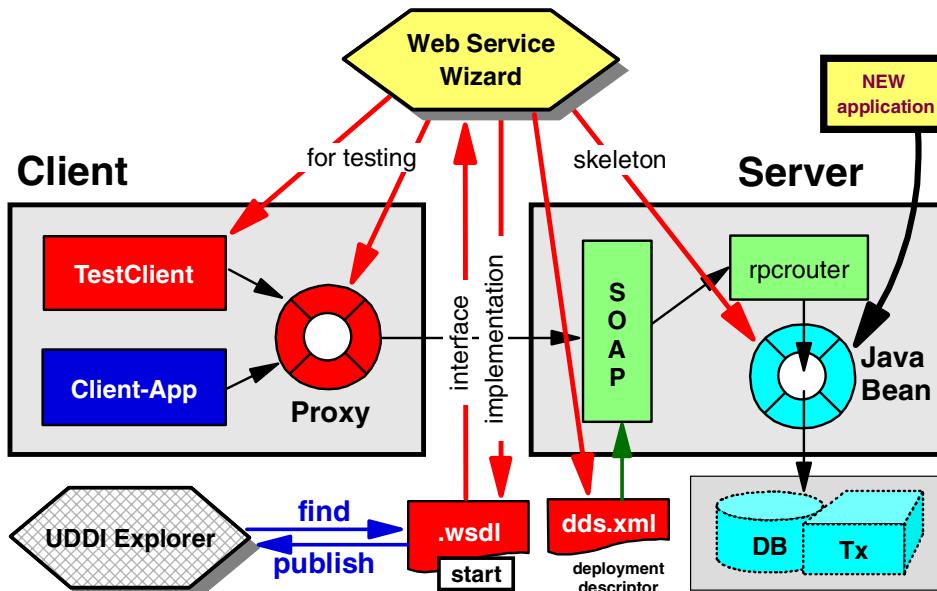


Figure 9-4 Web service skeleton JavaBean wizard

Web service DADX group configuration wizard

DAD extension (DADX) is an extension of the DB2 XML extender document access definition (DAD) file. A DADX document specifies how to create a Web service using a set of operations that are defined by SQL statements or DAD files. DAD files are files that describe the mapping between XML documents elements and DB2 database columns. They are used by the DB2 XML extender, which supports XML-based operations that store or retrieve XML documents to and from a DB2 database.

The Web services DADX group configuration wizard enables you to create a DADX group. The DADX group created with this wizard is used to store DADX documents. A DADX group contains connection (JDBC and JNDI) and other information that is shared between DADX files.

Once you have created a DADX group, you can import DADX files and then start the Web service wizard to create a Web service from a DADX file. You can create a DADX file by using the XML from SQL query wizard.

This process is demonstrated in “Creating a Web service from a DADX file” on page 479.

UDDI browser (import, export)

Application Developer contains a Web application that forms the IBM UDDI explorer or browser. With the IBM UDDI explorer you perform these functions against the IBM UDDI Test Registry, or the private IBM WebSphere UDDI Registry:

- ▶ Find a business entity (step 4b in Figure 9-1) demonstrated in “Importing the service WSDL from the UDDI registry” on page 394.
- ▶ Publish a business entity (step 4a in Figure 9-1) demonstrated in “Publishing a UDDI business entity” on page 387.
- ▶ Find a service interface and implementation (step 4b in Figure 9-1) demonstrated in “Importing the service WSDL from the UDDI registry” on page 394.
- ▶ Publish a service interface and implementation (step 4a in Figure 9-1) demonstrated in “Exporting the service WSDL to the UDDI registry” on page 390.

For all these operations, you have to create an account in the registry as explained in “Preparing for development” on page 383.

Web services toolkit (WSTK)

The WSTK is a toolkit that contains various tools, APIs, documents, and demonstrations, which help you in understanding, creating and deploying Web services. It is available from the IBM AlphaWorks Web site:

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

Relations between WSTK, Application Developer, and other IBM tools

WSTK is a toolkit that contains leading edge technologies in the area of Web services. Tools that first appear in WSTK may later on be adopted in other IBM products, such as Application Developer (the WSDL generator tool) or WebSphere Version 4.0 (for example UDDI4J technology and SOAP).

WSTK architecture

The WSTK architecture (Figure 9-5) is split into:

- ▶ Design-time Web services components
- ▶ Runtime components, that are either
 - Client components, called the Web services toolkit client runtime
 - Server components

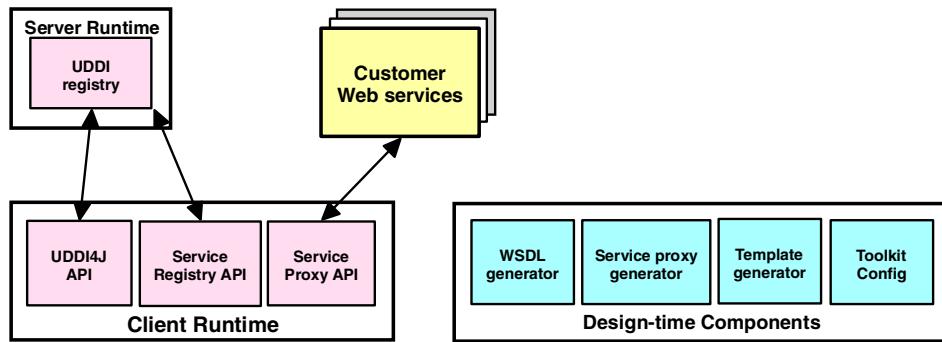


Figure 9-5 WSTK architecture

Design-time components

The design-time components of the WSTK include these tools:

WSDL generator tool

This stand-alone Java Swing application generates WSDL documents and SOAP deployment descriptors from existing JavaBeans, servlets, EJBs (conforming to the 1.0 specification) or COM objects. It is comparable to the Web service wizard in Application Developer and to step two in Figure 9-1 on page 308. This tool can generate serializers for complex data types. Serializers are Java class files that the Apache client and server use to serialize and deserialize complex types. How to write your own serializers is described in “Encodings and type mapping alternatives” on page 448.

To start the WSDL generator tool run the `wsdlgen` command located in the `WSTK_HOME/bin` directory.

Service proxy generator tool

Starting from an existing WSDL document, the service proxy generator tool generates a client proxy that interacts with a Web service. The functionality of this tool is similar to the Web service client wizard in Application Developer and maps to step 5 in Figure 9-1 on page 308.

To start the service proxy generator tool run the `proxogen -outputdir dir wsdl-document` command located in the `WSTK_HOME/bin` directory, where `outputdir` specifies the output directory and `wsdl-document` is the WSDL input document.

Service implementation template generator tool

This tool generates a JavaBean skeleton from an existing WSDL document. The tool creates the skeleton from the operations that are defined in the WSDL document. The functionality of this tool is similar to the Web service skeleton JavaBean wizard in Application Developer.

To run the service implementation template generator tool run the **servicegen wsdl-document -outputdir dir** command located in the WSTK_HOME/bin directory, where outputdir specifies the output directory and wsdl-document is the WSDL input document.

WSTK configuration tool

The configuration tool allows you to configure:

- ▶ Web application server configuration

Here you can choose to use embedded WebSphere, WebSphere 3.5, WebSphere 4.0 or Jakarta Tomcat as the application server within WSTK.

- ▶ UDDI registry used

You can choose to work with the IBM test and public UDDI registry, the Microsoft test and public UDDI registry, or another UDDI registry, such as the WebSphere UDDI Registry Preview available from AlphaWorks (<http://www.alphaworks.ibm.com>). The WebSphere UDDI Registry replaces the private UDDI registry in WSTK 2.4.

- ▶ Additional command line tools to configure Tomcat

WSDL4J APIs

These APIs are a set of Java classes used to work with WSDL documents programmatically.

Runtime components

The runtime components of the WSTK include these tools:

WSTK client runtime

The client runtime consists of a set of client runtime API's and their documentation. The use of the different API's is illustrated in "Working with the UDDI APIs" on page 398. The API's included in the client runtime are:

- ▶ UDDI4J API

This API allows you to perform the save, delete, find and get operations against a UDDI registry. This is step 6a of Figure 9-1 on page 308.

- ▶ Service registry API

This API allows you to perform the publish, unpublish, and find operations against a UDDI registry. These operations are similar to steps 4a and 4b of Figure 9-1 on page 308.

- ▶ Service proxy API

This API allows you to dynamically bind to Web services through the SOAP protocol.

The client runtime also contains an Apache SOAP pluggable provider feature to access non-Java Web services such as Microsoft COM objects.

Server runtime components

The server component consists of the UDDI registry that can be defined with the Web services toolkit configuration tool.

Other components and functionality

Besides the design-time Web services tools and the runtime components, WSTK contains various other useful things such as:

- ▶ Demonstrations that illustrate the use of the different tools and concepts
- ▶ Documentation, such as the specifications for WSDL 1.1, WSFL (Web services flow language) and HTTPR (reliable HTTP).
- ▶ WSTK 2.4 also contains a standalone version of the TCP/IP tunnel and monitor introduced in [cross reference to WS static]. It can be invoked with:
`tcptunnel hostname port`
- ▶ WSTK also ships the Apache SOAP user documentation, as well as the Javadoc for APIs such as SOAP, UDDI4J, and WSDL4J.

WebSphere Studio (classic) Version 4

WebSphere Studio 4.0 contains two wizards that help you in the development of Web services.

Web service creation wizard

This wizard is similar to the Web service wizard in Application Developer. Starting from an existing JavaBean, it generates the WSDL interface and implementation files and the required SOAP deployment descriptor. This wizard conforms to the bottom up approach described in “Development strategies for provider and requestor” on page 255, and is shown as step two in Figure 9-1.

Web service consumption wizard

This wizard is similar to the Web service client wizard in Application Developer. Starting from an existing WSDL document, it generates a Java proxy client and a JSP, which instantiates the Java proxy client, which in turn invokes the Web service itself. This wizard conforms to step 5 in Figure 9-1.

DB2

DB2 access through Web services is supported by DADX files. The DAD extension (DADX) is an extension of the DB2 XML Extender document access definition (DAD) file. A DADX document specifies how to create a Web service using a set of operations. The following operations are supported:

- ▶ XML document based (retrieve and store XML). The operations are described in DAD files that describe the mapping between XML documents elements and DB2 database columns. They are used by the DB2 XML Extender, which supports XML-based operations that store or retrieve XML documents to and from a DB2 database.
- ▶ SQL based (query, insert, delete, and update). This is demonstrated in “Creating a Web service from a DADX file” on page 479.
- ▶ DB2 stored procedure invocations

Versata

The Versata Logic Suite exploits the power of business logic automation to deliver highly dynamic and scalable business systems. It is powered by the Versata Logic Server (VLS), a container hosting an automation engine and execution services within IBM WebSphere. For more information, visit the Web site at: <http://www.versata.com>.

The Versata Transaction Logic Engine, at the core of the VLS, enables business rules execution for transactional business logic and provides a J2EE framework for necessary, reusable service, such as cross-object navigation, transaction sequencing, event-action synchronization, and more.

Working with Web services and Versata together is an obvious match. The automation engine provides a great mechanism for quickly building the transactional aspects of the Web service provider implementation, both for bottom up and top down scenarios. This integration can be already be achieved today by using a combination of the IBM WSTK and the new Versata XML, and a Web services toolkit, which we will describe in the next section.

Versata XML and Web services toolkit

The Versata XML and Web services toolkit, available as an additional feature to the VLS, enables the invocation of the deployed business logic by sending and receiving XML messages via a JMS queue using a fixed grammar, VXML (Versata XML).

As it is unlikely that the calling application wants to use VXML directly. The adapter also provides the capability of invoking an XSL transformation on both the inbound and outbound messages, and mapping it into a more appropriate format for the client. This is illustrated in Figure 9-6. Such transformations can be built using the XML tooling of Application Developer.

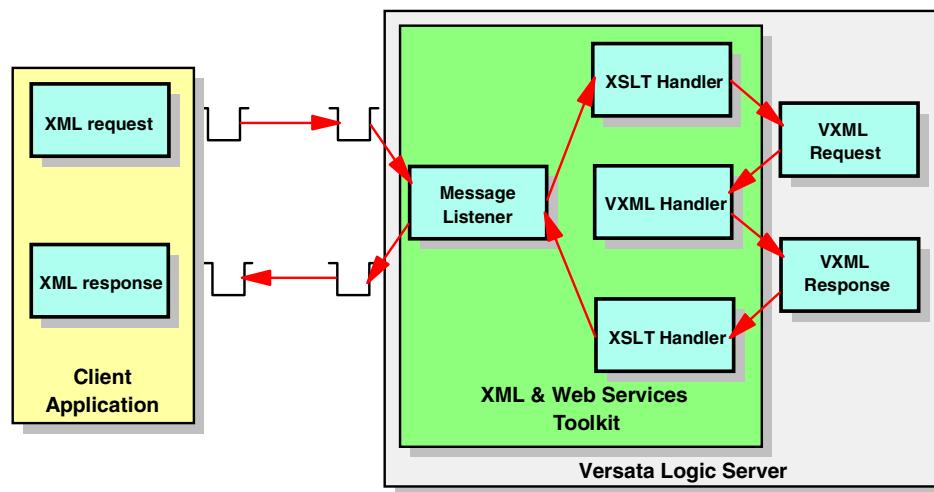


Figure 9-6 Runtime topology of the Versata XML and Web services toolkit

Exposing rules in the VLS as a Web service

A generic service processor has been developed by Versata which works in conjunction with the standard WebSphere or Apache SOAP RPC router. This service processor then acts as a client to the integration toolkit as shown in Figure 9-7.

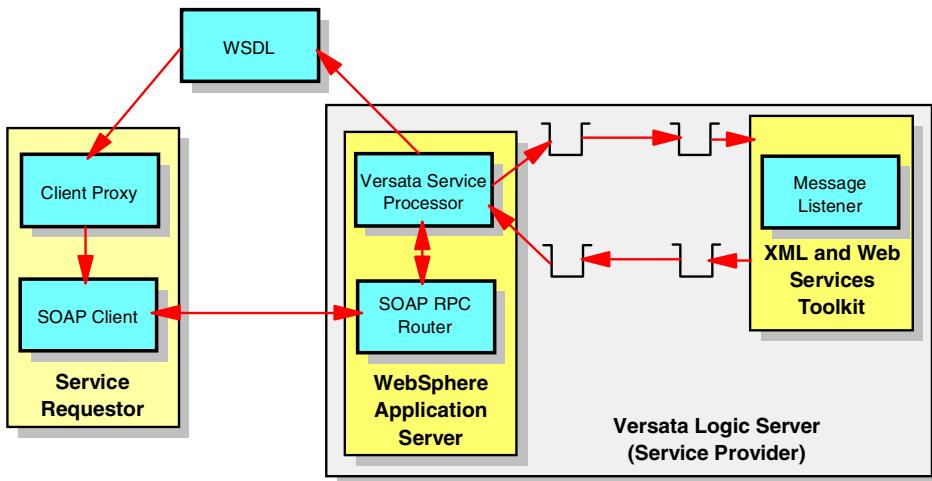


Figure 9-7 Runtime topology of the Versata Web service processor

The Web service implementation contains one operation, *process*, which accepts an XML element as an input parameter and returns the result of the rules execution as a second XML element. Service requesters can use the WSDL interface and implementation files supplied with the generic service processor to create client proxies for integration into their applications.

This generic service processor is available as a technology preview directly from Versata at the time of writing, but is expected to be made generally available by the end of 2001.

Calling a Web service from business rules

As is the case with most technology that is complementary to Web services, there is potential second scenario where the rules that are automated by Versata require the invocation of a Web service.

Application Developer can assist the Versata developer in creating a client proxy from the WSDL for the service. This client proxy, along with its accompanying class libraries can then be imported into Versata, added to the class browser and incorporated into business rules defined in the rules designer.

Summary

In this chapter we described some of the tools that assist in building applications based on Web services.

Quiz: To test your knowledge of the Web development features of Application Developer, try to answer the following five questions. The answers are provided in Appendix C, “Quiz answers” on page 559.

1. Which servlet invokes a Web service?
2. Does the Web service client wizard generate a skeleton client application?
3. Name at least two functions of the UDDI explorer?
4. Where can you find the Web services toolkit?

What Web service wizards does WebSphere Studio classic provide?



Static Web services

In this chapter we discuss how to develop simple, statically bound Web services using Application Developer. To illustrate this, we will extend the auto parts example applications we developed in Chapter 3, “Web development with Application Developer” on page 55, and Chapter 5, “EJB development with Application Developer” on page 133.

This chapter describes the following:

- ▶ How to create a Web service from an existing application using the bottom-up approach
- ▶ An explanation of the options available in the Application Developer Web service wizard
- ▶ A detailed investigation of the key files created
- ▶ Creating a client to a Web service using static service binding (no UDDI)
- ▶ Integrating the Web service into the Web application

Solution outline for auto parts sample Stage 2b

The Almaden Autos car dealership is now regularly finding itself in a situation where a mechanic requires a part that is not available in its current parts inventory system. When such situations occur, the mechanic currently calls the Mighty Motors National Parts Center and finds out if one of their regional warehouses has the part in stock. This is a slow process, and Mighty Motors is experiencing spiralling costs in its national call center as the dealership network expands. As a solution, it has decided to use Web services technology to provide an interface into its own national inventory system which can be easily integrated into the many heterogeneous systems used by its dealers.

Almaden Autos decides to get involved in the Mighty Motors Web services initiative, and is sent some WSDL describing the new Web service. Using the interface and implementation information provided, it modifies its current dealership inventory system to include an additional link for the mechanic that enables searching the national system if no parts are available locally.

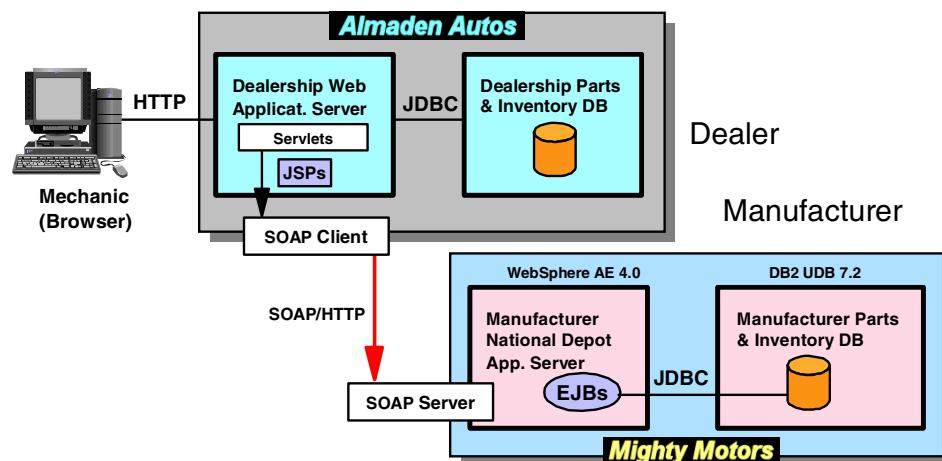


Figure 10-1 Basic design for the Web service provider and requester

The benefits of this approach is that neither Almaden Autos or Mighty Motors require any details on how their Web service has been implemented, or which type of technology has been used.

Using the terminology we introduced in Chapter 8, “Web services overview and architecture” on page 241, Mighty Motors is both the *service interface provider* and *service implementation provider*, and Almaden Autos is the *service requestor*.

Class and sequence diagrams

Figure 10-2 shows the class diagram, and Figure 10-3 shows the sequence diagram for the implementation.

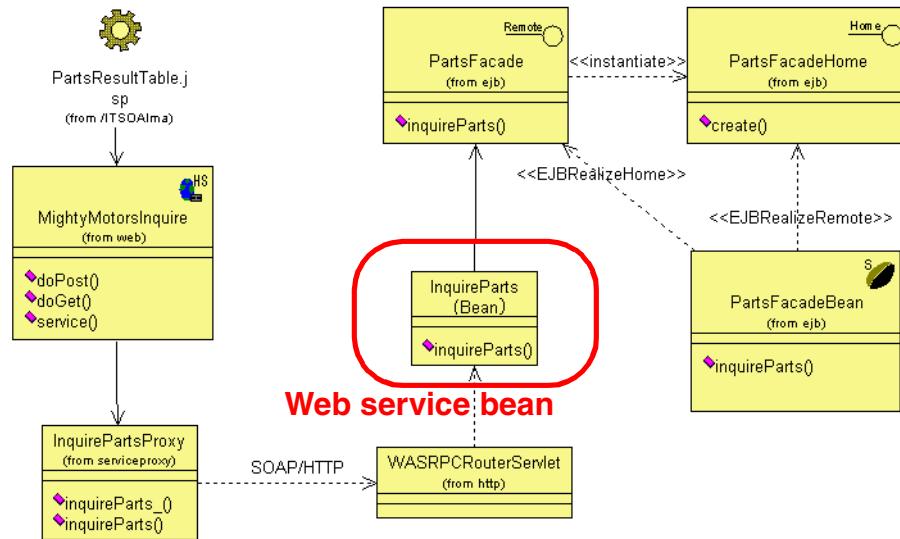


Figure 10-2 Class diagram for construction of Web service

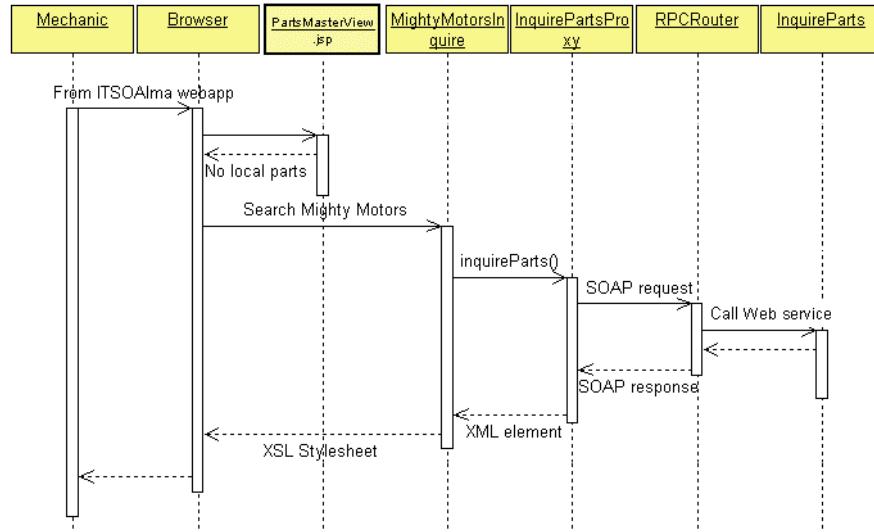


Figure 10-3 Sequence diagram for construction of Web service

Preparing for development

You should already have a number of projects added to the workspace from the previous development chapters of this redbook. These should include:

ITSOAlmaEAR	This is the EAR project containing the enterprise archive for the Almaden Autos parts inventory application.
ITSOAlmaWeb	The Web project for the Almaden Autos application, containing the servlets and JSPs for JDBC access which we created in Chapter 3, “Web development with Application Developer” on page 55. It is here that we will implement our <i>Web service client</i> .
ITSOMightyEAR	The second EAR project containing the enterprise archive for the Mighty Motors national inventory system.
ITSOMightyEJB	This is the EJB project we developed in Chapter 5, “EJB development with Application Developer” on page 133 which holds all of the source code for the Mighty Motors application. This will form the basis of our <i>service implementation</i> .
ITSOMightyXML	The XML project used in Chapter 4, “XML support in Application Developer” on page 91, to build the XML schema we are using to describe a parts inventory inquiry result set. This will be used in the <i>service interface</i> .
ITSOwsADServer	A server project containing one WebSphere test server instance and configuration, <code>ITSOwsADWebSphere</code> , which has both <code>ITSOAlmaEAR</code> and <code>ITSOMightyEAR</code> enterprise archives deployed in it.

If you have not completed the steps described in the first part of this book, an intermediate solution is available in the downloadable files that accompany it.

In the following sections, we will be performing the following configuration steps:

- ▶ Create a new Web project for Mighty Motors.
- ▶ Configure the new Web application.

Creating a new **ITSOMightyWeb** project

To create the project from the workbench take these steps:

- ▶ Select *File -> New -> Project* and create a *Web Project* from the *Web* category. Click *Next*.

- ▶ Enter a project name of ITS0MightyWeb and associate it with the ITS0MightyEAR EAR project.
- ▶ Set the Web application *Context root* to ITS0Mighty.
- ▶ Click *Next*.
- ▶ Select the ITS0MightyEJB.jar from the list of *Available dependent JARs* because the JavaBean wrapper we develop will have a dependency on the EJBs stored in this project.
- ▶ Click *Finish* to complete creating the project.

Web service types

The Application Developer Web service wizard supports the generation of a Web service from five different sources. These are as follows:

JavaBean Web service	Use an existing JavaBean in the application to generate the Web service interface. This uses the <i>bottom-up</i> development approach.
DADX Web service	Generate a Web service from a document access definition extension. This allows you to wrap DB2 extender or regular SQL statements into a Web service, and is discussed in more detail in Chapter 14, “Web services advanced topics” on page 447.
Skeleton JavaBean Web service	This enables you to create a skeleton JavaBean from an existing WSDL document if you would like to implement a Web service for a given standard. This uses the <i>top-down</i> development approach.
EJB Web service	Generate a Web service from an existing session EJB. This enables you to turn business methods into Web services.
URL Web service	This enables you to create a Web service with HTTP binding that invokes a given URL through HTTP POST/GET.

Creating the Mighty Motors InquireParts Web service

Our previously created PartsFacade session EJB returns a vector of items found in the inventory. A more portable solution would have been to return an XML document from the session EJB, however EJB 1.1 requires that the return type of an EJB must be serializable and XML elements are not serializable.

In order to create a clean layered architecture and a portable solution that returns an XML document, we will create a JavaBean to interface with our session EJB. This JavaBean will interrogate the session bean for the required information and then convert the results into an XML document. We will then generate the Web service from this JavaBean.

See “Creating a Web service from a session EJB” on page 374 for an example of creating a Web service directly from a session EJB.

Creating the JavaBean wrapper

This section details the process for creating the JavaBean that wraps the PartsFacade session EJB and returns an XML document.

In this section we will:

- ▶ Create a package for the JavaBean
- ▶ Create the JavaBean
- ▶ Complete JavaBean implementation
- ▶ Test the JavaBean

Creating the package for the JavaBean

We must first create a package for the Java wrapper. In the Web perspective:

- ▶ Select the source folder in the ITSOMightyWeb project.
- ▶ Select *New -> Other -> Java -> Java Package* from the context menu.
- ▶ Enter the package name of `itso.wsad.mighty.ejbclient`.
- ▶ Click *Finish*.

Creating the InquireParts JavaBean

To create the JavaBean take these steps:

- ▶ Select the newly created package and *New -> Other -> Java -> Java Class*.
- ▶ Complete the *Java Class* wizard as seen in Figure 10-4, to define the *InquireParts* class, and click *Finish*.

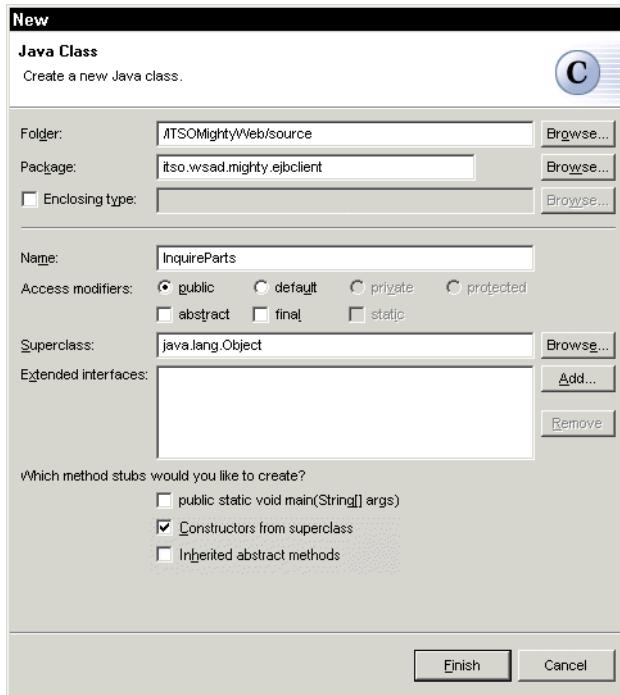


Figure 10-4 Java class wizard for *InquireParts*

Implementing the InquireParts JavaBean

As discussed previously, the *PartsFacade* session EJB returns a vector of inventory items. For portability and reuse we will implement the *InquireParts* JavaBean to return an XML document based on the vector returned from the session EJB.

The JavaBean code is available in `sampcode\wsstatic\InquireParts.java`.

Adding Xerces support to the project

Our JavaBean will be returning an XML Element type, so we must include an XML parser in the ITSOMightyWeb project build path. The XML parser we will use in the sample application is Apache Xerces (previously known as the IBM XML4J parser), which is included with Application Developer.

To add Xerces to the project:

- ▶ Open the Properties dialog for the project.
- ▶ Select the Java Build Path category, and switch to the Libraries tab.
- ▶ Click the *Add Variable* button on this panel. For the variable name field, browse to the WAS_XERCES entry.
- ▶ Leave the path extension field empty. Click *OK*.
- ▶ The resulting Java Build Path panel is shown in Figure 10-5.

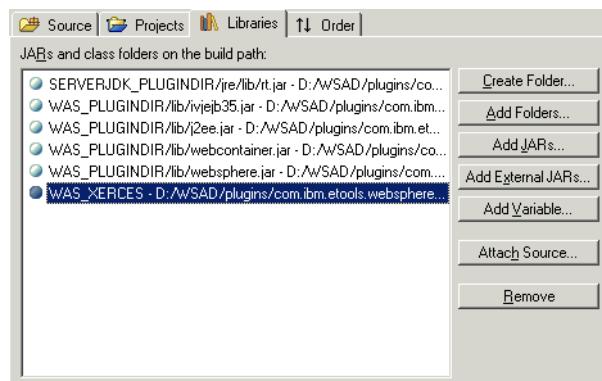


Figure 10-5 Adding Xerces to the Java build path

Adding import statements

This JavaBean will access the session EJB and create an XML document. This requires access to a number of classes in multiple packages. Add these import statements:

```
import javax.naming.*;  
import javax.ejb.*;  
import javax.rmi.*;  
import javax.xml.parsers.*;  
import java.util.Vector;  
import org.w3c.dom.Document;  
import org.w3c.dom.Element;  
import itso.wsad.mighty.ejb.*;  
import itso.wsad.mighty.bean.PartInventory;
```

Adding helper methods

To ensure that we only have a single instance of the PartsFacade EJB home and XML document builder, we define two helper methods:

getDocumentBuilder This method creates a singleton of the XML document builder.

getPartsFacadeHome This method creates a singleton for the home interface of the PartsFacade session EJB.

- ▶ Add the following declarations to the InquireParts JavaBean:

```
private PartsFacadeHome partsFacadeHome;
private DocumentBuilder builder;
```

- ▶ Now add the helper methods to the class (Figure 10-6).

```
/**  
 * getDocumentBuilder - share XML document builder  
 */  
private DocumentBuilder getDocumentBuilder() throws Exception {  
    if (builder == null) {  
        try {  
            DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();  
            builder = factory.newDocumentBuilder();  
        } catch (ParserConfigurationException ex) {  
            ex.printStackTrace();  
            throw new Exception("Error creating document builder: "+ex.getMessage());  
        }  
    }  
    return builder;  
}  
  
/**  
 * getPartsFacadeHome - share HOME of session bean  
 */  
private PartsFacadeHome getPartsFacadeHome() throws Exception {  
    if (partsFacadeHome == null) {  
        try {  
            InitialContext ic = new InitialContext();  
            Object objref = ic.lookup("itso/wsad/PartsFacade");  
            //Object objref = ic.lookup("java:comp/env/ejb/PartsFacade");  
            partsFacadeHome = (PartsFacadeHome)PortableRemoteObject.narrow  
                (objref,PartsFacadeHome.class);  
        } catch (NamingException ex) {  
            ex.printStackTrace();  
            throw new Exception("Error looking up PartsFacadeHome: "+ex.getMessage());  
        }  
    }  
    return partsFacadeHome;  
}
```

Figure 10-6 InquireParts helper methods

Adding the methods for creating the XML document

We now add several methods to assist in creating the XML document.

- ▶ Define a newElement method for creating an XML element for a given name and value pair. This method simplifies creating the document (Figure 10-7).

```
/**  
 * newElement - create XML name/value pair  
 */  
private Element newElement(Document doc, String name, String value) {  
    Element element = doc.createElement(name);  
    element.appendChild( doc.createTextNode(value) );  
    return element;  
}
```

Figure 10-7 Source code for element creation method

- ▶ Next, define a populatePart method that generates an XML element for the XML document, containing a part and inventory item (Figure 10-8).

```
/**  
 * populatePart - create XML for a part  
 */  
private Element populatePart(Document doc, PartInventory bean)  
    throws org.w3c.dom.DOMException {  
    Element elPart = doc.createElement("Part");  
    elPart.appendChild( newElement(doc, "ItemNumber", bean.getItemNumber().toString()) );  
    elPart.appendChild( newElement(doc, "Quantity",  
        (new Integer(bean.getQuantity())).toString() ) );  
    elPart.appendChild( newElement(doc, "Cost", bean.getCost().toString() ) );  
    elPart.appendChild( newElement(doc, "Shelf", bean.getShelf() ) );  
    elPart.appendChild( newElement(doc, "Location", bean.getLocation() ) );  
    elPart.appendChild( newElement(doc, "PartNumber", bean.getPartNumber() ) );  
    elPart.appendChild( newElement(doc, "Name", bean.getName() ) );  
    elPart.appendChild( newElement(doc, "Description", bean.getDescription() ) );  
    elPart.appendChild( newElement(doc, "Weight",  
        (new Double(bean.getWeight())).toString() ) );  
    elPart.appendChild( newElement(doc, "ImageURL", bean.getImageUrl() ) );  
    return elPart;  
}
```

Figure 10-8 Source code for XML element population method

Define the inquireParts method

Finally, we have to define the `inquireParts` method to generate the XML document from the matching parts returned from the `PartsFacade` session EJB (Figure 10-9).

```
/**  
 * inquireParts - find matching parts and return as XML  
 */  
public org.w3c.dom.Element inquireParts(String partNumber) throws Exception {  
    Element result = null;  
    Document doc = null;  
    DocumentBuilder builder;  
    PartsFacade partsFacade;  
  
    doc = getDocumentBuilder().newDocument();  
    result = doc.createElement("InquirePartsResult");  
    result.setAttribute("xmlns",  
        "http://www.redbooks.ibm.com/ITSOVSAD/schemas/InquireResults");  
    result.setAttribute("xmlns:xsi", "http://www.w3.org/2001/XMLSchema-instance");  
    result.setAttribute("xsi:schemaLocation",  
        "http://www.redbooks.ibm.com/ITSOVSAD/schemas/InquireResults  
        wsdl/InquireParts.xsd");  
    try {  
        partsFacade = getPartsFacadeHome().create();  
        Vector resultbeans = partsFacade.inquireParts(partNumber);  
        for (int i=0; i<resultbeans.size(); i++) {  
            result.appendChild( populatePart(doc,  
                (PartInventory)resultbeans.elementAt(i)) );  
        }  
    } catch (CreateException ex) {  
        System.out.println("Cannot find part: "+partNumber);  
        ex.printStackTrace();  
        throw new Exception("Error finding part: "+ex.getMessage());  
    } catch (java.rmi.RemoteException ex) {  
        ex.printStackTrace();  
        throw new Exception("Error with session EJB: "+ex.getMessage());  
    }  
    return result;  
}
```

Figure 10-9 Source code for main logic method

Note the XML schema location attribute is defined using a very confusing notation that separates the target name space from the location of the schema using a space. For more information on name space and location refer to Chapter 4, “XML support in Application Developer” on page 91.

The URL for the XML schema (`wsdl/InquireParts.xsd`) points to a nonexistent file. We have to copy the schema into a suitable folder of the Web application. We will do this after we generate the Web service.

Testing the InquireParts JavaBean

Before we continue on to generating a Web service from our newly created JavaBean, best practice indicates that we should test the bean. We can do this using the same universal test client used in “Testing the session EJB” on page 167.

To test the InquireParts JavaBean, switch to the server perspective:

- ▶ The ITSONWebSphere server must be restarted because we have changed the definition of the ITSONMightyEAR project when we added the ITSONMightyWeb project to it. Select the ITSONWebSphere server and restart it.
- ▶ Open the Web browser and enter the URL: <http://localhost:8080/UTC> to invoke the *universal test client*.
- ▶ Select the *EJB Page*. This will allow us to test our JavaBean (even though it is called the *EJB Page*).
- ▶ In the *References* pane, expand the *Utilities* and select *Load Class*.
- ▶ In the *Parameters* panel, enter `itso.wsad.mighty.ejbclient.InquireParts` as the class name and click *Load* (Figure 10-10).

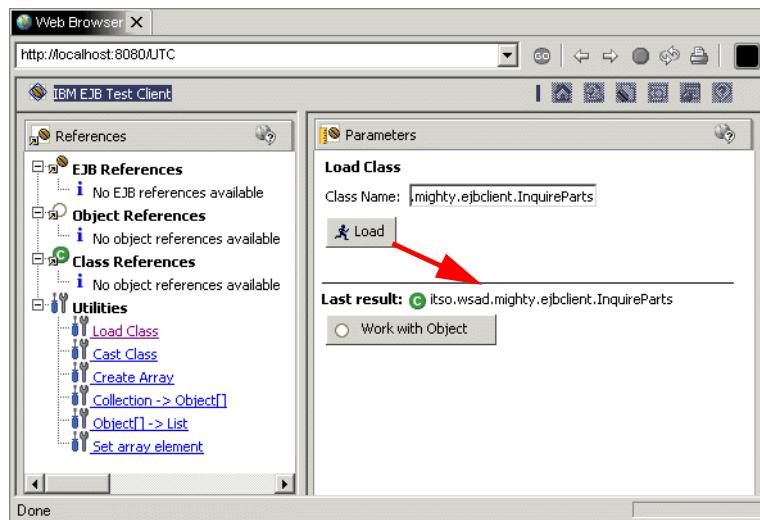


Figure 10-10 Loading the InquireParts class

- ▶ Click *Work with Object* and the InquireParts class appears under the *Class References* in the *References* pane.
- ▶ Expand the InquireParts class and select the InquireParts() constructor.

- ▶ Click *Invoke and Work with Object* and an instance of the InquireParts class appears under *Object References*.
- ▶ Expand the InquireParts object (under *Object References*) and select the Element inquireParts method.
- ▶ Enter the part number M10000003 and click *Invoke*.
- ▶ Click *Work with Object* and the returned Element appears under Object References (Figure 10-11).

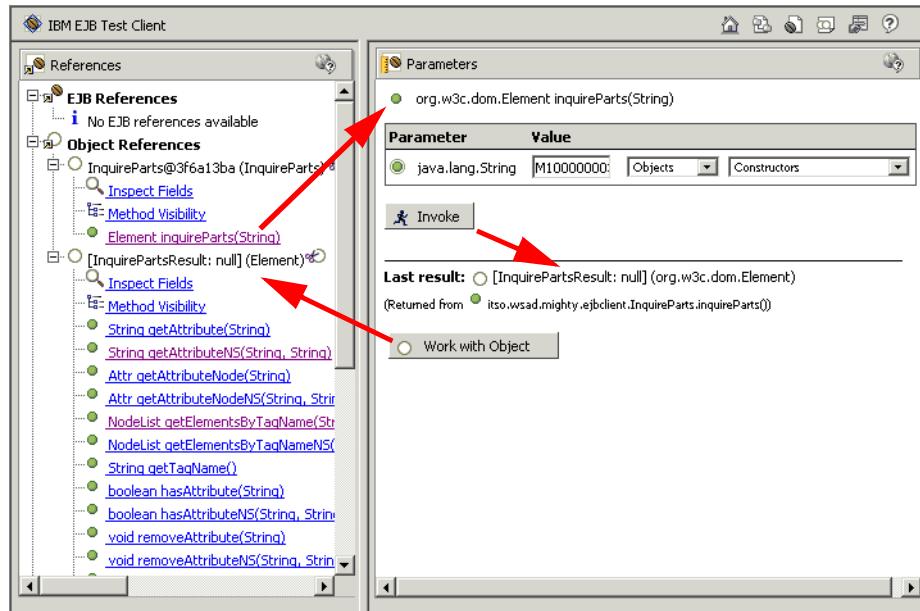


Figure 10-11 Test results from the JavaBean wrapper

- ▶ Expand the InquirePartsResult Element, select the getTagName method and invoke it.
- ▶ The tag name returned is a string with the value InquirePartsResult.

If you want to explore the tree structure of the returned XML element:

- ▶ Click on *Method Visibility* (under InquirePartsResult).
- ▶ Select both `org.w3c.dom.Element` and `org.w3c.dom.Node` and click *Set*. This action adds additional methods (from Node) to the InquirePartsResult object.
- ▶ You can now execute methods such as `getChildNodes` or `getFirstChild` to explore the XML tree structure.

You have now successfully implemented the JavaBean wrapper for the PartsFacade session EJB.

Close the browser window and stop the server.

Now we will use the wrapper to generate a Web service.

Creating the Web service

The following sections describe the steps that have to be performed to turn the wrapper JavaBean we just created into a powerful, self-describing Web service. The assets which will be generated by the wizard are shown in Figure 10-12.

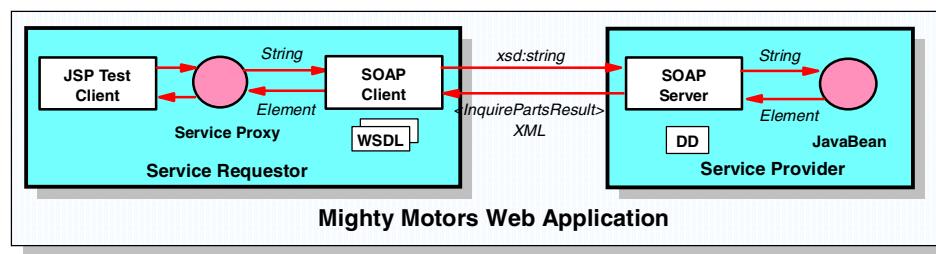


Figure 10-12 Assets generated from Web service wizard during this section

The Web service will have a similar interface to the InquireParts class—a single method inquireParts that accepts the part number as its input parameter and returns a XML element containing the results of the query.

The XML result shown in Figure 10-13 conforms to the XML schema (InquireParts.xsd) that we developed in “Generating an XML file and an XML schema from SQL” on page 111.

```
<InquirePartsResult xmlns="http://www.redbooks.ibm.com/ITSOADSAD/InquireResults"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.redbooks.ibm.com/ITSOADSAD/schemas/InquireResults
        http://localhost:8080/ITSO Mighty/wsdl/InquireParts.xsd">
    <Part>
        <ItemNumber>21000003</ItemNumber>
        <Quantity>10</Quantity>
        <Cost>59.99</Cost>
        <Shelf>L8</Shelf>
        <Location>San Francisco</Location>
        <PartNumber>M10000003</PartNumber>
        <Name>CR-MIRROR-R-01</Name>
        <Description>Large rear view mirror for Cruiser</Description>
        <Weight>4.6</Weight>
        <Image_URL>mirror03.gif</Image_URL>
    </Part>
    <Part>
        <ItemNumber>21000004</ItemNumber>
        <Quantity>10</Quantity>
        .....
    </Part>
</InquirePartsResult>
```

Figure 10-13 Sample returned XML element data for InquireParts Web service

There are a number of steps we perform when executing the wizard:

- ▶ Define the Web service type.
- ▶ Configure the Web service identity.
- ▶ Define the Web service scope.
- ▶ Define encoding styles and server mappings.
- ▶ Create a service proxy and client mappings.
- ▶ Create a sample test client.

Once completed, we will have developed the first Web service and be able to investigate its content.

Using the Web service wizard

Switch to the Web perspective. To launch the Web service wizard take these steps:

- ▶ Optionally select the ITSOMightWeb project or the InquireParts JavaBean (in source\itso\wsad\mighty\ejbclient) to prefill the wizards panels.
- ▶ Click on the *File -> New -> Other* menu.
- ▶ From the *Web Services* category, select *Web Service*. Click *Next*.

Complete the first Panel of the wizard (Figure 10-14).

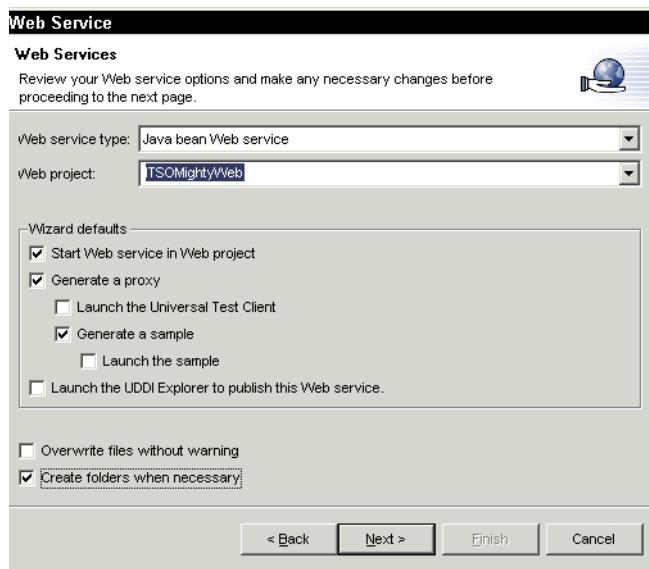


Figure 10-14 Web service wizard generation options

- ▶ Select the *Web service type* of *Java bean Web service*.
- ▶ Select the *ITSOMightyWeb* project.
- ▶ Select *Start Web service in Web project* (this will start the server), *Generate a proxy* (for testing) and *Generate a sample* (for testing using the proxy).
- ▶ Select *Overwrite files without warning* if you have to redo the wizard, and select *Create folders where necessary*. Click *Next*.
- ▶ On the *Web Service Java Bean Selection* dialog, set the *Bean* to *itso.wsad.mighty.ejbclient.InquireParts* and click *Next*.

Configuring the Web service identity

Figure 10-15 shows the next panel of the Web service wizard.

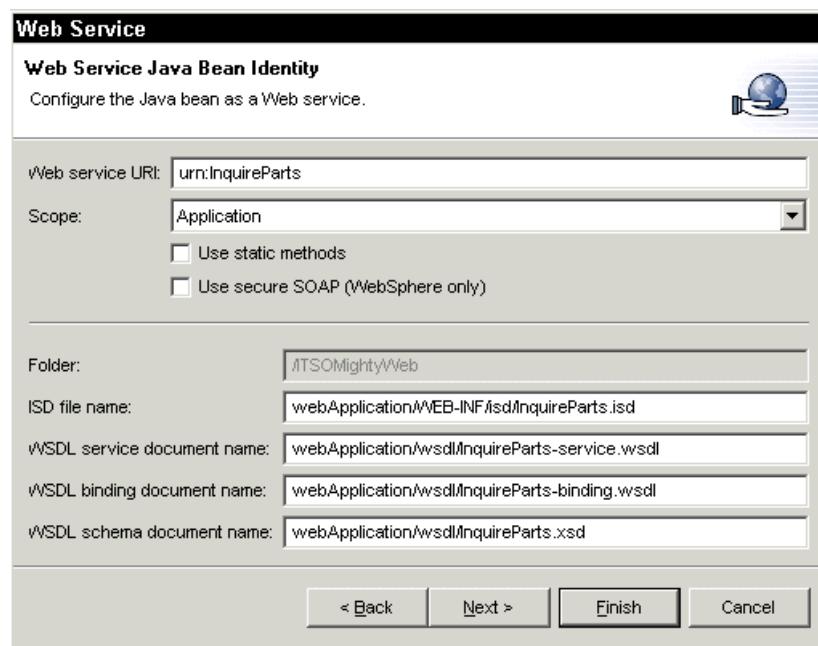


Figure 10-15 Defining the Web service identity

- ▶ Change the Web service URI to urn:InquireParts.
- ▶ Set the Scope value to Application (see “Web service scope” on page 338).
- ▶ Shorten the ISD file name to:

`webApplication/WEB-INF/isd/InquireParts.isd`

This ISD file will contain the deployment information for the Web service once the wizard is complete. We will describe it in more detail in a later step.

- ▶ Leave the default WSDL service and binding document names.
- ▶ Shorten the name of the *WSDL schema document name* to:

`webApplication/wsdl/InquireParts.xsd`

Web service scope

Three alternatives are available to define the scope of the Web service:

- | | |
|--------------------|--|
| Request | A new service bean is constructed by the server for each SOAP request. The object is available for the complete duration of the request. |
| Session | One service bean is constructed by the server for each new client session to the Web service, and is maintained by sending cookies to the client side. The bean is available for the complete duration of the session in a similar way to HTTP session data. |
| Application | A single instance of the service bean is constructed for the life of the server. This is the best option for good performance, but requires that the JavaBean is written in reentrant fashion so that multiple request can run through the code in parallel. |

This information will be part of the deployment descriptor for the Web service.

If the *Use static methods* option is selected, the class method that is made available is a static method, and therefore, no object will be instantiated.

Enable SOAP security

The Web service wizard also provides an option to apply security to the SOAP router servlet when deploying the Web services in the project to a WebSphere instance and configuration.

Because one instance of the SOAP router exists for each Web application (WAR file), this option can only be edited when creating the first Web service for the project. When creating further Web services with the Web service wizard, this option will be disabled, showing the status of the first selection.

This can be changed manually after generation by editing `web.xml`.

Selecting JavaBean methods and encoding styles

Let us continue with the Web service wizard:

- ▶ Click *Next* to progress to the next panel (Figure 10-16). This dialog prompts us to select which methods of the JavaBean we want to expose to the Web service interface, and the encoding style we want to use.

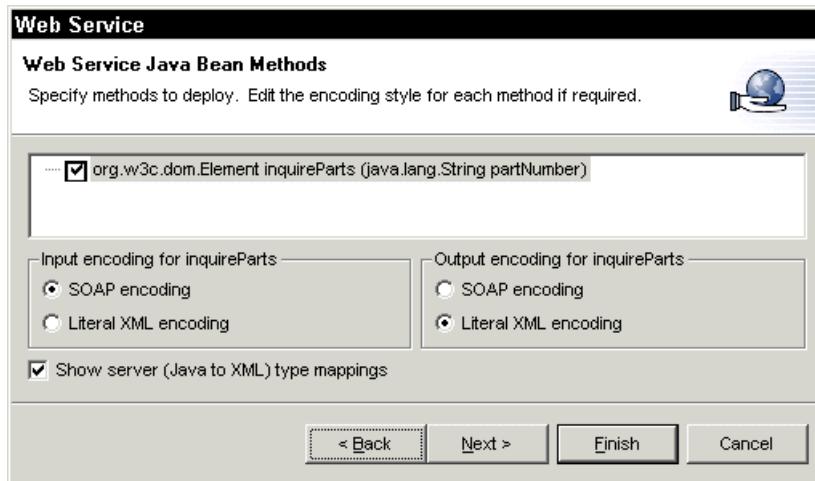


Figure 10-16 Selecting the JavaBean methods to include in the Web service

The wizard shows that one public method in our JavaBean is available to include. Make sure the `inquireParts` method is selected.

This panel also enables us to select the encoding style for the input and output data of each method. The encoding style defines how the Java types in the client and server are mapped to the XML types in the SOAP message. There are two encoding styles available:

Literal XML encoding This must be selected if the element to encode is of type `org.w3c.dom.Element`. The XML element is then inserted directly into the SOAP message.

SOAP encoding Use this option for all other datatypes, including simple Java types, standard Java classes and non-complex custom classes.

Mapping complex method signatures: There is a limitation in the current release of WSDL which only allows an encoding style to be defined at an input or output level for the method signature. For a complex input signature that requires a combination of XML elements and other datatypes, you must code a custom mapping and write your own serializers and deserializers. This is discussed in more detail in Chapter 14, “Web services advanced topics” on page 447.

Once we have selected the encoding style for each of our methods in the Web service interface, we must now define the details of the encoding mechanism:

- Select *Show server (Java to XML) type mappings* to examine the type mappings, and click *Next* to progress to the next panel of the wizard.

XML /Java mappings

It is worth spending some time explaining the process involved in encoding the SOAP message. Examine the diagram in Figure 10-17.

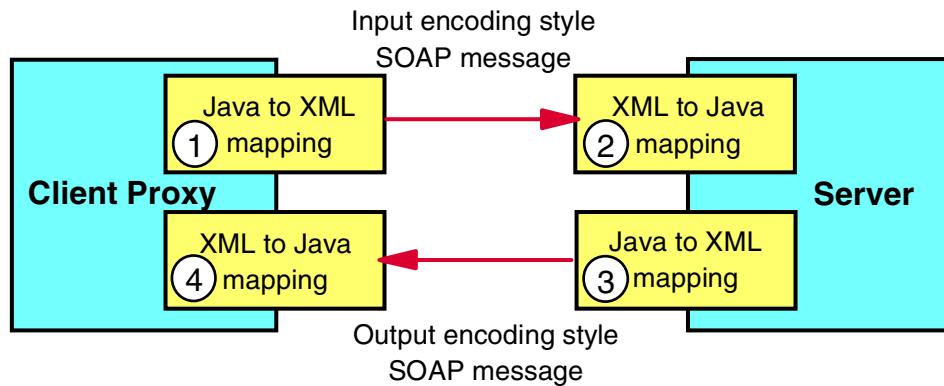


Figure 10-17 Illustration showing mapping and encoding stages for a Web service

This illustrates the mappings between Java and XML for the Web service to operate, assuming both a Java client and a Java server.

There are four steps in the process, indicated by the numbers in the illustration:

1. Client input mapping

This takes the parameter from the Java client and maps it using the input encoding style. In our example, this input parameter is the part number, defined using a Java String class and using the SOAP encoding style. This is serialized to an XML type name of xsd:string.

2. Server input mapping

The inbound part number is now deserialized from the SOAP encoding style in the message to a Java type of String, which is then used to invoke the method in the JavaBean.

3. Server output mapping

Once the JavaBean has completed its method execution, the returned value is inserted into the SOAP reply using the output encoding style. In our example, the returned type is an instance of the org.w3c.dom.Element class, containing the result set for our parts query. This data will be inserted into the SOAP message using the literal XML encoding style.

4. Client output mapping

The final stage is performed by SOAP for the client proxy, which maps the returned XML element encoding using literal XML style into an instance of the Java org.w3c.dom.Element data type.

Defining Java to XML mappings

On the *Web Service Java to XML Mappings* dialog (Figure 10-18), we define the mappings for the client input mapping and server output mapping (phase 1 and phase 3 from Figure 10-17).

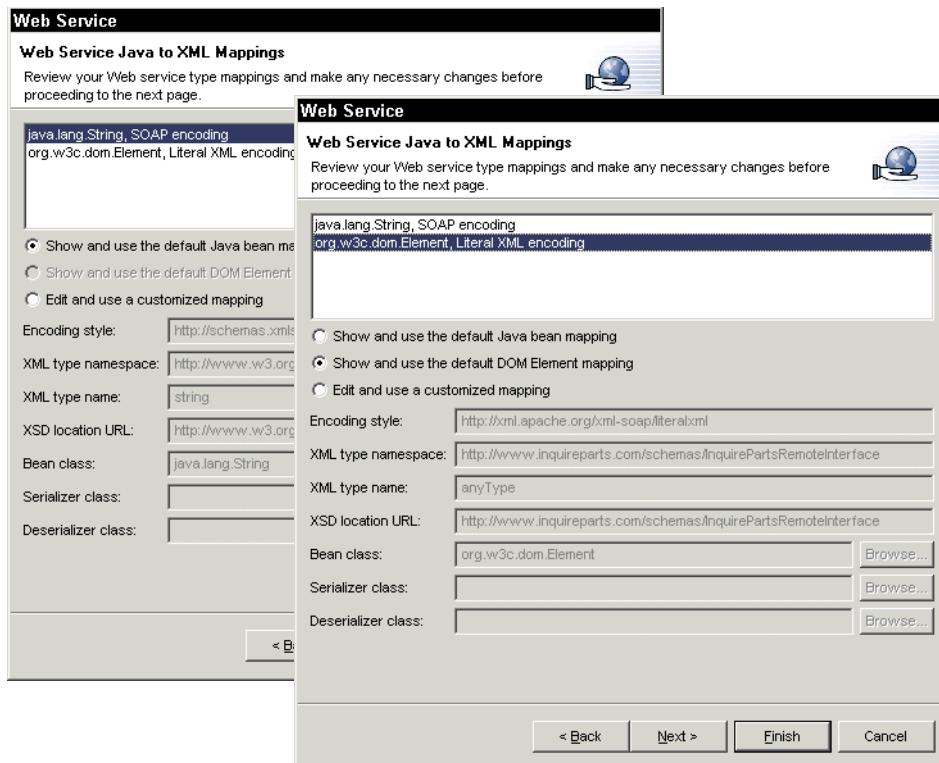


Figure 10-18 Java to XML mappings for input parameter and output result

- ▶ Select the *java.lang.String, Soap Encoding*, which is the encoding of the `java.lang.String` partNumber variable to XML type to be used in the SOAP message (phase 1, Figure 10-17).
We will leave *Show and use default Java bean mapping* to use the default mapping on this parameter.
- ▶ Select the *org.w3c.dom.Element, Literal XML encoding*, which is the encoding of the `org.w3c.dom.Element` returned from the `inquireParts` method (phase 3, Figure 10-17).
We will leave *Show and use default DOM Element mapping* to use the default mapping on this result.
- ▶ Click *Next*.

Proxy generation

The client proxy provides an RPC call interface to the Web service. While this proxy is not necessary for the implementation of the Mighty Motors application, it provides a very useful mechanism for testing the Web service before we make it available to third parties. Its usage was illustrated in Figure 10-12 on page 334. We will go into more detail about this later when we integrate the Almaden Autos Web application with the Mighty Motors Web service.

The *Web Service Binding Proxy Generation* panel of the wizard is shown in Figure 10-19.



Figure 10-19 Proxy generation for Web service

- ▶ The WSDL binding is preselected.
- ▶ Change the proxy class name to:
itso.wsad.mighty.wsstatic.proxy.InquirePartsProxy
- ▶ Select the *Show mapping* checkbox. This will allow us to specify the XML to Java mappings.
- ▶ Click *Next* to progress to the next panel.

We will examine the generated client proxy later in this chapter.

Defining the XML to Java mappings

On the *Web Service XML to Java Mappings* dialog (Figure 10-20), we define the client input mapping and server output mapping (phase 2 and phase 4 from Figure 10-17).

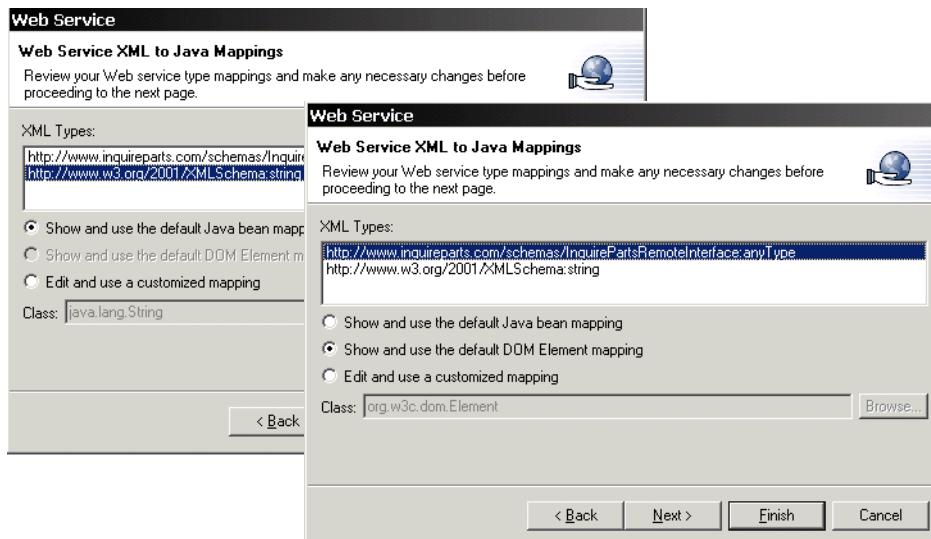


Figure 10-20 XML to Java mapping for parameter *partNumber*

- ▶ The String entry defines the mechanism to map the input XML data type to the Java type expected by our JavaBean, as shown in phase (2) in Figure 10-17. Leave the default mapping (string) for the input parameter.
- ▶ The InquirePartsRemoteInterface entry defines the mechanism to map the output from our JavaBean to the XML element in the reply message, shown as phase (4) in Figure 10-17. Leave the default mapping (DOM Element) for the result XML.
- ▶ Click *Next*.

Verifying the SOAP bindings

The next dialog of the wizard displays the encodings specified for the *SOAP encoding* and *Literal XML encoding*. These are set based on our previous definitions in the wizard and cannot be changed in our example. Click *Next*.

Web service test client

This dialog provides the option of automatically launching the Application Developer *universal test client*. We will first investigate the generated code and therefore, do not select this option. Click *Next*.

Generating a sample client

Because we selected to create a Java client proxy to our Web service, we also have the option to generate a Web application that uses this proxy to test the completed service (illustrated in Figure 10-12 on page 334).

Figure 10-21 shows the Web Service Sample Generation panel.

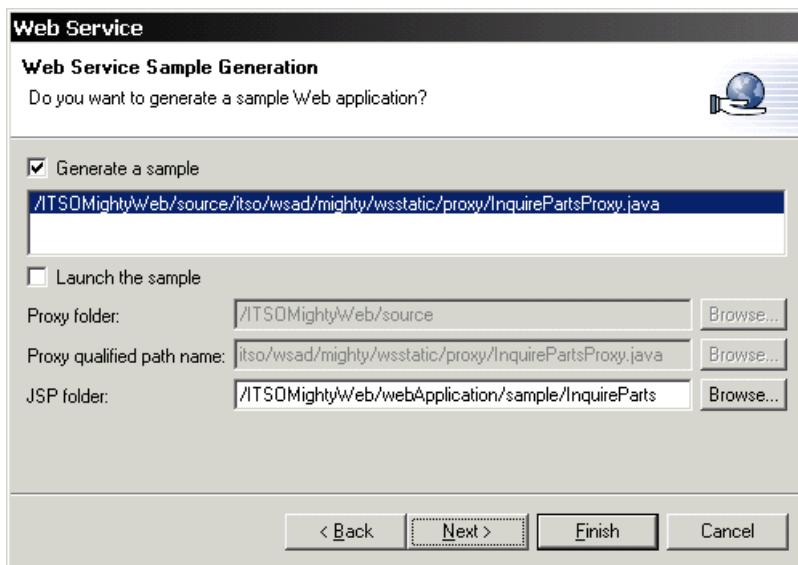


Figure 10-21 Creating a sample client for the Web service

- ▶ Select the *Generate a sample* checkbox.
- ▶ Leave the *Launch the sample* option unchecked (we will be examining the generated files before executing the application).

- ▶ Leave the folder location in the Web application (sample/InquireParts).
- ▶ Click *Next*.

Publishing the Web service

The final dialog of the Web service wizard enables you to publish the Web service by launching the UDDI explorer. We will not be publishing the Web service in this chapter.

- ▶ Click *Finish* to complete the wizard.

Application Developer now generates the Web service and associated files. The Web service is then published to the associated server and the server is started if it is not already running.

In this section we made a single pass through the Web service wizard and explained some of the terminology and options available. Upon completion, a number of files have been created. The next part of the chapter focuses on looking at these files in more detail.

Investigating the generated files

As we have seen in the previous sections, the Web service wizard is extremely comprehensive and powerful, and is responsible for automating a large quantity of artifacts based on its input. We must now gain a better understanding of what these files contain and how they relate to each other. It is intended to accompany the WSDL overview contained in “WSDL primer” on page 277.

We improve our understanding by completing the following steps:

- ▶ Work with the XML schema we created earlier in the book.
- ▶ Examine the WSDL files for the Web service.
- ▶ Edit the service implementation and interface.
- ▶ Explain the SOAP deployment descriptor.
- ▶ View the Web services through the SOAP admin application.
- ▶ Gain an understanding of the SOAP API by looking at the client proxy.
- ▶ Use the sample client to test the Web service.
- ▶ Work with the TCP/IP monitoring server to view the SOAP message contents as the Web service executes.

An overview of the generated folders and files is shown in Figure 10-22.

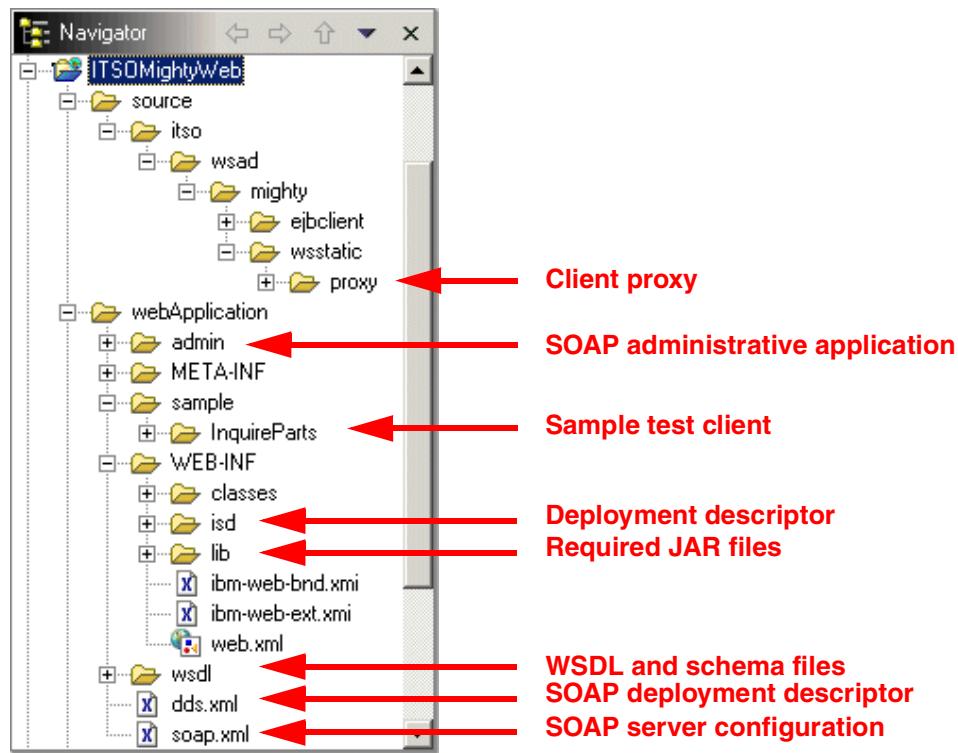


Figure 10-22 Generated folders and files

Working with the previously created XML schema

Examining the files created by the Web service wizard, we see the XSD file `InquireParts.xsd` (in the `wsdl` folder). This file was created when we specified the XSD filename in the wizard (Figure 10-15 on page 337), but it is only a skeleton because the wizard does not know our XML file structure.

We want to use the XSD we created previously in “Generating an XML file and an XML schema from SQL” on page 111. We have to replace the wizard generated XSD file with the one we previously developed. To do this copy the XSD file we created from the `ITSOMightyXML` project to the `ITSOMightyWeb` project:

- ▶ Select the `InquireParts.xsd` from the `ITSOMightyXML` folder.
- ▶ Select *Copy* from the context menu.
- ▶ Select the `ITSOMightyWEB/webApplication/wsdl` folder and click *OK*.
- ▶ Click *Yes* on the overwrite warning.

Web service WSDL files

The first files we examine after completing the Web service wizard are the WSDL files that define the service interface and implementation. Expand the wsdl folder in Figure 10-22; you can see the InquireParts.xsd file and two WSDL files:

InquireParts-service.wsdl	The service implementation file that defines where to invoke the service.
InquireParts-binding.wsdl	The service interface file that defines the specification of the service with parameters and results.

The XML schema file (InquireParts.xsd) is generated if you do not overwrite the default mapping (Figure 10-18 on page 341).

Changing the service implementation

Open the service implementation file, InquireParts-service.wsdl (Figure 10-23).

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="InquirePartsService"
    targetNamespace="http://localhost:8080/ITSOMighty/wsdl/InquireParts-service.wsdl"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:binding="http://www.inquireparts.com/definitions/InquirePartsRemoteInterface"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://localhost:8080/ITSOMighty/wsdl/InquireParts-service.wsdl">
    <import
        location="http://localhost:8080/ITSOMighty/wsdl/InquireParts-binding.wsdl"
        namespace="http://www.inquireparts.com/definitions/InquirePartsRemoteInterface" />
    <service name="InquirePartsService">
        <port binding="binding:InquirePartsBinding" name="InquirePartsPort">
            <soap:address
                location="http://localhost:8080/ITSOMighty/servlet/rpcrouter"/>
        </port>
    </service>
</definitions>
```

Figure 10-23 Generated service implementation file, InquireParts-service.wsdl

The elements shown in bold define the location of the service interface file and the URL of the SOAP router servlet required to invoke the service.

Note that there are two references (underlined) to a generated XML namespace (in addition to the two standards defined in <http://schemas.xmlsoap.org>), which we could not modify in the Web service wizard.

- ▶ Change both of the underlined entries (best in the Design view):

from: http://www.inquireparts.com/definitions/InquirePartsRemoteInterface
 to: http://www.redbooks.ibm.com/XXXXXXX/definitions/InquirePartsRemoteInterface
 XXXXXX = your last name or initials

Changing the service interface

Next open the service interface file, InquireParts-binding.wsdl (Figure 10-24).

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="InquirePartsRemoteInterface"
targetNamespace="http://www.inquireparts.com/definitions/InquirePartsRemoteInterface"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://www.inquireparts.com/definitions/InquirePartsRemoteInterface"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsd1="http://www.inquireparts.com/schemas/InquirePartsRemoteInterface">
  <import location="http://localhost:8080/ITSOMighty/wsdl/InquireParts.xsd"
    namespace="http://www.inquireparts.com/schemas/InquirePartsRemoteInterface" />
  <message name="inquirePartsRequest">
    <part name="partNumber" type="xsd:string"/>
  </message>
  <message name="inquirePartsResponse">
    <part name="result" type="xsd1:anyElement"/>
  </message>
  <portType name="InquireParts">
    <operation name="inquireParts" parameterOrder="partNumber">
      <input message="tns:inquirePartsRequest" name="inquirePartsRequest"/>
      <output message="tns:inquirePartsResponse" name="inquirePartsResponse"/>
    </operation>
  </portType>
  <binding name="InquirePartsBinding" type="tns:InquireParts">
    <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="inquireParts">
      <soap:operation soapAction="" style="rpc"/>
      <input name="inquirePartsRequest">
        <soap:body
          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          namespace="urn:InquireParts" use="encoded"/>
      </input>
      <output name="inquirePartsResponse">
        <soap:body
          encodingStyle="http://xml.apache.org/xml-soap/literalxml"
          namespace="urn:InquireParts" use="literal"/>
      </output>
    </operation>
  </binding>
</definitions>
```

Figure 10-24 Generated service interface file, InquireParts-binding.wsdl

The elements of the WSDL file highlighted in bold type show the key features required to define the interface of the Web service—the input and output parameters, together with any necessary XML schemas that are required and the name of the method to invoke. Table 10-1 shows the mapping between the elements in the WSDL service interface file, and the service implementation Java class.

Table 10-1 Mapping between WSDL and Java classes

WSDL	Java
Port type, Binding	Class
Operation	Method
Message	Input or output method signature
Part	Individual parameter
Part type	Java type for parameter

There are four items in this file that we have to change, all of which are underlined:

- ▶ We must change the XML namespaces in the interface file to match the namespace used in the implementation file. Change both targetnamespace and xmlns:tns entries (first two underlined):

```
from: http://www.inquireparts.com/definitions/InquirePartsRemoteInterface
to: http://www.redbooks.ibm.com/XXXXXXXX/definitions/InquirePartsRemoteInterface
      XXXXXXXX = your last name or initials
```

- ▶ We must change the XML namespaces for our XML in the interface file to match the namespace used in the XSD we previously built. Change both xmlns:xsd1 and the <import> namespace attribute (second two underlined) :

```
from: http://www.inquireparts.com/schemas/InquirePartsRemoteInterface
to: http://www.redbooks.ibm.com/ITSOWSAD/schemas/InquireResults
```

- ▶ Save the changes in both WSDL files and close the editors.

Attention: The changes in the WSDL files suggest a name of

```
http://www.redbooks.ibm.com/XXXXXXXX/definitions/InquirePartsRemoteInterface
```

Replace XXXXXXXX with a name of your own (for example your last name or initials). This avoids duplicate names when we publish the Web service to the IBM UDDI Test Registry.

However, do not change the namespace that refers to the XSD schema:

```
http://www.redbooks.ibm.com/ITSOWSAD/schemas/InquireResults
```

Generating SOAP deployment descriptors

The next files to examine, which are generated by the Web service wizard, are those which define how the Web service is deployed:

InquireParts.isd A .isd file is created by the wizard for each Web service that is generated, and contains the service definition required for deployment. Its only purpose is to be used to build the dds.xml file.

dds.xml The deployment descriptor for all of the Web services in the project. Each time the Web service wizard is completed, all of the .isd files in the project are concatenated together to form this file. This is loaded by the SOAP router servlet when it is initiated by the application server in order to determine the Web services that require deployment.

soap.xml This contains the configuration information for the Apache SOAP server and router servlet. The main purpose is to define the configuration manager that is used to load the deployment descriptor for the Web services. The default is com.ibm.soap.server.XMLDrivenConfigManager, which looks for the dds.xml file.

The loading hierarchy of the application and how it relates to the various deployment descriptors is shown in Figure 10-25.

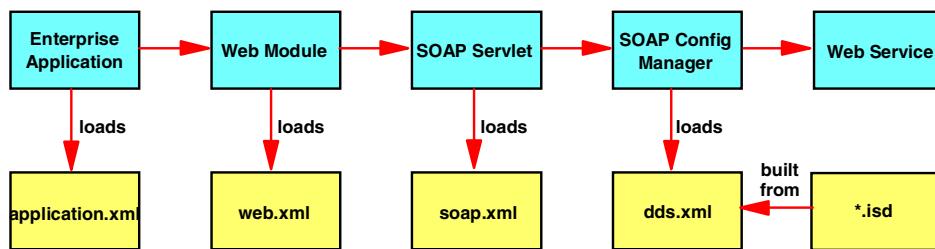


Figure 10-25 Loading hierarchy for a Web service

dds.xml

Open the dds.xml file (Figure 10-26). As you can see, this file defines the name of the class that implements the Web service and the method to invoke. The scope="Application" attribute is directly related to the Web service scope that we selected in Figure 10-15 on page 337.

Close all files.

```

<root>
<isd:service xmlns:isd="http://xml.apache.org/xml-soap/deployment"
  id="urn:InquireParts" checkMustUnderstands="false">
  <isd:provider type="java" scope="Application" methods="inquireParts">
    <isd:java class="itso.wsad.mighty.ejbclient.InquireParts"
      static="false"/>
  </isd:provider>
</isd:service>
</root>

```

Figure 10-26 Web service deployment descriptor, dds.xml

SOAP router servlets

The final element required to complete our server side Web service implementation is the definition of the SOAP router servlets in the deployment descriptor of our Web project.

- ▶ Open the ITSOMightyWeb projects web.xml file.
- ▶ Switching to the *Servlets* tab in the editor shows two servlets (Figure 10-27).

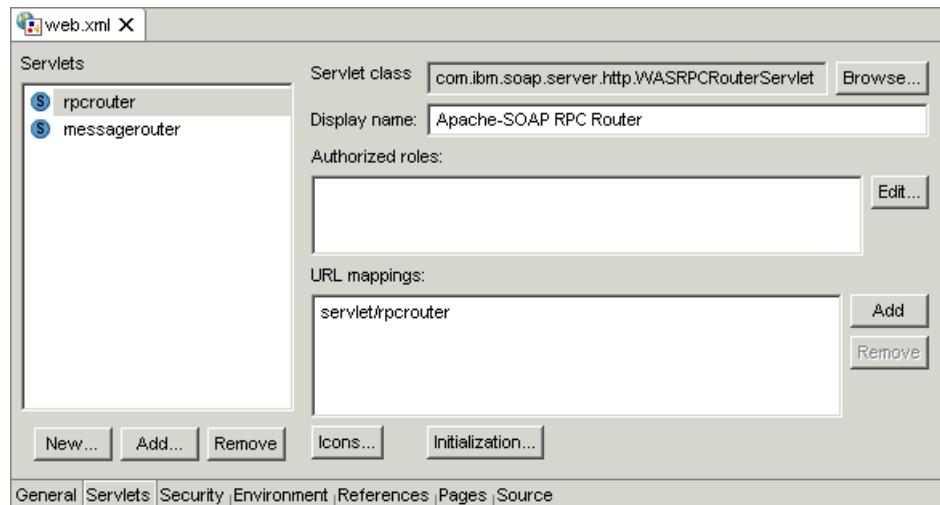


Figure 10-27 Servlet definitions for the SOAP routers

- | | |
|----------------------|---|
| rpcrouter | This is the SOAP router servlet invoked for all RPC style calls to the Web service. This will be used by our application. |
| messagerouter | This router is used for all message style calls to the Web service. |

As we saw in Figure 10-23 on page 347, the correct router to invoke is defined in the service implementation WSDL file with the <soap:address> tag. SOAP routers are defined for each Web application module deployed on the application server, so multiple routers on different URLs can exist on the same server configuration.

The implementation of these SOAP routers is provided in a file called soapcfg.jar file, which is added to a webApplication/lib folder by the Web service wizard when it is invoked for the first time in the project.

Close the web.xml file and ignore any changes made.

Viewing the deployed Web services

Included in our Web project is a Web application that can be used to view the deployed Web services in the Web module and to modify their configuration. These files have been added to the webApplication/admin folder of the Web project.

To invoke the admin client:

- ▶ Select the file called index.html and click on the *Run on Server* item in its context menu.
- ▶ Click on the *List all services* link on the left hand side of the home page. The resulting service listing, generated from the dds.xml file, appears (Figure 10-28).

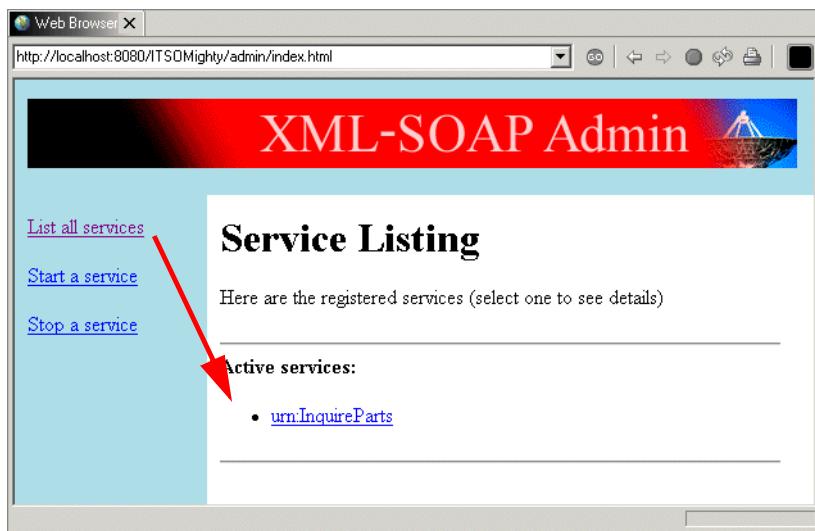


Figure 10-28 Service listing for the Web module

- As you can see, we have successfully deployed one Web service, urn:InquireParts. Click on the link for this service to see further details (Figure 10-29).

Property	Details
ID	urn:InquireParts
Scope	Application
Provider Type	java
Provider Class	itso.wsad.mighty.ejbclient.InquireParts
Use Static Class	false
Methods	inquireParts
Type Mappings	
Default Mapping Registry Class	

Figure 10-29 Service information for the InquireParts Web service

This allows us to view all of the important information about the Web service we have developed, including the provider class, its exposed methods, and the provider type—in this case, a Java class.

Note that the deployment process is not an explicit step—the Web service wizard generates an entry in `dds.xml`, which is picked up by the router servlet when it is initialized.

It is also possible to start and stop specific services from this administration console using the other links on the left hand side (Figure 10-28).

This can be performed while an application that uses the Web service is executing. This scenario may be useful when testing how a client application handles the scenario that the Web service is unavailable.

Close the browser.

Web service client proxy

An optional step in the Web service wizard generates a client proxy to the Web service. This was called `InquirePartsProxy.java` and can be found in the `itso.wsad.mighty.wsstatic.proxy` package in the Web projects source folder.

This class can be completely regenerated with access only to the generated interface and implementation WSDL files. It is worth investigating this class in more detail because it demonstrates how to programmatically invoke the Web service from a client.

One of the key instance variables in the class is defined as:

```
private Call call = new Call();
```

The `Call` class (from the `org.apache.soap.rpc` package) is the main component responsible for completing the SOAP call between the client and the server. Note that the API shields you from the implementation details of the SOAP message contents and significantly simplifies the calling mechanism.

The proxy class sets up the URL of the SOAP router servlet in the declaration of the `stringURL` variable. This is then used in the `getURL` method to determine the URL to be invoked from the Web service.

```
private String stringURL = "http://localhost:8080/ITSMighty/servlet/rpcrouter";
```

The proxy has a single method to invoke the service, `inquireParts`, which has the same method signature as the method included in the Web service implementation JavaBean on the server side.

The `inquireParts` method is listed Figure 10-30.

The `Call` object is set up before it is invoked by:

- ▶ Setting the method name and encoding style

```
call.setMethodName("inquireParts");
call.setEncodingStyleURI(Constants.NS_URI_LITERAL_XML);
```

- ▶ Setting the target URN to invoke the service

```
String targetObjectURI = "urn:InquireParts";
call.setTargetObjectURI(targetObjectURI);
```

- ▶ Setting up the call parameters

```
Vector params = new Vector();
Parameter partNumberParam = new Parameter("partNumber",
                                         java.lang.String.class, partNumber, Constants.NS_URI_SOAP_ENC);
params.addElement(partNumberParam);
call.setParams(params);
```

- Finally the Web service is invoked

```
Response resp = call.invoke(getURL(), SOAPActionURI);
```

```
public synchronized org.w3c.dom.Element inquireParts(java.lang.String partNumber)
throws Exception
{
    String targetObjectURI = "urn:InquireParts";
    String SOAPActionURI = "";

    if(getURL() == null)
    {
        throw new SOAPException(ConstantsFAULT_CODE_CLIENT,
        "A URL must be specified via InquirePartsProxy.setEndPoint(URL).");
    }

    call.setMethodName("inquireParts");
    call.setEncodingStyleURI(Constants.NS_URI_LITERAL_XML);
    call.setTargetObjectURI(targetObjectURI);
    Vector params = new Vector();
    Parameter partNumberParam = new Parameter("partNumber", java.lang.String.class,
                                                partNumber, Constants.NS_URI_SOAP_ENC);
    params.addElement(partNumberParam);
    call.setParams(params);
    Response resp = call.invoke(getURL(), SOAPActionURI);

    //Check the response.
    if (resp.generatedFault())
    {
        Fault fault = resp.getFault();
        call.setFullTargetObjectURI(targetObjectURI);
        throw new SOAPException(fault.getFaultCode(), fault.getFaultString());
    }
    else
    {
        Parameter refValue = resp.getReturnValue();
        return ((org.w3c.dom.Element)refValue.getValue());
    }
}
```

Figure 10-30 Source code for the inquireParts method

This code defines the URN required to invoke the Web service, the default encoding style for the proxy (SOAP encoding), and the URL of the target SOAP router. Again, these values can be obtained from the generated WSDL files.

The important aspects of the method are shown in bold type. The method name to invoke on the service is defined using setMethodName.

The encoding styles are set in the call object for both the parameter and the result of the Web service (Figure 10-31). The net effect of this is that SOAP encoding is used for the input message to the Web service call and literal XML encoding is used for the output message. This result matches what we defined in Figure 10-18 on page 341.

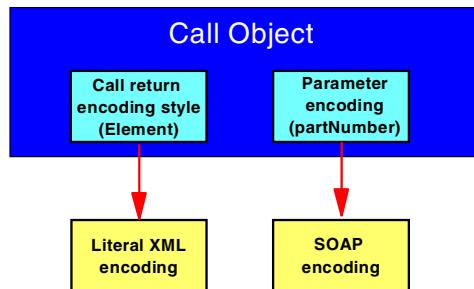


Figure 10-31 The encoding styles for the operation

The parameters for the Web service are constructed into a Vector of types `org.apache.soap.rpc.Parameter`.

To invoke the service, the proxy calls the `invoke` method, which returns an object of type `org.apache.soap.rpc.Response`. Finally, the `getReturnValue` method is called on the response to obtain the wrapped XML element that contains the results of the parts inquiry.

Close the Java editor, and ignore any changes made.

Web service sample client

Because we selected the option to generate a client proxy for your Web service, the wizard also provided the option to build a browser based sample client that tests the proxy and Web service. Like the EJB test client, this is a great way of performing unit and connectivity tests on the Web service.

Four files are generated in the `webApplication/sample/InquireParts` folder of the Web project for the test client:

- TestClient.jsp** This is the frameset for the sample client and should be the URL you use to launch the browser.
- Method.jsp** This page provides a list of the available Web service methods. When a link is selected, `Input.jsp` is invoked.
- Input.jsp** This provides a form for the input parameters for each method defined in `Method.jsp`. On submitting the form, `Result.jsp` is invoked.

Result.jsp	Contains an instance of the client proxy, and invokes it using the input parameters from the Input.jsp. On return it invokes a domWriter method (in the JSP) that writes the returned XML document to the output stream. This is a useful snippet of code that you may wish to use in your own client, and we actually used it as the basis of Figure 4-23 on page 126 in the XML chapter.
-------------------	--

To start the sample application:

- ▶ Select TestClient.jsp and execute the *Run on Server* method.
- ▶ The embedded browser should be launched, and after a short delay as the JSPs are compiled, the test client is displayed.
- ▶ There should only be one item listed on the *Methods* frame—*inquireParts*. Click on this link.
- ▶ Enter a valid part number in the input field such as M100000003. Click *Invoke*.

The results pane should display the XML document returned from the Web service (Figure 10-32).

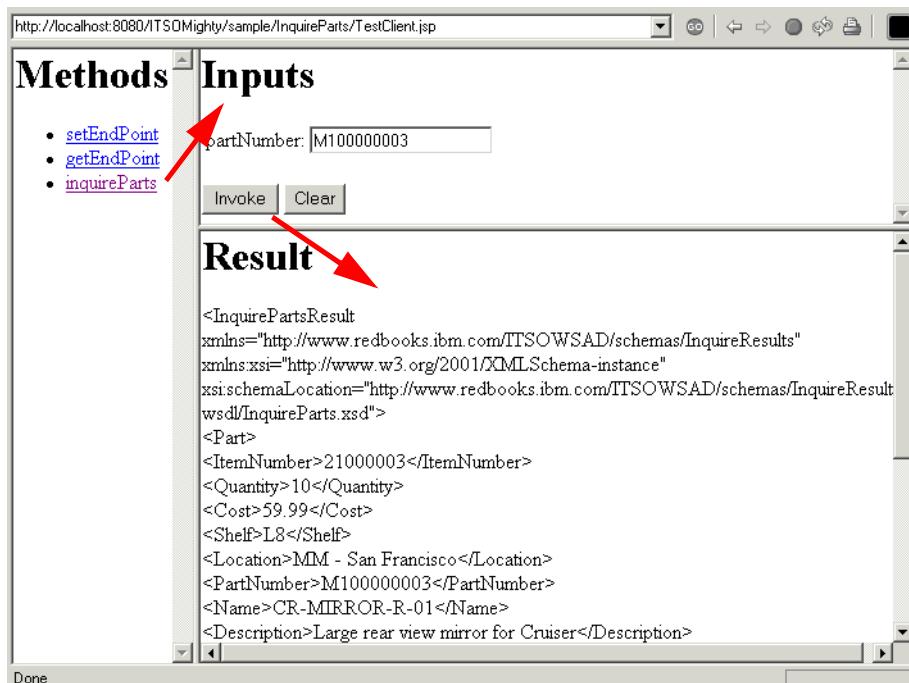


Figure 10-32 Web service sample application

Your Web service implementation is now complete and tested. Well done!

Using the TCP/IP Monitoring Server to view message contents

The SOAP API shields the developer from the exact content of the inbound and outbound SOAP messages. So how do we see what is travelling over the wire?

Application Developer provides an additional server instance type, *TCP/IP Monitoring Server* to address exactly this requirement.

In the server perspective, select the *ITSOwsADServer* server project and select the *File -> New -> Server Instance and Configuration* menu and complete the dialog (Figure 10-33), then click *Finish*.

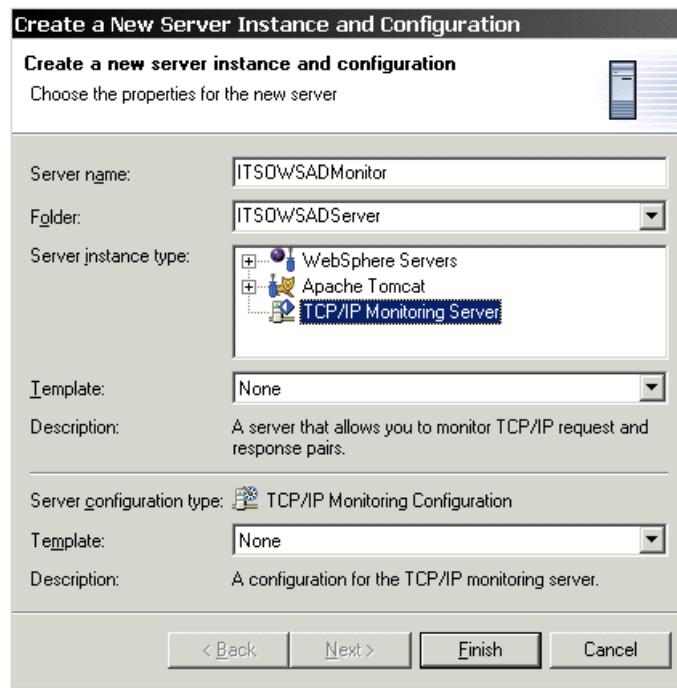


Figure 10-33 Creating a TCP/IP Monitoring Server

In the server perspective, a new entry should appear in the *Server Configuration* and *Servers* views. Note that unlike other server instance types, you cannot add or remove projects from the server configuration.

Ensure that your *ITSOwsADWebSphere* application server is already started and then start the new *ITSOwsADMonitor* instance. In the console, the following message appears:

```
Monitoring server started  
localhost:8081 -> localhost:8080
```

This means that any requests to port 8081 will be displayed to the client, then redirected to port 8080 to invoke the Web service. These ports can be changed by editing the properties of the server configuration.

Next, add the *TCP/IP Monitor* view to the server perspective by selecting *Perspective -> Show View -> TCP/IP Monitor*.

A new view appears (Figure 10-34). It consists of three panes, showing a list of requests, and the contents of each request and response. You can move the view over the Console/Servers view.

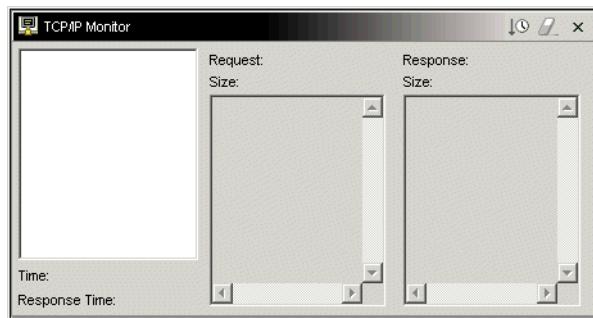


Figure 10-34 *TCP/IP Monitor* view

The monitor can be used for two purposes. First, we will use it to view the requests sent between the browser and the sample Web service client, as illustrated in Figure 10-35.

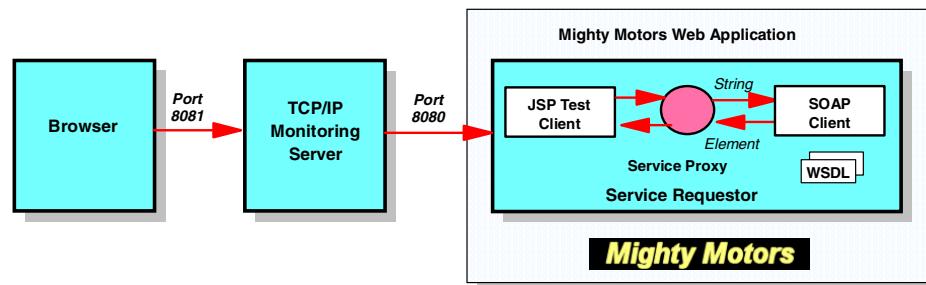


Figure 10-35 Using the TCP/IP monitoring server to view the HTTP requests

- ▶ Select the `TestClient.jsp` file and select *Run on Server*.
- ▶ Application Developer detects that there are servers this client could be invoked on, and prompts you with the dialog box shown in Figure 10-36.

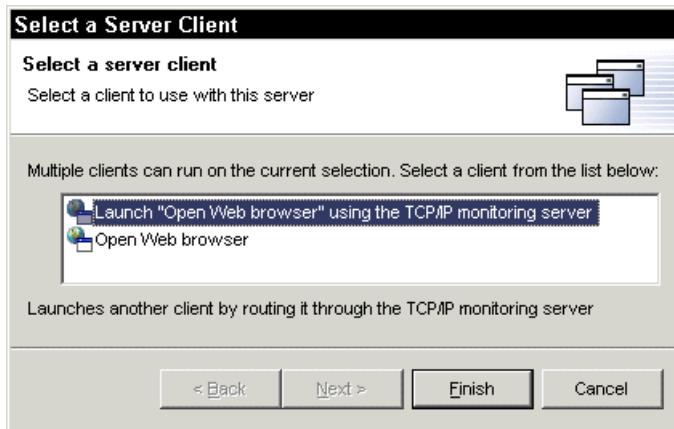


Figure 10-36 Execution server selection

- ▶ Select *Launch “Open Web Browser” using TCP/IP monitoring server*. Click *Finish*.
- ▶ Execute the application as before, invoking the *inquireParts* method with a part number of M100000003, then return to the *TCP/IP Monitor* view.
- ▶ Expand the host entry and select a request to see the responses (Figure 10-37).



Figure 10-37 Viewing the contents of a specific request

The second, and more interesting option, is to use the monitor to view only the communication between the Web service client proxy and the SOAP router servlet, as illustrated in Figure 10-38.

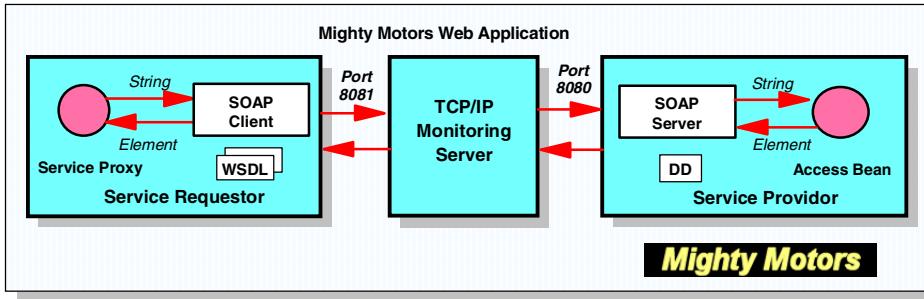


Figure 10-38 Using the TCP/IP monitoring server to view the SOAP requests

To configure this, we must change the binding URL defined in the client proxy class:

- ▶ While leaving both the WebSphere and TCP/IP monitoring servers executing, open the `InquirePartsProxy.java` file in the Java editor.
- ▶ Change the URL line:


```
from: private String stringURL = "http://localhost:8080/ITSOMighty/servlet/rpcrouter";
to:    private String stringURL = "http://localhost:8081/ITSOMighty/servlet/rpcrouter";
```
- ▶ Save the file and close the editor.
- ▶ Reinvoke the test client using *Run on Server* again, but this time selecting only the *Open Web Browser* option in the dialog, which avoids using the TCP/IP monitor for the browser requests. Click on *Finish*.
- ▶ Perform the same test as in previous steps, invoking the *inquireParts* method with a part number of M100000003.
- ▶ The TCP/IP Monitor view should now show a new request to the SOAP router URL /ITSOMighty/servlet/rpcrouter. Select this item, and you will be able to view the entire contents of the SOAP request and response, as shown in Figure 10-39.

Note that you have to scroll the pane to see the XML because the whole output is on one line.

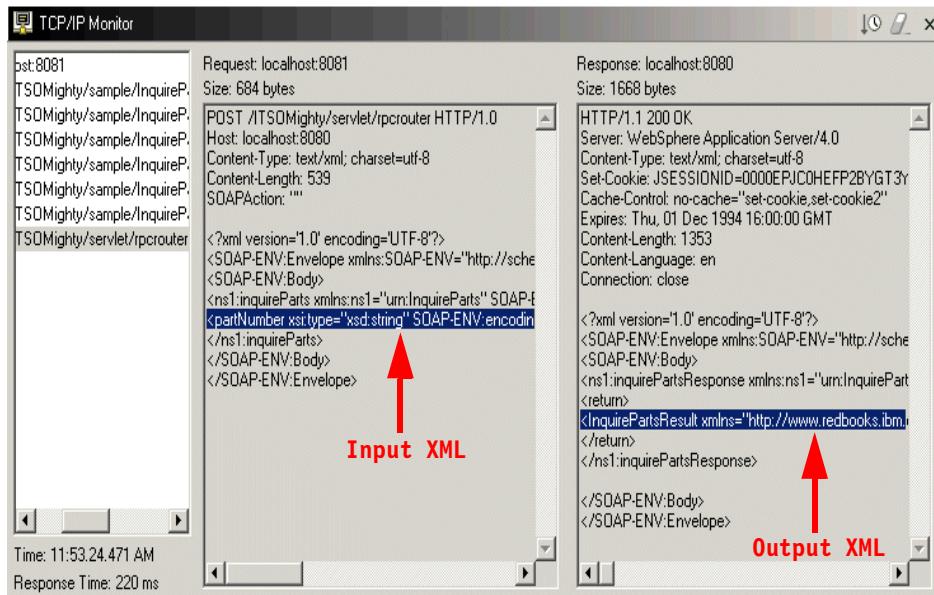


Figure 10-39 Monitoring the contents of the SOAP request

To execute the test client in future without the TCP/IP monitor, ensure that you change the port defined in the client proxy back from 8081 to 8080.

Stop the TCP/IP monitoring server, but leave WebSphere test environment instance running before progressing onto the next section.

The Web service has now been successfully developed. After running through the Web service wizard, we amended the generated WSDL files to support our schema namespace conventions and then executed the service in WebSphere using the generated client proxy and test client.

Creating the Almaden Autos Web service client

Now that we have successfully created and tested our Mighty Motors Web service, the next step is to create a client in the Almaden Autos Web application, which is used if the requested part is not available in their local inventory. This demonstrates how to build a Web service client from a given set of WSDL documents.

For the first phase of this implementation, we assume that Mighty Motors sends the WSDL files to Almaden Autos, and that they integrate the Web service into their application at development time using static bindings. This is a more realistic scenario than directly copying the client proxy between projects. The more advanced topic of dynamic binding to a Web service and searching for implementations in a UDDI registry is covered in the next chapter.

The scenario is illustrated in Figure 10-40, and should be compared to the generated types illustrated in Figure 10-12 on page 334.

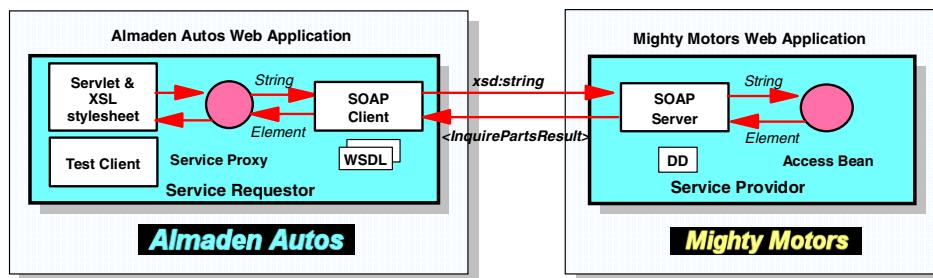


Figure 10-40 Basic design outline for Almaden Autos Web service requester

To complete the development of the Almaden Autos Web service requester, perform the following steps:

- ▶ Copy the generated WSDL from the Mighty Motors project to the Almaden Autos project.
- ▶ Use the Web service client wizard to generate a client proxy from the WSDL.
- ▶ View the generated files.
- ▶ Test the Web service using a new Web service test client.
- ▶ Build the client application that includes a new servlet and an XSL style sheet to transform the results into HTML displayed in the browser.
- ▶ Amend the Web application developed in Chapter 3, “Web development with Application Developer” on page 55 to include a link to our new servlet.
- ▶ Test the complete application.

Copying the WSDL service implementation

The Web service client proxy can only be created from a WSDL service implementation file contained in its own project:

- ▶ Create a wsdl folder in the webApplication folder of the ITS0A1maWeb project to contain the copied files.
- ▶ Create a static folder in the webApplication/wsdl folder.
- ▶ Copy the InquireParts-service.wsdl file from the ITS0MightyWeb project's webApplication/wsdl folder to the new webApplication/wsdl/static folder in the ITS0A1maWeb project.

Creating the Web service client

We will now run through the Web service wizard again, starting at the section for creating the client proxy. The generated proxy and test client should be similar to that initially created by the Web service wizard in the previous section, but in the Almaden Autos Web application.

To do this:

- ▶ Start the ITS0SADWebSphere server (if it is not running).
- ▶ Select the InquireParts-service.wsdl file in the ITS0A1maWeb project and *New -> Other -> Web Services -> Web Service client*. Click *Next*.
- ▶ ITS0A1maWeb should be preselected as Web project. Select the *Generate a sample* option and click *Next*.
- ▶ On the next panel, the WSDL file is preselected:
ITS0A1maWeb/webApplication/wsdl/static/InquireParts-service.wsdl
- ▶ Click *Next*.

At this stage, the Web service client wizard connects to the ITS0SADWebSphere server instance to retrieve the rest of the required Web service files, specifically the service interface file InquireParts-binding.wsdl and the XML schema file InquireParts.xsd.

- ▶ If this is successful, the wizard displays the WSDL bindings and prompts for the proxy class. Enter a proxy qualified class name of:
`itso.wsad.alma.wsstatic.proxy.InquirePartsProxy`
- ▶ Select the *Show mappings* option and click *Next*.
- ▶ View the XML to Java mappings. This should reflect the settings we defined earlier in the Web service wizard when we initially created the service (Figure 10-20 on page 343). The settings are default JavaBean mapping for the parameter and DOM Element mapping for the result. Click *Next*.

- ▶ Click *Next* on the SOAP binding mapping configuration.
- ▶ Click *Next* on the Web service test client (Do not select the option).
- ▶ Click *Next* on generate a sample (the option is selected, but do not launch the sample). The folder is `ITSOAlmaWeb/webApplication/sample/InquireParts`.
- ▶ Click *Finish*.

Examining the generated client files

As we have for each previous section, it is worth spending a few moments examining the files generated by the Web service client wizard as these differ slightly from those created by the Web service wizard that we used when initially building the service.

- ▶ In the `itso.wsad.alma.wsstatic.proxy` package we find the client proxy:
- InquirePartsProxy** This is the client proxy for the Web service, which provides a JavaBean interface to the SOAP call generated from the supplied WSDL definition. This class should be identical to the proxy class previously generated in the `ITSOMightyWeb` project (“Web service client proxy” on page 354).
- ▶ In `webApplication/sample/InquireParts` we find the sample test client, a set of four JSPs that use the client proxy to test the invocation of the Web service. This sample test client is identical to the test client generated in the `ITSOMightyWeb` project.

Testing the Web service requestor

To test the client proxy:

- ▶ Ensure that `ITSOWSADWebSphere` is the default server for the `ITSOAlmaWeb` project, and not the server used for remote testing in the previous chapter. In the properties dialog, select Server Preference and select the `ITSOWSADWebSphere` server.
- ▶ Start the sample client by selecting `TestClient.jsp` from the `webApplication/sample/InquireParts` folder in `ITSOAlmaWeb` and select *Run on Server*.
- ▶ When prompted on which server to use, select the *Open Web browser* option to directly test against the application server.
- ▶ Test the `inquireParts` method with a part number of `M100000003`.

The resulting frame should again display the contents of our XML element.

Building the client application

We now develop a simple client in the Web application to display the results of the Web service inquiry to the browser.

The current Almaden Autos Web application will be modified to include an additional link to a new servlet if no parts are returned from a local inventory inquiry. This new servlet invokes the Web service client proxy, and then passes the returned XML element to an XSL style sheet to display as HTML in the browser. The development of XSL style sheets from XML schemas is defined in Chapter 4, “XML support in Application Developer” on page 91.

We will be using the Apache Xalan XSL parser during this exercise, so add the class path variable WAS_XALAN into the ITSOAlmaWeb project’s *Java build path*.

Creating the XSL style sheet

We create a simple style sheet to display the returned result set in a table, similar in style to the table created by the Database Web pages wizard we used to create the initial Almaden Autos site.

In the Web perspective:

- ▶ Create a new folder in the ITSOAlmaWeb projects webApplication folder called stylesheets.
- ▶ Create a new folder in the ITSOAlmaWeb projects webApplication/stylesheets folder called static.
- ▶ In the webApplication/stylesheets/static folder of the ITSOAlmaWeb project, create a new file called InquirePartsResultTable.xsl. This can be found using *File -> New -> Other -> Simple -> File*.
- ▶ Open the new XSL file. Switch to the *Source* tab in the XSL editor and complete the style sheet as shown in Figure 10-41. (or import the code from the sampcode\wsstatic directory.)

```

<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:res="http://www.redbooks.ibm.com/ITSOVSAD/schemas/InquireResults">
<xsl:output method="html"/>
<xsl:template match="res:InquirePartsResult">
<HTML>
    <HEAD>
        <TITLE>Inventory Inquiry Results</TITLE>
    </HEAD>
    <BODY BGCOLOR="#fffff">
        <FONT COLOR="#0000ff">
            <CENTER>
                <IMG border="0" height="33" src="images/almadenautos.gif" width="400"/>
                <H1>Mighty Motors Inventory Inquiry Results</H1>
            </CENTER>
        </FONT>
        <table border = "1">
            <tr><bold>
                <TH BGCOLOR="#00ffff">Part Number</TH>
                <TH BGCOLOR="#00ffff">Name</TH>
                <TH BGCOLOR="#00ffff">Quantity</TH>
                <TH BGCOLOR="#00ffff">Cost</TH>
                <TH BGCOLOR="#00ffff">Location</TH>
            </bold></tr>
            <xsl:apply-templates/>
        </table>
    </BODY>
</HTML>
</xsl:template>

<xsl:template match="res:Part" >
    <tr>
        <td><xsl:value-of select="res:PartNumber"/></td>
        <td><xsl:value-of select="res:Name"/></td>
        <td><xsl:value-of select="res:Quantity"/></td>
        <td><xsl:value-of select="res:Cost"/></td>
        <td><xsl:value-of select="res:Location"/></td>
    </tr>
</xsl:template>
</xsl:stylesheet>

```

Figure 10-41 XSL style sheet to display Mighty Motors Web service results

This style sheet creates a results table containing the part number, name, quantity and cost and then creates a row in the table for each part in the XML document.

- Close the XSL editor and save all changes.

Creating the new servlet

To create a new servlet to handle our Web service request:

- ▶ Select *File -> New -> Other -> Web -> Servlet* and complete the servlet wizard (Figure 10-42). Click *Next*.

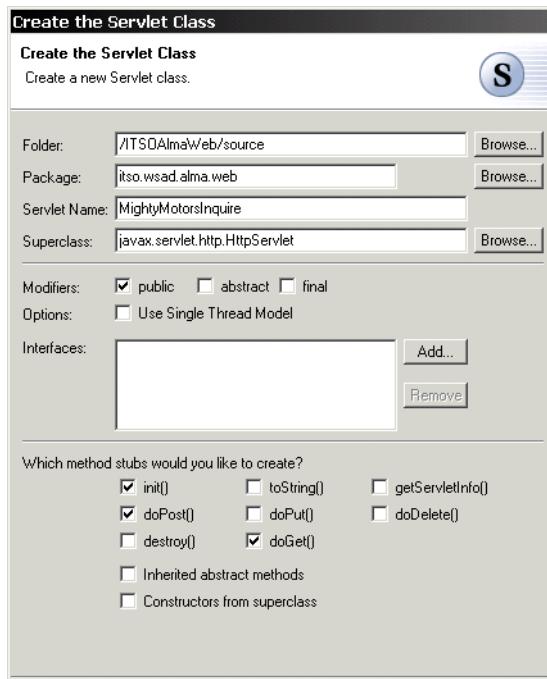


Figure 10-42 Creating a new servlet for the ITSOAlmaWeb project

- ▶ Accept the default values to add the servlet to web.xml and click *Finish*.

Completing the servlet

We must now code the body of the new servlet with the code shown in Figure 10-43. (The code is in the sg246292\sampcode\wsstatic directory.)

```

package itso.wsad.alma.web;

import javax.servlet.http.HttpServlet;

public class MightyMotorsInquire extends HttpServlet {

    public void init() throws javax.servlet.ServletException {
    }

    public void doPost(
        javax.servlet.http.HttpServletRequest request,
        javax.servlet.http.HttpServletResponse response)
        throws javax.servlet.ServletException, java.io.IOException {
            performTask(request, response);
    }

    public void doGet(
        javax.servlet.http.HttpServletRequest request,
        javax.servlet.http.HttpServletResponse response)
        throws javax.servlet.ServletException, java.io.IOException {
            performTask(request, response);
    }

    public void performTask(
        javax.servlet.http.HttpServletRequest request,
        javax.servlet.http.HttpServletResponse response) {
        try {
            response.setContentType("text/html; charset=UTF-8");
            javax.xml.transform.TransformerFactory tFactory =
                javax.xml.transform.TransformerFactory.newInstance();
            javax.xml.transform.Source xs1Source =
                new javax.xml.transform.stream.StreamSource(
                    new java.net.URL ("http://localhost:8080/ITSOAlma/stylesheets/
                        static/InquirePartsResultTable.xsl").openStream());
            javax.xml.transform.Transformer transformer =
                tFactory.newTransformer(xs1Source);
            java.io.PrintWriter out = response.getWriter();

            // For SOAP over HTTP
            itso.wsad.alma.wsstatic.proxy.InquirePartsProxy proxy =
                new itso.wsad.alma.wsstatic.proxy.InquirePartsProxy();
            org.w3c.dom.Element result = proxy.inquireParts
                (((String)request.getSession().getAttribute("partNo")));
            javax.xml.transform.Source xmlSource =
                new javax.xml.transform.dom.DOMSource(result);

            transformer.transform(xmlSource,
                new javax.xml.transform.stream.StreamResult(out));
        } catch (Throwable e) {
            e.printStackTrace();
        }
    }
}

```

Figure 10-43 Source code for the servlet to invoke the Web service client proxy

The two key methods are highlighted in bold. They demonstrate how to create a new instance of the Web service client proxy that invokes its method by passing in a part number obtained from the HTTP session, and how to invoke a XSL transformation using an XML DOM element as the input.

Linking the new servlet into the Web application

The final step is to link the new servlet into the existing Almaden Autos Web site. To do this we modify the PartsMasterView.jsp by:

- ▶ Adding a No Data Returned message
- ▶ Add an option for invoking the newly created MightyMotorsInquire servlet if no data is returned

Adding the “No Data Returned” message

To display the message we will add a scriptlet around the output table. The scriptlet will cause the message to be displayed instead of the empty table, if no data was returned in the results set.

Open the PartsMasterView.jsp file in the page designer (Figure 10-44).

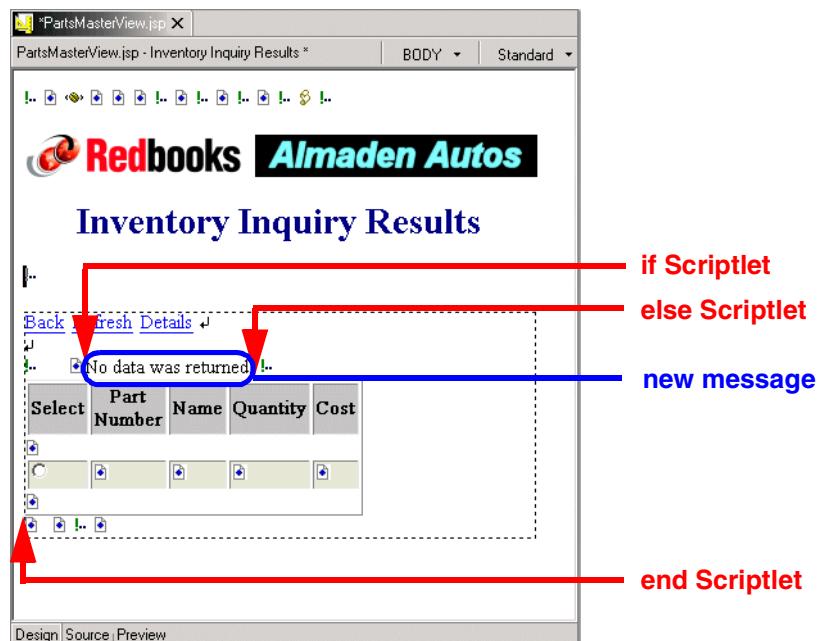


Figure 10-44 Inserting the starting and ending scriptlets

We now need to add three scriptlets to place an **if** statement around the table and our message:

- ▶ To insert the **if** scriptlet, place the cursor between the scriptlet and comment right above the table (Figure 10-44).
 - Type the text: No Data was returned.
 - Place the cursor in front of the text and select *JSP -> Insert Scriptlet* from the toolbar.
 - Complete the script dialog by placing the following code in the scriptlet (Figure 10-45):

```
if ( !masterViewDBBean.first() ) {
```

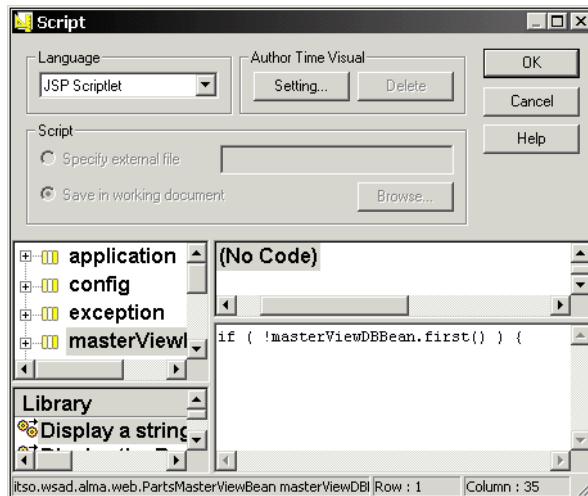


Figure 10-45 If scriptlet

- ▶ Add the **else** scriptlet the same way by placing the cursor after our message and inserting a scriptlet with the following code:

```
} else {
```

- ▶ To complete that code we need to place and **end** scriptlet immediately after the table. Place the cursor immediately preceding the existing scriptlet that follows the table and insert a scriptlet with the code:

```
}
```

Save the changes and test the Almaden Autos Web application as you did in Section , “Executing the Web application” on page 86 using part numbers M100000001 (result table) and M100000003 (no data returned message).

Linking to the MightyMotorsInquire servlet

We will now add the option to check the Mighty Motors inventory using our Web service:

- ▶ Open the PartsMasterView.jsp file in the page designer (Design view).
- ▶ Change the “No data returned” message to:

No Data Was Returned. Click [here](#) to search Mighty Motors.
- ▶ Select the word *here* and *Insert -> Link*. In the attributes dialog, switch to the *To URL* tab and enter a URL of:

MightyMotorsInquire

Note, we could have used the existing submit form Java script by entering a URL of javascript:submitForm('MightyMotorsInquire'). However, this invokes the ProcessController servlet with the MightyMotorsInquire value, and we would have needed to modify the initialization parameters of the controller servlet to include our new servlet, MightyMotorsInquire.

The PartsMasterView.jsp should now look as shown in Figure 10-46.

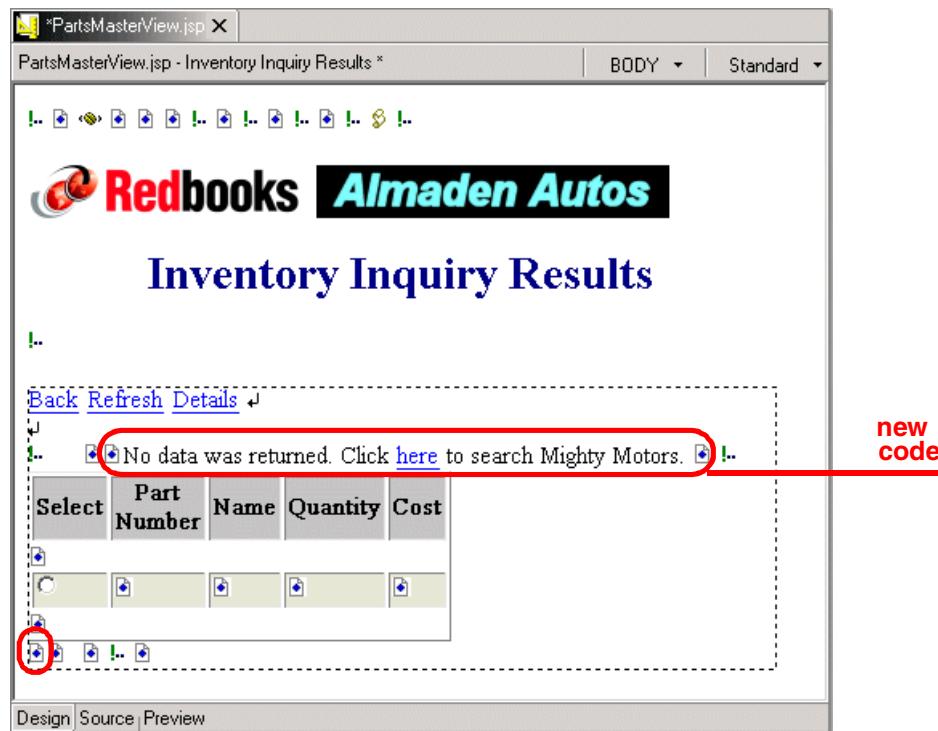


Figure 10-46 Completed PartsMasterView.jsp

Testing the Web service client

We are finally in a position to test the client of the Web service:

- ▶ Select the PartsInputForm.html file in the ITSOAlmaWeb project.
- ▶ Execute *Run on Server*, selecting the option to run it directly on the server.
- ▶ Complete the input form with a value of M100000003. The returned results page appears with the no data message (Figure 10-47).
- ▶ Select the here link to invoke the Mighty Motors Web service, which returns the XML result converted to an HTML table using the XSL style sheet.

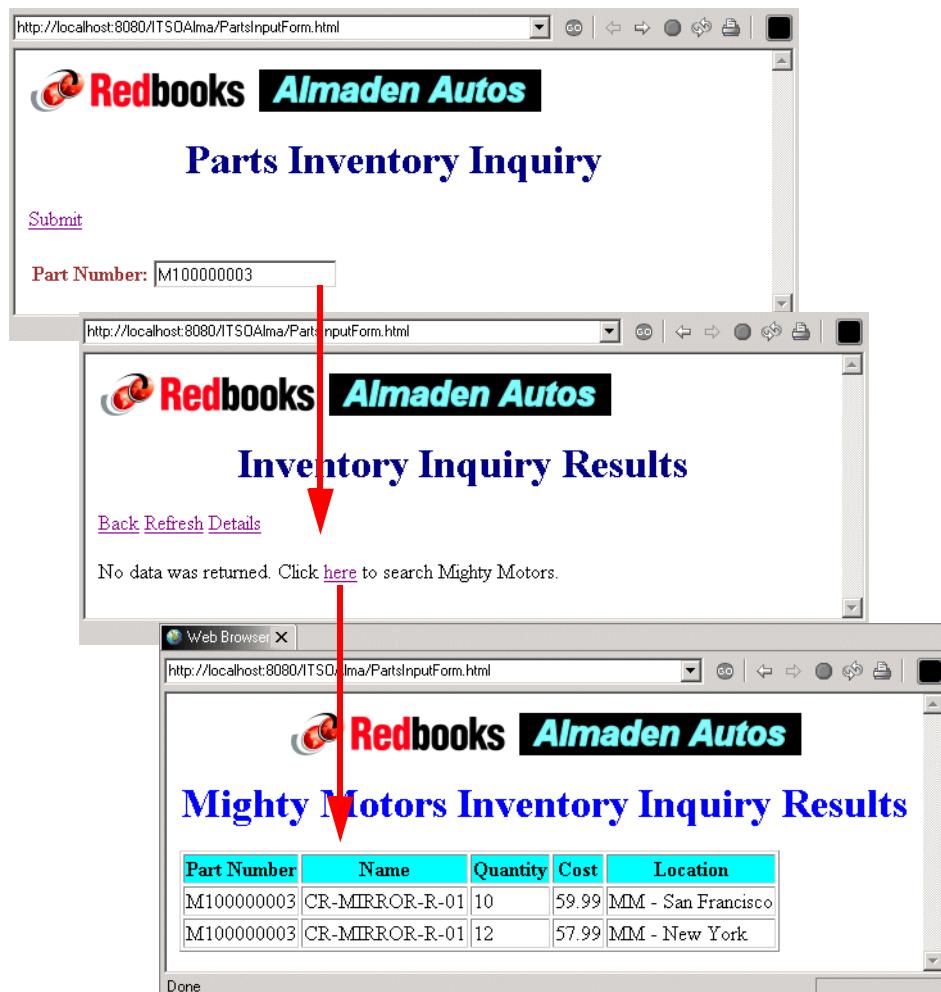


Figure 10-47 Inventory inquiry results page with Web service results

Considerations when enhancing this sample

In its current state, the Web service returns all of the inventory items that match the given part number, and the client displays a subset of this information (selected columns) through the style sheet.

It would be nice to enhance the client so that you can select an item from the returned table, and be able to see the details on that part. There are a number of alternatives to doing this:

- ▶ Create another servlet that calls the existing Web service again, returning all of the parts, and pass the selected item number into a new style sheet designed to display the details of one item. This has the disadvantage of retrieving all of the matching data from the database through the entity EJBs, and is quite a large overhead.
- ▶ Add the entire returned XML element, or a Java object representation of the entire result set, into the HTTP session and retrieve the selected item from the session for display on a new XSL style sheet. However, it is not good practice to put large amounts of data in the HTTP session.
- ▶ Create a second Web service that retrieves a single instance of a part, given a specific item number as parameter, and perhaps returning a custom Java type instead of an XML element.

The third option is the most attractive, but does require a significant amount of development at this stage. Creating Web services that use custom types is discussed in more detail in Chapter 14, “Web services advanced topics” on page 447.

Creating a Web service from a session EJB

In our example we created the Web service from a JavaBean that uses the session EJB to retrieve the part inventory items and returns the result as an XML Element.

A Web service can also be created directly from the session EJB, and return the result as a vector or array to the client. The client can then extract the results as Java classes and format the data in any desired way. This formatting can be done in a servlet or JSP, however, without having the nice XSL style sheet facility at hand.

Here are short instructions on how to create a Web service from the PartsFacade session bean. **Note that this section is optional and has no effect to future enhancements of the sample application.**

Create the session EJB Web service using the wizard

In the J2EE perspective, J2EE view:

- ▶ Select the PartsFacade session EJB and *New -> Other -> Web Services -> Web Service* and click *Next*.
- ▶ The Web service type is EJB Web Service, the Web project is ITSOMightyWeb. Select *Generate a sample* and click *Next*.
- ▶ Click *Next* in the EJB configuration.
- ▶ Set the URI to urn:PartsFacade, the ISD file name to /isd/ParstFacade.isd, and leave the default WSDL names. Click *Next*.
- ▶ Select both methods (*inquireParts* and *inquirePartsArray*) and leave *SOAP encoding* for all input and output. Do not select *Show server (Java to XML) type mappings*, we do not change anything. Click *Next*.
- ▶ Enter *itso.wsad.mighty.sessionejb.PartsFacadeProxy* as proxy class. Do not select *Show mappings*, and click *Next*.
- ▶ Do not launch the universal test client, click *Next*.
- ▶ Select *Generate a sample*, and leave the default sample/PartsFacade JSP folder. Click *Finish*.

Note: SOAP provides encoding for JavaBeans, vector and array results. See “[Implementing a Web service with a JavaBean as parameter](#)” on page 451.

Generated files

The generated files include:

- ▶ The proxy class (*itso.wsad.mighty.sessionejb.PartsFacadeProxy*)
- ▶ WSDL and schema files in the wsdl folder
- ▶ A sample test application in the sample\PartsFacade folder
- ▶ A *PartsFacade.isd* file in the WEB-INF\isd folder (it is worth opening this file and comparing it to the *InquireParts.isd* file)
- ▶ An updated *dds.xml* file that includes the content of both ISD files

Testing the session EJB Web service

The ITSOWSADWebSphere server has been started by the wizard. To test the Web service:

- ▶ Run the SOAP administration application by selecting *admin\index.html* and *Run on Server*. Check that both Web services are running.

- ▶ Run the sample test client by selecting `sample\PartsFacade\TestClient.jsp` and *Run on Server*. Invoke the two methods with a part number of M100000003 and see the results.
- ▶ You can also use the EJB test client to run the Web service. Start UTC using the URL of `http://localhost/UTC`:
 - Select the *EJB Page*.
 - Select *Load Class* (under Utilities), enter the Web service proxy class name (`itso.wsad.mighty.sessionejb.PartsFacadeProxy`), click *Load* and *Work with Object*.
 - Invoke the `PartsFacadeProxy` constructor (under Class References) and *Work with Object*.
 - Expand the `PartsFacadeProxy` (under Object References) and invoke the `Vector inquireParts` method with a part number of M100000003.
- ▶ To see the XML that is returned in the SOAP message, start the TCP/IP Monitoring server, change the proxy to port 8081, and run the sample test client again. The SOAP message looks like this:

```

<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope ...>
<SOAP-ENV:Body>
<ns1:inquirePartsResponse ...>
<return xmlns:ns2="http://xml.apache.org/xml-soap" ... >
<item ... xsi:type="ns3:itso.wsad.mighty.bean.PartInventory">
<name xsi:type="xsd:string">CR-MIRROR-R-01</name>
<cost xsi:type="xsd:decimal">59.99</cost>
<quantity xsi:type="xsd:int">10</quantity>
<weight xsi:type="xsd:double">4.6</weight>
<itemNumber xsi:type="xsd:long">21000003</itemNumber>
<shelf xsi:type="xsd:string">L8</shelf>
<imageUrl xsi:type="xsd:string">mirror03.gif</imageUrl>
<location xsi:type="xsd:string">MM - San Francisco</location>
<description xsi:type="xsd:string">Large rear view ... </description>
<partNumber xsi:type="xsd:string">M10000003</partNumber>
</item>
...
</return>
</ns1:inquirePartsResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>

```

- ▶ The generated `PartsFacade-schema.xsd` file has errors in the Tasks list. To remove the errors, delete the two `<import>` lines and the whole section `<complexity name="ArrayOfItso.wsad.manu.beans.PartInventory">`, then save the changes. The example still works fine.
- ▶ Stop the servers.

Summary

In this chapter, we used the capabilities of Application Developer to develop our first Web service for Mighty Motors and a client to this service in the Almaden Autos Web application, which reinforces the concepts explained in the other chapters.

Using the generated test clients and the TCP/IP Monitoring Server, we explored the runtime behavior of the service, along with investigating the components that were generated.

In the next chapter, we will take this example a stage further by changing the client from being a static service binding to a dynamic binding, and then we will examine working with a UDDI registry.

Quiz: To test your knowledge of building static Web services, try to answer the following five questions. The answers are provided in Appendix C, “Quiz answers” on page 559.

1. What are the three types of Web service scope, and where are they defined?
2. What is the difference between Literal XML encoding and SOAP encoding?
3. Which of the following is the service implementation file, and which is the service interface file?
 - InquireParts-service.wsdl
 - InquireParts-binding.wsdl
4. What are the ISD files used for?
5. Which of the two WSDL files are used as the basis of the client proxy generation?



Dynamic Web services

The previous chapter demonstrated how to develop, deploy and use a *static* Web service. We used a fixed 1:1 relationship between the service requestor (the dealership) and the service provider (the vehicle manufacturer). In this chapter, we take the scenario one step further and discuss the development of *dynamic* Web services with Application Developer.

As discussed in “Web services overview and architecture” on page 241, there are two fundamentally different types of dynamic Web service requesters:

- ▶ A *provider-dynamic requestor* loads the service type specification at build-time and dynamically discovers all service providers implementing this service at runtime. The requester accesses a UDDI registry for the discovery step.
- ▶ A *type-dynamic requestor* dynamically retrieves both the service type definitions, and the service implementation access information from a UDDI registry.

We will develop a provider dynamic service requestor. This chapter features the UDDI explorer as well as the APIs for programmatic access to the UDDI registry, where we will:

- ▶ Publish a business entity, business service, binding and service type
- ▶ Develop a client side proxy and use the UDDI4J APIs to dynamically locate and invoke the published implemented services

Solution outline for auto parts sample Stage 3

This chapter adds Stage 3 to the auto parts example (refer to Chapter 1, “Auto parts sample application” on page 3):

The mechanics of Almaden Autos are still not satisfied with the parts inquiry service that searches the inventory systems of both the dealership and the vehicle manufacturer Mighty Motors.

A new class of service providers, the part manufacturers, are therefore connected to the inquiry application of the dealership as well. We will only implement one such part provider, *Plenty Parts*. Both Mighty Motors and Plenty Parts publish their *InquireParts* Web service to a UDDI registry (Figure 11-1).

To simplify the example, we add another actor, the *Auto Parts Association*. This business entity takes ownership of the WSDL interface document from Mighty Motors. All services are registered under this business entity; the manufacturers themselves do not appear in the UDDI registry.

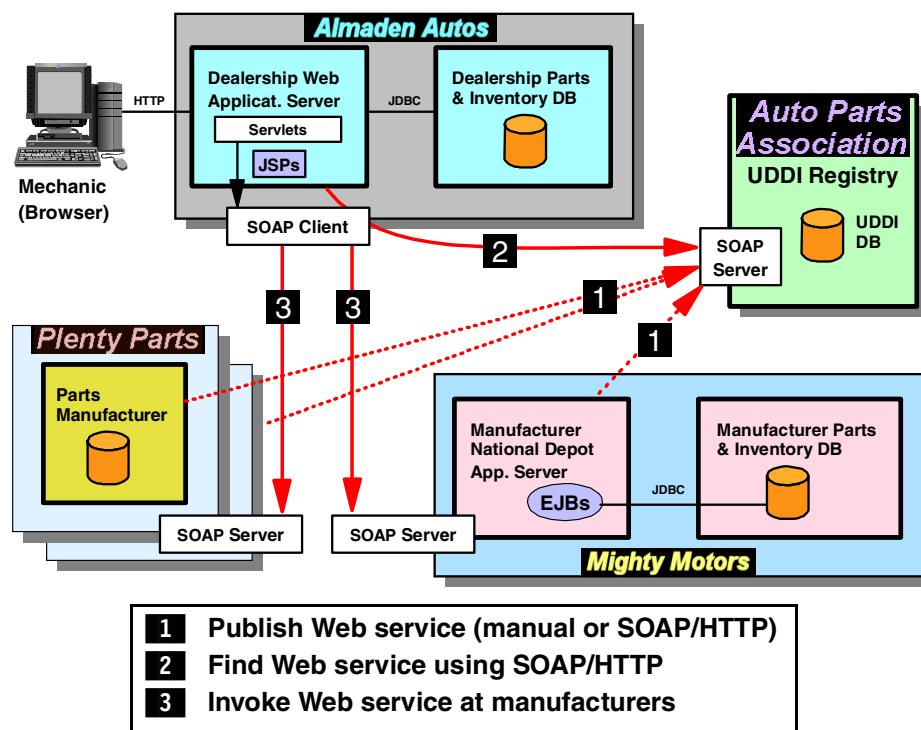


Figure 11-1 Dynamic Web service with UDDI registry lookup

Table 11-1 shows in which project we implement these roles.

Table 11-1 Roles and projects for auto parts Stage 3

Company	Role	Project
Almaden Autos	service requester (provider dynamic)	ITSOAlmaWeb (update)
Mighty Motors	service provider	ITSO Mighty Web, ITSO Mighty EJB
Plenty Parts	service provider	ITSO Plenty Web (new project)
Auto Parts Association	service interface provider	(UDDI entry only)

At runtime, the Almaden Autos application queries the UDDI registry to find implementations of the InquireParts Web service by the Auto Parts association business entity. There can be multiple implementations, for example, by Mighty Motors and Plenty Parts.

The SOAP addressing information is extracted from the UDDI registry, and each Web service is then dynamically invoked by the Almaden Autos application. The inquiry results are displayed, and the mechanic can choose from which parts manufacturer to order the part.

We mainly work in the Java and the Web perspective in this stage. As a UDDI registry, we use the public IBM Test Registry, but the code has also been tested using the IBM WebSphere UDDI Registry Preview (a private registry).

Class diagram

The component model for auto parts Stage 3 is displayed in Figure 11-2. We will explain the classes, attributes, and methods when we implement them.

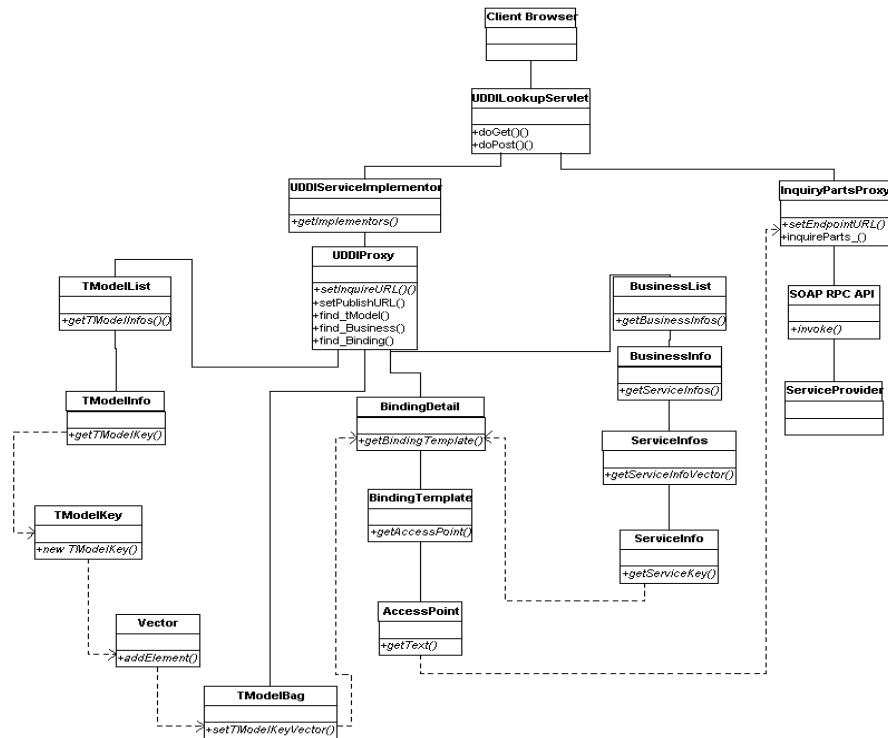


Figure 11-2 Class diagram for Web service discovery

Sequence diagram

The component interactions for the discovery part of auto parts Stage 3 are displayed in Figure 11-3. Use it as a reference in the subsequent discussions.

The interactions for the service invocation are very similar to those in auto parts Stage 2. They are therefore not displayed in the figure. See “Static Web services” on page 321 for further information.

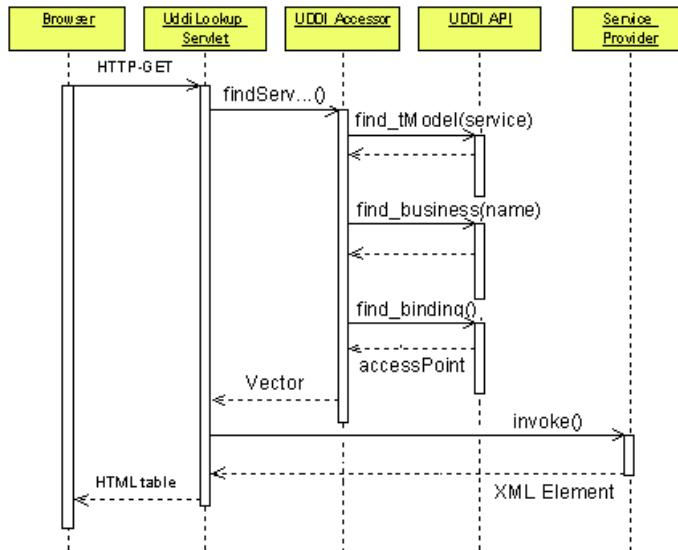


Figure 11-3 Sequence diagram for Web service discovery

Preparing for development

The following prerequisites apply to this stage of the auto parts sample:

- ▶ You require a registered and already activated UDDI account. You can register at

www-3.ibm.com/services/uddi/testregistry/protect/home.jsp

Note that you have to activate your account manually by logging in to the IBM *Business* registry. Do so, even if later on we only use the *Test* registry.

- ▶ No business entity registered for that account should exist, you have to unpublish any existing ones (otherwise, skip the step *Creating an UDDI business entity*).
- ▶ If your workstation is located behind a firewall, your client workstation has to be enabled for socks, because we require a HTTPS connection to the test registry. An example of a socks client for Windows is the Hummingbird SOCKS implementation, which can be downloaded from:

<http://www.hummingbird.com/products/nc/socks/index.html>

After having installed Hummingbird, take a look at the `sockd` entry in your `C:WINNT\system32\socks.cnf` file. It must point to the socks server in use.

- ▶ Starting with WSTK 2.4, HTTP and HTTPS proxies are supported as well.

- ▶ You have to install the service provider implementation for Plenty Parts. We do not feature it here, because it is essentially the same as the Mighty Motors implementation developed in Chapter 10, “Static Web services” on page 321.

Installing the Plenty Parts Web service

The Plenty Parts InquireParts Web service was created by using the Web service wizard to create a Web service from the WSDL service interface file. The service implementation itself uses JDBC to access the Plenty Parts tables in the database in order to satisfy the inquiry.

To install the Plenty Parts Web service:

- ▶ Import the ITSOPlenty.ear file into the ITSOPlentyEAR project.
Select *File -> Import -> EAR File*, click *Next*, click *Browse* to locate the sampcode\wsdynamic\ITSOPlenty.ear file, and enter ITSOPlentyEAR as the enterprise application project name.
- ▶ Add the variables WAS_XERCES and SOAPJAR to the ITSOPlentyWeb Java build path on the Libraries page (this fixes the errors in the Tasks list).
- ▶ Add the ITSOPlentyEAR project to the ITSOWSADWebSphere server configuration.
- ▶ Start the server and validate the Plenty Parts Web service by running the generated sample with part number M100000003:

<http://localhost:8080/ITSOPlenty/sample/InquireParts/TestClient.jsp>

Static Web services (revisited)

Before we continue, take a moment to recap the following aspects covered by the previous stage (Figure 11-4):

- ▶ Two WSDL files are generated for each Web service, one containing the protocol binding and the service type (interface file, x-binding.wsdl), and one containing the service implementation (x-service.wsdl). The implementation file imports the interface file.
- ▶ A Web service is identified by its URN and its network address. The URN appears in the protocol binding in the WSDL interface file. For SOAP over HTTP, the hostname and port form the network address; they appear in the port element of the WSDL implementation file.
- ▶ The Web service wizard uses the following naming conventions:
ClassNameRemoteInterface is the name of the service type,
ClassNameService the name of the Web service.

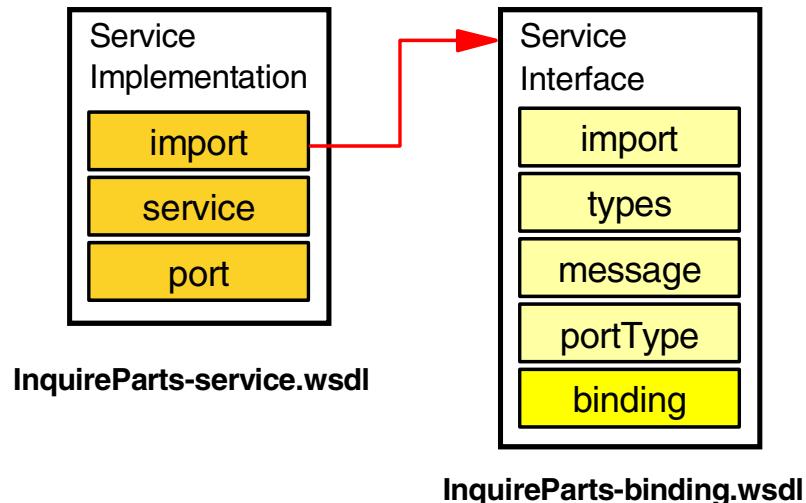


Figure 11-4 Service interface and implementation WSDL files

To distinguish between the Web services of Plenty Parts and Mighty Motors, we want to assign more meaningful service names in the implementation files (`webApplication/wsdl/InquireParts-service.wsdl`):

- For Plenty Parts we specified the service name as `InquirePlentyParts`, instead of `InquirePartsService`:

```
<service name="InquirePlentyParts">
```

- For Mighty Motors, edit the `InquireParts-service.wsdl` file and change the service name:

```
From: <service name="InquirePartsService">  
To: <service name="InquireMightyMotors">
```

Note: Changing the name of the service in the implementation file does not affect the runtime environment:

- The service name is not stored in the deployment descriptor.
- Each Web application has its own deployment descriptor and SOAP administration application.
- The service is accessed through its URI (`urn:InquireParts`).
- Our Web services have the same URI but run in different Web applications.

Working with the Application Developer UDDI browser

To provide both types of requestors with the required service information, the service provider must publish a business entity, at least one business service, and binding templates, as well a corresponding service type to the UDDI registry. These operations are supported by the *IBM UDDI Explorer*, a tool that ships with Application Developer.

In the steps that follow, we use the UDDI explorer and its built in categorization helper utility to:

- ▶ Publish a business entity (the service provider)
- ▶ Publish a service interface and implementation
- ▶ Find and import a service interface and implementation

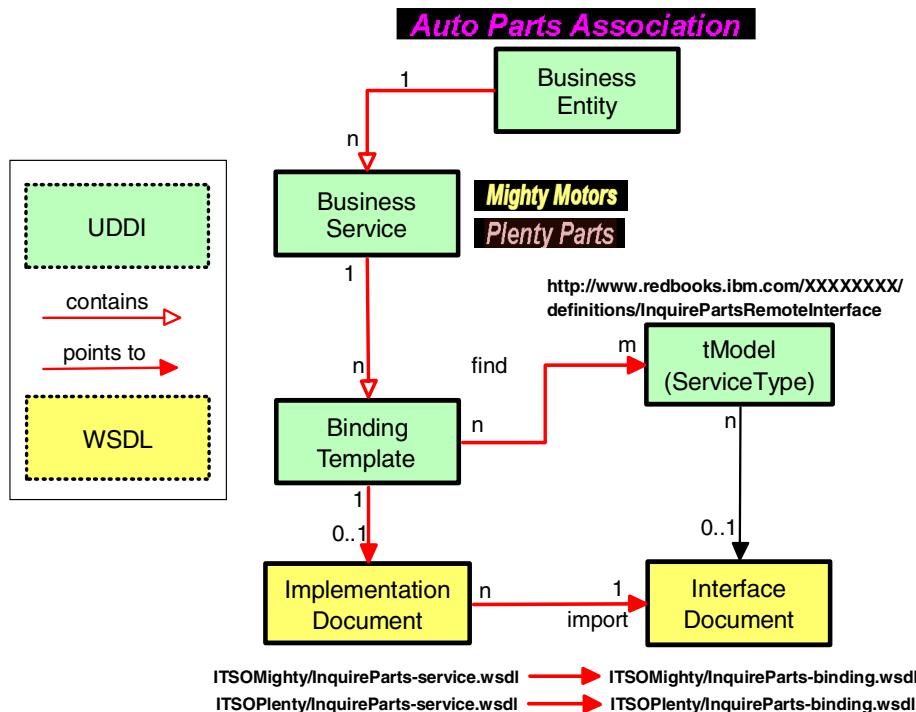


Figure 11-5 UDDI entities and their relationships

Refer to “UDDI overview” on page 293 for an introduction to these concepts. Note that you have to unpublish these entities first if you (or somebody else using the same user ID) have performed these steps before. Refer to “Unpublishing and updating entries in the UDDI registry” on page 396 or the help perspective.

Publishing a UDDI business entity

The IBM Test Registry allows you to walk through the Web service registration process without placing the service in the official UDDI Business Registry. As explained in “Preparing for development” on page 383, you must create an account before you can publish a Web service to the IBM Test Registry.

Restriction: The IBM Test Registry allows you to publish only one business entity per registered user, up to four services per business entity, two binding templates per business service, and ten tModels per registered user. Similar restrictions apply to the IBM Business Registry, which we do not use.

As a first step, we open the IBM UDDI explorer:

- ▶ In the Navigator view, select the ITS0MightyWeb project and *File -> Export* to open the export wizard.
- ▶ Select *UDDI* as export source, click *Next* to open the UDDI export wizard, and click *Finish* to open the IBM UDDI explorer.

Note: The UDDI explorer communicates to a UDDI registry through an internal server that is started the first time the UDDI explorer is opened. This server is not visible in the server perspective. The port number seems to be assigned in random order.

Figure 11-6 shows the UDDI browser GUI after selecting the IBM Test Registry.

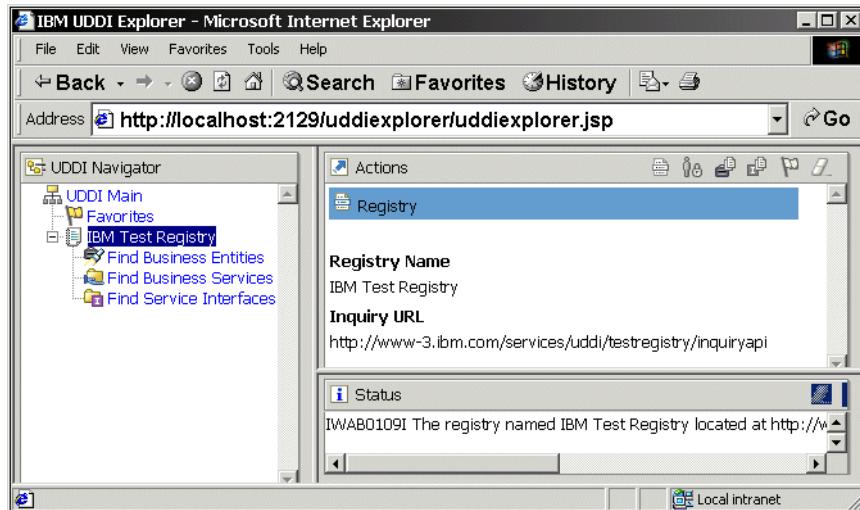
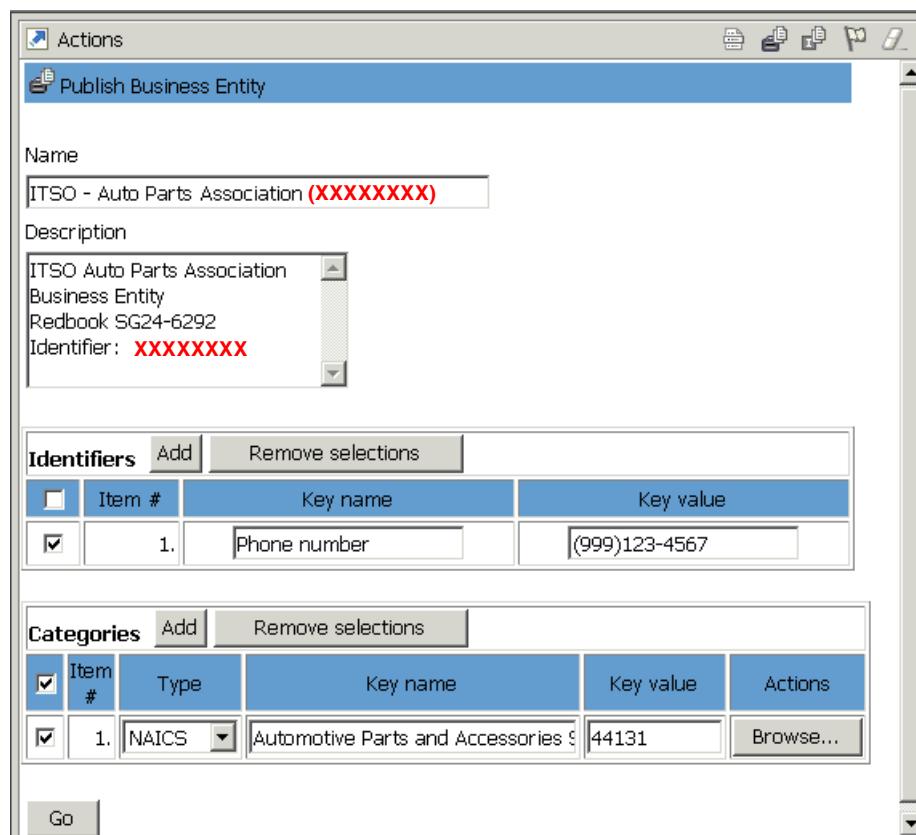


Figure 11-6 UDDI browser GUI

Now we can publish the Auto Parts Association business entity:

- ▶ In the UDDI Navigator pane, select *IBM Test Registry*.
- ▶ Click the *Publish Business Entity* icon .
- ▶ In the *Actions* pane, enter your user ID and password to logon, click *Go*.
- ▶ The browser should display *Login to registry IBM Test Registry succeeded* in the Status pane (bottom right).
- ▶ Define the attributes for the entity to be published (Figure 11-7):



The screenshot shows the 'Publish Business Entity' dialog box. At the top, there's a toolbar with icons for back, forward, search, and other actions. Below it is a blue header bar with the title 'Publish Business Entity'. The main area contains several input fields and tables.

Name: ITSO - Auto Parts Association (XXXXXXX)

Description: ITSO Auto Parts Association
Business Entity
Redbook SG24-6292
Identifier: XXXXXXXX

Identifiers:

Add	Remove selections		
<input type="checkbox"/>	Item #	Key name	Key value
<input checked="" type="checkbox"/>	1.	Phone number	(999)123-4567

Categories:

Add	Remove selections			
Item #	Type	Key name	Key value	Actions
<input checked="" type="checkbox"/>	1. NAICS	Automotive Parts and Accessories	44131	Browse...

At the bottom right is a 'Go' button.

Figure 11-7 ITSO - Auto Parts Association business entity attributes

- Enter *ITSO - Auto Parts Association (XXXXXXX)* and a meaningful description of the business in the *Name* and *Description* fields.

Attention: Replace XXXXXX with your last name or initials to distinguish from other users that go through this exercise.

- In the *Identifiers* box, click *Add* to create an identifier. In the *Key name* text field, enter *Phone Number*. In the *Key value* text field enter (999)123-4567.
- In the *Categories* box, click *Add* to create a category.
- Click *Browse* to open the NAICS category helper (Figure 11-8). Navigate through the *NAICS* category hierarchy to select *Automotive Parts and Accessories Stores* from the first *Retail Trade* entry as a key name. Double click on the entry; the key name and key value are added to the category list. Close the category helper.

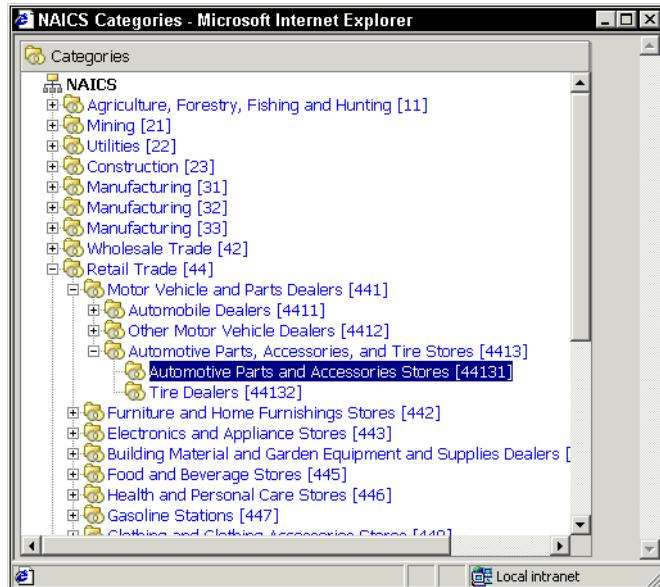


Figure 11-8 NAICS category helper

- ▶ Click *Go* in the Actions panel. The explorer displays a success message in the Status panel if the publish operation went through. An error message is shown otherwise.
- ▶ The business entity is added to the test registry (Figure 11-9).



Figure 11-9 ITSO - Auto Parts Association business entity

We have just created a business entity in the UDDI registry. The UDDI explorer GUI is automatically updated with this new business entity. Refer to “UDDI overview” on page 293 or more information about this entity type and its attributes.

Exporting the service WSDL to the UDDI registry

In this step, we use the UDDI explorer to publish a business service and binding template that reference our the service implementation WSDL file `InquireParts-Service.wsdl`. Refer to “UDDI overview” on page 293 for more information about these UDDI entities.

When publishing a service *implementation*, the UDDI browser automatically exports the referenced service *interface* file `InquireParts-binding.wsdl` as well.

Mighty Motors

We start with the publishing of Mighty Motors’ `InquireMightyMotors` Web service by taking these steps:

- ▶ Ensure that the WSDL file is accessible from your workstation, for example:
`http://localhost:8080/ITS0Mighty/wsdl/InquireParts-service.wsdl`
- ▶ Start the ITS0WSADWebSphere server if it is not running. You will receive an HTTP error 500 or a `NullPointerException` otherwise because the explorer gets the WSDL file from that URL. In order to verify that the URL can be served, open a new Internet Explorer and direct it to the URL. The XML representation of the WSDL specification should be displayed.
- ▶ In the UDDI Navigator pane, select the business entity we have created in the previous step.
- ▶ In the Actions pane, click the *Publish Business Service* icon  and the actions panel for a business service opens (Figure 11-10).
- ▶ Click *Browse* and locate the `ITS0MightyWeb` project and select the `InquireParts-service.wsdl` file. Click *GO*.

Note: If you select the WSDL file before opening the UDDI explorer, then it is already filled in this panel.

- ▶ In the Actions pane, enter a meaningful explanation of the service in the *Description* text box.
- ▶ Click *Add* to add category types, key names, and key values for your Web service. Click *Browse* and navigate through the NAICS category hierarchy to select the same *Key name* as in the previous step. The *Key value* defaults when a key name is selected.
- ▶ Click *Go* in the Actions panel.

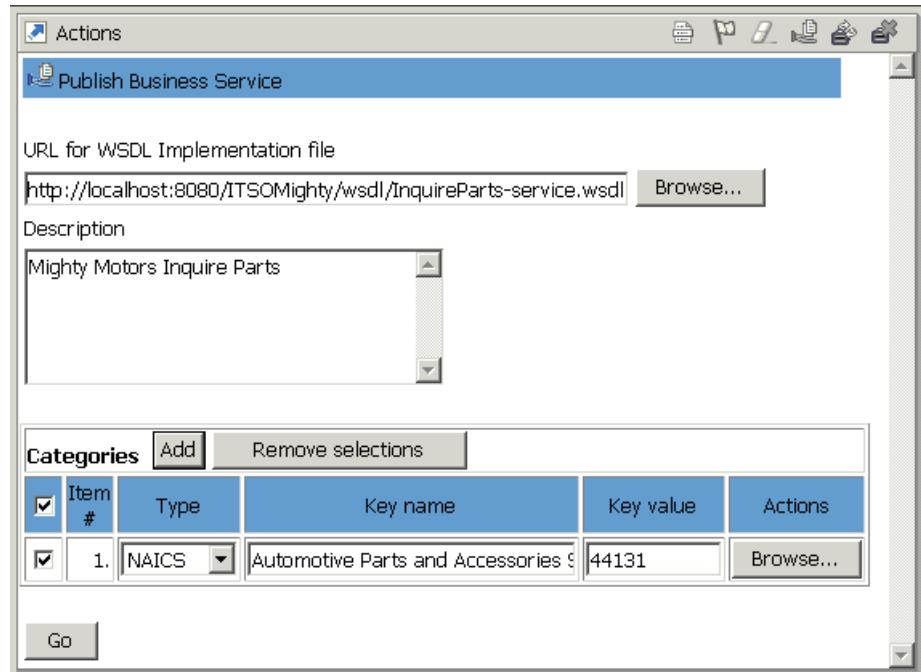


Figure 11-10 Mighty Motors business service

The UDDI explorer is automatically updated with the published Web service. The explorer automatically publishes the WSDL binding document InquireParts-Service-binding.wsdl as well.

Three entities are created in the UDDI registry: a business service, a binding template, and a tModel (service type).

If your update is successful, the status pane displays:

```
IWAB0136I Service interface http://www.redbooks.ibm.com/XXXXXXXXX  
/definitions/InquirePartsRemoteInterface was successfully updated.  
IWAB0140I Business service InquireMightyMotors was successfully published.
```

Attention: The .wsdl files must contain valid URLs that are accessible from your workstation; if both the service provider and the service requester run on the same machine, the generated code does not have to be modified. If they do not, replace all occurrences of localhost with the hostname for the development environment of the service provider. Use the correct hostname in the *URL for WSDL Implementation file* input field of the UDDI browser as well. In our example, localhost is fine because all processes run on one machine.

Plenty Parts

Publish the service implementation file in ITSOPlentyParts in the same way.

Note that in the UDDI browser, two services with the same interface are displayed (Figure 11-11). They only differ in the URLs pointing to the WSDL files on their respective Web sites.

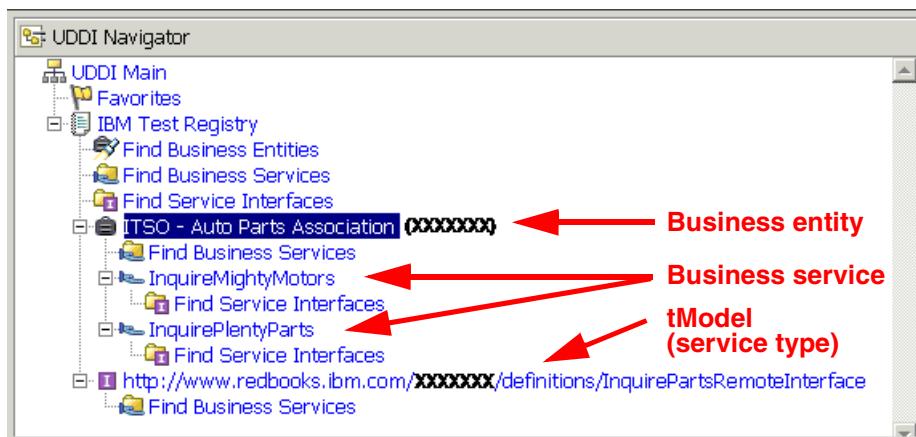


Figure 11-11 Mighty Motors and Plenty Parts successfully published

Tip: You can run multiple instances of the Web service on the same machine. In this case, you either have to register and lookup the Web service under multiple URNs (not recommended). Alternatively, you can deploy the different instances of the Web service into different Web applications. Because each of these Web applications has its own URL prefix and RPC router servlet, the SOAP server address (the URLs of the router servlet, that is), is guaranteed to be unique.

Please note that the IBM Test Registry is not a production site; on the other hand, the IBM Business Registry should not be used for development and test purposes.

Verification

With the two publish operations we have just performed, we have created a complete UDDI tree structure conforming to the data model presented in the UDDI overview in “Web services overview and architecture” on page 241.

There is a business entity, two business services and binding templates, as well as a tModel (or service type).

Using a Web browser to access the Test Registry

As an optional step, you can manually log in to the IBM UDDI Test Registry to lookup your business and service entities. Use any standard browser and go to <http://www.ibm.com/services/uddi>. Login to the test registry using the link provided on the displayed page. Use the user ID and password combination you used for the registration. The test registry displays your business entity, services, and the service type (Figure 11-12).

The screenshot shows a Microsoft Internet Explorer window displaying the IBM UDDI Business Test Registry. The address bar shows the URL <https://www-3.ibm.com/services/uddi/testregistry/protect/publish;jsessionid:XXXXXX>. The main content area is titled "UDDI Business Test Registry" and describes it as "Universal Description, Discovery, and Integration". A sidebar on the left contains links for "Select a country", "Logout", "Publish", "Find", and "Related links" (Web Services and UDDI, IBM UDDI Business Test Registry). The main content area shows three sections: "Registered Businesses: 1 found", "Services for ITSO - Auto Parts Association (ITSOSJC): 2 found", and "Registered Service Types: 1 found". Each section has a table with columns for Show Services, Business Name, Description, and Actions (Edit, Delete, Change Owner). The "Registered Businesses" table shows one entry: "ITSO - Auto Parts Association" with description "ITSO Auto Parts Association Business Entity Redbook SG24-6292 Identifier: XXXXXX". The "Services for ITSO - Auto Parts Association" table shows two entries: "InquireMightyMotors" and "InquirePlentyParts", both with description "Mighty Motors Inquire Parts" and "Plenty Parts Inquire Parts" respectively. The "Registered Service Types" table shows one entry: "http://www.redbooks.ibm.com/XXXXXX/definitions/InquirePartsRemoteInterface" with description "None".

Figure 11-12 IBM UDDI Test registry sample population

Click on each of the entries to inspect the respective attribute details. For example, if you click on *ITSO - Auto Parts Association* and then the link under *Discovery URL*, you can download an XML file containing a registry dump of the business entity and all contained entities.

If you click on the service type, you can see the URL of the WSDL interface file displayed in the *Overview URL* table. If your server in the Application Developer is running, you can click on the link to display the WSDL file.

Note: Although the WSDL file was used to publish the Web service to the UDDI registry, the WSDL file itself is not stored in the UDDI registry.

Click on *InquireMightyMotors* as well. Note that the details of the service do not display any service type information. In order to see it, you have to go into the edit mode. Go back to the first page and click on the *Edit* link in the *InquireMightyMotors* row.

From the *Edit Service* page, click on the *Edit* link of the only row in the *Access Point(s)* table. The *Edit Access Point* page lists the pointer to the tModel as the only entry under *Service Type Details*. Click on the *Edit* link of this entry. On the *Edit Service Type Information* page, the *Implementation Details Overview* contains the link to our WSDL implementation file.

Note: You are probably confused by now. So were we when we started. Unfortunately, the UDDI explorer does not expose its WSDL-to-UDDI entity mapping strategy, and the terminology used in the UDDI browser GUI, the UDDI data model, and the UDDI explorer differ from each other.

Importing the service WSDL from the UDDI registry

In this step we switch from the service (interface) provider role to the service requester role.

Note that we do not directly exchange code between the Application Developer projects that implement the provider and the requestor (ITS0MightyWeb and ITS0A1maWeb, respectively). We rather use UDDI as the only interface between the two projects.

First, we locate the service interface file (*InquireParts-binding.wsdl*) we have just published in the UDDI registry. Then, we download it from the service provider's Web site. Both of these steps are supported by one single function of the UDDI browser.

It is possible to discover a Web service by searching for a business entity, business service, or service interface. In this sample, we query the registry for the Auto Parts Association business entity.

Find business entity and services

To locate a business entity, perform these steps (Figure 11-13):

- ▶ Select the ITSOAutoWeb project in the Navigator view of the Web perspective.
- ▶ Select *File -> Import -> UDDI* to launch the UDDI explorer.
- ▶ Click the *Expand* icon  to expand the *IBM Test Registry* node.
- ▶ In the UDDI Navigator pane, select *Find Business Entities*. In the Actions pane, enter *Query Inquire Auto Parts Association* as the query name. Enter *ITSO - Auto Parts Association (XXXXXXX)* in the *Business Name* text field.
- ▶ Click *Go*.

Note that it would be sufficient to type the first few characters *ITSO*; there is an implied wildcard mechanism. However, this retrieves entries from other people who have gone through this exercise.

- ▶ Expand the *Query Inquire Auto Parts Association* query and the *ITSO - Auto Parts Association*.
- ▶ Click *Find Business Services*. Name the query *Query Find Business Services* and click *Go*. This locates the two services we published.
- ▶ In the UDDI Navigator pane, expand the *Query Find Business Services* query result.
- ▶ Select the *Inquire Plenty Parts* service and expand it.
- ▶ Click *Find Service Interfaces*.
- ▶ Name the query *Query Plenty Parts Service Interfaces* and click *Go*.
- ▶ In the query result, select on the only entry:

<http://www.redbooks.ibm.com/...InquirePartsRemoteInterface>



Figure 11-13 Plenty Parts service implementation in the UDDI registry

Import the WSDL file

In the Actions pane, click the *Import to workbench* icon  to import the WSDL document into the ITSOA1maWeb Web project in Application Developer. Click *Go*.

If the import succeeds, a status message is displayed:

```
IWAB0131I InquireParts-binding.wsdl was successfully imported into web  
project ITSOA1maWeb
```

The `InquireParts-binding.wsdl` file is imported into the `webApplication` folder of the Web project.

Remember how we performed the equivalent step in “Static Web services” on page 321, where we directly copied the WSDL files from one project to another. Here, the transfer involves the UDDI registry.

Also, note that we only import the `InquireParts-binding.wsdl` service type file, because we implement a provider dynamic requester. The UDDI browser would have allowed us to import one of the two service implementation files as well.

As an optional step, you can go to the Navigator view and double-click on the imported WSDL file to launch the XML editor. Take a look at its message, port type, and binding elements.

Restriction: You might want to validate the WSDL file. However, Application Developer can only check for well-formedness because the WSDL XSD is not accessible.

Unpublishing and updating entries in the UDDI registry

With the UDDI browser, you can easily unpublish business entities, service types, and (business) services from a UDDI registry. For example, you may want to remove the entities we have just created after having completed the example.

Simply locate the entity you want to delete and select the *Unpublish* icon from the upper right corner of the window. Make sure that you are logged in with an appropriate user ID and password combination.

Do not delete the entries now; otherwise, our provider dynamic service requester will not find any business services in the UDDI registry. You might want to perform this step after having completed this redbook to cleanup the registry.

To update business entities, service types, and (business) services, locate the entry and click the *Update* icon to change the data.

Generating a SOAP client proxy from a WSDL file

Next, we use the Web service client wizard in order to generate a provider dynamic client side proxy for the Web service interface we have just imported from the UDDI registry.

- ▶ In the ITSOAlmaWeb project's webApplication/wsdl, create a folder called dynamic.
- ▶ Move the InquireParts-binding.wsdl to the wsdl/dynamic folder.
- ▶ Make sure the ITSOWSADWebSphere server instance is started.

Now start the Web service client wizard:

- ▶ Select the dynamic/InquireParts-binding.wsdl and *File -> New -> Other -> Web Services -> Web Service client*. Click *Next*.
- ▶ Do not select *Generate a sample*. Click *Next*.

We do not generate a test client because the generated test clients only work with proxies created from the WSDL service implementation file, which includes the URL of the RPC router servlet. We demonstrated the generation of such a static test client in “Generating a sample client” on page 344.

- ▶ Verify the WSDL file name is:
`/ITSOAlmaWeb/webApplication/wsdl/dynamic/InquireParts-binding.wsdl`
- ▶ Make sure that your WebSphere server instance is running so that the wizard can access the .wsdl file. Click *Next*.
- ▶ On the *Web Service Binding Proxy Generation* window change the qualified path name to:
`itso.wsad.alma.wsdynamic.proxy.InquiryPartsProxy`
- ▶ The rest of the defaults are correct. Click *Finish*.

Inspect the generated proxy class `InquirePartsProxy`, now appearing in the source/itso/wsad/alma/wsdynamic/proxy folder of your project. Note that its default URL (`stringURL`) data member is set to null.

To generate this proxy, we used the `InquireParts-Binding.wsdl` file describing the service interface, as opposed to Stage 2, where we started from the service implementation file. The generated Java code is very similar, however. The only difference is that the proxy no longer hardcodes the SOAP address of the service provider.

Note that we could have generated the client proxy on a different machine than the server implementation. The only link between the two systems is the WSDL file.

Working with the UDDI APIs

In this section we will work with the UDDI4J APIs to access the UDDI registry programmatically. We will then make use of this code in the UDDILookupServlet built in “UDDI lookup servlet” on page 405.

In review of the UDDI registry, Figure 11-14 shows an overview of the relationship between the UDDI entities.

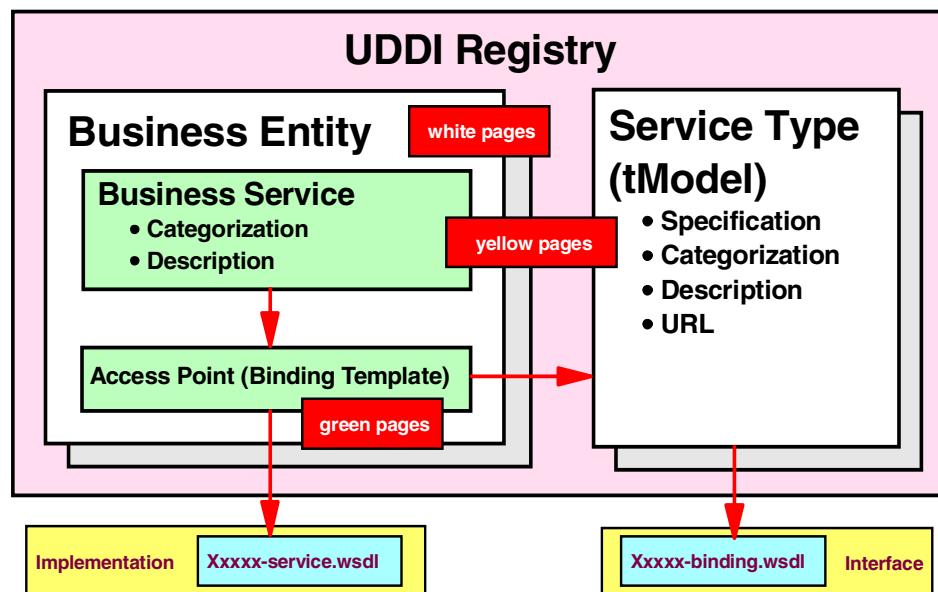


Figure 11-14 UDDI registry overview

In “Publishing a UDDI business entity” on page 387, we published the Mighty Motors and Plenty Parts Web services to the UDDI registry by way of the Application Developer UDDI explorer. We did this by exporting the Mighty Motors and Plenty Parts implementations, the InquireParts-service.wsdl files, to the UDDI registry. The resulting publication can be summarized as follows.

UDDI registry	The IBM Test Registry. We will use the IBM Test Registry in our implementation, however, using the UDDI Private Registry is only a matter of changing the URL of the registry in our implementation. The UDDI Private Registry is explored in “Advanced UDDI topics” on page 489.
Business entity	When we installed the business entity we called it ITSO - Auto Parts Association (XXXXXXXX) . This is the business entity we will use to look up our available business services.

Service type	We defined our service type to be: <pre>http://www.redbooks.ibm.com/XXXXXXXX/definitions/ InquirePartsRemoteInterface</pre>
Business service	This name is from the targetNamespace attribute in the interface wsdl (InquireParts-binding.wsdl): <pre><definitions name=... targetNamespace=xxxxxxxxxx .../></pre>
Access points	The business services we have registered are InquireMightyMotors and InquirePlentyParts . These came from the service name attribute in the implementation wsdl (InquireParts-service.wsdl): <pre><service name="InquirePlentyParts"></pre>
Implementation	The associated access points are the URLs of the defined services: <pre>http://localhost:8080/ITSO_Mighty/servlet/rpcrouter http://localhost:8080/ITSO_Plenty/servlet/rpcrouter</pre>
	These came from the soap:address element in the implementation wsdl (InquireParts-service.wsdl): <pre><soap:address location="http://localhost:8080/....></pre>

With this in mind, we will now proceed to programmatically access this information and invoke the discovered Web services.

Finding the service providers in the UDDI registry

We will first look at locating the service providers in the UDDI registry. This will be done using a UDDIAccessor class implemented in this section.

UDDIAccessor class

To create the UDDIAccessor class:

- ▶ First add two variables, **DDI4JJAR** and **MAILJAR**, to the ITSOAlmaWeb project Java build path. Define them as:

DDI4JJAR <WSAD_INSTALL_DIR>plugins/com.ibm.etools.websphere.runtime/lib/uddi4j.jar

MAILJAR <WSAD_INSTALL_DIR>plugins/com.ibm.etools.servletengine/lib/mail.jar

- ▶ Create a new `itso.wsad.alma.wsdynamic.uddi` package in the source folder.
- ▶ Import the `UDDIAccessor` class into the newly created package (from `sampcode\wsdynamic`).

UDDIAccessor processing

Let us take a closer look at the `UDDIAccessor` class. The UDDI lookup is done in the `findServiceImplementors` method. The goal of this method is to collect all the access points defined for a given provider and service type and return the list as a Java vector.

The `findServiceImplementors` method takes two parameters, the service provider and the service type, in our case:

ITSO - Auto Parts Association (XXXXXXXX)
<http://www.redbooks.ibm.com/XXXXXXX/definitions/InquirePartsRemoteInterface>

To understand how the access points are located for the specified service and provider, let us examine the relationship between the UDDI entities available through the UDDI4J APIs.

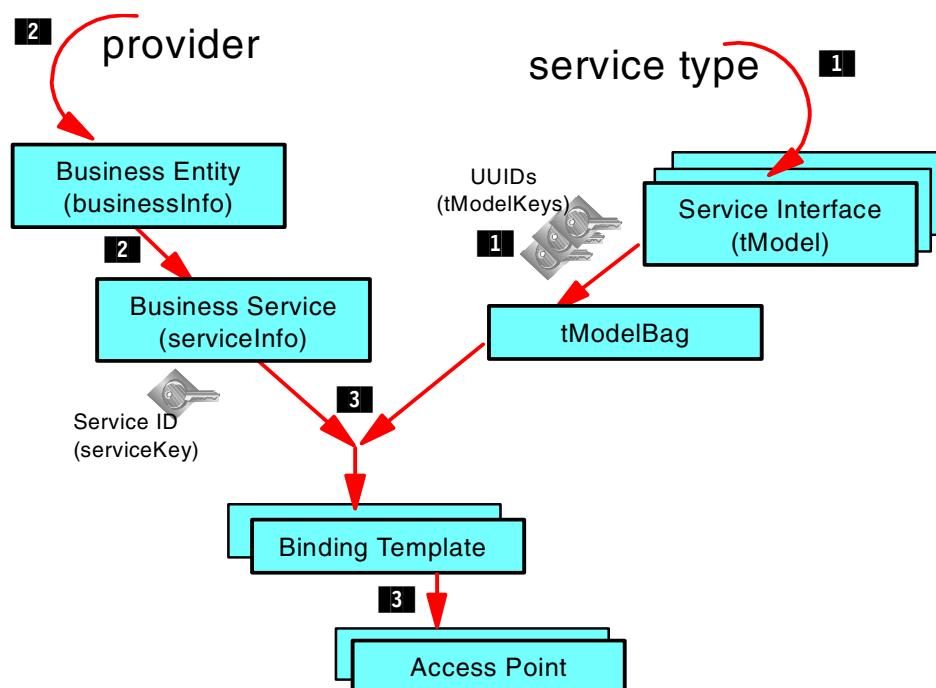


Figure 11-15 Locating the access points from the service type and business entity

Let us explore the code of the `UDDIAccessor` class in small pieces.

Accessing the registry

First we set which UDDI registry to access:

```
// IBM Test Registry
String inquiryAPI =
    "http://www-3.ibm.com/services/uddi/testregistry/inquiryapi";
String publishAPI =
    "http://www-3.ibm.com/services/uddi/testregistry/protect/publishapi";
// IBM Registry Preview
//String inquiryAPI = "http://localhost/services/uddi/inquiryAPI";
//String publishAPI = "http://localhost/services/uddi/publishAPI";
UDDIProxy proxy = new UDDIProxy();
Vector result = new Vector();
proxy.setInquiryURL(inquiryAPI);
proxy.setPublishURL(publishAPI);
```

Collect tModels

Using the service type parameter, we find all the tModels and collect their keys in a tModelBag:

```
// find tmodel for the service name and fill tModelBag with keys
TModelBag tmb = new TModelBag();
Vector tmbv = new Vector();
tmb.setTModelKeyVector(tmbv);
TModelList tml = proxy.find_tModel(servicename, null, 0);
TModelInfos tmis = tml.getTModelInfos();
Vector v1 = tmis.getTModelInfoVector();
for (int i1=0; i1 < v1.size(); i1++) {
    TModelInfo tmi = (TModelInfo)v1.elementAt(i1);
    String tmk = tmi.getTModelKey();
    System.out.println("TModel="+tmi.getNameString()+" "+tmk);
    TModelKey tmKey = new TModelKey(tmk);
    tmbv.addElement(tmKey);
    if (i1==14) { System.out.println("Too many tModels-stop");
                  break; }
}
```

Find the business entity

Using the provider parameter, we find the business entity. Note that the method returns a list, even if there is only one (most API methods accept wildcards):

```
// find business by provider
BusinessList bl = proxy.find_business(provider, null, 0);
BusinessInfos bis = bl.getBusinessInfos();
Vector v2 = bis.getBusinessInfoVector();
```

Find the business services of the business entity

For each business entity we find the business services:

```
for (int i2 = 0; i2 < v2.size(); i2++) {
```

```

BusinessInfo bi = (BusinessInfo)v2.elementAt(i2);
String bkey = bi.getBusinessKey();
System.out.println(" Business="+bi.getNameString()+" "+bkey);

// find service of business
ServiceInfos sis = bi.getServiceInfos();
Vector v3 = sis.getServiceInfoVector();

```

Find bindings of service that implements the tModel

For each business service find the bindings for the service that implements the service type (tModel), using the business service key and the tModelBag:

```

for (int i3 = 0; i3 < v3.size(); i3++) {
    ServiceInfo si = (ServiceInfo)v3.elementAt(i3);
    String skey = si.getServiceKey();
    System.out.println(" Service="+si.getNameString()+" "+skey);

    // find binding of service that implements tmodel
    BindingDetail bd = proxy.find_binding(null, skey, tmb, 0);
    Vector v4 = bd.getBindingTemplateVector();
    if (v4.size()==0)
        System.out.println(" no binding templates");
    for (int i4 = 0; i4 < v4.size(); i4++) {
        BindingTemplate bt = (BindingTemplate)v4.elementAt(i4);

```

Collect access points from the bindings

Extract the access point from each binding and store in the result vector:

```

// get access point
AccessPoint accessPoint = bt.getAccessPoint();
System.out.println("AccessPoint="+accessPoint.getText());

// add access point to result
result.addElement(accessPoint.getText());

```

The access points can then be used to invoke the service dynamically, which we describe when we implement the servlet that uses the UDDIAccessor class (see “UDDI lookup servlet” on page 405).

Retrieve WSDL name

For the purpose of example, the UDDIAccessor class goes on to programmatically find the implementation of the service (Xxxx-service.wsdl). The implementation could then be analyzed using the WSDL APIs if desired. We will only determine the file in our example—it will not be used in the application:

```

TModelInstanceDetails tid = bt.getTModelInstanceDetails();
Vector v5 = tid.getTModelInstanceInfoVector();
for (int i5 = 0; i5 < v5.size(); i5++) {
    TModelInstanceInfo tii =

```

```

        (TModelInstanceInfo)v5.elementAt(i5);
        InstanceDetails inst = tii.getInstanceDetails();
        OverviewDoc od = inst.getOverviewDoc();
        OverviewURL ou = od.getOverviewURL();
        System.out.println("      WSDL="+ou.getText());
    }
}

```

UDDIAccessor testing

In the spirit of iterative development, we will now test the UDDIAccessor class before integrating it into our application.

We implement a UDDIAccessorTest program to exercise the UDDIAccessor class we just imported. Define the class in the `itso.wsad.alma.wsdynamic.uddi` package and enter the code shown in Figure 11-16 (or import the class from `sampcode\wsdynamic`).

```

// change XXXXXXXX according to your business entity and WSDL file

package itso.wsad.alma.wsdynamic.uddi;

import java.util.Vector;

public class UddiAccessorTest {

    public static void main(String[] args) {
        String provider = "ITSO - Auto Parts Association (XXXXXXX)";
        String service  = "http://www.redbooks.ibm.com/XXXXXXX/definitions/
                           InquirePartsRemoteInterface";
        UddiAccessor uddi = new UddiAccessor();
        Vector result = uddi.findServiceImplementations(provider, service);
        for (int i=0; i < result.size(); i++) {
            System.out.println("Result="+String(result.elementAt(i)));
        }
    }
}

```

Figure 11-16 UDDI Accessor test program

Running this class with our service type and provider parameters produces the output shown in Figure 11-17. (The in the listing indicate the UUIDs in the registry.)

To run the program, change the `XXXXXXX` to your specifications, and start the program in the Java perspective.

```

TModel=http://www.redbooks.ibm.com/XXXXXXXX/definitions/InquirePartsRemoteInterface .
Business=ITSO - Auto Parts Association (XXXXXXXX) ...
Service=InquireMightyMotors ...
AccessPoint=http://localhost:8080/ITSOMighty/servlet/rpcrouter
WSDL=http://localhost:8080/ITSOMighty/wsdl/InquireParts-service.wsdl
Service=InquirePlentyParts ...
AccessPoint=http://localhost:8080/ITSOPlenty/servlet/rpcrouter
WSDL=http://localhost:8080/ITSOPlenty/wsdl/InquireParts-service.wsdl
END
Result=http://localhost:8080/ITSOMighty/servlet/rpcrouter
Result=http://localhost:8080/ITSOPlenty/servlet/rpcrouter

```

Figure 11-17 UDDI Accessor test output

Analyzing the output we detect the following:

- ▶ One tModel was found matching the specified service type. The service type (tModel name) and the UUID (tModelKey) is displayed.

Depending on what else has been registered by other users, you may see several tModels. This is why we suggested the XXXXXXXX is your own name or initials.

- ▶ Next we only find one business entity that matches our provider description. The business entity name, ITSO - Auto Parts Association (XXXXXXXX), and its key are displayed.

Note. If you change the code of the provider parameter to ITSO - Auto Parts Association, (without the XXXXXXXX), then you will find multiple business entities.

- ▶ Next, we see that our business entity implements two services, InquireMightyMotors and InquirePlentyParts. (Note that all services are found, even those that do not implement our tModel.)
- ▶ Each service is followed by the one binding/access point that implements the tModel, and is associated with a WSDL implementation file.
- ▶ The program finally lists the collected access points.

Congratulations, you have successfully used the UUDI4J APIs to programmatically access the UDDI registry. We will now proceed by integrating this into our existing application.

Updating the Almaden Autos Web application

We are now ready to integrate the UDDI find operation into the already existing Web application. We will implement a `UDDILookupServlet` that dynamically finds the Web services and invokes them. We will then add a link to the current Almaden Autos Web application to provide the mechanics another option for finding parts.

UDDI lookup servlet

Select the `itso.wsad.alma.web` package and *New -> Servlet* to create a new servlet called `UDDILookupServlet`. On the second panel of the servlet creation wizard, check *Add to web.xml* and leave the mapping as `UDDILookupServlet`. The code is available in `sampcode\wsdynamic`.

- ▶ Add the following import statements:

```
import javax.xml.transform.*;
import javax.xml.transform.stream.StreamSource;
import javax.xml.transform.stream.StreamResult;
import javax.xml.transform.dom.DOMSource;
import org.w3c.dom.Element;
import itso.wsad.alma.wsdynamic.proxy.InquirePartsProxy;
import itso.wsad.alma.wsdynamic.uddi.UddiAccessor;
import java.util.Vector;
```

- ▶ Put the code displayed in Figure 11-18 into the `doGet` method of the new servlet (yes, according to best practices for Web programming, the HTML output should rather be part of a JSP).
- ▶ Call `doGet(request, response)` from `doPost` or create an additional `service(...)` method called both from the `doGet` and the `doPost` method.

Processing

The `UDDILookupServlet` first places some initial HTML into the output stream.

The key portion of the servlet comes next and is highlighted in bold. The servlet uses the `UDDIAccessor` we implemented in “`UDDIAccessor class`” on page 399 to get a list of all the access points (`endpointURLs`) defined in the registry for our provider and service type.

The servlet then uses the `InquirePartsProxy` generated in “`Generating a SOAP client proxy from a WSDL file`” on page 397 to exercise each of the web services identified by the access points.

For each Web service exercised successfully, the returned data is transformed into HTML using an XSL style sheet and added to the output stream.

```

public class UDDILookupServlet extends HttpServlet {

    String provider = "ITSO - Auto Parts Association (XXXXXXX)";
    String service =
        "http://www.redbooks.ibm.com/XXXXXXX/definitions/InquirePartsRemoteInterface";

    public void doGet(
        javax.servlet.http.HttpServletRequest request,
        javax.servlet.http.HttpServletResponse response) {
        try {
            response.setContentType("text/html; charset=UTF-8");
            java.io.PrintWriter out = response.getWriter();
            out.println("<HTML><HEAD><TITLE>Inventory Inquiry Results </TITLE></HEAD>
                        <BODY BGCOLOR=\"#fffff\>");
            out.println("<CENTER>");
            out.println("<FONT COLOR=\"#0000ff\"><IMG border=\"0\" height=\"33\"
                        src=\"images/autopartsassociation.gif\" width=400\>
                        <CENTER><H1>Parts Inquiry</H1></CENTER></FONT>\"");
            out.println("</CENTER>");
            TransformerFactory tFactory = TransformerFactory.newInstance();
            Source xslSource = new StreamSource( new java.net.URL(
                "http://localhost:8080/ITSOAlma/stylesheets/dynamic/InquirePartsResultTable.xsl")
                .openStream() );
            Transformer transformer = tFactory.newTransformer(xslSource);
            InquirePartsProxy proxy = new InquirePartsProxy();
            String partNumber = (String)request.getSession().getAttribute("partNo");

            UddiAccessor uddi = new UddiAccessor();
            Vector endpoints = uddi.findServiceImplementations(provider, service);
            for (int i=0; i < endpoints.size(); i++) {
                String endpoint = (String)endpoints.elementAt(i);
                proxy.setEndPoint( new java.net.URL(endpoint) );
                out.println("<h2>" + endpoint + "</h2>");
                try {
                    Element result = proxy.inquireParts(partNumber);
                    Source xmlSource = new DOMSource(result);
                    transformer.transform(xmlSource, new StreamResult(out));
                } catch (Exception e) {
                    out.println("<FONT COLOR=\"#0000ff\"><CENTER><H3>
                                Web Service at this URL not available.</H3></CENTER></FONT>\"");
                }
            }
            out.println("</BODY></HTML>");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

Figure 11-18 UDDI lookup servlet processing

Creating the XSL style sheet

We now create a simple style sheet to display each returned result set in a table similarly to the table created for our static Web service in “Creating the XSL style sheet” on page 366.

- ▶ Create a new folder in the ITSOAlmaWeb project:

webApplication/stylesheets/**dynamic**

- ▶ Create the InquirePartsResultTable.xsl file in the new folder with the code of Figure 11-19, or import the file from sampcode\wsdynamic.

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:res="http://www.redbooks.ibm.com/ITSOADSAD/schemas/InquireResults">
<xsl:output method="html"/>
<xsl:template match="res:InquirePartsResult">
  <table border = "1">
    <tr><bold>
      <TH BGCOLOR="#00ffff">Part Number</TH>
      <TH BGCOLOR="#00ffff">Name</TH>
      <TH BGCOLOR="#00ffff">Quantity</TH>
      <TH BGCOLOR="#00ffff">Cost</TH>
      <TH BGCOLOR="#00ffff">Location</TH>
    </bold></tr>
    <xsl:apply-templates/>
  </table>
</xsl:template>
<xsl:template match="res:Part" >
  <tr>
    <td><xsl:value-of select="res:PartNumber"/></td>
    <td><xsl:value-of select="res:Name"/></td>
    <td><xsl:value-of select="res:Quantity"/></td>
    <td><xsl:value-of select="res:Cost"/></td>
    <td><xsl:value-of select="res:Location"/></td>
  </tr>
</xsl:template>
</xsl:stylesheet>
```

Figure 11-19 Style sheet implementation for the dynamic Web service

This style sheet creates a results table containing part number, name, quantity, and cost columns, and then creates a row in the table for each part in the XML document. The rest of the HTML is generated in the servlet.

Updating the existing Web application

We now have to make the final changes to make our new dynamic capability available to the mechanics. We will update the existing ITSOAlmaWeb application.

- ▶ Import the autopartsassociation.gif file to the ITSOAlmaWeb project webApplication/images folder.
- ▶ Add the following HTML snippet in the PartsMasterView.jsp after the “No data was returned” line, as we did in “Linking to the MightyMotorsInquire servlet” on page 372:

```
<br> Alternatively, click <A href="UDDILookupServlet">here</A> to  
discover other part providers in the ITSO - Auto Parts Association.<br>
```

The complete code in front of the table is:

```
<% if ( !masterViewDBBean.first() ) { %>  
No data was returned.  
Click <A href="MightyMotorsInquire">here</A> to search Mighty Motors.<BR>  
Alternatively click <A href="UDDILookupServlet">here</A> to discover other  
part providers in the ITSO - Auto Parts Association. <BR>  
<% } else { %>
```

This is a link to the UDDILookupServlet we just developed. Make sure to use the name specified when registering the servlet into the web.xml file with the servlet creation wizard in “UDDI lookup servlet” on page 405.

This completes the implementation of the dynamic Web service example. We are now ready to test the implementation.

Unit testing

As in the previous steps, the unit testing is performed in the WebSphere Test environment included in Application Developer. Use the TCP/IP Monitoring Server introduced in “Static Web services” on page 321, to intercept and trace the communication between the service requester and the two service providers.

We assume the Plenty Parts service provider implementation is available. In “Preparing for development” on page 383, you can find the instructions on how to install it if you have not already done so.

Now you can start the server instance and load the Almaden Autos homepage (PartsInputForm.html) into the browser in the same way as shown in “Static Web services” on page 321 (Figure 11-20).

http://localhost:8080/ITSOAlma/PartsInputForm.html

Redbooks Almaden Autos

Parts Inventory Inquiry

[Submit](#)

Part Number:

Done

Figure 11-20 Almaden Autos entry page

Select a part that is not locally available, M100000003. and click *Submit*. No parts are found, and the updated HTML with the new link is displayed (Figure 11-21).

Web Browser X

http://localhost:8080/ITSOAlma/PartsInputForm.html

Redbooks Almaden Autos

Inventory Inquiry Results

[Back](#) [Refresh](#) [Details](#)

No data was returned. Click [here](#) to search Mighty Motors.
Alternatively click [here](#) to discover other part providers in the ITSO - Auto Parts Association.

Figure 11-21 New link for dynamic UDDI lookup

Click on the new link that allows you to discover part manufacturers in the UDDI registry. The search results are shown in Figure 11-22.

The screenshot shows a web browser window with the URL <http://localhost:8080/ITSOAlma/PartsInputForm.html>. The page title is "Redbooks | Auto Parts Association" and the main heading is "Parts Inventory Inquiry". Below the heading is the URL <http://localhost:8080/ITSOMighty/servlet/rpcrouter>. A table displays part information:

Part Number	Name	Quantity	Cost	Location
M100000003	CR-MIRROR-R-01	10	59.99	MM - San Francisco
M100000003	CR-MIRROR-R-01	12	57.99	MM - New York

Below this is another table for the URL <http://localhost:8080/ITSOPlenty/servlet/rpcrouter>:

Part Number	Name	Quantity	Cost	Location
M100000003	CR-MIRROR-R-01	42	49.99	PP - Heidelberg

A "Done" button is visible at the bottom left of the browser window.

Figure 11-22 Dynamic UDDI lookup results

Note that the ITSOMightyWeb and ITSOPlentyWeb applications have to be up and running so that the ITSOAlmaWeb application can invoke the Web services.

Deploying the enhanced application to WebSphere 4.0

This step is very similar to the deployment of Stage 2 of the example. Refer to Chapter 10, “Static Web services” on page 321 for further information.

Make sure that your class path settings include the JAR files mentioned in “Unit testing” on page 408.

Summary

In this chapter we used the UDDI registry to store two implementations of our Web service. We then implemented a client Web application that interrogates the UDDI registry to find implementers of the requested Web service. All (trusted) implementations are then invoked to assemble the dynamic output.

Quiz: To test your knowledge about dynamic Web services, try to answer the following five questions. The answers are provided in Appendix C, “Quiz answers” on page 559.

1. Name the two different types of dynamic Web service clients, and explain the difference between them. Which type is likely to be implemented more often?
2. List at least three different ways in which you can inspect the content of a UDDI registry.
3. Which two HTTP operations must a service provider support, after publishing a service to the UDDI registry?
4. Name at least three artifacts in the UDDI object model.
5. Which WSDL file is the implementation file and which file, is the interface file?



Composed Web services

In this chapter we build a Web service that calls another Web service, known as a composed Web service.

To implement the composed Web service we:

- ▶ Create and implement a Web service using a JavaBean skeleton generated from an existing WSDL document.
- ▶ We then integrate that Web service into an existing Web service, demonstrating the ability of a Web service to invoke another Web service.

To be able to do this chapter you must have completed Chapter 10, “Static Web services” on page 321. It is not necessary to have finished Chapter 11, “Dynamic Web services” on page 379.

Solution outline for auto parts sample Stage 4

Almaden Autos decides to expose its own parts inventory system as a Web service so that other car dealers can search for parts in its inventory. Therefore, it uses the existing WSDL of Mighty Motors and implements the Web service for its own inventory system. Almaden Autos hopes that other car dealers will follow this initiative, so that eventually it is possible not only to search for parts in the depots of car manufactures and third party part manufacturers, but also, in the inventories of other dealers (Figure 12-1).

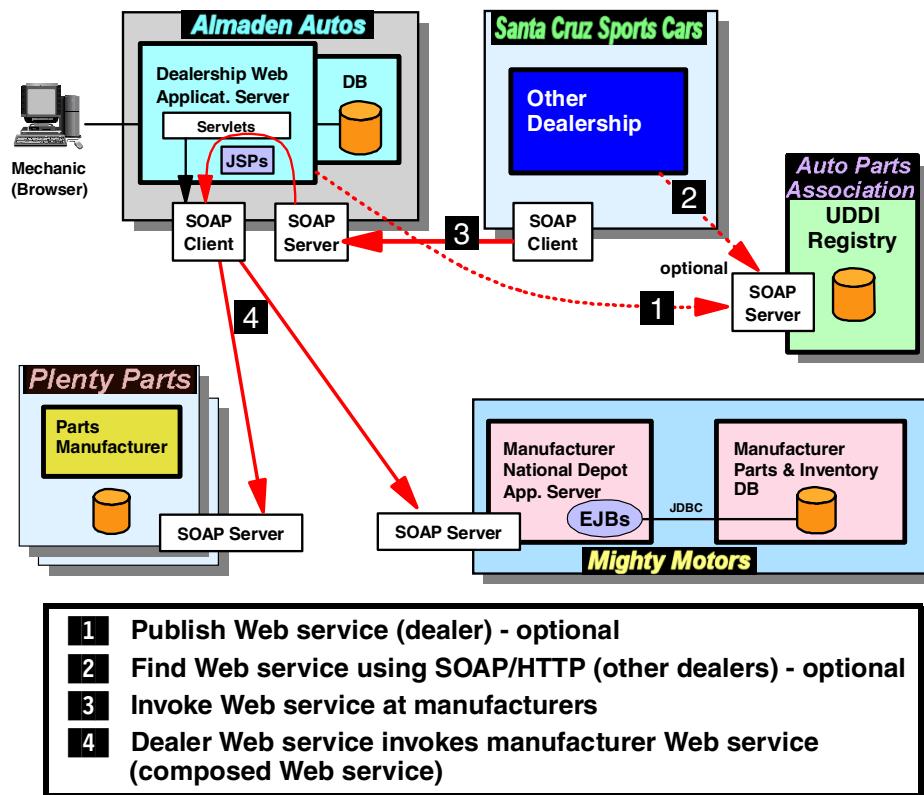


Figure 12-1 Cross dealership enablement

We introduce another car dealer, Santa Cruz Sports Cars, which uses the Web service of Almaden Autos to look up parts. Using the terminology introduced in Chapter 8, “Web services overview and architecture” on page 241 Santa Cruz Sports Cars is the *service requestor* and Almaden Autos is now the *service interface provider* and *service implementation provider*.

We use a static binding between Santa Cruz Sports Car and Almaden Autos. We also want to illustrate that a Web service can invoke another Web service transparent to the initial requestor. To show this, we keep the relationship between Almaden Autos as a *service requestor* and Mighty Motors as a *service interface provider* and *service implementation provider*. This means that Santa Cruz Sports Car invokes the Web service of Almaden Autos to search for a part. If the part is not in the inventory of Almaden Autos, then the Almaden Autos Web service will invoke the Mighty Motors Web service to see if the part is in its inventory.

We do not show how Santa Cruz Sports Cars in turn can expose its inventory system as a Web service, because this step is identical to the case of Almaden Autos we describe in this chapter.

Note: This scenario is somehow set up to illustrate a composed Web service. In a real business scenario, Almaden Autos, Santa Cruz Sports Cars, Mighty Motors, and all other car dealers, part and car manufacturers would register themselves in the public UDDI registry. Anyone looking for a specific part would do a lookup in the UDDI and invoke all of the registered Web services to look for the part. Advanced UDDI topics are discussed in “Advanced UDDI topics” on page 489.

Moreover, a composed Web service would usually invoke Web services with a different signature than the original one. For instance, we could have a parts inquiry Web service that invokes in its implementation another Web service which looks up the prices for the parts found. It then sends back all information (parts + prices) to the requestor. For simplicity reasons, we do not introduce a Web service with a different signature than the one created in Chapter 10, “Static Web services” on page 321.

Class and interaction diagrams

The component model for auto parts stage 4 is displayed in Figure 12-2. We will explain the classes, attributes, and methods when we implement them.

The interaction diagram (Figure 12-3) shows the scenario in which nothing is found in the local database of Santa Cruz Sports Cars, and nothing is found in the Almaden Autos database. Eventually, the part is found in the Mighty Motors database.

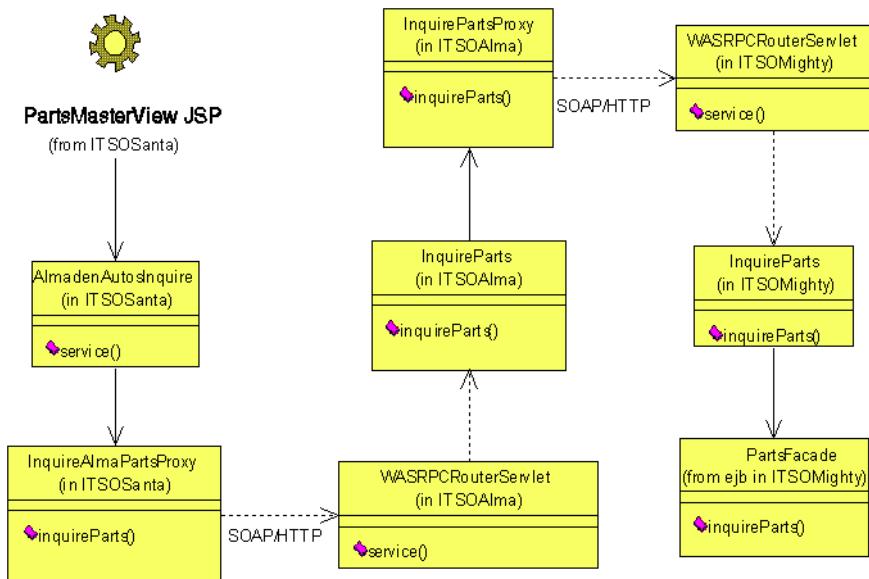


Figure 12-2 Class diagram

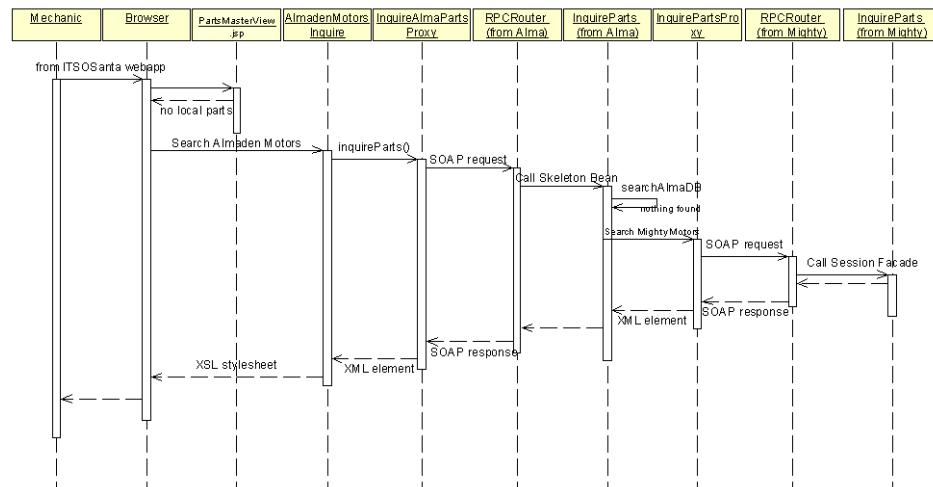


Figure 12-3 Interaction diagram

Preparing for development

You must have completed Chapter 10, “Static Web services” on page 321 before starting this chapter. The completion of Chapter 11, “Dynamic Web services” on page 379 is not mandatory. The projects you should already have in the workspace before the start of this chapter include:

- | | |
|------------------------|---|
| ITSOAlmaEAR | This is the EAR project containing the enterprise archive for the Almaden Auto parts inventory application. |
| ITSOAlmaWeb | The Web project that contains the Almaden Autos Web application with the servlets and JSPs developed in Chapter 3, “Web development with Application Developer” on page 55. This Web application provides JDBC access to its own database and contains a servlet that invokes the Web service of Mighty Motors developed in Chapter 10, “Static Web services” on page 321. If you also finished Chapter 11, “Dynamic Web services” on page 379 then you will have a second Web service client proxy to the Mighty Motors or Plenty Parts inventory. |
| ITSOMightyEAR | The EAR project containing the enterprise archive for Mighty Motors. |
| ITSOMightyWeb | The Web project developed in Chapter 10, “Static Web services” on page 321 that contains the Web service of Mighty Motors. |
| ITSOMightyEJB | The EJB project developed in Chapter 5, “EJB development with Application Developer” on page 133. |
| ITSOMightyXML | The XML project used in Chapter 4, “XML support in Application Developer” on page 91, which contains the XML schema that we are using to describe the parts inventory inquiry result set. |
| ITSOADSADServer | The server project created in Chapter 3, “Web development with Application Developer” on page 55, which contains a WebSphere test server instance and configuration that we will continue to use to test our Web applications. |
| ITSOPlenty... | If you completed Chapter 11, “Dynamic Web services” on page 379, your workspace also contains <i>ITSOPlentyEAR</i> , the enterprise archive for the Plenty Parts manufacturer and the corresponding Web project, <i>ITSOPlentyWeb</i> . You do not need to have these projects in your workspace to complete this chapter |

Creating a new dealer

As discussed in the introduction of this chapter, we introduce another dealer, Santa Cruz Sports Cars. This additional car dealer uses the Web service of Almaden Autos, developed in this chapter, to search for parts.

To make it easy, Santa Cruz Sports Cars is (besides the name and the database tables it uses) identical to Almaden Autos developed in Chapter 3, “Web development with Application Developer” on page 55. Therefore, you can follow the guidelines of that chapter in order to create the Santa Cruz Sports Cars dealer. You can skip the section “Creating the ITSOWSAD sample database” on page 61 because the ITSOWSAD already contains the tables necessary to construct the Santa Cruz Sports Cars dealer. You should use the SSINVENTORY and SSPARTS tables instead of the AAINVENTORY and AAPARTS tables throughout the guidelines.

If you do not want to go through all the steps of Chapter 3, “Web development with Application Developer” on page 55, you can just import the necessary projects into the workspace by taking the following steps:

- ▶ In the Web perspective select *File -> Import*. Select *EAR file* and click *Next*.
- ▶ Locate the sampcode\wscomposed\ITSOSanta.ear file and specify *ITSOSantaEAR* as the project name. Click *Finish*.

After importing the EAR file, two new projects are added to the workspace:

- EAR project **ITSOSantaEAR**
- Web project **ITSOSantaWeb**
- ▶ Add the ITSOSantaEAR project to the ITSOWSADWebSphere server configuration.

Make sure the Santa Cruz Web application is working before continuing. You can test the Web application the same way as you test the Almaden Autos Web application (use part number M10000001 as test input).

Creating the Almaden Autos Web service

In this section, we develop the Almaden Autos Web service. In Chapter 10, “Static Web services” on page 321, we started from an existing JavaBean and generated the WSDL document. Unlike in the static Web services construction, this time we start from an existing WSDL document and generate the JavaBean skeleton, which we then implement. The reason why we start from an existing WSDL document, is that we want to comply to the existing Web service interface of Mighty Motors.

Creating the JavaBean skeleton

In this section we create the new Web service in Almaden Autos from the existing WSDL file. We use the Web service wizard to generate the JavaBean skeleton from the existing InquireParts-binding.wsdl document (Figure 10-24 on page 348), which we copy from the ITSOMightyWeb project.

Note: In a real-life situation you cannot copy the file from the Mighty Motors Web application as we do here. Instead you could download the file through the URL pointer in the UDDI registry as illustrated in Chapter 11, “Dynamic Web services” on page 379, or you could receive it through e-mail.

Copy the Web service interface file

We copy the file InquireParts-binding.wsdl from the Mighty Motors Web application:

- ▶ In the Web perspective, create a new folder called composed in the ITS0AlmaWeb project’s /webApplication/wsdl folder.
- ▶ Select ITSOMightyWeb/webApplication/wsdl/InquireParts-binding.wsdl and *Copy*.
- ▶ Select ITS0AlmaWeb/webApplication/wsdl/composed as the destination and click *OK*.

Copy the XML schema and update the WSDL file

We prefer to have the XML schema in the ITS0AlmaWeb application instead of being dependent on the Mighty Motors application, therefore:

- ▶ Copy the InquireParts.xsd file:

```
from: ITSOMightyWeb/webApplication/wsdl  
to:   ITS0AlmaWeb/webApplication/wsdl/composed
```
- ▶ Edit the InquireParts-binding.wsdl file and change the location:

```
from: http://localhost:8080/ITSOMighty/wsdl/InquireParts.xsd  
to:   http://localhost:8080/ITS0Alma/wsdl/composed/InquireParts.xsd
```

Note: In a real life you cannot just copy the file. Instead, you can invoke the link for the schema in a browser, and download it to your file system; afterwards, importing it into your workspace. Alternatively, this XML schema can be hosted at a central location, such as the Auto Parts Association standards body.

Start the server

Start the ITSOWSADWebSphere server instance; it will be used to retrieve the XML schema from the WSDL binding file.

Web service wizard

To generate the skeleton, we run the Web service wizard:

- ▶ The starting point is the existing WSDL interface file. Select the webApplication/wsdl/composed/InquireParts-binding.wsdl file and *New->Other -> Web Services -> Web Service*. Click *Next*.
- ▶ In the first panel of the wizard:
 - Set the *Web service type* to *Skeleton Java bean Web service*. This should be already set since we are generating from the service WSDL.
 - Select the ITSOAlmaWeb project as the target Web project.
 - Select *Generate a proxy* and *Generate a sample*.
 - Leave the *Overwrite files without warning* checkbox unmarked and the *Create folders where necessary* checkbox marked. Click *Next*.
- ▶ The WSDL file is preselected (composed/InquireParts-binding.wsdl), click *Next*.
- ▶ The next panel in the wizard is the *Web Service Skeleton Java Bean Configuration* panel (Figure 12-4).

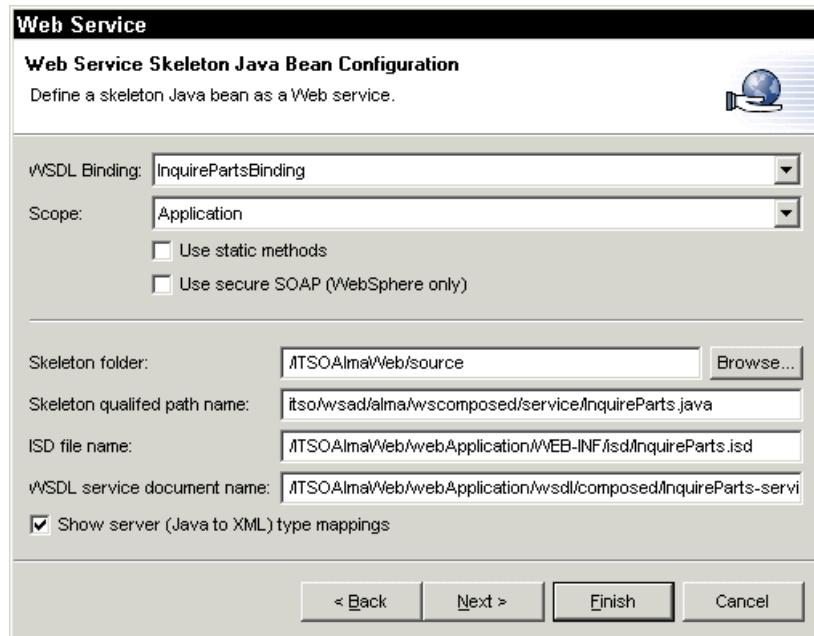


Figure 12-4 Web service skeleton JavaBean generation

- Leave the *WSDL Binding* set to *InquirePartsBinding*.

- Set the *Scope* to *Application*. The Web service scope is explained in “Web service scope” on page 338.
- Change the *Skeleton qualified path name* to:
`itso/wsad/alma/wscomposed/service/InquireParts.java`
 Note that the name is entered with slashes and not with periods.
- Shorten the *ISD file name* to:
`/ITSOAlmaWeb/webApplication/WEB-INF/isd/InquireParts.isd`
- Change the *WSDL service document name* to:
`/ITSOAlmaWeb/webApplication/wsdl/composed/InquireParts-service.wsdl`
- Select *Show server (Java to XML) type mappings*.
- Click *Next*.
- ▶ The next panel is the *Web Service Skeleton XML to Java mappings* panel.
 - The first entry *string, SOAP encoding* defines how the input XML parameter (the partnumber) is mapped to the skeleton JavaBean. You can leave the selection as *Show and use the default Java Bean mapping*.
 - The second entry, *InquirePartsResult*, defines the mapping between the output of our skeleton JavaBean to the XML element in the reply message. You can leave the *Show and use the default DOM Element mapping*.

More information about the server mappings is described in “XML /Java mappings” on page 340.
- ▶ In the proxy generation panel set the proxy class to:
`itso.wsad.alma.wscomposed.proxy.InquirePartsProxy`
- ▶ In the sample generation panel, set the sample folder to:
`/ITSOAlmaWeb/webApplication/sample/Composed`
- ▶ Click *Finish*.

Investigating the generated files

The JavaBean skeleton wizard generated a number of files, which we will describe in this section.

WSDL implementation file

Starting from the interface file (`InquireParts-binding.wsdl`) the wizard generated the implementation file (`InquireParts-service.wsdl`) shown in Figure 12-5. This file can be used unchanged.

```

<?xml version="1.0" encoding="UTF-8"?>
<definitions name="InquirePartsBinding_service"
    targetNamespace="http://localhost:8080/ITSOAlma/wsdl/composed/
        InquireParts-service.wsdl" xmlns="http://schemas.xmlsoap.org/wsdl/"
        xmlns:tns="http://localhost:8080/ITSOAlma/wsdl/composed/InquireParts-service.wsdl"
        xmlns:binding="http://www.redbooks.ibm.com/ITSOSJC/definitions/
            InquirePartsRemoteInterface" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">
<import namespace="http://www.redbooks.ibm.com/ITSOSJC/definitions/
    InquirePartsRemoteInterface"
location="http://localhost:8080/ITSOAlma/wsdl/composed/InquireParts-binding.wsdl"/>
<service name="InquirePartsBinding_service">
    <port name="InquirePartsBinding_port" binding="binding:InquirePartsBinding">
        <soap:address location="http://localhost:8080/ITSOAlma/servlet/rpcrouter"/>
    </port>
</service>
</definitions>

```

Figure 12-5 Generated WSDL implementation file

SOAP deployment and administration files

The folders and files for the SOAP server and for testing are:

admin	SOAP administration application
WEB-INF/isd	InquireParts.isd file
dds.xml, soap.xml	SOAP deployment files
lib	SOAP deployment JAR files

Web service testing files

The folders and files for testing of the Web service are:

proxy	itso.wsad.alma.wscomposed.proxy.InquirePartsProxy
sample/Composed	Sample test client (TestClient.jsp)

Generated JavaBean skeleton

Figure 12-6 shows the JavaBean skeleton generated by the Web service wizard. We have to complete the `inquireParts` method to implement the Web service. This method, with the `partNumber` as input and the `DOM` element as output, matches the operation defined in the service interface WSDL file. We will complete the `inquireParts` method in “Implementing the JavaBean skeleton” on page 423.

```
package itso.wsad.alma.wscomposed.service;

import org.w3c.dom.*;

public class InquireParts
{
    public org.w3c.dom.Element inquireParts(java.lang.String partNumber)
    {
        // implement your code here
        return null;
    }
}
```

Figure 12-6 Generated JavaBean skeleton

Attention: Although we specified the qualified path for the skeleton JavaBean as `itso/wsad/alma/wscomposed/service/InquireParts.java`, the code was generated into a folder named `itso.wsad.alma.wscomposed.service`.

To fix this, create a service folder under `itso/wsad/alma/wscomposed` and move the `InquireParts.java` file into the new folder. Note that the package name in the code is correct.

Implementing the JavaBean skeleton

The Web service wizard generated the skeleton JavaBean, `InquireParts`, that implements the Web service (Figure 12-6).

All we have to do is to complete this JavaBean. We already developed the code to retrieve the data from the local database, to convert SQL results into an XML element, and to invoke the Mighty Motors Web service. We assemble all the pieces into the `InquireParts` class.

Import the `InquireParts.java` file from the `sampcode\wscomposed` directory and overwrite the class in the `itso.wsad.alma.wscomposed.service` package.

Now let us take a closer look at the methods in the `InquireParts` class:

- ▶ The `inquireParts` method (Figure 12-7) implements the Almaden Autos Web service. It allocates and executes the `PartsMasterViewBean` to retrieve local data and converts the local data by calling the `convertXml` method. If no local data is found, it invokes the Mighty Motors Web service using the proxy generated for the static Web service.

```

public org.w3c.dom.Element inquireParts(java.lang.String partNumber)
{
    //Variables
    PartsMasterViewBean masterViewBean = new PartsMasterViewBean();
    Element result = null;
    try {
        // search local DB
        masterViewBean.setpartNo(partNumber);
        masterViewBean.setDataSourceName("jdbc/ITSOWSAD");
        masterViewBean.execute();
        if ( masterViewBean.first() ) {
            System.out.println("Almaden Auto Result");
            // convert result to XML
            result = convertToXml(masterViewBean);
            return result;
        }
        // nothing found in local Database, look for MightyMotors
        itso.wsad.alma.wsstatic.proxy.InquirePartsProxy proxy =
            new itso.wsad.alma.wsstatic.proxy.InquirePartsProxy();
        result = proxy.inquireParts(partNumber);
        if (result!=null) System.out.println("Mighty Motors Result");
        return result;
    } catch (Throwable e) {
        e.printStackTrace();
    }
    return result;
}

```

Figure 12-7 Web service implementation method

- ▶ The convertToXml method (Figure 12-8) converts a local SQL result data to a DOM Element to match the result of the Mighty Motors Web service. First the result is initialized and then the populatePart method is called for each result row of the SQL view bean.
- ▶ The populatePart method retrieves the column values of the SQL row to generate the XML for one inventory item by calling the newElement method for each column value.
- ▶ The newElement method creates one name/value pair.
- ▶ The getDocumentBuilder method instantiates a shared document builder.

The last three methods are identical or similar to the methods used for the static Web service in “Implementing the InquireParts JavaBean” on page 327.

```

private org.w3c.dom.Element convertToXml(PartsMasterViewBean masterViewBean)
                                         throws Exception
{
    Document doc = null;
    Element result = null;
    doc = getDocumentBuilder().newDocument();
    result = doc.createElement("InquirePartsResult");
    result.setAttribute("xmlns",
                        "http://www.redbooks.ibm.com/ITSOVSAD/schemas/InquireResults");
    result.setAttribute("xmlns:xsi",
                        "http://www.w3.org/2001/XMLSchema-instance");
    result.setAttribute("xsi:schemaLocation",
                        "http://www.redbooks.ibm.com/ITSOVSAD/schemas/InquireResults
                        wsdl/composed/InquireParts.xsd");
    try {
        do {
            result.appendChild(populatePart(doc, masterViewBean));
        }
        while (masterViewBean.next() );
    } catch (Throwable e) {
        e.printStackTrace();
    }
    return result;
}

```

Figure 12-8 Method to convert SQL result into DOM element

This scenario is described in the interaction diagram in Figure 12-3 on page 416.

Testing the Web service

Start the ITSOVSADWebSphere server (actually it should be running from generating the Web service).

Select the sample test client (sample\Composed\TestClient.jsp) and *Run on Server*.

Run the Web service with part numbers of M100000001 and M100000003. The first test retrieves local data and the second test invokes the Mighty Motors Web service. Both sets of XML result data are identical in format.

Creating the Santa Cruz Sports Cars Web application

To access the new Web service implemented for Almaden Autos, we create a client Web application for Santa Cruz Sports Cars.

We assume that Almaden Web autos sends the WSDL files to Santa Cruz Sports Cars. Santa Cruz Sports Cars then integrates the Web service into their existing Web application using static bindings.

Copying the WSDL file to the ITSOSantaWeb project

To run the Web service client wizard, we have to copy the WSDL file from ITS0AlmaWeb to ITS0SantaWeb:

- ▶ Create a wsdl folder under ITSOSantaWeb/webApplication.
- ▶ Copy the InquireParts-service.wsdl file:
 - from ITS0AlmaWeb webApplication/wsdl/composed
 - to ITSOSantaWeb webApplication/wsdl

Starting the server

To run the Web service client we have to start the ITS0SADWebSphere server instance so that the wizard is able to connect and retrieve the rest of the Web service implementation (the XML schema InquireParts.xsd and the service interface file InquireParts-binding.wsdl).

Creating the Web service client

The task of creating a Web service client for the Almaden Web service is similar to “Creating the Almaden Autos Web service client” on page 363:

- ▶ In the ITSOSantaWeb project select the InquireParts-service.wsdl file and *New -> Other -> Web services -> Web Service client*. Click *Next*.
- ▶ The ITS0AlmaWeb project is selected on the first panel. Select *Generate a proxy* and *Generate a sample*, click *Next*.
- ▶ The WSDL service file is preselected, click *Next*.
- ▶ On the next panel, mark enter the proxy class name, select *Show mappings*, click *Next*.
`itso.wsad.santa.wsstatic.proxy.InquireAlmaPartsProxy`
- ▶ For the XML to Java mappings, leave the defaults and click *Next*.
- ▶ Click *Next* on the next two panels.

- ▶ On the sample generation panel the sample is generated into:
`/ITSOSantaWeb/webApplication/sample/InquireAlmaParts`
- ▶ Click *Finish*.

Testing the proxy

Select the `sample\InquireAlmaParts\TestClient.jsp` and *Run on Server*. Enter part numbers of M100000001 and M100000003 and check the XML results.

We have now completed the generation and testing of the proxy to call the Almaden Autos Web service. Next we have to integrate that into the Santa Cruz Sports Cars Web application.

Adding the XSL style sheet

The returned XML element from the Web service is converted to HTML by applying the same XSL style sheet created in “Creating the XSL style sheet” on page 366.

Therefore copy the `InquirePartsResult.xsl` file:

- ▶ from `ITSOAlmaWeb/webApplication/stylesheets/static`
- ▶ to `ITSOSantaWeb/webApplication`

Edit the XSL style sheet and change the name of the image file from `almadenautos.gif` to `santacruzsportscars.gif`, and the heading to `<H1>Almaden Autos Inventory Inquiry Results</H1>`.

Because we use the Apache Xalan XSL processor, we have to add the `xalan.jar` into the `ITSOSantaWeb` project *Java build path*. You can do this by adding the `WAS_XALAN` variable to the build path.

Creating the servlet to invoke the Web service

To invoke the Almaden Autos Web service from Santa Cruz Sports Cars we have to extend the `ITSOSantaWeb` application. We develop a servlet to invoke the Web service through the proxy and then add it to the Santa Cruz Sports Cars Web application by adding a link to the mechanics Web page.

To create the servlet that invokes the Almaden Web service:

- ▶ In the Web perspective, for the `ITSOSantaWeb` project select the `source/itso/wsad/santa/web` folder and click on the *Create a Java Servlet class* icon .
- ▶ Specify the servlet name as `AlmadenAutosInquire`. Click *Finish*.

Open the AlmadenAutosInquire servlet (Figure 12-9):

- ▶ Complete the servlet code by copying the methods of the MightyMotorsInquire servlet in the ITSOAlmaWeb project.
- ▶ Change the two statements (in bold) to point to the ITSOSanta project:
 - The address of the XSL file
 - Instantiate the InquireAlmaPartsProxy

```
package itso.wsad.santa.web;

import javax.servlet.http.HttpServlet;

public class AlmadenAutosInquire extends HttpServlet {

    public void doPost( ..... ) { ..... }
    public void doGet( ..... ) { ..... }
    public void performTask( ..... ) {
        try {
            response.setContentType("text/html; charset=UTF-8");
            javax.xml.transform.TransformerFactory tFactory =
                javax.xml.transform.TransformerFactory.newInstance();
            javax.xml.transform.Source xslSource =
                new javax.xml.transform.stream.StreamSource(
                    new java.net.URL
                    ("http://localhost:8080/ITSOSanta/InquirePartsResultTable.xsl").openStream());
            javax.xml.transform.Transformer transformer =
                tFactory.newTransformer(xslSource);
            java.io.PrintWriter out = response.getWriter();

            // For SOAP over HTTP
            itso.wsad.santa.wsstatic.proxy.InquireAlmaPartsProxy proxy =
                new itso.wsad.santa.wsstatic.proxy.InquireAlmaPartsProxy();
            org.w3c.dom.Element result = proxy.inquireParts
                ((String)request.getSession().getAttribute("partNo"));
            javax.xml.transform.Source xmlSource = new
                javax.xml.transform.dom.DOMSource(result);
            transformer.transform(xmlSource,
                new javax.xml.transform.stream.StreamResult(out));
        } catch (Throwable e) {
            e.printStackTrace();
        }
    }
}
```

Figure 12-9 Client servlet that invokes the Web service

Linking the servlet into the Web application

The last step is to add a link to the new servlet into the existing Santa Cruz Sports Cars Web application. This is identical to the change we did for the Almaden Auto Web application in “Linking the new servlet into the Web application” on page 370.

- ▶ In the Web perspective of the ITS0SantaWeb project, open the PartsResultTable.jsp file in the page designer. Switch to the *Source* tab and enter this code before and after the table:

```
<% if ( !masterViewDBBean.first() ) { %>No data was returned. Click <A href="AlmadenAutosInquire">here</A> to search Almaden Autos.  
<% } else { %>  
<!--Table-->  
<TABLE border="1">  
.....  
</TABLE>  
<% } %>
```

- ▶ Close the JSP and save the changes.

Testing the Santa Cruz Sports Cars Web application

Now we are ready to test the Santa Cruz Sports Car Web application, which now contains a link to invoke the Almaden Autos Web service.

Select the PartsInputForm.html file in the ITS0SantaWeb project and select *Run on Server*. The browser opens and the parts inventory inquiry input page is displayed.

We can test three scenarios with three different input part numbers as shown in Table 12-1.

Table 12-1 Scenarios for composed Web services

Part number	Part found in	Web services invoked
M100000001	Santa Cruz tables	none
M100000002	Almaden Autos tables	Almaden Web service
M100000003	Mighty Motors tables	Almaden Web service Mighty Motors Web service

The three outputs are shown in Figure 12-10.

Part in Santa Cruz tables

Select	Part Number	Name	Quantity	Cost
<input type="radio"/>	M100000001	CR-MIRROR-L-01	10	89.99

Part in Almaden Autos tables

Part Number	Name	Quantity	Cost	Location
M100000002	CR-MIRROR-R-01	5	99.99	AA - Almaden

Part in Mighty Motors tables

Part Number	Name	Quantity	Cost	Location
M100000003	CR-MIRROR-R-01	10	59.99	MM - San Francisco
M100000003	CR-MIRROR-R-01	12	57.99	MM - New York

Figure 12-10 Santa Cruz Sports Cars: Part retrieval scenarios

When running the scenario with part number M100000003 you should see two rpcrouter servlets initialized in the console, indicating that two different SOAP servlets have been invoked, one for the Almaden Autos Web application, and one for the Mighty Motors Web application.

Summary

In this chapter we illustrated:

- ▶ How to build a composed Web service that can be called and that calls other Web services
- ▶ How to generate a JavaBean skeleton from an existing WSDL file, and what tasks you currently must perform to complete the Web service implementation

Quiz: To test your knowledge of composed Web services in Application Developer, try to answer the following five questions. The answers are provided in Appendix C, “Quiz answers” on page 559.

1. True or false: If you have the service interface and service implementation files of a Web service, you can generate a JavaBean that implements the Web service?
2. What JAR file contains the WAS Message Router Servlet?
3. From which WSDL file do you start to generate the JavaBean skeleton: interface file, or implementation file?
4. Which WSDL file is generated when you create the JavaBean skeleton: interface file, or implementation file?

Which file is the SOAP deployment descriptor?



Deployment of Web services to WebSphere

In this chapter we deploy the static, dynamic, and composed Web services applications to WebSphere Application Server. We describe the deployment for:

- ▶ WebSphere Application Server Advanced Edition 4.0 Single Server (AEs)
- ▶ WebSphere Application Server Advanced Edition 4.0 Full Version (AE)

Preparing for deployment

In this chapter we assume that you have finished Chapter 12, “Composed Web services” on page 413. However, you can easily deploy the sample applications after Stage 2b or Stage 3, applying the instructions explained in this chapter. We deploy four enterprise applications:

- ▶ Almaden Autos enterprise application introduced in Chapter 3, “Web development with Application Developer” on page 55, and extended in the other chapters
- ▶ Mighty Motors enterprise application introduced in Chapter 5, “EJB development with Application Developer” on page 133 and further extended with a Web service in Chapter 10, “Static Web services” on page 321
- ▶ Plenty Parts enterprise application introduced in Chapter 11, “Dynamic Web services” on page 379
- ▶ Santa Cruz Sports Cars enterprise application introduced in Chapter 12, “Composed Web services” on page 413

URL considerations

If we take a look at the different resources of the four sample applications, we see that there are quite some references to `localhost` and port 8080:

- ▶ In the `ITSOAlmaWeb` application, the `InquireMightyMotors` servlet contains a URL reference to the XSL style sheet which is used to display the XML results in HTML format:
`http://localhost:8080/ITSOAlma/InquirePartsResultTable.xsl`
- ▶ All the generated WSDL files contain references to `localhost:8080`.
- ▶ The URL for invoking the SOAP servlets in the generated proxy clients contain references to `localhost:8080`. For example, `InquireAlmaPartsProxy` in the `ITSOAlmaWeb` application contains the following line in its constructor:
`new URL("http://localhost:8080/ITSOAlma/servlet/rpcrouter")`

Tip: To see all references to `localhost:8080`, in the Web perspective select: *Edit -> Search*, select *Text Search* and type *localhost:8080*.

Normally, you change all these references when you deploy Web applications to a production environment. There are two approaches:

- ▶ Replace the absolute URL paths with relative URL paths. You can do this only in certain cases such as in HTML links. You can, however, not change the URL reference for the `rpcrouter` servlet in the client proxy to a relative path

because you are referencing a servlet outside the current Web application. Moreover, the `java.net.URL` constructor requires an absolute URL as a parameter.

- ▶ Replace all occurrences of `localhost` with the real hostname of the production server. Port 8080 will be omitted in most cases because you probably want to use the default HTTP port 80.

In this chapter, however, we have chosen not to replace the URL references of `localhost:8080`. Instead, we configure the application server in such a way that is also listens to port 8080. Such a configuration is not suggested for a real environment; we only want to demonstrate the deployment of Web services.

Export the enterprise applications

In this section we create EAR files for deployment by exporting the enterprise applications to EAR files. We will also make use of the application assembly tool (AAT) of the WebSphere Application Server to modify the Almaden Autos EAR file once it is created.

Export EAR files

To deploy the four enterprise applications, we export them as a .ear files in Application Developer:

- ▶ Select *File -> Export -> select EAR file*. Click *Next*.
- ▶ Select one the four enterprise applications you want to export and specify a path and filename for the .ear file to be exported.
- ▶ Click *Finish*.

Add the SOAP admin client

The SOAP admin client is a Web application which allows you to see the deployed Web services and to start and stop them. The use of the SOAP admin client is described in “Viewing the deployed Web services” on page 352.

When generating the Web service from the JavaBean, described in “Using the Web service wizard” on page 336, the SOAP admin client was added to the Web application. However, this was not the case in the Web service JavaBean skeleton wizard illustrated in “Creating the JavaBean skeleton” on page 419.

Attention: The final Application Developer product code does generate the admin client into the Web application, but learning about AAT is good anyway.

We can add the SOAP admin client to the Almaden enterprise application by using the application assembly tool that comes with WebSphere 4.0:

- ▶ Start AAT by selecting *Start -> Programs -> IBM WebSphere -> Application Server V4.0 AEs or AE -> Application Assembly Tool*.
- ▶ On the *Welcome To Application Assembly Tool* switch to the *Existing* tab and click *Browse*.
- ▶ Browse to the directory where you exported the *ITSOAlma.ear* file and double-click on the file. Back in the *Existing* panel click *OK*.
- ▶ *ITSOAlma.ear* is now open in AAT (Figure 13-1).

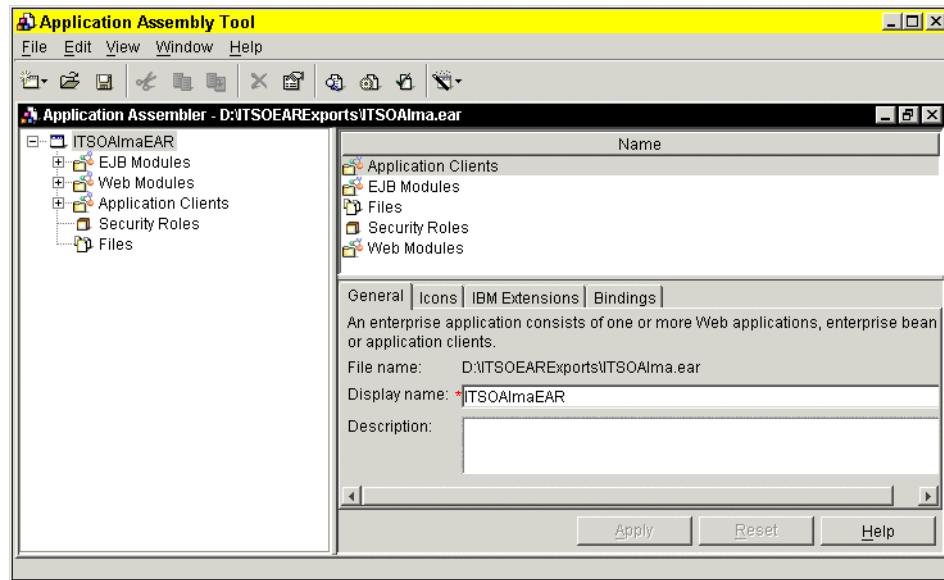


Figure 13-1 *ITSOAlma.ear* opened in AAT

- ▶ Expand *Web Modules*, *ITSOAlmaWeb* and *Files*.
- ▶ Right-click on *Resource Files* and select *Add Files*.
- ▶ In the *Add Files* dialog, click *Browse* and select the directory:
`WSAD_ROOT\plugins\com.ibm.etools.webservice\runtime\admin\websphere`
Where `WSAD_ROOT` is the root directory where you installed Application Developer (for example, `C:\WSAD`). Click *Select*.
- ▶ Select all files and click *Add* so that they appear in the list of *Selected Files* (Figure 13-2).
- ▶ Click *OK*. The admin client files are now in the list of Resource files of the *ITSOAlmaWeb* module.

- Select *File -> Save* to save the *ITS0Alma.ear* file and exit AAT.

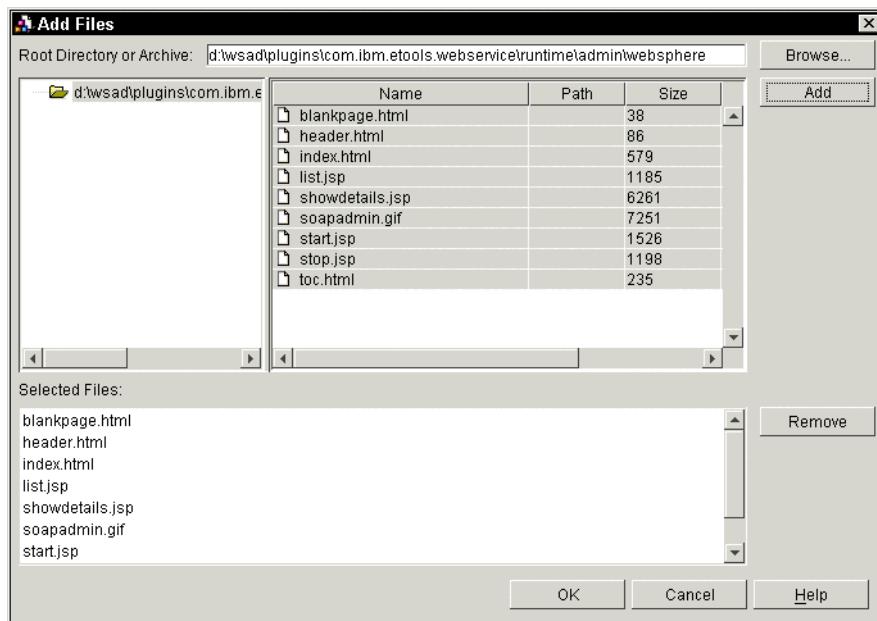


Figure 13-2 Adding the admin client files

Note: The application assembly tool (AAT) that ships with WebSphere is a tool that enables you to assemble enterprise application into an EAR file. With AAT you can:

- Modify and add modules to applications
- Generate deployment code for modules
- Verify archives
- View deployment descriptors
- Specify JNDI bindings and resolve references
- Convert EJB JAR files from 1.0 to 1.1 format

We have not been using the Application Assembly Tool so far, because of two major reasons:

- Unlike VisualAge for Java 4.0 or WebSphere Studio 4.0, Application Developer is capable of exporting EAR files.
- Application Developer is capable of generating deployed code for the EJB 1.1 specification.

Deployment to WebSphere AEs

First we describe the deployment to WebSphere Advanced Single Server (AEs).

Uninstall the previous enterprise applications

In Chapter 6, “Deployment of Web and EJB applications to WebSphere” on page 169, we deployed the Mighty Motors and the Almaden Autos enterprise applications. If you completed that chapter, you have to remove those enterprise applications, except for the EJB test client, from the Application Server in order to complete this chapter.

The sample applications we install in this chapter use the same data source we defined in “Creating the JDBC driver and data source” on page 174. You must complete that section in order to complete this chapter.

To uninstall the previous versions of the sample applications:

- ▶ Start the Application Server as described in “Starting the AEs server” on page 172.
- ▶ Start the Administrator’s Console as described in “Starting the Administrator’s Console” on page 173.
- ▶ Expand *Nodes*, *YourNodeName*, and click on *Enterprise Applications*.
- ▶ In the list of Enterprise Applications mark the checkbox next to *ITSOAlmaEAR* and *ITSOmightyEAR* and click *Stop*.
- ▶ You receive a message indicating that the enterprise applications have been stopped. Again mark the same checkboxes and click *Uninstall*.
- ▶ In the *Specify the application that you want to install* panel, leave the settings to *No* for *Export* and *Yes* for *Delete all related files*.
- ▶ Click *Uninstall*. Almaden Autos and Mighty Motors enterprise application are removed from the list of enterprise applications.

Once you uninstalled the Almaden Autos and Mighty Motors enterprise applications, the Administrator’s Console warns you to regenerate the plug-in and to save the configuration. You can postpone this until you finished the installation of the new EAR files described in the next section.

Uninstall by command

You can also uninstall enterprise application using the `SEAppInstall` command:

```
seappinstall -uninstall ITSOAlmaEAR -delete true
```

Install the enterprise applications

You can install the enterprise applications using the Administrator's Console or the SEAppInstall.

Using the Administrator's Console

The installation of the Almaden Autos, Mighty Motors, Plenty Parts and Santa Cruz Sports Cars enterprise applications using the Administrator's console is similar to the installation described in "Deployment to WebSphere AEs" on page 170.

To install the Almaden Autos, Santa Cruz Sports Cars and Plenty Parts enterprise applications (which contain no EJB modules) follow the guidelines described in "Installing the Almaden Autos enterprise application" on page 176.

For the Mighty Motors enterprise application, which contains an EJB module, follow the instruction described in "Installing the Mighty Motors enterprise application" on page 178. Make sure you specify ITSO as the *Schema Name* in the *Database and Data Source settings* and that you deselect the *Re-Deploy option* for the EJB modules.

Using SEAppInstall

When can also install the enterprise applications with the command line tool SEAppInstall, located in the WAS_ROOT/bin directory. To see all available options with this tool, enter the command SEAppInstall.

We assume that the exported EAR files are in a directory called c:\temp.

- ▶ To install the Almaden Autos enterprise application (and Santa Cruz Sports Cars and Plenty Parts) enter this command:

```
seappinstall -install c:\temp\ITSOAlma.ear -interactive false
```

- ▶ To install the Mighty Motors enterprise application enter:

```
seappinstall -install c:\temp\ITSOmighty.ear -interactive false  
-ejbdeploy false -schemaName ITSO
```

Define port 8080 for AEs

As discussed in "URL considerations" on page 434, we have to enable the application server to accept requests on port 8080, if we do not change the source code. Therefore, we have to define two things:

- ▶ Add an HTTP Transport in the Web Container setting for port 8080
- ▶ Add an alias for virtual host *default_host*

We also could have changed the configuration of IBM HTTP Server to listen on port 8080. But then, we would only be able to test our sample applications with the IBM HTTP Server and not with the embedded HTTP server. By just specifying an HTTP Transport of 8080 and an alias, all requests to port 8080 will directly go to the embedded HTTP server, and not to the IBM HTTP Server.

Add an HTTP transport

An HTTP transport describes how requests flow from the embedded HTTP server to the Web container. To enable the embedded HTTP server to listen on port 8080 we have to add an HTTP Transport:

- ▶ Expand *Nodes* -> *YourNodeName* -> *Application Servers* -> *Default Server* -> *Web Container*.
- ▶ Click on *HTTP Transports*. In the right panel click on *New*.
- ▶ Leave the *Host Name* to * and set the *port* to 8080 (Figure 13-3). Click *OK*.

[HTTP Transports:](#)

Configuration for using an HTTP transport to communicate requests to the Web container

[*For more information...](#)

Properties	
Host Name:	* *
Port:	* 8080
<input type="checkbox"/> SSL Enabled	Whether to use SSL for the connection between the Web server and the Web container in WebSphere Application Server
<input type="checkbox"/> External	Specifies whether this transport is for internal or external use.

Buttons: OK | Reset | Cancel

Figure 13-3 HTTP transport settings in AEs

Add an alias

An alias is the TCP/IP hostname and port number used to request a Web module resource. For the virtual host *default_host* to accept requests for port 8080, we have to add an alias:

- ▶ Expand *Virtual Hosts*, *default_host*.
- ▶ Click on *Aliases* and click *New* in the right panel. Specify a * for the *Host Name* and 8080 for the *port*. Click *OK*.

Regenerate the plug-in and save the configuration

Just like we did in Chapter 6, “Deployment of Web and EJB applications to WebSphere” on page 169, we have to regenerate the plug-in configuration so that the IBM HTTP Server becomes aware of the installed enterprise applications by reading its new plug-in configuration (`plugin-cfg.xml`). Follow the instructions in “Verifying the enterprise applications” on page 185.

Having regenerated the plug-in, save your configuration as described in “Saving the configuration” on page 181, and close the Administrator’s console. Now restart the IBM HTTP Server to enable the new plug-in configuration.

Add the necessary JAR files to the AEs class path

In “Working with the UDDI APIs” on page 398 we introduced some new API’s. We added various JAR files to the build path of the Almaden Web application and to the WebSphere server instance we used for unit testing.

It is necessary to add the JAR files that were added to the class path of the ITSOWSAD WebSphere server instance in “Unit testing” on page 408, to the WebSphere Application Server class path:

- ▶ Copy the `mail.jar`

```
from: WSAD_ROOT\plugins\com.ibm.etools.servletengine\lib  
to:   WAS_ROOT\lib\ext
```

- ▶ Check that `soap.jar` and `uddi4j.jar` are already in `WAS_ROOT\lib`.

Test the applications with AEs

Before we are able to test the deployed sample applications we have to restart the application server:

- ▶ Open a command prompt and enter `stopserver`.
- ▶ When the server is stopped enter `startserver` to start it again.

Use the `-configFile` option for both commands to specify the server configuration file, if you saved your configuration in a file other than the default (`server-cfg.xml`)

Test the Almaden Autos Web application

To test the Almaden Autos Web application with the embedded HTTP server or with the IBM HTTP Server, open a browser and enter the URL:

```
http://localhost:9080/ITSOAlma/PartsInputForm.html  
http://localhost/ITSOAlma/PartsInputForm.html
```

Enter the values in Table 13-1 for the part number to test the different scenarios:

Table 13-1 Scenarios for testing Almaden Autos

Part number	Part found in	Web services invoked
M100000001	local DB	none
M100000003	Mighty Motors tables and/or Plenty Parts tables (depending on the link you follow)	Mighty Motors or Plenty Parts tables (depending on the link you follow)

Test the Santa Cruz Sports Cars Web application

To test the Santa Cruz Sports Cars Web application enter the URL:

`http://localhost:9080/ITSOSanta/PartsInputForm.html`
`http://localhost/ITSOSanta/PartsInputForm.html`

Enter the values in Table 13-2 for the part number to test the different scenarios:

Table 13-2 Scenarios for testing Santa Cruz Sports Cars

Part number	Part found in	Web services invoked
M100000001	local Santa Cruz Sports Cars tables	none
M100000002	Almaden Autos tables	Almaden Autos
M100000003	Mighty Motors tables	Almaden Autos and Mighty Motors

Test the SOAP admin client

To test the SOAP admin client for the Almaden Autos and Mighty Motors Web applications enter the URL:

`http://localhost:9080/ITSOSanta/admin/index.html`
`http://localhost/ITSOSanta/admin/index.html`

`http://localhost:9080/ITSO_Mighty/admin/index.html`
`http://localhost/ITSO_Mighty/admin/index.html`

Deployment to WebSphere AE

For WebSphere Advanced (full) we deploy the same EAR files that we used in “Deployment to WebSphere AEs” on page 438.

Uninstall the previous enterprise applications

First, we have to remove the Almaden Autos and Mighty Motors enterprise applications deployed in Chapter 6, “Deployment of Web and EJB applications to WebSphere” on page 169, from the Application Server in order to complete this chapter.

The sample applications that we install in this chapter use the same data source we defined in “Creating the JDBC driver and data source” on page 191. You must complete that section in order to complete this chapter. We also use the same ITSOWSAD application server we defined in “Creating an application server” on page 192.

To uninstall the previous versions of the sample applications:

- ▶ Start the Admin Server as described in “Starting the Admin Server” on page 190.
- ▶ Start the Administrator’s Console as described in “Starting the Administrator’s Console” on page 190.
- ▶ In the Administrator’s Console, stop the ITSOWSAD application server if it is running.
- ▶ Expand enterprise applications, select ITSOAlmadenAutos and click *Remove*. Choose *No* when asked for exporting the files and *yes* for deleting the files.
- ▶ Repeat the same for the ITS0MightyMotors enterprise application.

Install the enterprise applications

The installation of Almaden Autos, Mighty Motors, Plenty Parts and Santa Cruz Sports Cars enterprise applications is similar to the installation described in “Deployment to WebSphere AE” on page 190.

In fact, to install Almaden Autos, Santa Cruz Sports Cars, and Plenty Parts enterprise applications (which contain no EJB modules) follow the guidelines described in “Installing the Almaden Autos enterprise application” on page 194.

For Mighty Motors, which contains an EJB module, follow the instruction described in “Installing the Mighty Motors enterprise application” on page 195. Make sure not to choose to regenerate the deployed code for the EJB modules.

Define port 8080 for AE

Because we did not change the source code, we want the application server to accept requests on port 8080, and we have to:

- ▶ Add an HTTP Transport in the Web container setting for port 8080
- ▶ Add an alias for virtual host *default_host*, indicating the port 8080

Add an HTTP transport

To do add an HTTP transport:

- ▶ In the Administrator's Console expand the *Application Servers*.
- ▶ Select ITSOWSAD and in the right panel click on the *Services* tab.
- ▶ Select *Web Container Service* and click *Edit properties*.
- ▶ The *Web Container Service* panel pops up. Select the *Transport* tab as shown in Figure 13-4. Under *HTTP Transports* you see the port used by the embedded HTTP server for the selected application server.
- ▶ Click *Add*. Specify * for the *Transport Host name* and 8080 for the *Transport Port*. Click *OK* to return to the *Web Container Service* Panel. Click *OK* again.
- ▶ Click *Apply* to enable the changes.

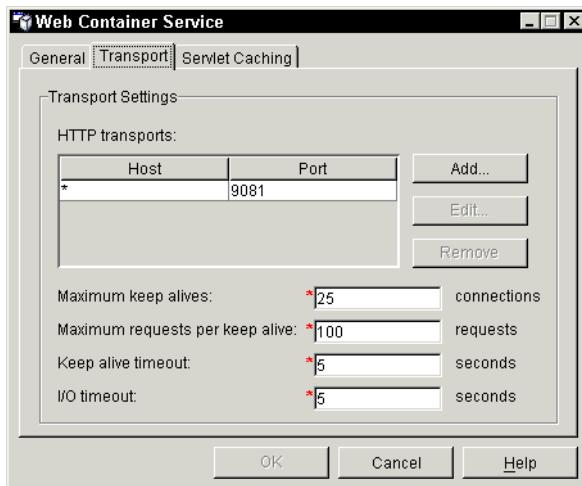


Figure 13-4 HTTP transport settings in AE

Add an alias

To define a virtual host alias for the *default_host* for port 8080:

- ▶ In the Administrator's Console select *Virtual Hosts*.

- ▶ Click on *default_host* and click on the *Add* button (next to the list of aliases) and add *localhost:8080* as an alias.

Add the necessary JAR files to the AE class path

Repeat the steps performed for WebSphere AEs in “Add the necessary JAR files to the AEs class path” on page 441.

Test the applications with AE

It is necessary to stop and restart the Admin Server before you can test the applications. Then, start the Administrator’s Console and start the ITSOWSAD application server if it is not running.

To have the HTTP Server pick up the new plug-in configuration, restart the IBM HTTP Server as well.

To test the applications, follow the instructions described in “Test the applications with AEs” on page 441. However, for testing with the embedded Web server you have to use the port defined in Figure 13-4, instead of 9080. This port can be the same as 9080 depending on the application servers already defined on your node.

SOAPEARENabler tool

In the `WebSphere/AppServer/bin` directory you can find a command line tool, `soapearenabler.bat`, for enabling a set of SOAP services within an EAR file. We did not have to run this tool with our deployed EAR files, because they already contained the necessary SOAP servlets.

For example, you have to use this tool when you create a Web service with WebSphere Studio Version 4.0. Typically, with WebSphere Studio Version 4.0, you export the Web application in a WAR file. Then you assemble the WAR file into an EAR file using AAT. Then you run the `soapearenabler` tool, which adds a Web module containing the SOAP router servlets to the EAR file. If desired, the tool can add the SOAP admin Web application as well.

Summary

In this chapter we covered the deployment of Web services to WebSphere AEs and AE.

Quiz: To test your knowledge of the Web services deployment with Application Developer, try to answer the following five questions. The answers are provided in Appendix C, “Quiz answers” on page 559.

1. What WebSphere tool can be used to assemble an EAR file?
2. What command line deployment tool is provided with WebSphere AEs?
3. What is the use of the SOAP admin client?
4. Which application server, AEs or AE, can automatically update the Web server plug-in?
5. What tool can enable SOAP services in an EAR file?



Web services advanced topics

In this chapter, the following topics are discussed:

- ▶ Advanced SOAP programming
 - Custom mappings, including a practical example
 - Message oriented communication style, including programming example
- ▶ Web services support in other IBM products:
 - DB2 XML Extender, DADX
- ▶ Advanced UDDI topics
 - Types of registries
 - Programmatic access to UDDI registry using UDDI4J and SOAP
- ▶ Moving up the Web services protocol stack
 - WSFL

Advanced SOAP programming

We discuss the following topics in this section:

- ▶ Programmatic deployment
- ▶ Encodings, default mappings for data types, and custom mappings
- ▶ Message oriented SOAP communication

Programmatic deployment

The code snippet of Figure 14-1 shows how Web services can be deployed to Apache SOAP using the SOAP API.

```
Document doc;
DeploymentDescriptor deploymentDescriptor;
DocumentBuilder xdb = XMLParserUtils.getXMLDocBuilder();

// Read and parse the deployment descriptor document
doc = xdb.parse(new InputSource(xmlFile));

// Get the deployment descriptor
deploymentDescriptor =
    DeploymentDescriptor.fromXML(doc.getDocumentElement());

// Get client interface to communicate with the SOAP server
ServiceManagerClient smc =
    new ServiceManagerClient(routerURL);

// Deploy service using the specified deployment descriptor
smc.deploy(deploymentDescriptor);
```

Figure 14-1 Programmatic SOAP deployment

The ServiceManagerClient class exposes the required API functionality. It provides a deploy method that receives an XML deployment descriptor argument.

Encodings and type mapping alternatives

This section is structured in the following way:

- ▶ Overview on encodings and mappings

- ▶ Using the Apache SOAP BeanSerializer to map a JavaBean
- ▶ Implementing a simple custom mapping
- ▶ Discussion on how to handle method signatures containing data types not supported under a single encoding

Overview of encodings and mappings

Refer to “An introduction to SOAP” on page 261, for an introduction to the terms *encoding* and *mapping*.

In a nutshell, an encoding is merely a globally unique name for a set of mappings, consisting of (de)serializers for pairs of Java/XML data types. Mappings are either predefined defaults provided by the SOAP runtime or custom developed (Figure 14-2).

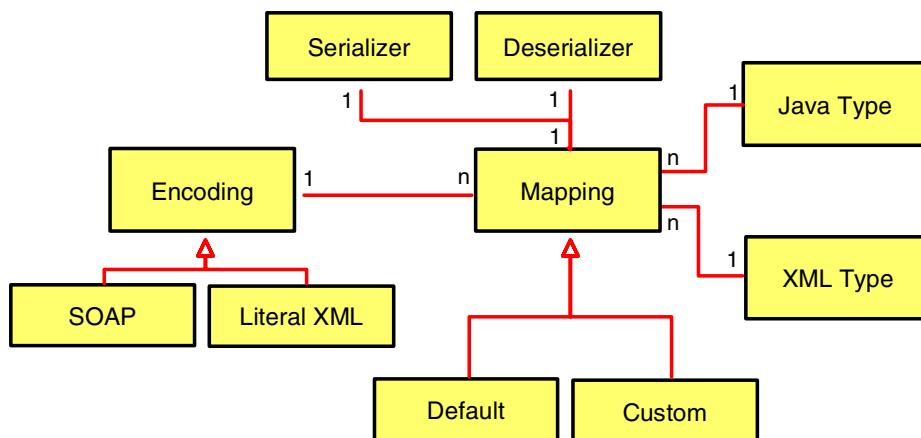


Figure 14-2 Relationship between encodings and mappings

Table 14-1 lists the defined encodings.

Table 14-1 Available encodings

Name	Defined in	Comments
SOAP	W3C SOAP 1.1 specification	http://schemas.xmlsoap.org/soap/encoding/ Supports the data types from the SOAP data model.
Literal XML	Apache SOAP 2.2	http://xml.apache.org/xml-soap/1literalxml Supports XML elements.
XMI	Apache SOAP 2.2	Allows to encode arbitrary objects.

The XMI encoding is not supported by Application Developer, and therefore not discussed any further here. The SOAP and literal XML encodings support the type mappings listed in Table 14-2.

Table 14-2 Types and mappings for SOAP and Literal XML encoding

Type	Encoding	Mapping support
Java primitive types such as int, float, and so forth, and their wrappers (<code>java.lang.Integer</code> , <code>java.lang.Float</code> and so forth)	SOAP	Default, custom
<code>org.w3c.dom.Element</code>	Literal XML	Default
<code>org.w3c.dom.Element</code>	SOAP	Custom (see note)
<code>java.lang.String</code>	SOAP	Default, custom
<code>java.util.Date</code> <code>java.util.GregorianCalendar</code>	SOAP	Default, custom
<code>Java arrays</code> <code>java.util.Vector</code> <code>java.util.Hashtable</code> <code>java.util.Map</code>	SOAP	Default, custom
<code>java.math.BigDecimal</code>	SOAP	Default, custom
<code>javax.mail.internet.MimeBodyPart</code> <code>java.io.InputStream</code> <code>javax.activation.DataSource</code> <code>javax.activation.DataHandler</code> <code>org.apache.soap.util.xml.QName</code> <code>org.apache.soap.rpc.Parameter</code>	SOAP	Default, custom
<code>java.lang.Object</code> (a deserializer for null objects only)	SOAP	Default, custom
other	any	Custom

Each mapping carries several pieces of information:

- ▶ An unique name describing the encoding style (URI format)
- ▶ A qualified name (QName) for the XML element
- ▶ The Java class to be encoded from/decoded to
- ▶ The name of the Java class to act as the serializer (custom mapping only)
- ▶ The name of the Java class to act as the deserializer (custom mapping only)

The Java classes acting as serializers/deserializers must implement the interfaces `org.apache.soap.util.xml.Serializer`, defining a `marshall` method, as well as the interface `org.apache.soap.util.xml.Deserializer`, defining an `unmarshall` method.

Any SOAP runtime environment holds a table of such mapping entries, the `SOAPMappingRegistry`. The mappings are looked up by the SOAP runtime when (de)serializing request and response parameters.

At startup time, this table is pre-loaded with entries for all available default mappings. On the server side, additional entries can be configured in the deployment descriptor:

```
<isd:mappings>
    <isd:map encodingStyle="encoding-uri"
        xmlns:x="qname-namespace" qname="x:qname-element"
        javaType="java-type" java2XMLClassName="serializer"
        xm12JavaClassName="deserializer"/>
    ...
</isd:mappings>
```

On the client side, it is necessary to add custom mappings through the SOAP client API, as we will see in the next section.

Implementing a Web service with a JavaBean as parameter

In “Static Web services” on page 321, we saw how the Web service wizard GUI exposes these encodings and type mappings. The method signature we used there was quite simple, and we did not modify any of the default settings, either. We will do so now.

As a first step, let us investigate how the standard SOAP encoding deals with JavaBeans. As an example, we implement a new Web service `GetInvItem` returning a JavaBean holding detailed information about inventory items.

Note that we do not have to implement a custom mapping. SOAP 2.2. provides a bean serializer class. This class can marshall and unmarshall instances of any Java class conforming to the JavaBean specification.

We implement, describe, and test the Web service in the same way as described in “Static Web services” on page 321.

Prepare for development

Perform the following two steps in order to prepare for this exercise:

- ▶ Create a new Web project called `ITSOAdvancedWeb` and a corresponding EAR file `ITSOAdvancedEAR`. Set the context root of the Web application to `ITSOAdvanced`.
- ▶ Add the `ITSOAdvancedEAR` project to the `ITSOwsADWebSphere` server instance.
- ▶ Add the `WAS_XERCES` variable to the build path of the new Web project.
- ▶ Define a new package `itso.wsad.alma.custom`.

Next, we implement the Web service following the bottom up approach as defined in “Development strategies for provider and requestor” on page 255.

Implement the Web service

Figure 14-3 contains our service provider side implementation of `GetInvItem`. It defines a single method returning a dummy instance of the `InvItemBean` JavaBean class.

```
package itso.wsad.alma.custom;
import org.w3c.dom.Element;
public class GetInvItem {

    public InvItemBean getAsBean(String partNumber, long itemNumber) {
        System.out.println("In getAsBean of GetInvItem (default)");
        System.out.println("partNumber is " + partNumber);
        System.out.println("itemNumber is " + itemNumber);

        InvItemBean result = new InvItemBean();
        result.setCost(33.33F);
        result.setDescription("An item.");
        result.setImageURL("Part-"+partNumber+".gif");
        result.setItemNumber(itemNumber);
        result.setLocation("Here.");
        result.setPartNumber(partNumber);
        result.setQuantity(66);
        result.setShelf("S1");
        result.setWeight(999.9);
        result.setName("Magic part");
        return result;
    }
}
```

Figure 14-3 Web service implementation with JavaBean parameter

- ▶ Create this class in the `itso.wsad.alma.custom` package (import from `sampcode\wsenhanced\AdvancedSAOP`).
- ▶ Copy it as `GetInvItem2` in the same package.

This allows us to run the Web service wizard multiple times without loosing any generated code. Later on we can compare the old and new wizard output when working with custom mappings.

The `InvItemBean` instantiated as Web service result is shown in Figure 14-4.

```
package itso.wsad.alma.custom;
public class InvItemBean
{
    long itemNumber;
    int quantity;
    float cost;
    String shelf;
    String location;
    String partNumber;
    String name;
    String description;
    double weight;
    String imageURL;

    public InvItemBean() { }

    public long getItemNumber() {
        return itemNumber;
    }
    public void setItemNumber(long itemNumber) {
        this.itemNumber = itemNumber;
    }
    // other public getters and setters skipped
}
```

Figure 14-4 InventoryItem JavaBean

- ▶ Create this class in the `itso.wsad.alma.custom` package (import from `sampcode\wsenhanced\AdvancedSAOP`).
- Make sure you implement all getters and setters; otherwise the Apache bean serializer will fail at runtime. To do so, you can select each data member in the Outline view and select *Generate Getter and Setter*.

Run the Web service wizard

Next, we use the GetInvItem Java class to create a Web service, following the bottom up strategy.

- ▶ Select GetInvItem.java in the Navigator view and select *New -> Other -> Web Services -> Web Service* and then *Next* to open the wizard.
- ▶ Create the Web service in the same way as described in “Using the Web service wizard” on page 336.
 - Generate a proxy and a sample
 - Set URI to urn:GetInvItem
 - Shorten ISD file name to: webApplication/WEB-INF/isd/GetInvItem.isd
 - Set the proxy name to: itso.wsad.alma.custom.proxy.GetInvItemProxy
 - Accept all other defaults and click *Finish*.

Important: The Web service wizard for generating a Web service from a JavaBean tries to publish to the defined WebSphere instance even if you do not choose to launch the sample. In this process it tries to start the server instance. If the ITSOAdvancedEAR project is not associated with the running server instance, the wizard will fail. Make sure the project is associated with the running server instance or stop all server instances.

Inspect the deployment descriptor

The wizard has defined a *mapping element* in the ISD file and in dds.xml (Figure 14-5).

```
<isd:service id="urn:GetInvItem"
  xmlns:isd="http://xml.apache.org/xml-soap/deployment">
  <isd:provider type="java" scope="Application" methods="getAsBean">
    <isd:java class="itso.wsad.alma.custom.GetInvItem" static="false"/>
  </isd:provider>
  <isd:mappings>
    <isd:map encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      xmlns:x="http://www.getinvitem.com/schemas/GetInvItemRemoteInterface"
      qname="x:itso.wsad.alma.custom.InvItemBean"
      javaType="itso.wsad.alma.custom.InvItemBean"
      java2XMLClassName="org.apache.soap.encoding.soapenc.BeanSerializer"
      xml2JavaClassName="org.apache.soap.encoding.soapenc.BeanSerializer"/>
  </isd:mappings>
</isd:service>
```

Figure 14-5 Type mapping information in the deployment descriptor

Investigate the SOAP client proxy

Lets take a look at the generated client side proxy. It populates its SOAPMappingRegistry (smr) with additional entries (Figure 14-6):

```
.....
public class GetInvItemProxy
{
    .....
    private String stringURL =
        "http://localhost:8081/ITSOAdvanced/servlet/rpcrouter";
    .....
    {
        org.apache.soap.encoding.soapenc.BeanSerializer ser_4 = new
            org.apache.soap.encoding.soapenc.BeanSerializer();
        org.apache.soap.encoding.soapenc.BeanSerializer deSer_4 = new
            org.apache.soap.encoding.soapenc.BeanSerializer();
        smr.mapTypes("http://schemas.xmlsoap.org/soap/encoding/",
            new QName
                ("http://www.getinvitem.com/schemas/GetInvItemRemoteInterface",
                    "itso.wsad.alma.custom.InvItemBean"),
            itso.wsad.alma.custom.InvItemBean.class, ser_4, deSer_4);
    }
}
```

Figure 14-6 Populating the SOAPMapping registry in the client proxy

- ▶ The mapping information in the client side registry matches with the server side deployment descriptor entry in Figure 14-5. In both cases, encoding, qualified XML type name, and Java type name are mapped to pairs of (de)serializer classes. The serializer and deserializer class and instance can be (but do not have to be) identical.
- ▶ Make sure to update the port information to the port your TCP/IP tunnel/monitor listens to so that we can investigate the exchanged messages. Refer to “Using the TCP/IP Monitoring Server to view message contents” on page 358 for instructions how to set up the monitor server instance, which defaults to port 8081, or use the standalone version of the monitor that comes with Apache SOAP (port 4040).

Test the bean serialization and deserialization

To test the serialization and deserialization, perform these steps:

- ▶ Start the TCP/IP Monitoring Server.
- ▶ Start the generated test client by selecting the TestClient.jsp and *Run on Server*.
- ▶ Invoke the `getAsBean` method with a part number of M100000003 and an inventory item number of 21000003.
- ▶ The SOAP runtime delivers the input parameters to the service provider and returns a meaningful bean result (Figure 14-7).

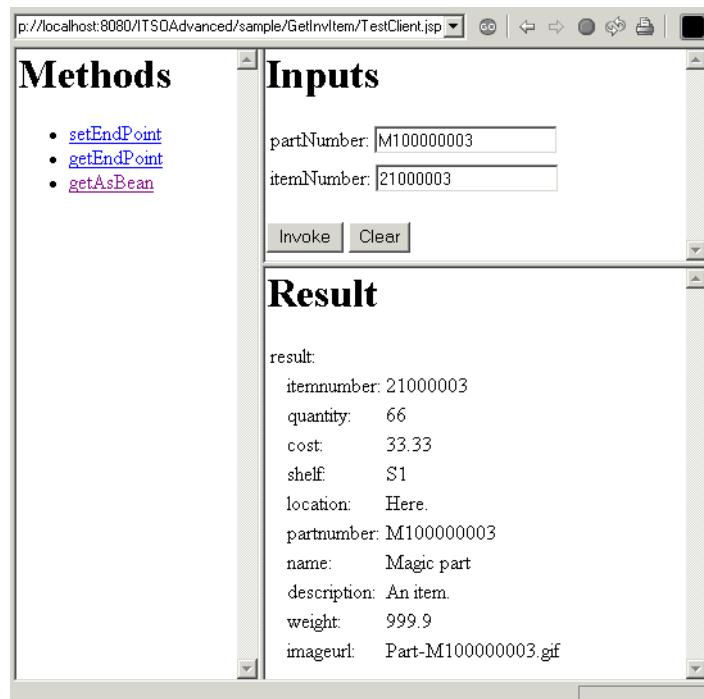


Figure 14-7 GetInvItem Web service output

Use the TCP/IP Monitor view to inspect the exchanged SOAP messages. You can see the XML representation of the JavaBean in the response message (Figure 14-8).

TCP/IP Monitor

```

Request: localhost:8081
Size: 666 bytes
POST /ITSOAdvanced/servlet/ipcrouter HTTP/1.0
Host: localhost:8080
Content-Type: text/xml; charset=utf-8
Content-Length: 519
SOAPAction: ""

<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Body>
<ns1:getAsBean xmlns:ns1="urn:GetInvItem" SOAP-ENV:encoding="utf-8">
<partNumber xsi:type="xsd:string">M100000003</partNumber>
<itemNumber xsi:type="xsd:long">21000003</itemNumber>
</ns1:getAsBean>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>

XML result

```

```

Response: localhost:8080
Size: 1386 bytes
HTTP/1.1 200 OK
Server: WebSphere Application Server/4.0
Content-Type: text/xml; charset=utf-8
Set-Cookie: JSESSIONID=00002UKG1QZIEPPS1J3L
Cache-Control: no-cache="set-cookie.set-cookie2"
Expires: Thu, 01 Dec 1994 16:00:00 GMT
Content-Length: 1071
Content-Language: en
Connection: close

<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Body>
<ns1:getAsBeanResponse xmlns:ns1="urn:GetInvItem" SOAP-ENV:encoding="utf-8">
<return xmlns:ns2="http://www.getinvitem.com/schema">
<name xsi:type="xsd:string">Magic part</name>
<cost xsi:type="xsd:float">33.33</cost>
<quantity xsi:type="xsd:int">66</quantity>
<imageURL xsi:type="xsd:string">Part-M100000003.gif</imageURL>
<weight xsi:type="xsd:double">999.3</weight>
<itemNumber xsi:type="xsd:long">21000003</itemNumber>
<shelf xsi:type="xsd:string">S1</shelf>
<location xsi:type="xsd:string">Here.</location>
<description xsi:type="xsd:string">An item.</description>
<partNumber xsi:type="xsd:string">M100000003</partNumber>
</return>
</ns1:getAsBeanResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>

```

Figure 14-8 SOAP messages from testing the GetInvItem Web service

Note: In “Creating a Web service from a session EJB” on page 374 we described a Web service that returns a vector or array of JavaBeans. The mapping is very similar to the example described here.

Complete the Almaden Autos Web application

As an optional step, you can now complete the Web application of Almaden Autos. Use this new Web service to retrieve the detail information on parts and inventory items from the manufacturer. Such a solution is very similar to the application we developed in “Static Web services” on page 321, therefore, we do not go into details here.

Create a custom mapping

If a method signature contains data types not supported by the SOAP runtime, you have to implement and register *custom mappings*.

As a simple example, we write our own String serializer to demonstrate the technique.

Note: Apache SOAP does provide string (de)serialization capabilities; the string deserializer in our version shows an interesting behavior. However, when XML reserved characters such as <Hi>You & me</Hi> occur in the string, their XML wire format <Hi>You & me</Hi> is passed to the application. You can verify this by entering a string containing XML reserved characters to the part number parameter when testing the GetInvItem Web service. If this behavior is not desired, a custom string serializer can be a useful tool.

We have to implement two interfaces, each defining one method. This is the org.apache.soap.util.xml.Serializer interface:

```
public void marshall(
    java.lang.String inScopeEncStyle,
    java.lang.Class javaType, java.lang.Object src,
    java.lang.Object context, java.io.Writer sink,
    org.apache.soap.util.xml.NSStack nsStack,
    org.apache.soap.util.xml.XMLJavaMappingRegistry xjmr,
    org.apache.soap.rpc.SOAPContext ctx);
```

The org.apache.soap.util.xml.Deserializer interface is:

```
public org.apache.soap.util.Bean unmarshall(
    java.lang.String inScopeEncStyle,
    org.apache.soap.util.xml.QName elementType,
    org.w3c.dom.Node src,
    org.apache.soap.util.xml.XMLJavaMappingRegistry xjmr,
    org.apache.soap.rpc.SOAPContext ctx)
        throws java.lang.IllegalArgumentException;
```

These functions are called by the SOAP layer both on the client and the server side when serializing/deserializing request and response parameters.

To run the Web service wizard with a custom mapper, we have to make the mapper available first.

Implement the marshall and unmarshall methods

Define the StringSerializer class in the itso.wsad.alma.custom.mappings package (import from sampcode\wSENHANCED\AdvancedSOAP). As we have already seen, two methods have to be implemented:

- marshall** Performs the Java to XML conversion (*serialization*). A serializer receives a Java object and provides an XML representation for it. The XML representation comprises the accessor tag name as defined by the SOAP RPC style and the value itself.
- unmarshall** Implements the XML to Java conversion (*deserialization*). A deserializer receives the XML element as it appears in the SOAP message body, and instantiates a Java object from it.

Figure 14-9 contains the class definition of the StringSerializer and the marshall method.

```
package itso.wsad.alma.custom.mappings;

import org.apache.soap.util.xml.Deserializer;
import org.apache.soap.util.xml.Serializer;

public class StringSerializer implements Serializer, Deserializer {

    public void marshall(java.lang.String inScopeEncStyle,
                         java.lang.Class javaType, java.lang.Object src,
                         java.lang.Object context, java.io.Writer sink,
                         org.apache.soap.util.xml.NSStack nsStack,
                         org.apache.soap.util.xml.XMLJavaMappingRegistry xjmr,
                         org.apache.soap.rpc.SOAPContext ctx) {
        nsStack.pushScope();

        // ** Java to XML
        String tagname = context.toString();
        System.out.println("Marshalling: " + tagname + ": " + src.toString());

        try {
            sink.write("<" + tagname + " xsi:type='xsd:string'>" +
                      + src.toString() + "</" + tagname + ">");
        }
        catch(java.io.IOException e) {
            throw new java.lang.IllegalArgumentException("marshall:IOException");
        }
        finally {
            nsStack.popScope();
        }
    }
}
```

Figure 14-9 StringSerializer: marshall

The *in* parameter context contains the accessor name, *src* is the object to be serialized. Type information is available as well.

Figure 14-10 shows the corresponding unmarshal method. It receives the XML representation of the string as a org.w3c.dom.Node instance, extracts its only child element, a text node containing the value for the Java string instance, and instantiates an org.apache.soap.util.Bean object from it. Such a bean is a wrapper for a Java object and its type, which makes it possible to send a typed null pointer.

```
public org.apache.soap.util.Bean unmarshal(
    java.lang.String inScopeEncStyle,
    org.apache.soap.util.xml.QName elementType,
    org.w3c.dom.Node src,
    org.apache.soap.util.xml.XMLJavaMappingRegistry xjmr,
    org.apache.soap.rpc.SOAPContext ctx)
throws java.lang.IllegalArgumentException {
    org.apache.soap.util.Bean result = null;
    org.w3c.dom.Element element = null;
    org.w3c.dom.Node textNode = null;
    try {
        System.out.print("Unmarshalling string element " + src.getNodeName());
        int nodeType = src.getNodeType();
        String nodeName = src.getNodeName();
        if(nodeType!=org.w3c.dom.Node.ELEMENT_NODE)
            throw new IllegalArgumentException("unmarshal:invalid node type");
        else
            element = (org.w3c.dom.Element) src;
        org.w3c.dom.NodeList nl = element.getChildNodes();
        if(nl.getLength()!=1)
            throw new IllegalArgumentException(
                "unmarshal: invalid child list length");
        else {
            textNode = nl.item(0);
            nodeType = textNode.getNodeType();
        }
        if(nodeType!=org.w3c.dom.Node.TEXT_NODE)
            throw new IllegalArgumentException(
                "unmarshal: invalid child node type");
        String myString = ((org.w3c.dom.Text) textNode).getNodeValue();
        result = new org.apache.soap.util.Bean(String.class, myString);
    }
    catch(IllegalArgumentException e) {
        e.printStackTrace(System.err);
        throw(e);
    }
    return result;
}
```

Figure 14-10 *StringSerializer: unmarshal*

Define custom mappings with the Web service wizard

Invoke the Web service wizard as in the previous example, but use the second Java class GetInvItem2 (which is a copy of GetInvItem) this time:

- ▶ Generate a proxy and a sample.
- ▶ Set URI to urn:GetInvItem2.
- ▶ Shorten ISD file name to: webApplication/WEB-INF/isd/GetInvItem2.isd.
- ▶ Stay with the default SOAP encoding. Select *Show server (Java to XML) type mappings*.
- ▶ Select the *java.lang.String, SOAP encoding* list entry (Figure 14-11).
- ▶ Click on *Edit and use a customized mapping*.
- ▶ Enter `itso.wsad.alma.custom.mappings.StringSerializer` both as *Serializer class* and *Deserializer class*.

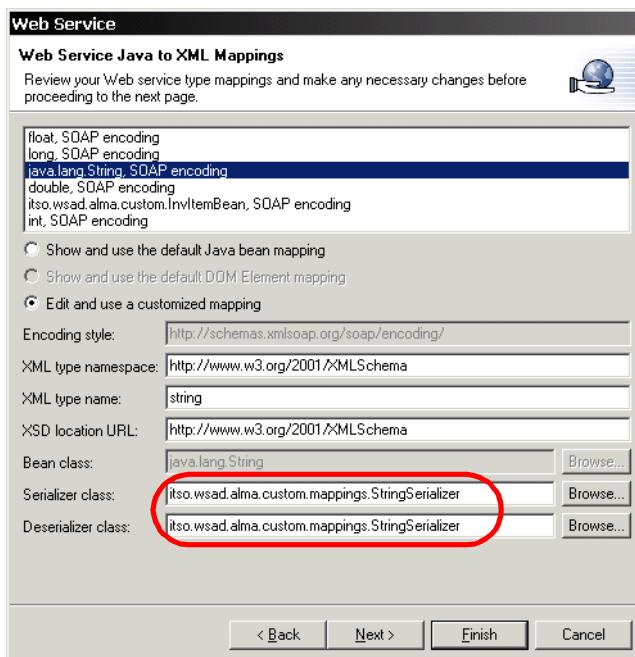


Figure 14-11 Server side mapping configuration in the Web service wizard

- ▶ Set the proxy name to: `itso.wsad.alma.custom.proxy.GetInvItem2Proxy` and select *Show mappings*. The custom mapping is predefined for *String*.
- ▶ The SOAP binding mapping configuration shows the serializer class for *String* (Figure 14-12). Then click *Finish*.

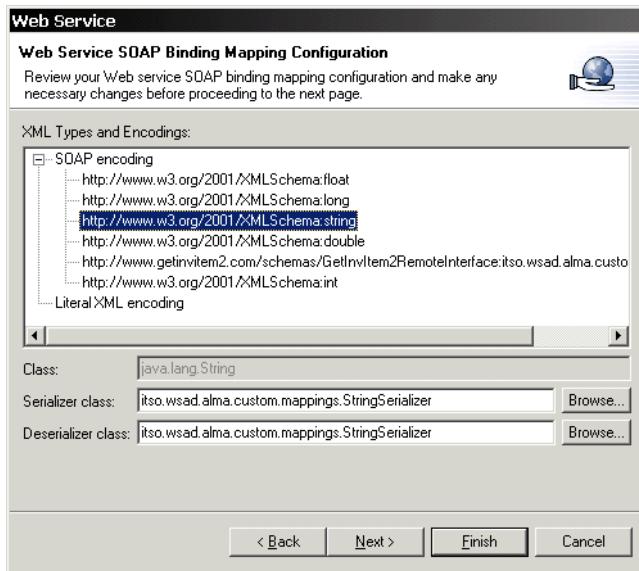


Figure 14-12 Client side mapping configuration in the Web service wizard

Take a look at the generated client proxy and the ISD file (Figure 14-13), which is also added to the dds.xml file.

```

<isd:service id="urn:GetInvItem2"
    xmlns:isd="http://xml.apache.org/xml-soap/deployment">
    <isd:provider type="java" scope="Application" methods="getAsBean">
        <isd:java class="itso.wsad.alma.custom.GetInvItem2" static="false"/>
    </isd:provider>
    <isd:mappings>
        <isd:map encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
            xmlns:x="http://www.w3.org/2001/XMLSchema" qname="x:string"
            javaType="java.lang.String"
            java2XMLClassName="itso.wsad.alma.custom.mappings.StringSerializer"
            xml2JavaClassName="itso.wsad.alma.custom.mappings.StringSerializer"/>
        <isd:map encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
            xmlns:x="http://www.getinvitem2.com/schemas/GetInvItem2RemoteInterface"
            qname="x:itso.wsad.alma.custom.InvItemBean"
            javaType="itso.wsad.alma.custom.InvItemBean"
            java2XMLClassName="org.apache.soap.encoding.soapenc.BeanSerializer"
            xml2JavaClassName="org.apache.soap.encoding.soapenc.BeanSerializer"/>
    </isd:mappings>
</isd:service>

```

Figure 14-13 ISD file with custom mappings defined

Test the custom mapping

As in the previous step, patch the client proxy to use port 8081 of the TCP/IP tunnel rather than 8080:

- ▶ Start the TCP/IP Monitoring Server.
- ▶ Select the TestClient.jsp page in the webApplication/sample/GetInvItem2 folder and select *Run on Server*.
- ▶ Invoke the getAsBean method, passing a part and an inventory number as before. The SOAP runtime delivers the in parameters to the service provider and return a meaningful bean result to the test client.
- ▶ Take a look at the TCP/IP Monitor view to inspect the SOAP messages.
- ▶ Standard output of the client and the server is displayed in the Application Developer console (Figure 14-14). Note that the marshalling that takes place before the SOAP call noted by the invocation of the rpcrouter. (Note that you can only see the Console when the TCP/IP server is not running.)

```
[tsp] = [2/9/02 21:40:03:265 PST] timestamp

[tsp] 507d010a SystemOut      U Marshalling: partNumber: M100000003
[tsp] 24fb010b WebGroup       I SRVE0091I: [Servlet LOG]: rpcrouter: init
[tsp] 24fb010b SystemOut      U Unmarshalling string element: partNumber
[tsp] 24fb010b SystemOut      U In getAsBean of GetInvItem (default)
[tsp] 24fb010b SystemOut      U partNumber is M100000003
[tsp] 24fb010b SystemOut      U itemNumber is 21000003
[tsp] 24fb010b SystemOut      U Marshalling: name: Magic part
[tsp] 24fb010b SystemOut      U Marshalling: imageURL: Part-M100000003.gif
[tsp] 24fb010b SystemOut      U Marshalling: shelf: S1
[tsp] 24fb010b SystemOut      U Marshalling: location: Here.
[tsp] 24fb010b SystemOut      U Marshalling: description: An item.
[tsp] 24fb010b SystemOut      U Marshalling: partNumber: M100000003
[tsp] 507d010a SystemOut      U Unmarshalling string element: name
[tsp] 507d010a SystemOut      U Unmarshalling string element: imageURL
[tsp] 507d010a SystemOut      U Unmarshalling string element: shelf
[tsp] 507d010a SystemOut      U Unmarshalling string element: location
[tsp] 507d010a SystemOut      U Unmarshalling string element: description
[tsp] 507d010a SystemOut      U Unmarshalling string element: partNumber
```

Figure 14-14 Console output from custom string serializer

Unlike the SOAP deserializer, our StringSerializer implementation returns the original string to the application, even if it contains XML reserved characters.

It is worth noting that our string serializer is not just used for the part number parameter; it is recursively invoked during the serialization of the InvItemBean as well.

Conclusion

In this section, we have defined a custom mapping for string values. The SOAP runtime no longer automatically converts strings into XML elements and vice versa; our own class has taken over this responsibility.

The `java.lang.String` class served as a simple example; custom mappings for more complex data types implement the same interface and follow the same design pattern.

In cases when you are starting to write your own custom deserializers, make sure to check the Apache SOAP source code before doing so. For debugging purposes, compare your message bodies with the ones created by already existing serializers.

Messages with XML element parts and other parameters

There is an interesting design gap between WSDL and SOAP; for example:

- ▶ In WSDL, you can specify only one encoding per operation message. As listed in Table 14-2 on page 450, none of the existing SOAP encodings support all data types, however.
- ▶ If the `in` parameter list of a method contains a combination of data types not supported by a single encoding (for example, a `org.w3c.dom.Element` and a `java.lang.String`) the SOAP proxy generated from the WSDL uses an unsupported encoding for at least one parameter, which causes runtime errors to occur.

A simple solution is to modify the generated proxy. You have to change the encoding on the parameter level for the data type reporting the encoding problem. Compare your settings with Table 14-2 on page 450. Usually any `org.w3c.dom.Element` parameters should use the literal XML and all others the SOAP encoding.

Tip: In the SOAP 2.2 API, an encoding *can* optionally be specified on the call level. If it is there, it is used to define the encoding style for the response. It serves as a default value for the parameter level as well. On the parameter level, an encoding parameter has to be passed to the constructor; this parameter can be set to `null`, however.

If Apache SOAP cannot find an encoding specification, either on the call, or on the parameter level, it uses the SOAP encoding by default.

For more information, consult the FAQ list that comes with SOAP 2.2.

As a more complex solution to this problem, you can also define a custom mapping for `org.w3c.dom.Element` and register it for the SOAP encoding. To do so, you would have to implement an `ElementSerializer` class similar to the `StringSerializer` from the previous section, supporting marshalling and unmarshalling of `org.w3c.dom.element` objects.

As a further design alternative, you can also consider to provide custom mappings for the non XML data types in the method and register them for the Literal XML encoding.

As a final design alternative, you can also follow the meet in the middle development approach and modify the Java method signature, moving the data carried by the non XML parameters into the XML element.

Summary

Mappings implement the deserialization functionality required to encode and decode data values under certain encodings. Table 14-3 and Table 14-4 summarize where default and custom mappings are available.

Table 14-3 Encoding and mapping styles (default mapping)

	Basic types	Complex types	XML element
SOAP encoding	Yes	Yes for most types	No
Literal XML	No	No	Yes

Table 14-4 Encoding and mapping styles (custom mapping)

	Basic types	Complex types	XML element
SOAP encoding	Yes	Yes	Yes (patches to generated files required)
Literal XML	No	No	Possible in theory (not needed, however)

Custom mappings can be defined whenever a mapping for a certain type is missing in the one of the encodings. We recommend to add mappings to the default SOAP encoding scheme. Do not try to define your own encoding schemes.

In general, you should not to implement custom mappings unless you cannot avoid it. This is due to the fact that the custom mapping couples the service requestor and provider quite tightly, because two additional software components have to be made available at the two ends of the communication channel.

These two components have interoperate with each other, which might introduce software distribution and versioning issues. Moreover, the tighter coupling compromises the openness and flexibility of your solution.

Custom mappings can be a solution, however, to the problem that methods containing both XML element parameters and other data types, are by default not supported by any of the available encodings. A simpler alternative is to patch the generated SOAP client, however.

Message-oriented communication

As described in “Web services overview and architecture” on page 241, SOAP does not only offer an RPC communication style; there is a *message* or *document oriented* interface as well. At the time of writing, most SOAP tutorials and programming samples focus on the RPC style, just as we did up to this point in the book.

A message-oriented model sometimes fits better to the problem domain. For instance, it is well suited under the following circumstances:

- ▶ Some message elements do not map well to programming language types.
- ▶ One way communication is desired, the service provider cannot return meaningful response data.
- ▶ Non XML data is supposed to be returned.
- ▶ Direct access to transport layer is required in the Web serve implementation.

Creating message-oriented services in Apache SOAP is slightly more complex than creating a simple RPC-based service.

In a message-oriented service, the service implementation is responsible for processing the contents of the SOAP envelope. The service implementation must extract the information it requires to process the request from the envelope. The SOAP API offers the required functionality.

If the message oriented service participates in a request-response protocol using SOAP messages in both directions, then it is also responsible for generating the appropriate response envelope.

A message-oriented service does not have to return a SOAPEnvelope, but instead can return anything it wants. The service can even decide not to return anything (unless the underlying transport is bidirectional and expects some sort of response).

Solution outline for example

As an example, we will implement the following scenario:

- ▶ New content for the part database of Almaden Autos has to be uploaded periodically.
- ▶ The content catalog is provided by a third party content syndication agency. It is formatted as an XML document.
- ▶ Almaden Autos provisions the third party with a document-oriented Web service speeding up the upload process.

Figure 14-15 contains the class diagram for our message client.

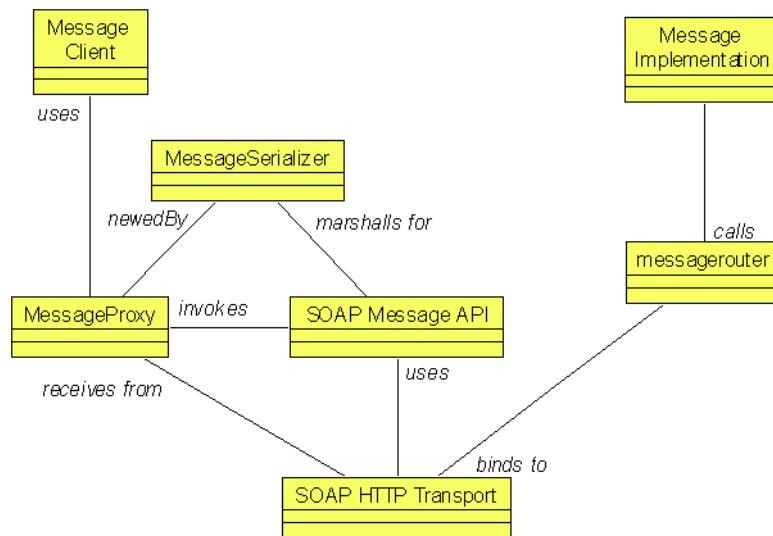


Figure 14-15 Class diagram for SOAP message communication

We will explain these classes and their interactions (Figure 14-16) as we develop the solution.

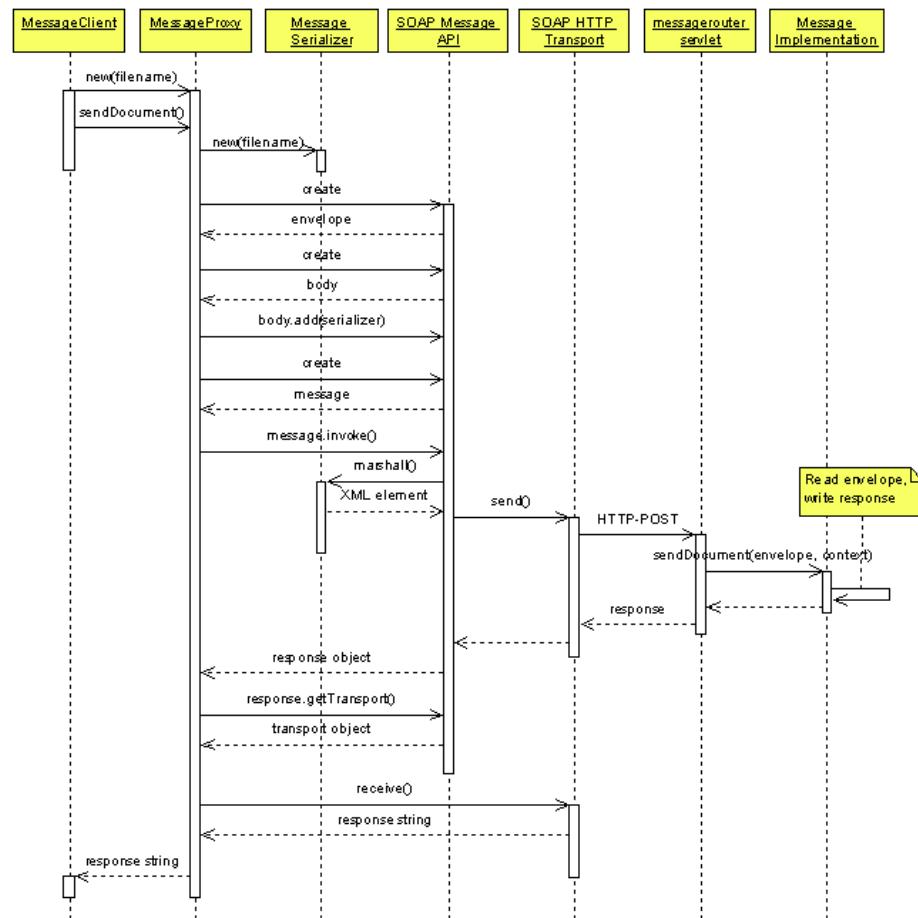


Figure 14-16 Sequence diagram for SOAP message communication

Prepare for development

We use the same projects as in the previous step, ITS0AdvancedWeb and ITS0AdvancedEAR. Create these projects now in case you skipped the last section, and set the context root to ITS0Advanced.

Define a new package `itso.wsad.alma.message` in the Web project.

A Web service enabled Web application has to be available. In the last step, we generated such a Web application. Now, run the Web service wizard and create a dummy RPC Web service in case you skipped that step.

The only API we use is Apache SOAP 2.2, which was already added to build path by the wizard. Also required is `mail.jar`. Add the `MAILJAR` variable for this library to the build path.

Implementing a message service on the server side

Because message oriented services have full control over the SOAP envelopes, any XML document can be passed as part of the envelope body. When the service receives the SOAP envelope, it is free to extract the body, and process it in any way.

Unlike RPC service implementations, message service implementations must all conform to the same interface:

```
void name(Envelope requestEnvelope, SOAPContext requestContext,  
          SOAPContext responseContext);
```

<code>name</code>	Name of the method
<code>requestEnvelope</code>	Envelope containing the incoming message
<code>requestContext</code>	SOAP context for the incoming message
<code>responseContext</code>	SOAP context which may be used for a response if one is needed

Import the code

Import five classes from `sampcode\wsenhanced\AdvancedSOAP` into the `itso.wsad.alma.message` package:

```
MessageService  
MessageProxy  
MessageSerializer  
MsgClient  
XMLUtils
```

We will describe these classes in sequence.

Service implementation

Figure 14-17 shows a simple service implementation in the `MessageService` class.

```
package itso.wsad.alma.message;

import java.util.*;
import org.w3c.dom.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
import org.apache.soap.messaging.*;

public class MessageService {
    public void sendDocument(org.apache.soap.Envelope requestEnvelope,
                           org.apache.soap.rpc.SOAPContext requestContext,
                           org.apache.soap.rpc.SOAPContext responseContext) {
        try {
            System.out.println("Received a sendDocument message.");
            Body b = requestEnvelope.getBody();
            Vector v = b.getBodyEntries();

            for (int i=0;i<v.size();i++) {
                Element elem = (Element) v.elementAt(i);
                System.out.println( "Body element with index " + i +
                                   " is " + elem.getNodeName());
                String inElem = XMLUtils.domWriter(elem, new StringBuffer());
                System.out.println(inElem);
            }
            responseContext.setRootPart("Message received.", "text/plain");
        }
        catch(Exception e ) {
            e.printStackTrace(System.err);
        }
    }
}
```

Figure 14-17 `MessageService` class

In this case, the service implementation does not do much with the received body element(s). It merely prints them to the standard output. To do so, it uses a helper class `XMLUtils`, which we will implement later.

A response is sent through the `responseContext`. Note that it is not a SOAP envelope, not even XML, but a simple string. Any MIME type can be sent; attachments are possible as well.

Note: We could have thrown a SOAPException to indicate any server side processing error. If we do so, the messengerouter servlet returns a SOAP fault message to the client. We do not require this functionality for our small example.

Deployment

In Application Developer, message-oriented services are deployed directly by editing the deployment descriptor file; there is no wizard support.

The easiest way for us to manually configure our service is to copy and paste an existing entry, and then change the URN to urn:Message, the method name to sendDocument, and the class name to itso.wsad.alma.message.MessageService.

We have to add the optional parameter type="message", instructing the SOAP server to assign this service to the messengerouter servlet rather than the rpcrouter. You can delete any existing <isd:mappings> entries for this service because the SOAP message layer does not deal with data types.

Compare your deployment descriptor (dds.xml) with Figure 14-18.

```
<root>
<isd:service .....
    .... existing entries
</isd:service>
<isd:service xmlns:isd="http://xml.apache.org/xml-soap/deployment"
              id="urn:Message" type="message" checkMustUnderstands="false">
    <isd:provider type="java" scope="Request" methods="sendDocument">
        <isd:java class="itso.wsad.alma.message.MessageService" static="false"/>
    </isd:provider>
</isd:service>
</root>
```

Figure 14-18 Deployment descriptor with message service

Accessing the message service from the Web service client

Writing clients to access SOAP message services requires you to interact with a lower-level set of SOAP APIs than you would otherwise use, if you were writing an RPC based client. Message-oriented services provide you with a finer grain of control over what is actually transmitted over SOAP.

The basic steps for creating a client that interacts with a SOAP message-oriented service are as follows:

- ▶ First obtain the interface description of the SOAP service, so that you know what the format of the SOAP message is (what headers it should have, what the body should look like, and so forth).
- ▶ Construct an org.apache.soap.Envelope containing the information required by the SOAP service.
- ▶ Create an org.apache.soap.messaging.Message object as a local representative of the message exchange.
- ▶ Invoke the send method on this message object. Pass the URL of the port that provides the service, the SOAP action URI for the HTTP header, and the envelope.
- ▶ If your service returns data and the transport supports two-way interaction, you have to retrieve the SOAPTransport object from the message object by using the getSOAPTransport method. You can then invoke the receive method on this transport object in order to access the returned data.

The Application Developer wizards currently do not support the generation of message style proxies; we cannot use or patch a generated RPC proxy either, because we have to code against a completely different API.

Therefore, we write our own message layer proxy, MessageProxy, which encapsulates access information and message invocation (Figure 14-19).

```
package itso.wsad.alma.message;
import java.io.*;
import java.net.*;
import java.util.*;
import org.w3c.dom.*;
import org.apache.soap.*;
import org.apache.soap.messaging.*;
import org.apache.soap.transport.*;
import org.apache.soap.util.*;

public class MessageProxy
{
    private URL url = null;
    private String soapActionURI = "urn:Message";

    public MessageProxy() throws MalformedURLException { }
    public synchronized void setEndPoint(URL url) {
        this.url = url;
    }
    // code for sendDocument method goes here
}
```

Figure 14-19 MessageProxy class

The sendDocument method implements the steps described above (Figure 14-20).

```
public synchronized String sendDocument(java.lang.String filename)
    throws Exception {
    if(url == null) {
        throw new SOAPException(ConstantsFAULT_CODE_CLIENT,
            "A URL must be specified via " + "ExchangeProxy.setEndPoint(URL).");
    }

    // ** instantiate SOAP message and its elements
    Envelope env = new Envelope();
    Body body = new Body();
    Vector bodyElementVector = new Vector();
    Message msg = new Message();

    // ** create body element and add it to body
    MessageSerializer ms = new MessageSerializer(filename);
    Bean bean = new Bean(MessageSerializer.class, ms);
    bodyElementVector.add(bean);
    body.setBodyEntries(bodyElementVector);
    env.setBody(body);

    // ** send message
    System.out.println("Proxy: sending message ...");
    msg.send(this.url, soapActionURI, env);
    System.out.println("Proxy: done.");

    // ** receive response
    System.out.println("Proxy: receiving response ...");
    SOAPTransport t = msg.getSOAPTransport();
    BufferedReader br = t.receive();
    System.out.println("Proxy: done.");

    // ** create response string
    String nextLine = null;
    StringBuffer result = new StringBuffer();
    while ((nextLine = br.readLine()) != null) {
        result.append(nextLine);
    }

    return result.toString();
}
```

Figure 14-20 SOAP message invocation in sendDocument

Compare this code with one of the generated RPC proxies from the previous stages. Neither the Call, nor the Parameter, nor the SOAPMappingRegistry classes are used.

All elements in the message body have to implement the Serializer interface, because they are responsible for the marshalling of the contained data. We, therefore, need the new class MessageSerializer. The message proxy creates an instance of this class, wrapped in a SOAP bean instance, and adds it to the message body (Figure 14-21).

```
package itso.wsad.alma.message;

import java.io.*;
import org.apache.soap.rpc.*;
import org.apache.soap.util.xml.*;

public class MessageSerializer implements Serializer {

    StringBuffer message = null;

    public MessageSerializer(String filename) {
        try {
            BufferedReader br = new BufferedReader(new FileReader(filename));
            String nextLine = br.readLine(); // skip first line of document
            message = new StringBuffer();
            while ((nextLine = br.readLine()) != null) {
                message.append(nextLine);
            }
        } catch (Exception e) {
            e.printStackTrace(System.err);
            message.append("<messageContent></messageContent>");
        }
    }

    public void marshall(String inScopeEncStyle, Class javaType, Object src,
                         Object context, Writer sink, NSStack nsStack,
                         XMLJavaMappingRegistry xjmr, SOAPContext ctx)
        throws IllegalArgumentException, IOException {
        nsStack.pushScope();
        sink.write("<ns1:sendDocument xmlns:ns1=\"urn:Message\>");
        sink.write(message.toString());
        sink.write("</ns1:sendDocument>");
        nsStack.popScope();
    }
}
```

Figure 14-21 MessageSerializer class

The MessageSerializer reads an XML document from the file system and puts it into the message body. It is invoked by the SOAP runtime.

We introduced the Serializer interface in “Encodings and type mapping alternatives” on page 448.

Figure 14-22 shows the main program, Message Client.

```
package itso.wsad.alma.message;

import java.net.*;

public class Message Client {

    public static void main(String[] args) {
        try {
            String response = null;
            String filename = "catalog.xml";
            MessageProxy mp = new MessageProxy();
            mp.setEndPoint(new
                URL("http://localhost:8080/ITSOAdvanced/servlet/messagerouter"));
            response = mp.sendDocument(filename);
            System.out.println("Client: response to sendDocument is: " +
                response);
        }
        catch( MalformedURLException e ) {
            e.printStackTrace(System.err);
        }
        catch( Exception e ) {
            e.printStackTrace(System.err);
        }
    }
}
```

Figure 14-22 MessageClient test class

Note that we could use the TCP/IP Monitor port (8081) to inspect the exchanged messages (try that with a binary protocol such as IIOP). Use port 8080 if you are not interested in the trace messages.

The XMLUtils helper class that the message service uses to display the incoming XML element is shown in Figure 14-23. Note that the helper does not support XML entity references and processing instructions.

```

package itso.wsad.alma.message;

import org.w3c.dom.*;

public class XMLUtils {
    public static java.lang.String domWriter(Node node,
        java.lang.StringBuffer buffer) {
        if ( node == null )
            return "";

        int type = node.getNodeType();
        switch ( type ) {
        case Node.DOCUMENT_NODE: {
            buffer.append("<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n");
            domWriter(((org.w3c.dom.Document)node).getDocumentElement(),buffer);
            break;
        }
        case Node.ELEMENT_NODE: {
            buffer.append("<" + node.getnodeName());
            NamedNodeMap attrs = node.getAttributes();
            int length = (attrs != null) ? attrs.getLength() : 0;
            for ( int i = 0; i < length; i++ ) {
                Attr attr = (org.w3c.dom.Attr) attrs.item(i);
                buffer.append(" " + attr.getNodeName() + "=" );
                buffer.append(attr.getNodeValue() + "=");
            }
            buffer.append(">");
            NodeList children = node.getChildNodes();
            if ( children != null ) {
                int len = children.getLength();
                for ( int i = 0; i < len; i++ ) {
                    if(((Node)children.item(i)).getNodeType() == Node.ELEMENT_NODE)
                        buffer.append("\n");
                    domWriter(children.item(i),buffer);
                }
            }
            buffer.append("</" + node.getnodeName() + ">");
            break;
        }
        case Node.CDATA_SECTION_NODE:
        case Node.TEXT_NODE:{ 
            buffer.append(node.getNodeValue());
            break;
        }
        }
        return buffer.toString();
    }
}

```

Figure 14-23 XMLUtils class with domWriter method

Test the message service

To test the message service protocol:

- ▶ Add any XML file to the Application Developer install directory and name it catalog.xml, which is the filename used in the MessageClient.

Restriction: The file, catalog.xml in this example, must reside in the install directory of the Application Developer. Copy it to there or you receive a FileNotFoundException.

- ▶ Start the server.
- ▶ Start the SOAP admin GUI to verify that the message service is deployed.
- ▶ Run the client as a Java application. (Select the class in the Java perspective and click the *Run* icon.)
- ▶ Take a look at the console messages both on the server and the client side:

- The client displays:

```
Proxy: sending message ...
Proxy: done.
Proxy: receiving response ...
Proxy: done.
Client: response to sendDocument is: Message received.
```

- The server side output is:

```
Received a sendDocument message.
Body element with index 0 is ns1:sendDocument
<ns1:sendDocument xmlns:ns1="urn:Message">
<InquirePartsResult xmlns="http://...../InquireResults">
<Part>
<ItemNumber>2100003</ItemNumber>
...
</Part>
</InquirePartsResult>
</ns1.sendDocument>
```

- ▶ Stop the server.
- ▶ Done!

Summary

In this section, we developed a simple message service and a client for it. We described how the SOAP message and RPC APIs differ.

A lower-level client API, which is slightly more complex than the RPC API, has to be used. The SOAP RPC layer itself is actually implemented against this API. On the server side, a different router servlet called `messagerouter` is used; the method signature for any service implementation is fixed. Both service requestor and provider have full control over the SOAP envelope (headers, body). The response can be anything, or nothing at all.

We expect additional tools and convenience APIs to come out soon. For example, the Web services toolkit (WSTK) 2.4 contains such an API defining a `ServiceProxy`, a `ServiceRequest`, and a `ServiceResponse` class as well as a message demo. These classes are similar in function to our `MessageProxy`.

Check the Apache SOAP 2.2 User's Guide and API documentation for more information. If you prefer do read the API source code, feel free to do so (Apache SOAP is open source, and available at <http://www.apache.org/soap>).

Additional Web service support

In this section we explore two additional wizards in Application Developer:

- ▶ Creating a Web service from a DADX file
- ▶ Generating a Web service from a URL

Creating a Web service from a DADX file

In this section, we provide a small example of how to create a Web service from a DADX file. We use the same database *ITSOWSAD*, which was used in the other chapters. You can follow the instructions in “Creating the *ITSOWSAD* sample database” on page 61, to create the database. Besides that, you do not have to go through the other chapters to make this example, although a basic understanding of what a Web service is, is required.

As discussed in “Web service DADX group configuration wizard” on page 312, DADX is an extension of the XML extender document access definition (DAD) file. A DADX document specifies how to create a Web service using a set of operations that are defined by SQL statements or DAD files. DAD files are files that describe the mapping between XML documents elements and DB2 database columns. They are used by the DB2 XML extender that supports XML-based operations that store or retrieve XML documents from a DB2 database. In this section, we show how you how to build a DADX file based on SQL statements.

To complete this section JDBC 2.0 must be enabled.

This example illustrates:

- ▶ Creating an SQL statement using the SQL wizard
- ▶ Creating a DADX group using the DADX group configuration wizard
- ▶ Creating a DADX file from the SQL statement
- ▶ Running the Web service wizard from the DADX file
- ▶ Testing the Web service using the generated sample application

Prepare a project

Create a *ITS0DadxWeb* Web project related to a new EAR project called *ITS0DadxEAR*. Set the context root to *ITS0Dadx*. Add the *ITS0DadxEAR* project to the *ITSOWSADWebSphere* server configuration.

Create a folder *ITS0WSADDB* in the *ITS0DadxWeb* project to contain the database information and the SQL statement.

Create the SQL statement

In the data perspective, *Data view*:

- ▶ Select *File -> New -> Other -> Data -> SQL Statement*, and click *Next*.
- ▶ In the SQL Statement wizard select *Be guided through creating an SQL Statement* and *Connect to a database and import a new database model*. Specify *selectParts* for the *SQL Statement Name* (Figure 14-24).

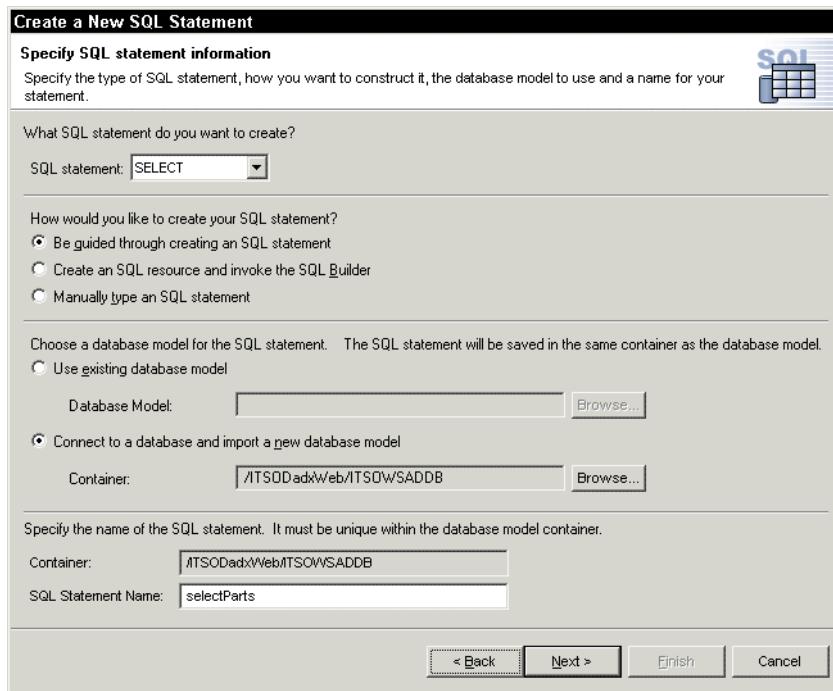


Figure 14-24 Create an SQL statement

- ▶ Click the *Browse* button next to *Connect to a database and import a new database model*. Select the *ITSOWSADDB* folder in the *ITSODadxWeb* project as the target folder, click *OK*.
- ▶ Click *Next*.
- ▶ Complete the *Database Connection* panel (Figure 14-25):
 - Specify *ITSODadxDB* for the *Connection Name*, *ITSOWSAD* for the *Database* and make sure to specify a valid *User ID* and *Password* for DB2 (or leave the fields empty).
 - Click *Filters*, enter *AAPARTS* (the only table we will use), and set the *Predicate* to *LIKE*. Click *OK*.

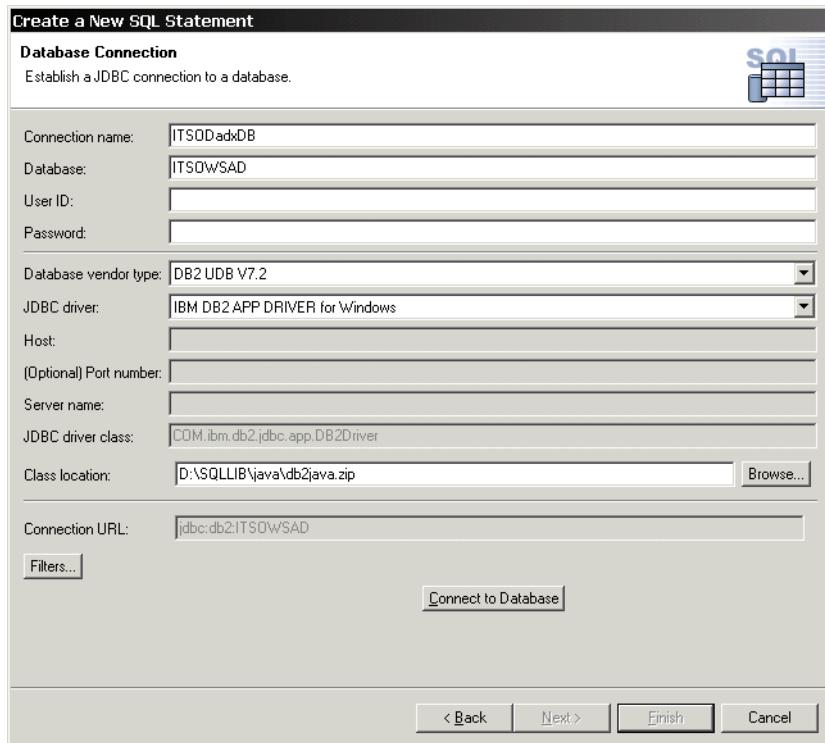


Figure 14-25 Database connection

- ▶ Click *Connect to Database*. This opens the Construct an SQL Statement wizard:
 - On the *Tables* tab, expand *ITSO* and *Tables*. Select the *AAPARTS* table, and click the arrow button to move the table to the *Selected tables* panel.
 - Switch to the *Columns* tab of the *Construct a SQL statement wizard*, expand *AAPARTS*. Select the *PARTNUMBER* and *NAME* columns and move them to the *Selected columns* list.
 - Click *Next* and the SQL statement is displayed:

```
SELECT ITSO.AAPARTS.PARTNUMBER, ITSO.AAPARTS.NAME
FROM ITSO.AAPARTS
```
 - Click *Execute* if you want to test the SQL Statement.
- ▶ Click *Finish* to end the wizard. You can find the generated information (connection, database, tables, SQL statement) in the *ITSOVSADD* folder in the Data and Navigator perspectives.

Create a DADX group

The Web service DADX group configuration wizard enables you to create a DADX group that is used to store DADX documents. A DADX group contains connection (JDBC and JNDI) and other information that is shared between DADX files. You can import DADX files into the DADX group, and then start the Web service wizard to create a Web service from a DADX file.

In the Web perspective:

- ▶ Select *New -> Other -> Web Services -> Web Service DADX Group Configuration*. Click *Next*.
- ▶ In the *Web Service DADX Group Configuration wizard*, select the *ITSOAdvancedWeb* project and click *Add group*.
- ▶ Specify *ITSOGroup* as the name for the DADX group. Click *OK*.
- ▶ Expand the *ITSOAdvancedWeb* project, select the *ITSOGroup* and *Group properties*.
- ▶ In the *Dadx Group Properties Page*, change the database name in the *DB URL* to *jdbc:db2:ITSOVSAD* and provide a valid *User ID* and *Password* (or empty fields). Click *Ok* (Figure 14-26).
- ▶ Click *Finish* to end the wizard. A folder *groups\ITSOGroup* with the properties is added under the source folder.

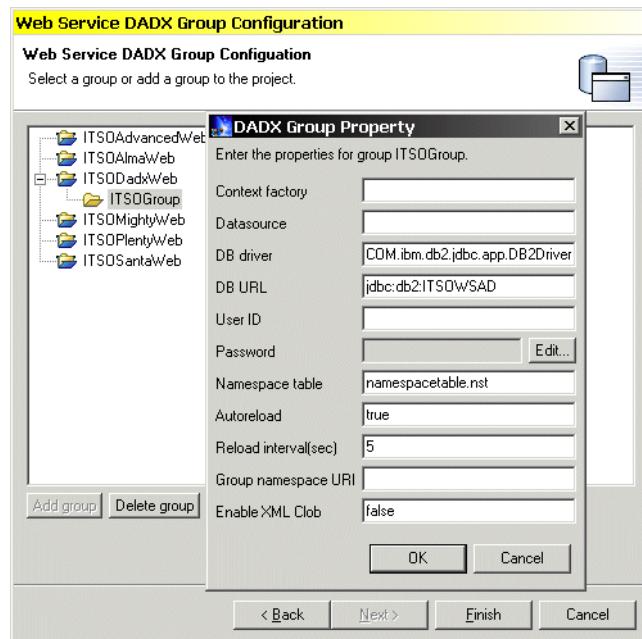


Figure 14-26 DADX group configuration

Create a DADX file from the SQL statement

In the data perspective, Data view:

- ▶ Select New -> Other -> XML-> XML from SQL Query. Click Next.
- ▶ Select Create DADX from SQL query. Click Next.
- ▶ Expand ITSODadxWeb until you find the selectParts SQL statement (Figure 14-27) and select it. Click Next.

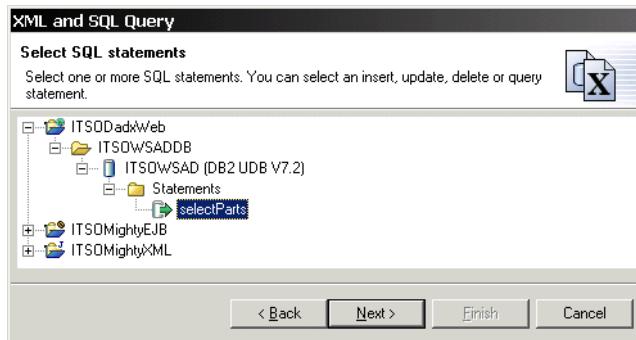


Figure 14-27 Select the SQL statement

- ▶ Skip the Select DAD files panel.
- ▶ In the DADX generation wizard specify ITSOSelectPartsDadx.dadx as the file name and select /ITSODadxWeb/source/groups/ITSGroup (the folder of the DADX Group created in the previous step) as the output folder (Figure 14-28). Click Finish.

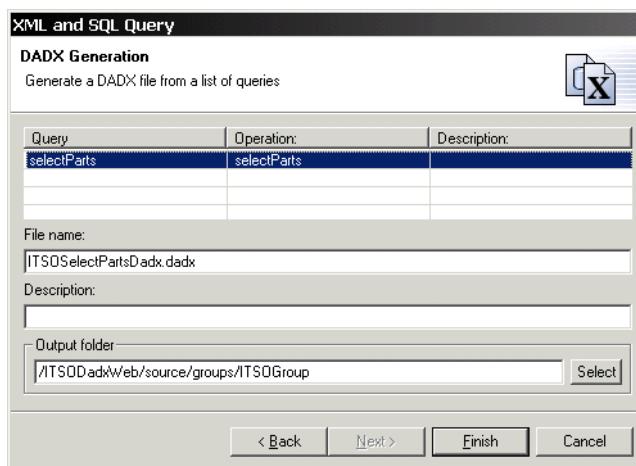


Figure 14-28 DADX generation

The Wizard generates the DADX file, `ITSOSelectPartsDadx.dadx`, in the `source/groups/ITSOGroup` folder in the `ITSODadxWeb` project and opens the file in the editor (Figure 14-29).

```
<?xml version="1.0" encoding="UTF-8"?>
<dadx:DADX xmlns:dadx="http://schemas.ibm.com/db2/dxx/dadx"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xsi:schemaLocation="http://schemas.ibm.com/db2/dxx/dadx dadx.xsd
        http://schemas.xmlsoap.org/wsdl/ wsdl.xsd">
    <dadx:operation name="selectParts">
        <wsdl:documentation xmlns="http://www.w3.org/1999/xhtml">

        </wsdl:documentation>
        <dadx:query>
            <dadx:SQL_query>
                <![CDATA[
                    SELECT ITSO.AAPARTS.PARTNUMBER, ITSO.AAPARTS.NAME FROM ITSO.AAPARTS
                ]]>
            </dadx:SQL_query>
        </dadx:query>
    </dadx:operation>
</dadx:DADX>
```

Figure 14-29 Generated DADX file, `ITSOSelectPartsDadx.dadx`

This file defines one operation, `selectParts`, which refers to the SQL statement we defined. The SQL statement itself, shown in bold, is also contained within the DADX file.

Verify the Web project and the server

Before running the wizard, verify that the `ITSODadxEAR` project was added to the `ITSOWSADWebSphere` servers and that this server is the default server instance for the `ITSODadxWeb` project.

- ▶ In the Web perspective, select the `ITSODadxWeb` project and *Properties*. In the *Properties* panel make `ITSOWSADWebSphere` the default server in the *Server Preferences* panel.

Run the Web service wizard from the DADX file

To read more about the wizard, and the various options and files it generates, see “Using the Web service wizard” on page 336. Now we start the Web service wizard from the DADX file:

- ▶ In the Web perspective, select the `ITSOSelectPartsDadx.dadx` file and *New -> Other -> Web Services -> Web Service*.
- ▶ On the first panel make sure *DADX Web service* and `ITSDadxWeb` is selected. Select *Generate a proxy* and *Generate a sample*. Click *Next*.
- ▶ The `ITSOSelectPartsDadx.dadx` file is preselected, click *Next*.
- ▶ On the *Web Service DADX Group properties*, the properties defined in Figure 14-26 on page 482 are displayed. Click *Next*.
Be patient, the server is started and the Web application is published.
- ▶ On the *Binding Proxy Generation* panel, change the proxy class to:
`itso.wсад.dadx.proxy.ITSOSelectPartsDadxProxy`
Select *Show mappings* and click *Next*.
- ▶ On the *Web Service XML to Java Mappings* panel, select *Show and use the default DOM Element mapping*. Click *Next*.
- ▶ You can skip the rest of the panels, or go through them, and click *Finish*.

These files listed here are generated:

- ▶ A new servlet `ITSOGroup`, with `DxxInvoker` as the servlet class, is added to the Web application deployment descriptor. This servlet is a subclass of the `rpcrouter` servlet that is used in “SOAP router servlets” on page 351. It handles the incoming SOAP requests and sends the response back. The URL mapping, `/ITSOGroup/*`, matches the name of the DADX group.
- ▶ The client proxy class in `itso.wсад.dadx.proxy`.
- ▶ A mapping class named `SelectPartsResult_ElementContentType` in a new `mappings` folder.
- ▶ The SOAP admin application in the `webApplication/admin` folder.
- ▶ The test client in the `sample/ITSOSelectPartsDadx` folder.
- ▶ The WSDL service interface file (`ITSOSelectPartsDadx-binding.wsdl`) and the service implementation file (`ITSOSelectPartsDadx-service.wsdl`) are generated in the `wsdl` folder.
- ▶ The `ITSOSelectPartsDadx.isd` file in the `WEB-INF/isd/dadx` folder.
- ▶ The SOAP deployment files, `dds.xml` and `soap.xml`.
- ▶ A `worf` folder with an HTML-based Web service test facility. See “Using the Web services object runtime framework (WORF)” on page 487.

Client proxy

An extract of the client proxy is shown in Figure 14-30.

```
public class ITSOSelectPartsDadxProxy
{
    private Call call = new Call();
    private URL url = null;
    private String stringURL =
        "http://localhost:8080/ITSOSelectPartsDadx/ITSOGroup/ITSOSelectPartsDadx.dadx/SOAP";
    private SOAPMappingRegistry smr = call.getSOAPMappingRegistry();
    .....
    public synchronized org.w3c.dom.Element selectParts() throws Exception
    {
        String targetObjectURI = "http://tempuri.org/ITSOGroup/ITSOSelectPartsDadx.dadx";
        String SOAPActionURI = "http://tempuri.org/ITSOGroup/ITSOSelectPartsDadx.dadx";
        .....
        call.setMethodName("selectParts");
        call.setEncodingStyleURI(Constants.NS_URI_LITERAL_XML);
        call.setTargetObjectURI(targetObjectURI);
        Vector params = new Vector();
        call.setParams(params);
        Response resp = call.invoke(getURL(), SOAPActionURI);
        ....
    }
}
```

Figure 14-30 Client proxy (extract)

Notes:

- ▶ We could not overwrite the URI of the Web service in the wizard:
`http://tempuri.org/ITSOGroup/ITSOSelectPartsDadx.dadx`
This value is in the ISD file (and therefore in `dds.xml`) and in the client proxy.
- ▶ The target URL points to `/ITSOGroup`, which is mapped to the `DxxInvoker` servlet defined in the Web application:
`http://localhost:8080/ITSOSelectPartsDadx/ITSOGroup/ITSOSelectPartsDadx.dadx/SOAP`

Test the Web service

To test the Web service, launch the sample application:

`/webApplication/sample/ITSOSelectPartsDadx/TestClient.jsp`

- ▶ Click on the `selectParts` method link in the left frame. Click `Invoke` in the right frame. The results of the query should appear in XML format in the right bottom frame as shown in Figure 14-31.

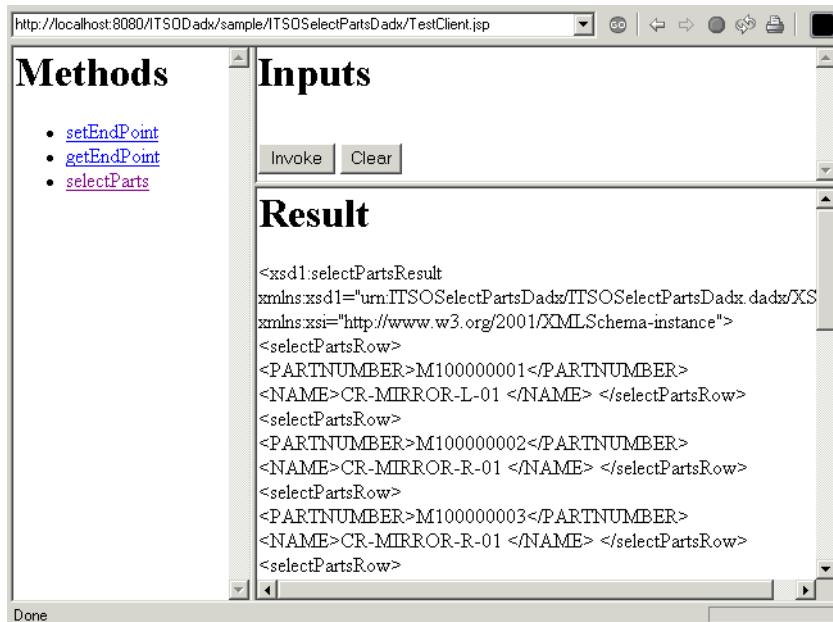


Figure 14-31 Testing the Web service using the sample test client

Using the Web services object runtime framework (WORF)

WORF is shipped with the Application Developer 4.0.2 and can be used to test the Web service without using the client proxy. The Web service wizard generated a set of JSP and HTML files into the `worf` folder of the Web application.

To test the Web service using WORF, paste this URL into the internal or an external Web browser:

`http://localhost:8080/ITSODadx/ITSOGroup/ITSOSelectPartsDadx.dadx/TEST`

This is almost the same URL as used in the client proxy (Figure 14-30 on page 486), but with TEST as suffix.

The dialog shown in Figure 14-32 opens:

- ▶ Click on the `selectParts` method in the left frame.
- ▶ Click `Invoke` in the right frame.
- ▶ The XML output of the Web service appears in the bottom frame. Note the formatting of the XML output done by the Web browser.
- ▶ Compare the output to the output of the sample test client.

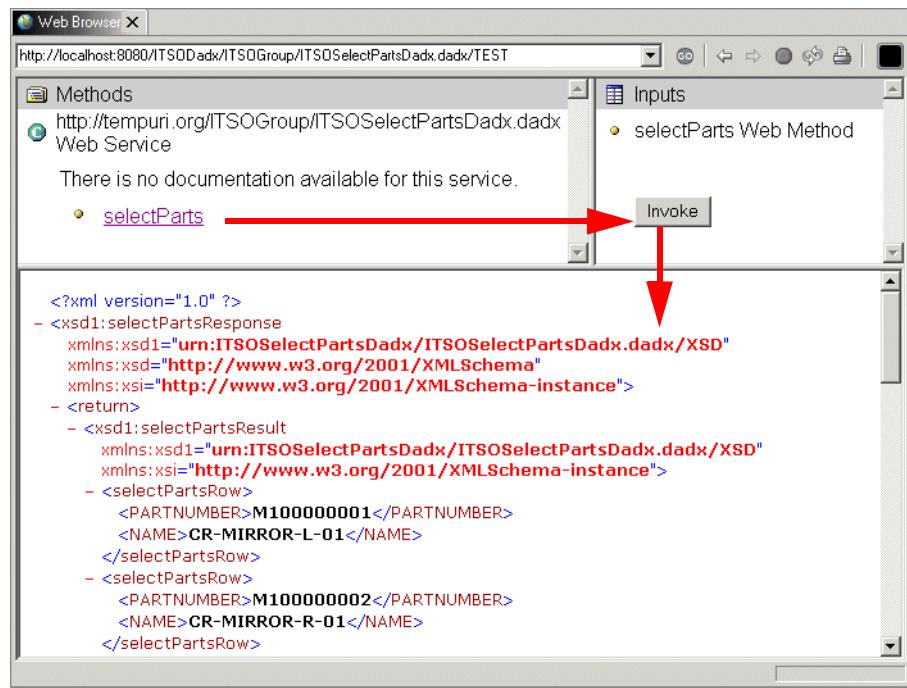


Figure 14-32 Web service testing using WORF

Generating a Web service from an URL

A Web service can also be generated from an existing running Web site (only HTTP POST/GET operations are supported, no full SOAP).

We did not experiment with this feature for this redbook, because it was not available in the beta code.

Summary

In this section we demonstrated how to create a Web service from a DADX group based on an SQL statement.

Advanced UDDI topics

This section should be read in association with “UDDI overview” on page 293.

This section outlines the six identifiable types of UDDI business registries and provides recommendations as to which is the most appropriate for different business scenarios. These variants are as follows:

- ▶ UDDI operator cloud
- ▶ e-marketplace UDDI
- ▶ Portal UDDI
- ▶ Partner catalog UDDI
- ▶ Internal enterprise application integration UDDI
- ▶ Test bed UDDI

We also investigate how to use the IBM WebSphere UDDI Registry product provided on the WebSphere Developer Domain:

<http://www7b.boulder.ibm.com/wsdd/downloads/UDDIregistry.html>

The UDDI specification was designed to address the requirements of finding and searching for business entities and services that were published in the global UDDI business registry currently operated by IBM, Microsoft, and Ariba (which will be replaced by HP).

When most people discuss UDDI, this is the registry they refer to. However, other types of UDDI nodes are supported by the specification and are actually of more practical use when developing Web services today. These are critical to the emergence of a dynamic style of service-oriented architectures, and are described in more detail in the following sections.

UDDI operator cloud

The UDDI operator cloud is a series of UDDI nodes that can be accessed from:

<http://www.uddi.org> or <http://www.ibm.com/services/uddi>.

Changes made in each node are replicated on a regular basis to the other nodes. The runtime topology for this architecture is shown in Figure 14-33.

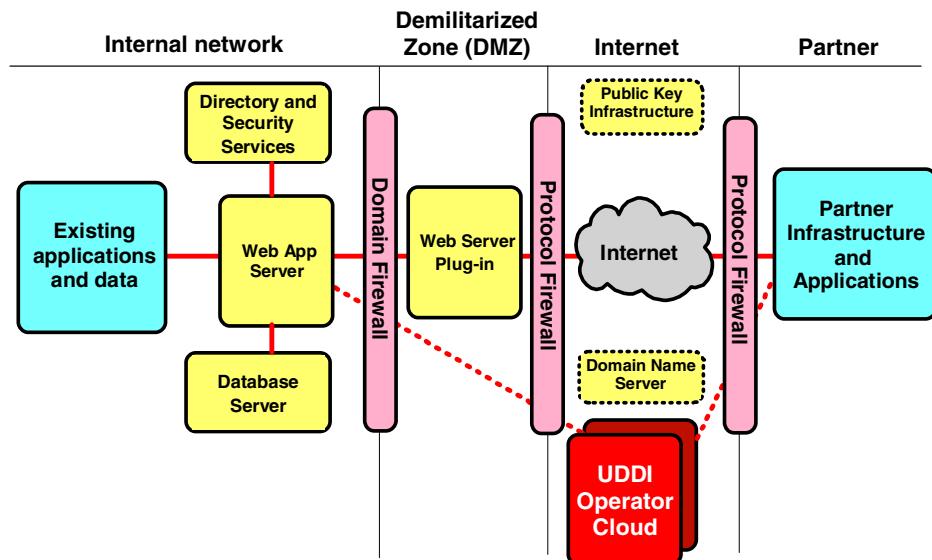


Figure 14-33 Runtime topology for using the UDDI operator cloud

The UDDI operator cloud contains Web services data for many different kinds of businesses in various geographies and industries. Some of this data represents legitimate businesses and information, and the business process exposed as Web services. Some of this data represents organizations posing on the transactional Web, but do not provide Web services. Finally, some of this data can represent fraudulent or misleading information on business intent. In order to use this registry successfully for dynamic e-business, we must be able to distinguish between meaningful entries and entries that are a waste of time.

If Web services discovery relies only on the operator cloud, all discovery operations must be performed at design time. As the tModel field contains text-prose information, it must be examined in detail and invoked using hand crafted code based on the description.

e-marketplace UDDI

This is a form of private UDDI registry that can be hosted by an e-marketplace, a standards body or a consortium of organizations that participate and complete in the industry. In this scenario, the *publish* and *find* UDDI APIs are deployed for access over the Internet by any of the member organizations. This uses a similar runtime topology to that shown in Figure 14-33.

This private registry can relate to a particular geography or industry. The membership process should involve validation of the legitimacy of the business entry. It could also provide value-added features such as quality of service monitoring on the response times of the Web service, or monitoring of the business practices of the member organizations.

Currently, very few of these e-marketplace UDDI nodes exist, and this presents a huge business opportunity for new players in the industry or existing e-marketplace operators, who want to ensure they are not made redundant by the UDDI operator cloud.

Such e-marketplace registries can limit the entries to a subset of the existing Web service taxonomies (such a small number of the NAICS categories) as well as providing standard tModels for common business processes in the industry. In some circumstances, it may be necessary to provide completely custom taxonomies if no appropriate standards exist.

The real benefits of such a registry are that as a developer, you have a reasonable guarantee that using the *find* and *bind* operations against the registry for dynamic binding at runtime using a single service proxy, will be successful.

Portal UDDI

Just as a company can have a presence of the traditional document-based Web (<http://www.acme.com>), it can also have a presence on the transactional Web (<http://www.acme.com/uddi>).

The portal UDDI sits inside the organizations demilitarized zone as shown in Figure 14-34.

This portal UDDI server contains only metadata related to the company's Web services that it wishes to provide to external partners. The *publish* APIs will be removed for users accessing the registry from the Internet, restricting publishing rights to internal applications only. This gives the organization complete control over how their Web services metadata can be used and retrieves information about who the interested parties are. It could also potentially restrict *find* access to a specific list of trusted partners.

This topology can be used in conjunction with the UDDI operator cloud or e-marketplace UDDI registries, although a replication mechanism must be put in place to ensure that the entries are synchronized. This is discussed in the later section, "Managing the relationships between UDDI nodes" on page 494. The URL for the UDDI portal server can be used as the *discoveryURL* in the *businessEntity* record for the company in the UDDI operator cloud.

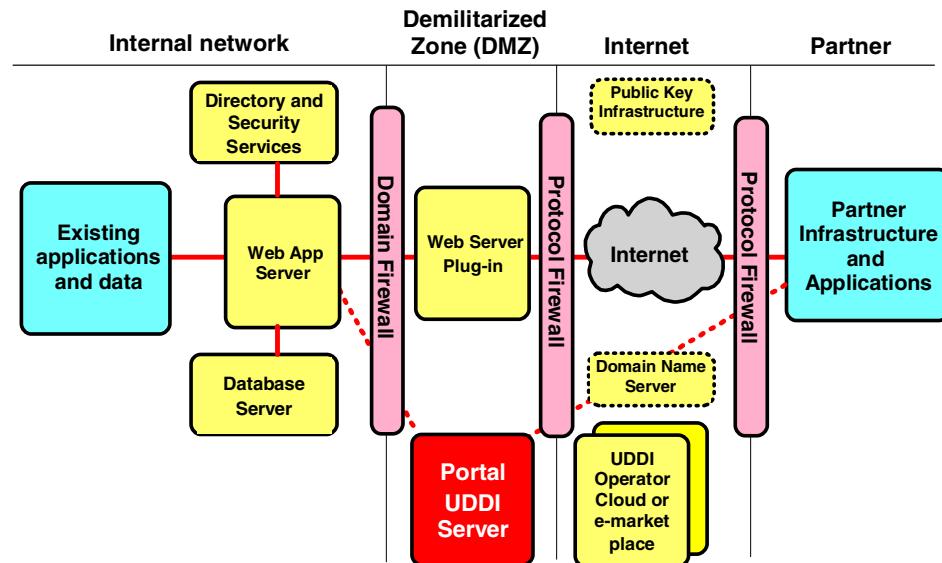


Figure 14-34 Runtime topology for using a Portal UDDI server

Partner catalog UDDI

Potentially the most useful runtime topology, the partner catalog UDDI server is a private registry that sits behind an organizations firewall and contains only Web service metadata published by trusted partners. The metadata is added only by internal processes that validate information retrieved from the partners portal UDDI server, an e-marketplace server, or the UDDI operator cloud.

This setup is especially useful when the nature of the business requires extensive legal contracts to be agreed to before the two organizations can integrate their applications. It provides all of the benefits of dynamic binding at runtime, but against a controlled and validated set of partners.

The runtime topology for this approach is illustrated in Figure 14-35.

This topology again raises the question of how these entries are continually synchronized with the parent UDDI node, which is discussed later.

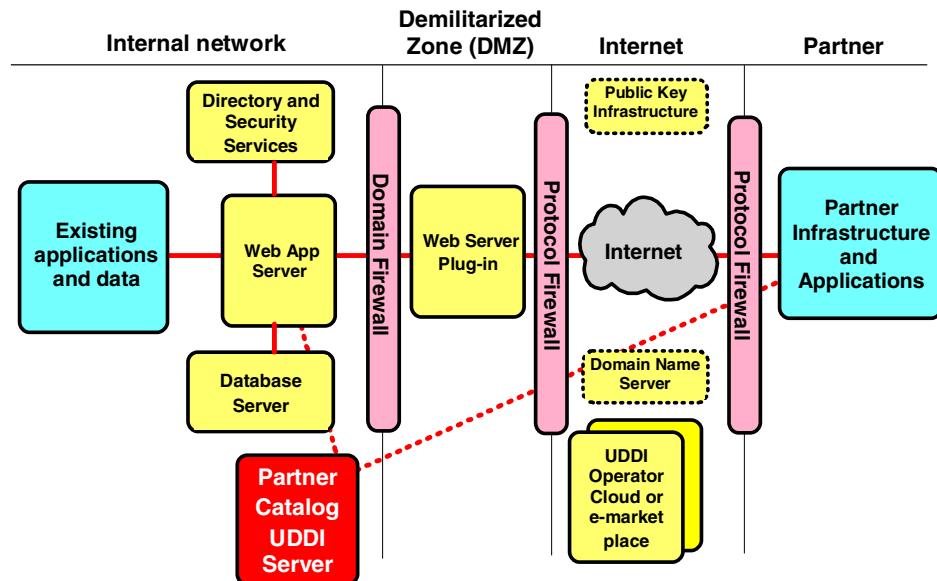


Figure 14-35 Runtime topology for using a partner catalog UDDI server

Internal enterprise application integration UDDI

Before companies begin to utilize dynamic Web services with their business partners, it is likely that they will trial the technology first by integrating their own applications internally.

This runtime topology is very similar to the partner catalogue variety, except that the entries are for Web services provided by other departments or groups within an organization. The administrators of the registry have the power to define the Web services standards that will be used in the organization by limiting the tModels available.

The runtime topology for this approach is illustrated in Figure 14-36.

This may potentially be a very attractive approach for organizations who have selected a multi-vendor approach to application development, and require interoperability between loosely-coupled applications. However, this should not be confused with a scalable EAI architecture solution, such as MQSeries or MQSeries Integrator, which provides a higher quality of service, and a greater number of integration platforms.

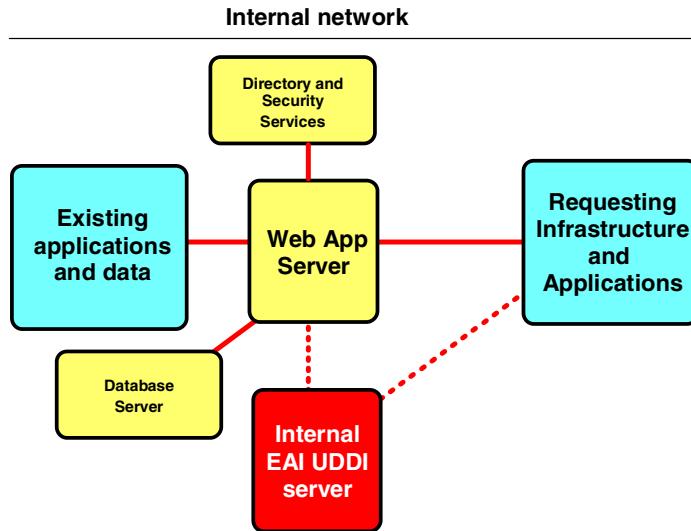


Figure 14-36 Runtime topology for using an internal EAI UDDI server

Test bed UDDI

The final style of UDDI node is a node that is used to privately test applications. This can be for both requester applications and provider Web services.

A test node provides an environment where developers can test that the UDDI entries for their Web service are accurate, and that applications can use the metadata to generate proxies from the UDDI entry to access the service.

Before publishing to either the UDDI operator cloud, e-marketplace, or local UDDI portal, it is strongly advised that an organization installs such a test bed registry and performs a suite of tests that simulate a number of approaches for finding and binding to the Web service from each platform that its partners would use. This should also include tests which measure performance and response times when using a UDDI portal, so that an appropriate quality of service can be achieved.

Managing the relationships between UDDI nodes

It is likely that adopters of Web services technology may find themselves building a hierarchy of UDDI registries. A potential scenario is illustrated in Figure 14-37.

When developing such a hierarchy, issues arise relating to the replication of metadata information between UDDI nodes.

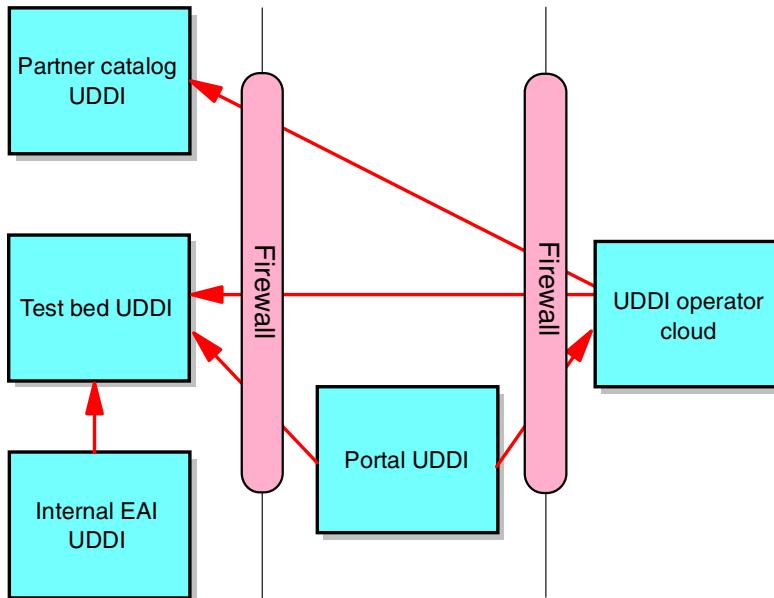


Figure 14-37 Relationships and replication requirements between UDDI nodes

A number of limitations exist in the current version of the UDDI specification, which make managing private UDDI nodes a complex task. This is primarily because the specification is designed primarily to support the single UDDI operator cloud, where the assignment of unique identifiers (UUIDs) for each Web service is the responsibility of the server, and not the publisher of the service.

To be able to properly manage private registries, we must understand how information can be replicated between private nodes where the keys may not necessarily be unique. In addition to this, a publish/subscribe interface should be available to inform registries (such as our partner catalog) that the entry in the UDDI operator cloud for the service has been updated, so that a synchronization of the metadata may be initiated. Some of these features are being proposed for future releases of the UDDI specification.

The implications of these limitations is that while it is likely that the previously described scenarios are attractive application topologies, caution should be used during implementation, and steps taken to ensure that developed functions are isolated from future changes to the specification.

IBM WebSphere UDDI Registry

In addition to the Web services toolkit available from IBM alphaWorks, IBM also provides a private UDDI registry product that is available for download from the WebSphere developer domain URL:

<http://www7b.boulder.ibm.com/wsdd/downloads/UDDIregistry.html>

The WebSphere UDDI Registry is currently (March 2002) available as:

- ▶ **Version 1.1:** On Windows 2000, Windows NT, and Linux (RedHat and SuSe) implementing UDDI Version 2 specification, requires WebSphere Application Server 4.02 and DB2 7.2 Fixpack 5
- ▶ **Preview:** On Solaris Version 8, implementing UDDI Version 1 specification, requires WebSphere Application Server 4.0 and DB2 7.1 (or better)

Attention:

- ▶ When we wrote the redbook we used the WebSphere UDDI Registry **preview** code on Windows. This code is not available any more. The **version 1.1** code on Windows does not work in the same way as the preview code and not all functions of the Application Developer UDDI Explorer are supported.
- ▶ We suggest that you use the IBM Business Test Registry for UDDI tests with the Application Developer.
- ▶ Integration of the Application Developer with the WebSphere UDDI Registry should be possible again in the future when a newer level of the WebSphere UDDI Registry or the Application Developer is available.

Business process management with Web services

Throughout this book, we have demonstrated how Web services can be used to integrate disparate systems across multiple vendors. We have also discussed approaches for creating composed Web services from others. However, there is one final piece of the Web services jigsaw that has not yet been discussed—the implications this technology can have on performing business process management.

Developers often consider business process management techniques unnecessary when similar results can be obtained by using standard programming approaches. However, it is important to abstract the definition of a business process from the technology used to implement it, by building a good understanding of the big picture business requirements before rushing in to build an application.

Traditionally, business process modeling starts with a picture of what the process looks like as a whole, broken down in terms of the specific *activities* that must be completed, the order in which they must be completed, the *dependencies* between those activities, and a definition of who is responsible for making sure those activities are completed.

Each activity can either represent a new piece of business logic or an interface to an existing system using some middleware. Web services are an ideal technology to use for creating such activities, as they provide the flexibility of easily interchanging the implementation without impacting the defined business process.

In this section, we outline some of the activities IBM is currently involved in to provide a standardized grammar for describing the flow through a series of Web services. We also describe how Web services technology can already be applied to products that are available today, such as IBM's MQSeries Workflow and MQSeries Integrator. It is not intended to provide a comprehensive description of how business process management works, but it provides enough food for thought to hopefully encourage the reader to investigate the implications of this technology further.

IBM WSFL proposal

IBM intends to work with partners on the creation of a standard in the Web services flow language subject area. The current Web services flow language (WSFL) specification is the IBM input to the corresponding standardization effort. This section describes the content of the first draft of the specification, which can be downloaded from:

<http://www-4.ibm.com/software/solutions/webservices/resources.html>

Introduction

WSFL is a new XML grammar for defining software workflow processes within the framework of a Web services architecture. It considers two aspects of Web services compositions:

- ▶ The sequence in which the services should be invoked
- ▶ The information flow between the services

In itself, WSFL is not a specification dealing with how to create a business process model. Rather, you would use WSFL to create an XML representation of the business model, then feed that XML-representation into a middleware application designed to invoke and manage the process. It is anticipated that the currently available business process modelling tools will be adapted to generate WSFL in the future.

WSFL fits naturally into the Web services computing stack. It is layered on top of WSDL, which it uses for the description of services interfaces and their protocol bindings. WSFL also relies on an envisioned endpoint description language to describe the non-operational characteristics of service endpoints, such as quality-of-service properties. This has provisionally been referred to as the Web services endpoint language (WSEL), although even the first draft of this specification is not yet complete at the time of writing.

One final aspect to note is that WSFL provides support for what is known as “recursive composition”. This means that a flow defined in WSFL can itself be exposed as a Web service and used in a different WSFL flow. This provides scalability to the language and support for progressive refinement of the design, with increasing granularity, as well as aggregation.

Other competing XML-based workflow definition grammars have also been proposed. These include:

- ▶ Microsoft's XLANG, used in the heart of their BizTalk server
- ▶ The ebXML business process specification
- ▶ BPML—the business process modelling language from the business process management initiative
- ▶ Hewlett-Packard and UDDI.org's WSCL—the Web services conversation language

These proposals are all far from complimentary. As to what will be the most successful in the future is anyone's guess, although the number of vendors who see this as an important aspect of Web services technology is reassuring.

WSFL flow model

Figure 14-38 is taken directly from the WSFL specification, and illustrates the elements that are contained with a WSFL document.

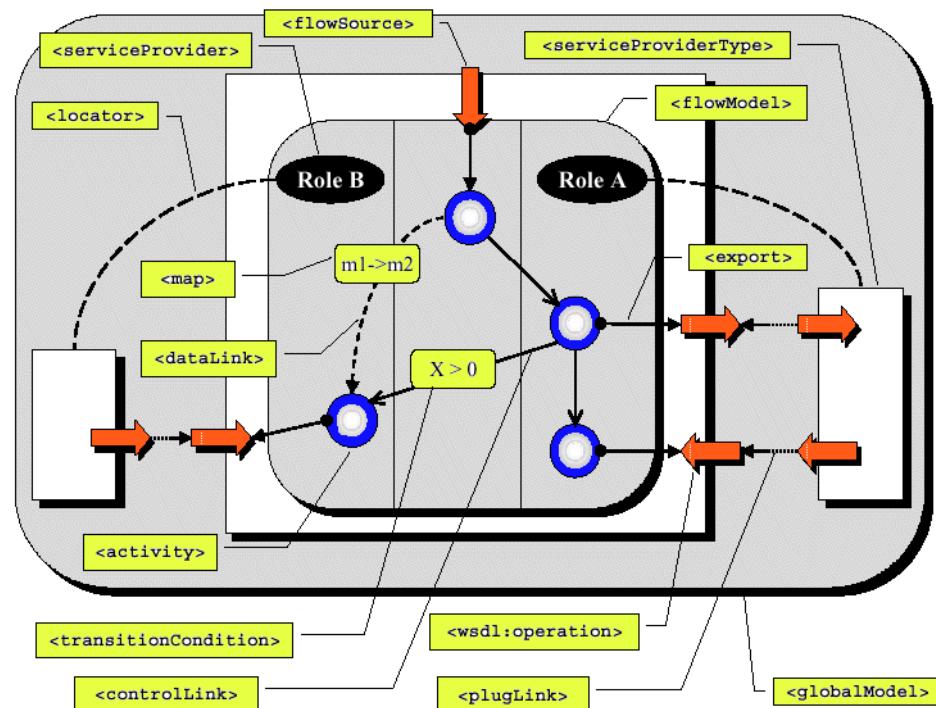


Figure 14-38 WSFL grammar elements

There are two types of models illustrated in this diagram:

Flow model This is the representation of the flow of the business process comprised of Web services.

Global model Because of the decentralized or distributed nature of these Web service interactions, a global model is used to detail the messages sent between the Web services in the flow model as it is executed. It does not contain any information about the execution sequence, but defines how to bind the services together and map their operations. We will postpone a discussion of the global model until later in this chapter.

Each process defined in the flow is referred to as an *activity*. There are four types of activities that are available: local operations, static Web service bindings, UDDI lookups, or dynamic bindings based on a given criteria.

In the illustration, two types of communication are shown between the activities in the flow model:

Control links Shown by a solid line. This is the mechanism through which the workflow processor walks through each of the activities in the business process.

Data links Shown by a dashed line. This defines the flow of data through the business process, passed between the comprising Web services by the flow engine. In most cases, this closely follows the control flow.

When defining control links, a mechanism is required to define when a particular activity is finished and when the next activity can be invoked. This is known as a *transition condition*. This is a true or false statement that the processor can use to determine the current state of any activity.

WSFL also supports activities that *fork* when they have more than one outgoing control link defined. Each activity defined following the fork is then executed in parallel. Similarly, once a flow has been forked it must be synchronized at a later time. This is done through a *join* activity that has more than one incoming control link. By default, activities defined after the join are only executed when all parallel activities are complete.

Activities that have no incoming control link are by definition *start activities*, and those without an outgoing control link are *end activities*. When the last end activity completes, the output of the overall flow is determined and returned to its invoker.

Each data link is weighted by a *map* specification. This prescribes how fields in the target's input message are constructed from fields in the output message. Data links are also used to describe the input and output of the flow itself. Any links which target activities in the flow from outside are known as *flow sources*, and links from activities inside the flow that target activities outside the flow are known as *flow sinks*.

Every activity in the flow can be performed by a different business partner. This is defined by specifying a *service provider*, who is responsible for providing the appropriate Web service for the activity.

Flow model example

Extending the auto parts sample used elsewhere in this book, let us consider a sample scenario where the Almaden Autos mechanic wants to order a part from one of the parts suppliers to fit in a car being serviced.

A sample workflow process is shown in Figure 14-39.

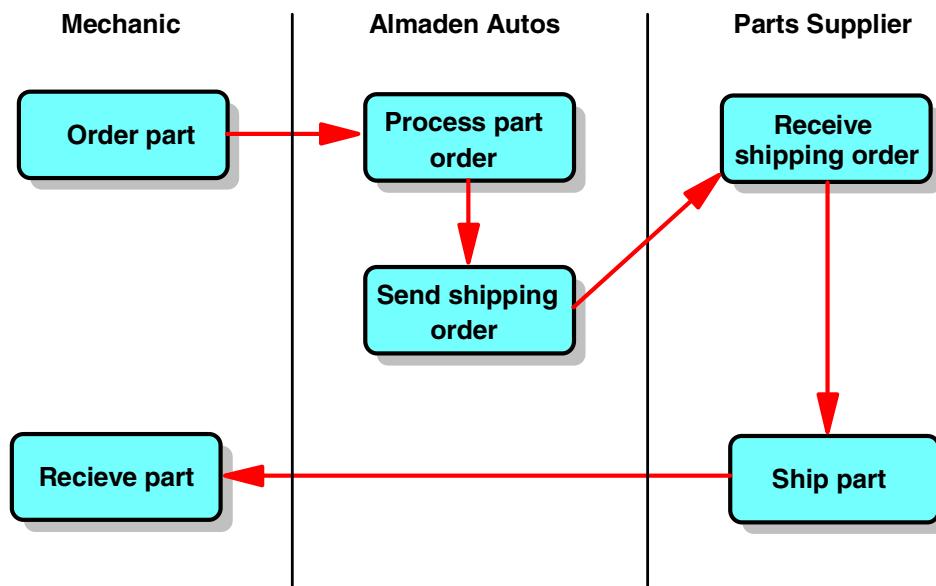


Figure 14-39 Sample flow model for ordering a part from a supplier

Each of the shaded boxes illustrates an activity in our business process, and the arrows show the control flow. The corresponding WSFL for a fragment of the example is shown in Figure 14-40.

This WSFL code shows the flow model entries that describe the control flow between the *Send shipping order* and *Receive shipping order* activities. The text highlighted in bold shows the links between the service providers, their activities, and the corresponding control and data links.

```

<flowModel name="partsOrderFlow" serviceProviderType="partsOrder">
    <serviceProvider name="almadenAutos" type="dealership"/>
    <serviceProvider name="partsCompany" type="partsSupplier"/>
    <!-- other service providers here -->
    <activity name="sendShippingOrder">
        <performedBy serviceProvider="almadenAutos"/>
        <implement>
            <export>
                <target portType="partsRequestor" operation="sendShippingOrder"/>
            </export>
        </implement>
    </activity>
    <activity name="receiveShippingOrder">
        <performedBy serviceProvider="partsCompany"/>
        <implement>
            <export>
                <target portType="partsShipment" operation="receiveShippingOrder"/>
            </export>
        </implement>
    </activity>
    <!-- other activites here -->
    <controlLink source="processPartOrder" target="sendShippingOrder"/>
    <controlLink source="sendShippingOrder" target="receiveShippingOrder"/>
    <controlLink source="receiveShippingOrder" target="shipPart"/>
    <!-- other control links here -->
    <dataLink source="sendShippingOrder" target="receiveShipingOrder">
        <map sourceMessage="shippingOrder" targetMessage="shippingOrder"/>
    </dataLink>
    <!-- other data links here -->
</flowModel>

```

Figure 14-40 WSFL fragment showing the control flow between two activities

WSFL global model

The global model describes the interactions between existing Web services and how to define new Web services as a composition of existing services—in essence the binding of services to the flow model, and the exposure of the models interface.

A *service provider type* defines the interface of a Web service with a set of port types that declare the operations supported. Each *service provider* fulfilling a role within the process must implement these port types and define them in the WSDL describing their Web service interface. The service provider types are defined using the grammar shown in Figure 14-41. Again note how the names underlined match those in Figure 14-40.

```
<definitions name="partsOrderDefs">
  <serviceProviderType="dealership">
    <portType name="partsRequestor" />
  </serviceProviderType>
  <serviceProviderType="partsSupplier">
    <portType name="partsShipment" />
  </serviceProviderType>
</definitions>
```

Figure 14-41 Service provider type definitions in the global model

The service provider used at runtime is determined by a *locator*, using a *selection policy*, which could be based on using the first service in the list, selecting a service at random, or using some user-defined algorithm.

The interaction between a set of service providers in the global model are declared using *plug links*.

Recursive composition of flows

As we explained earlier, a WSFL flow can be itself called using a Web service interface. Just like any other B2B system, the flow has a *public interface* and a *private interface*. The private interface contains the flow model and the global model. WSFL allows you to take the individual parts of that process, and export them as part of the business processes public interface.

When interacting with Web services that are external to the flow, the activities must be exposed using an *export* definition through to the public interface, and defined as *plug links* to provide the flow engine with the information necessary to perform the operation, and tie that back into the business process being executed.

Finally, a WSFL implementation engine could take the global model, run it through a WSDL generation process, and create the Web service interface (WSDL port type) definition containing the public interface. This can then be used to generate a Web service client as described in Chapter 10, “Static Web services” on page 321.

Web services and MQSeries Workflow

MQSeries Workflow is the process engine at the heart of IBM’s business process management software. It executes process definitions captured during modeling, and ensures that business functions are performed reliably and correctly using the transactional integrity of MQSeries. Such processes could run from sub-seconds to many months.

This product is the ideal host for the implementation of a workflow engine for Web services. This integration is provided on two levels:

- ▶ Exposing the workflow as a Web service
- ▶ Being able to call a Web service from inside a workflow

The most recent release at the time of writing, MQSeries Workflow Version 3.3, does not provide these capabilities out of the box. However, a download is now available from the IBM alphaWorks site called the IBM Web services process management toolkit (WSPMTK), which provides these features. It is available at the following URL:

<http://www.alphaworks.ibm.com/tech/wspmt>

IBM Web services process management toolkit

The toolkit is designed to work together with MQSeries Workflow (MQWF) and the IBM Web services toolkit (WSTK) to provide a business process management engine for Web services. It contains the following key components:

- ▶ A generic Web service provider: This is a SOAP provider implementation that intercepts SOAP requests, transforms them into MQWF XML requests, transmits them to MQWF using MQSeries, transforms and passes the MQWF response back to the SOAP requester.
- ▶ Generic Web services activity: This component allows you to call a Web service as an activity implementation or user-defined program execution service (UPES) of a process. The MQWF UPES either uses client proxy code or a WSDL document to invoke the service.
- ▶ Flow Definition Language (FDL) to WSDL generation: A generator that creates a WSDL Web service definition based on FDL.

WSPMTK and WSFL: It is important to note at this point that the current release of the WSPMTK does not generate WSFL, but instead uses its non-XML-based Flow definition language (FDL) as the basis for generating the WSDL interfaces.

Exposing a workflow as a Web service

This section describes the steps to make a given workflow process available as a realization of a Web service (Figure 14-42).

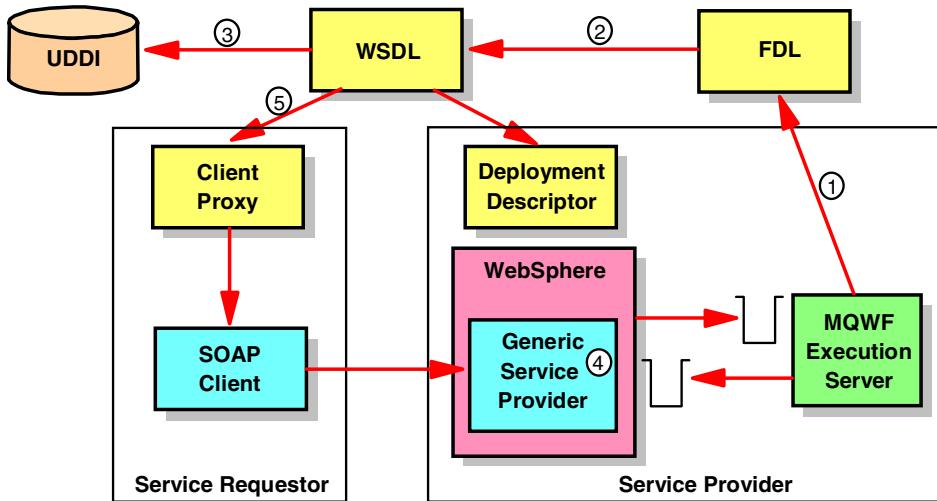


Figure 14-42 Process used to expose an MQWF process as a Web service

- ▶ The service provider models a process using the MQWF process modeler, deploys it to the workflow engine, and generates a FDL representation of this process from the runtime representation using the MQWF fmcibie utility (1).
- ▶ The service provider executes the fd12wsdl translator (provided by the WSPMTK) against the FDL and generates the following artifacts:
 - Serializers and deserializers for all structures used within the process
 - A deployment descriptor, which is used by the generic server provider to binds specific URIs to various workflows
 - A single WSDL file, which provides the Web service interface description for service requests to the workflow (2)
- ▶ The WSDL file is published to an appropriate UDDI registry (3).
- ▶ The generic Web service provider shipped (4) with the WSPMTK is added to the Web application, and is required for routing the SOAP messages to MQWF, along with the generated proxy and serializer classes. When the Web application is deployed to an application server instance, the service is ready for execution.
- ▶ Clients of the Web service import the WSDL definition into Application Developer, and use it to create client proxies for the service (5). These can be integrated into the client application as described elsewhere in this book.

Calling a Web service from inside a workflow

A Web service can provide the implementation for an activity with the context of an MQWF process (Figure 14-43):

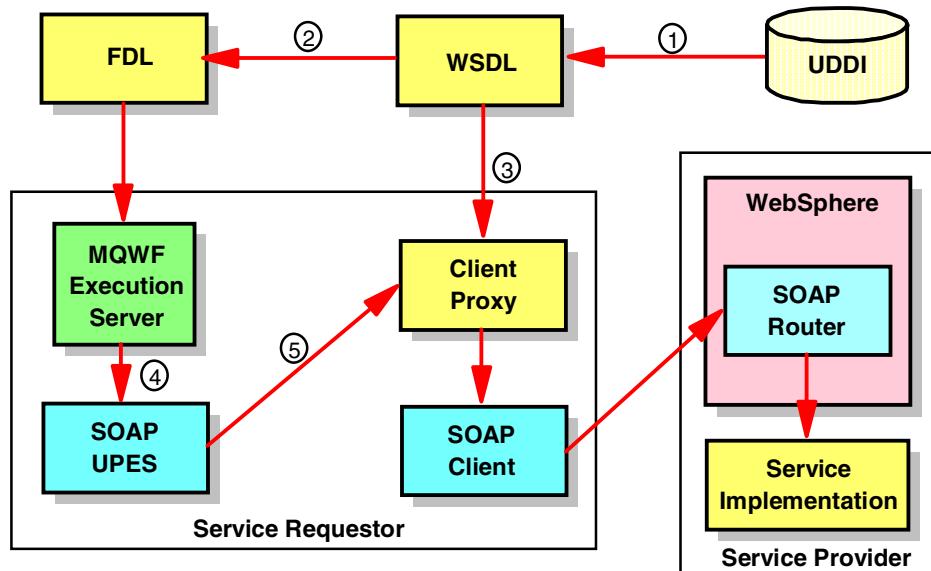


Figure 14-43 Consuming a Web service as an MQWF activity

- ▶ At build time, the designer of the business process (service requestor) searches for a Web service in a UDDI registry. When a suitable service has been discovered, the WSDL document is downloaded from the service provider (1).
- ▶ The service requestor then generates an FDL stub based on the WSDL (2). Currently, the WSPMTK does not provide a tool to assist in this process.
- ▶ Client proxy code is generated from the services WSDL, using either Application Developer or the WSTK (3).
- ▶ The process designer completes modeling the business process in the MQSeries workflow modeling tool, exports it into FDL and imports the process into the MQWF runtime.
- ▶ When the process is executed and the activity that is implemented by the Web service is started, control is given to the Web services UPES shipped with the WSPMTK (4).
- ▶ The Web services UPES collects the necessary information to invoke the Web service based on the client proxy code, or based on the WSDL document.

- ▶ The Web services UPES then calls the Web service (5). When the invocation returns, the Web services UPES returns the result to MQWF, and process navigation continues.

Static specification of a Web service as implementation for an activity during design time is just one option. Dynamic lookup and binding of a Web service during the execution of a process is an advanced option. This could be performed by an interaction with a UDDI registry during runtime to find and bind a Web service implementation based on some criteria specified at design time.

Web services and MQSeries Integrator

IBM's MQSeries Integrator (MQSI) transforms, augments, and applies rules to message-based data, then routes and distributes it between high-performance systems. Processes in MQSI are known as *message flows*.

One of the primary roles MQSI can play in the context of Web services is that of the *service mediator* (a new role in the Web services terminology). For example, on receiving a Web service request, a flow may:

- ▶ Transform this request message into an alternate Web service request
- ▶ Route the request to a different system
- ▶ Even capture data from the request for in-flight logging in a warehouse or audit system

MQSI WSSoapNode

Additionally, MQSI Version 2 now provides a custom WSSoapNode, which implements a Web services requestor that can invoke a Web service and use it to augment or transform the document in the message flow. This is available in the latest MQSeries Integrator Java Plug-in support pac (IA76), downloadable from:

<http://www-4.ibm.com/software/ts/mqseries/ttxppacs/ia76.html>

Mediating a message flow to a Web service

Message flows can be used to implement operations of Web services that require mediation between the originating request and the service providers, or when delegation to a set of service providers is required and their answers aggregated. Another option is to wait for the fastest response of one provider.

At a basic level, this requires transforming the content and representation of a request from the format expected from by the client, to the format offered by the available service providers. In this way, the mediation or transformation separates the service call from the core business implementation, and is more adaptable as the needs of the business change, or as new providers become available.

A message flow can also provide sophistication in value-add services, such as audit, warehousing, content-based access control, nonrepudiation, intelligent routing or time-of-day-based processing, and deliver these sophistications in a flexible and non-intrusive way. They can also be used to implement dynamic service aliasing, based on a variety of quality-of-service considerations, or to provide dynamic lookups for internal, rather than external, Web services names (Figure 14-44).

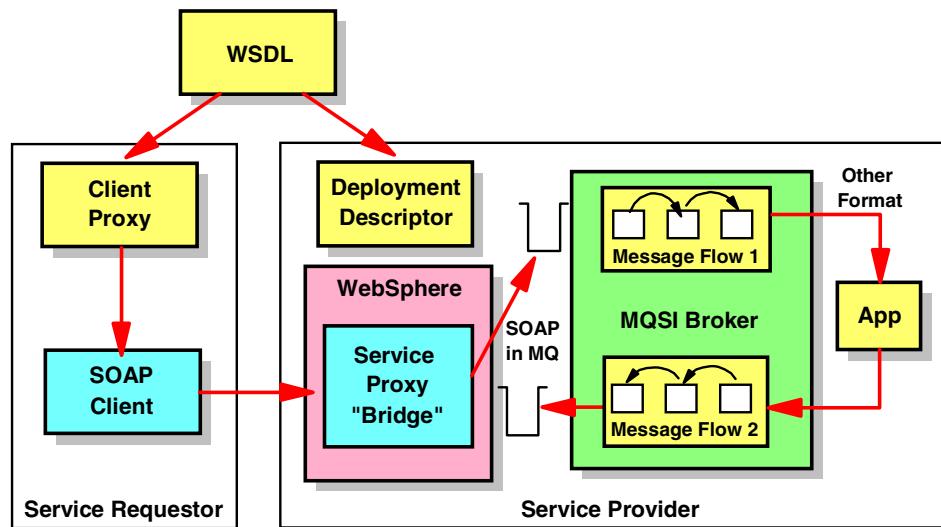


Figure 14-44 Using an MQSI message flow as a service mediator

In general, you can expect to develop a pair of flows (or a complex flow with two execution paths) to implement a Web service, as shown in Figure 14-45.

One flow mediates the inbound request, and the other flow mediates the outbound response. Additionally, a single flow could be used to perform a synchronous set of operations and reply immediately to the initiator of the flow.

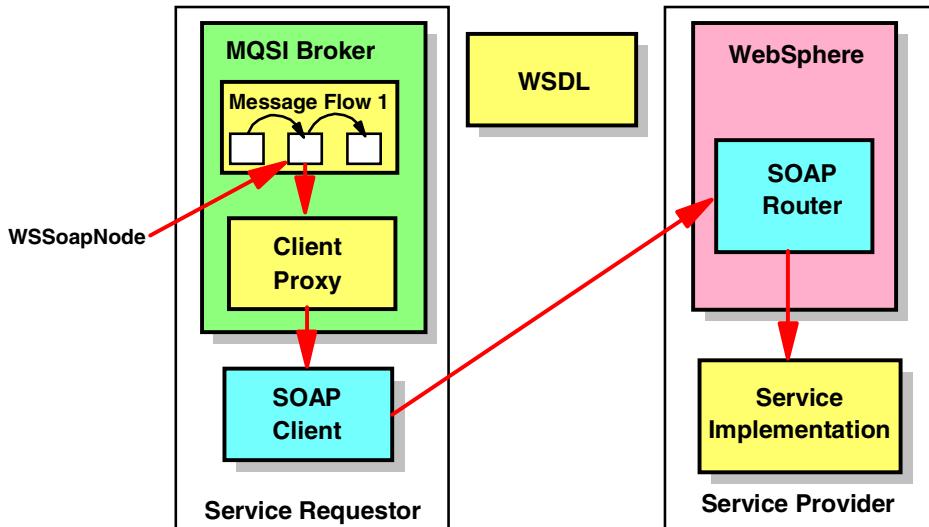


Figure 14-45 Calling a Web service from an MQSI message flow

MQSI does not currently provide support for inbound or outbound SOAP requests over HTTP. The simplest approach to invoke a message flow is to develop a Web service proxy that acts as a bridge to the MQSI runtime by sending and receiving MQSeries messages, presumably using the Java Message Service (JMS). The simple implementation passes the entire body of the SOAP request directly into the body of the message, and sends it to the MQSI input queue to invoke the message flow. In general, this then requires additional transformations at the start and end of the flow to transform it between SOAP, and the form required by the particular implementation. This can be performed using the standard MQSI tools.

The publish and subscribe features of the product also provide an opportunity to try a different Web services model using push technology. This involves Web service requesters registering themselves for events in the Web service, and information pushed to them using the matching engine. Again, a proxy service still has to be developed to broker between SOAP over HTTP messages, and the MQSeries transport mechanism used by MQSI.

Calling a Web service from inside a message flow

The intent of the **WSSoapNode** in the latest MQSI Java plug-in is to illustrate the ability to perform synchronous SOAP calls using the WSTK from within a MQSI message flow. The node uses the features provided by the standard MQSI Java node, **CSIJava**, to parse the input message, extract the SOAP parameters, and then invoke the SOAP call. Once the response has been received, the node then processes the returned objects and forms the XML output.

To define a Web service invocation inside MQSI:

- ▶ Create a message flow that has, as a minimum, input and output queues and a CSIJava node. Connect the message flow to completion.
- ▶ Update the properties of the CSIJava custom node, so that the value for the application field is: com.ibm.csi.mqsi.webService.WSSoapNode (Figure 14-46).

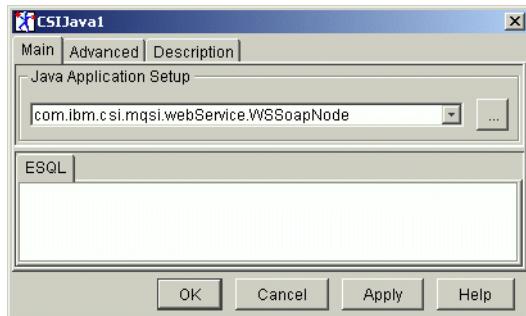


Figure 14-46 Configuration of the MQSI Java node to use the SOAP service

- ▶ Ensure that the element tree for the input message contains the expected SOAP element tree structure.

Summary

In this chapter we described various advanced topics around Web services and demonstrated through examples how such advanced functions can be implemented.

Quiz: To test your knowledge on the topics we covered in this chapter, try to answer the following five questions. The answers are provided in Appendix C, “Quiz answers” on page 559.

1. Name the two encodings supported by the Application Developer Web service wizard. Do they support `java.lang.boolean`?
2. Assign the following class names to the two styles (SAP RPC vs SOAP message-oriented): Message, Envelope, Body vs. Call, Parameter, SOAPMappingRegistry.
3. Decide which SOAP communication style to use for the following two scenarios that a news portal provider is confronted with:
 - When processing incoming HTTP requests, the news portal wants to translate the HTTP request parameters into a Web service invocation to one of its content providers. The request parameters are fixed and known at build time.
 - The interface between the content management system and the news portal is supposed to be reimplemented; each night, a set of XML files needs to be transferred from the content management system to the portal.
4. Name at least three possible scenarios for UDDI registries.
5. What does WSFL stand for and what is it used for?



Architecture and design considerations

In this chapter, we discuss these topics:

- ▶ Architectural positioning of Web services technology
- ▶ Design considerations
- ▶ Some early best practices for development of Web services
- ▶ Troubleshooting and debugging
- ▶ Current limitations of the Web service wizard in the Application Developer beta code

Architecture

This section contains (some) answers to the following questions:

- ▶ What are the typical usage scenarios for Web services?
- ▶ What is the appropriate architectural positioning of Web service technology?
- ▶ On which tiers of a multi-tiered architecture can Web Service technology be exploited?
- ▶ How do Web services and the model-view-controller design pattern relate to each other?
- ▶ How do Web services fit into an overall J2EE solution architecture?
- ▶ What is the impact on the operational model of your Web application?
- ▶ Are there common runtime patterns?
- ▶ How does Web services technology compare to other distributed programming approaches?
- ▶ When and how to apply Web services technology to my project/existing IT system?
- ▶ How to get started?

What are common usage scenarios for Web services?

We expect solutions from the following problem domains to be among the early adopters of the SOA:

Enterprise application integration (EAI)

- ▶ Web services give the architects of EAI solutions an additional design alternative. The message oriented SOAP communication style can be very useful to loosely couple existing IT systems (such as content management systems).

Portals and marketplaces

- ▶ Due to the hub architecture of such solutions, there is a strong need for non browser clients to interact with Web sites. For example, the catalog upload from a supplier to a procurement marketplace can be implemented as a Web service.

Business to business (B2B)

- ▶ Any B2B solution that need systems to interact with each other is suited for using the SOA. An automated supply chain management for the automobile industry would be an example.

Any solution using fat or thick clients

- A non HTML client that needs to communicate with an application server can use SOAP rather than RMI/IOP, or other binary protocols to do so.

COM to Java bridge

- The SOA is language neutral; there are bindings both for Microsoft COM technology, as well as for Java 2 Enterprise Edition (J2EE). Hence, *mediator* Web services connecting these two worlds can be defined. Any Windows based desktop application can be Web enabled this way.

Peer-to-peer computing is one of the disciplines the Web service technology promises to deliver previously unavailable building blocks for.

What is the appropriate architectural positioning of Web services?

From an overall point of view, Web services are an additional architecture element. They need to fit into existing enterprise and solution architectures if existing systems are to be leveraged.

Let us first take a look at a classical multi-tiered Web application architecture (Figure 15-1).

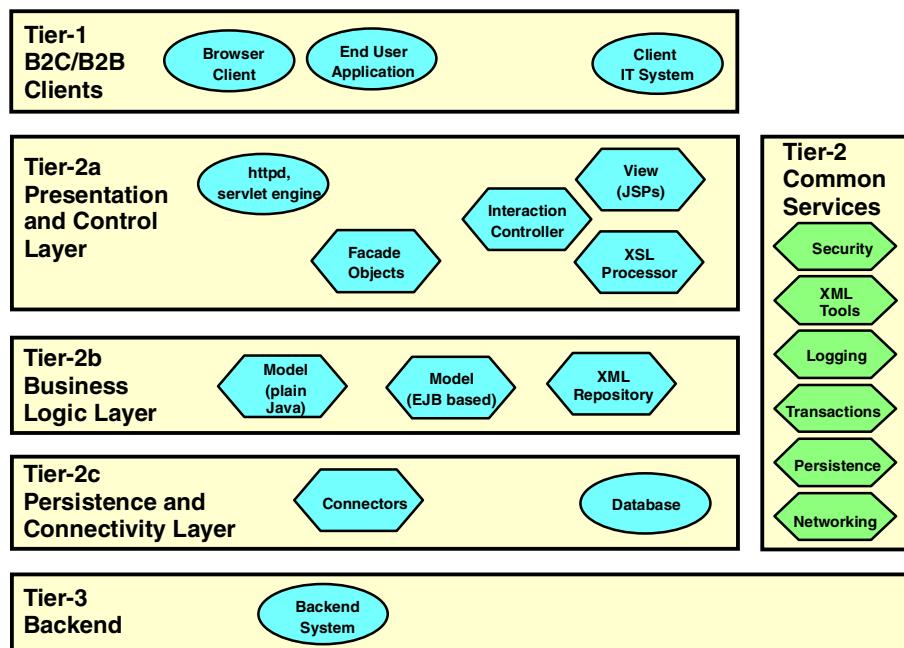


Figure 15-1 Canonical B2C/B2B architecture without Web services

Tier-1 comprises clients such as browsers, other end user applications, and other IT systems. The middle tier-2 is application sever supported; according to the principle *separation of concerns*, there are different layers for

- ▶ Presentation/control functions
- ▶ Business logic
- ▶ Connectivity and persistence
- ▶ Miscellaneous support functions

Tier-3 groups any existing systems that have to be integrated.

Together these architecture elements form the conceptual J2EE architecture; there are no Web services involved yet.

On which tiers of a multi-tiered architecture can Web services be exploited?

- ▶ In a 1+3+1 tiered system (as outlined in Figure 15-1) the SOA service requestor is either part of the client tier, or resides on the lower connectivity layer of the middle tier. The service provider s located in the upper controller layer (just as any other servlet).

Figure 15-2 outlines this positioning.

How do Web services and J2EE relate to each other?

- ▶ Web services provide a *description of and access to* server side functionality; J2EE is *one* implementation alternative for the Web service requestor and provider. Java is a good choice due to the availability of a wide range of APIs such as for XML, SOAP, UDDI, WSDL. The power of the open source community working with Java benefits this approach.
- ▶ The SOAP RPC router servlet fits nicely into any Web application server based architecture.
- ▶ Note that the tier 2-b business logic can, but does not have to be EJB based.
- ▶ One option for connecting to the back-end is the Java connector architecture. A Java interface is all you need. It is straightforward to Web service enable your model or connectors (see Figure 15-2).

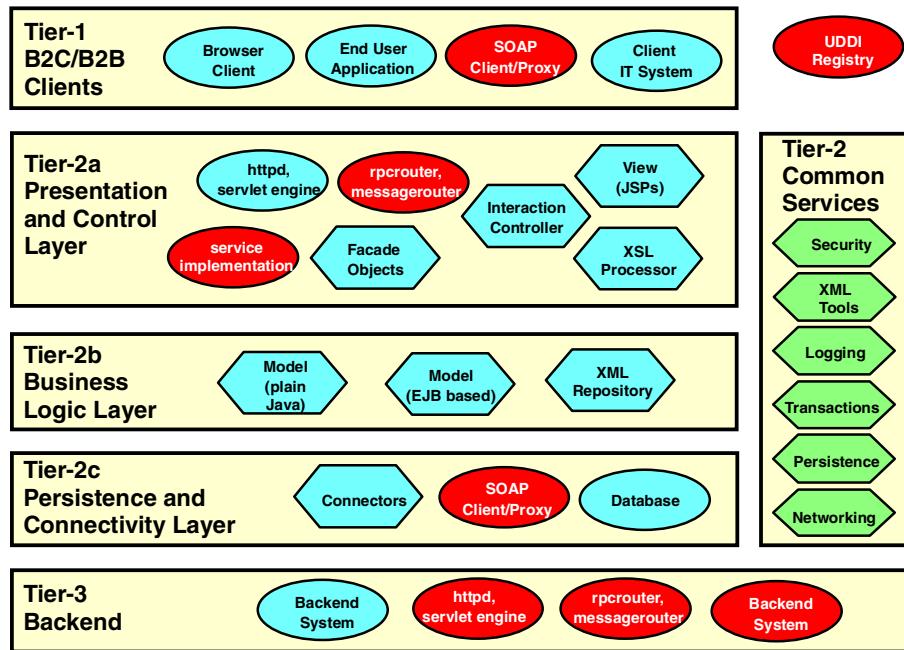


Figure 15-2 SOA element positioning into an overall solution architecture

How do Web services and the model-view-controller (MVC) design pattern fit?

- ▶ The MVC pattern can nicely be applied to the provider side of the SOA: the SOAP server acts as controller, the server side Web service implementation (and custom mappers if existing) provide(s) an XML view of the business data. The model remains unchanged, with the Web service implementation class acting as a facade to it.
- ▶ The requestor side of the SOA (SOAP client and proxy) implement the proxy or wrapper pattern.

Are there any additional architectural patterns?

This topic currently receives a lot of attention in the industry. Here are two examples:

- ▶ The SOAP message communication style is suited for document oriented message exchange. For instance, self-describing queries become possible; the client does not have to be changed when server upgrades.
- ▶ Peer-to-peer computing and decentralized networking over the Internet

Check the developerWorks Web services zone for articles on this topic.

What is the impact on the operational model?

The main focus of this book is on development aspects; however, here are some first thoughts on the runtime aspects.

Firewall considerations

- ▶ The server side RPC router is just another servlet that is accessed through HTTP; an additional HTTPS connection to an external system, the UDDI registry, might be required in order to be able to publish a service.
- ▶ A service requestor is just another HTTP client (assuming that HTTP is used as transport protocol).

Load balancing considerations

- ▶ The same considerations and techniques as for normal servlets apply, with one exception.
By default, there is no hot failover for *session* type Web services. If failover is a requirement, it should be handled on the next layer. For example, in a servlet acting as Web service implementation, or by using a redundantly configured EJB session facade.
- ▶ A stateless *request* style Web service instance can be pooled and load balanced because it is a Java object in the JVM of the application server. Failover is also supported for *request* style Web services.
- ▶ If *session* style is used, scalability is best. However, the Web service implementation class must be serializable and small.
- ▶ If *application* style is used, load balancing is not an issue, because there is only one instance whose lifetime is identical to the one of the application server hosting it. However, the Web service implementation class must be thread safe.

Are there common runtime patterns?

- ▶ Yes, as we have described in this book, service requestors can either be static, or provider dynamic or type dynamic, depending on whether they use a straight static SOAP connection, access UDDI at build or deploy time, and UDDI access at runtime. Refer to our discussion on advanced UDDI topics (see “Advanced UDDI topics” on page 489).
- ▶ For more information check the IBM Patterns for e-business Web site at:
<http://www.ibm.com/software/developer/web/patterns>

How do Web services relate to similar technologies?

Web services are often compared to classical remote procedure calls (RPCs) and distributed computing frameworks such as CORBA, DCOM, and J2EE. However, Web services are *not* just another distributed computing or RPC mechanism. Let us start with some of the aspects we covered in this book:

- ▶ The Web service are self-explaining, the meta-information travels with the messages. Web services are dynamic, standards-based, composable, and language independent.
- ▶ There is support for document-oriented messaging support.
- ▶ Web services leverage a proven technological foundation: XML is a standard for portable data exchange. HTTP POST is already in wide use as a B2B communication mechanism. The message payload serialization is typically either based on XML, or a simple, solution specific string syntax.
- ▶ Web services complement the CORBA, COM, EJBs programming models and do not compete with them. Web services provide universal access, CORBA, COM, and J2EE are implementation alternatives.

Table 15-1 gives an overview of some important differences between Web services and the other technologies. (J2EE uses RMI/IOP as transport protocol, so there is no J2EE column in the table.)

Table 15-1 Differences between distributed computing technologies

Technology	DCOM	CORBA	Web services
Endpoint naming	OBJREF	Interoperable object reference (IOR)	URL/URN
Interfaces	Single	Multiple	Multiple - WSDL
Payload type	Binary	Binary	Text
Payload parameter value format	Network data representation (DR)	Common data representation (CDR)	XML
Server address resolution	(Directory)	Naming service	IP routing, URN
Message dispatch	ID based	ID based	Namespace and parameter types
Client/server coupling	Tight	Tight	Loose

Dan Gisolfi discusses this issue in the third part of his series of articles called *Web services architect* on developerWorks:

<http://www.ibm.com/developerworks/webservices/library/ws-arc3/>

What about security?

In the open, dynamic environment that we are envisioning with the service oriented architecture (SOA), an end-to-end view on the security issue is required. Security requirements include authentication, authorization, integrity, privacy, auditing, and non repudiation.

Let us have a word on the port 80 discussion. If we simply say *with SOAP client systems can access Web applications through the same port as browsers* we are missing the point. The SOA concept is *not* about sneaking through an already open port—firewalls are there for a reason, and your security administrator will probably not approve such an approach.

As a first step, a separate port such (81, for instance) can be used to distinguish from user-oriented HTML communication. The key difference is that the protocol the SOA uses is text based. The firewall can therefore inspect the messages (if they are not encrypted). A new type of application level gateways performing such checks could evolve.

But this step does not solve the problem; as a matter of fact, the security requirements have to be addressed on every layer of the SOA protocol stack. Web service security is a partially open issue; standard and products implementing them are expected to be available in the near future.

As of today, there following architecture alternatives exist:

- ▶ Leverage transport layer security, for example:
 - HTTPS ensuring integrity and privacy
 - HTTP basic authentication headers
- ▶ The features of your reverse proxy, Web server and WebSphere application server can be fully leveraged because SOAP server (rpccrouter) is just another servlet. For example, Tivoli Policy Director or WebSphere security can act as authorization engines.
- ▶ Enhancing the SOAP message layer with security features:
 - XML signatures, addressing the integrity requirement
 - XML encryption, addressing the privacy requirement

Additional SOAP headers can be used to transport the required security elements such as credentials and signatures. Both XML signatures and encryption are forthcoming W3C standards.

The Web services toolkit (WSTK) contains both demos and supporting information on them:

<http://www.alphaworks.ibm.com/tech/webservicestoolkit>

An implementation of XML signatures and XML encryption, the IBM XML Security Suite, is available at IBM alphaWorks:

<http://www.alphaworks.ibm.com/tech/xmlsecuritysuite>

What about management and quality of service?

Just like any other component in a distributed system, Web service implementations and the infrastructure hosting them must be managed. Security, fault, configuration, and performance management are some of the key disciplines. Other aspects are technical and business process level logging.

Because a SOAP server essentially is a servlet based Web application running in an application server such as IBM WebSphere, already existing system management procedures can be fully leveraged.

Security aspects are discussed separately (see “What about security?” on page 520).

In WebSphere, there is an *m:m:m* relationship between the application server, its Web applications, and the servlets in the Web application, with the rpcrouter and the messagerouter being two of them. There is a *1:1* relationship between the Web application and each of these router servlets. The process running the JVM is the application server. Fault and performance management should therefore be carried out on this level.

From a deployment point of view, the SOAP server components and the Web service implementation are bundled as a .war file, which in turn is part of a .ear file. Therefore, configuration management tasks such as installing new versions of .jar files should be carried out on the application server level as well.

In addition to these WebSphere capabilities, the SOAP server has a simple management interface, with which you can start and stop existing Web services and deploy additional services. The configuration of the server is file based. This can be useful if local properties of the Web service implementation change. For example, you could switch to using a different database without having to shut down the application server.

A group of Web services supporting the same business process should be deployed in the same Web application. Note that each of them must have a different URN.

Technical logging for a service implementation can be handled in the same way as for any other Java Web application. It is worth noting, however, that the internal tracing and logging capabilities of the Apache SOAP implementation are very limited. It is always possible, however, to use a non-GUI version of a TCP/IP tunnel to intercept the message flow in order to record the client/server interactions. The UDDI and WSDL APIs offer some more support; for example, there is a debug flag in the `wstkProperties.xml` configuration file.

Business logging of Web services translates into accounting and billing of Web service invocation. A first proposal is described in the following article on developerWorks:

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

On the requestor side, the SOAP API and the proxy on top of it are just another communication component. We expect them to fit nicely into any client side management policy.

What about performance?

A frequently articulated concern regarding the SOA is that a string-based protocol has a high overhead in terms of bandwidth and CPU requirements. Meta information such as XML tags and namespace prefixes adds to the problem.

On the other hand, a text based protocol such as SOAP is not necessarily slower than binary ones, taking into account that client stubs supporting binary protocols such as IIOP or DCOM might have to be downloaded and initialized before the message exchange can start.

There is room for both technologies; in a heterogeneous world such as the Internet, however, an open, interoperable protocol such as SOAP has clear advantages. As usually, the solution to be preferred in a concrete project situation depends on the actual functional and nonfunctional requirements (the expected message exchange patterns, for example). A proof of technology (PoT) can help to gain an understanding whether Web services are the right fit for your environment.

It should be noted that all server side Java and servlet performance optimization and load balancing techniques are available.

Graham Glass touches this issue in the following developerWorks article:

<http://www.ibm.com/developerworks/webservices/library/ws-peer3/?dwzone=ws>

What about interoperability?

The SOAP protocol is designed to be open and interoperable, the W3C owns its specification. Interoperability has already been proven in real world projects.

Regarding Apache SOAP 2.2 and Microsoft's .NET SOAP, the two protocol implementations are generally interoperable. One known problem is that the Microsoft implementation does not always send all type information Apache expects. A patch from James Snell is available. Consult the Apache SOAP FAQ list for more information.

Apache SOAP is an open source project, which makes it easier to apply patches to the SOAP runtime, in case unexpected problems occur anyway.

In regard to WSDL support, a few minor differences between the IBM and the Microsoft toolkits exist as of today.

IBM's jStart team (see Chapter 16, “IBM’s jStart program” on page 531) has successfully demonstrated interoperability between the IBM and the Microsoft toolkits in several customer projects. For more information about interoperability issues, refer to:

- ▶ Apache: <http://xml.apache.org/soap/index.html>
- ▶ Yahoo discussion group: <http://groups.yahoo.com/groups/soapbuilders>
- ▶ IBM developer site:
<http://www-106.ibm.com/developerworks/library/ws-ref3/?n-ws-5241>
- ▶ MSDN on SOAP interoperability:
<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnsoap/htm1/soapinteropbknd.asp>

Are there any gaps in the current version of the SOA?

Web services are emerging technology. Therefore, it is not surprising that not all aspects are already fully covered by the standards. Reliability and support for distributed transactions are two important areas that fall in this category. We recommend that you follow the ongoing discussion about these topics on forums such as developerWorks.

Reliable messaging

Take a look at the following paper presenting a preview of how a reliable HTTP protocol could look like:

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

Transactional Web services

The ACID properties (*atomicity, consistency, isolation, durability*) and two phase commit (2PC) capabilities provided by existing transaction monitors, cannot yet be fully exposed on the Web service level (note that it is possible to initiate a transaction, however).

The XMA specification, an XML version of the X/Open XA interface for resource managers, promises to standardize how distributed transactions involving multiple Web service invocation should be handled. The SOAP header can be used to pass transactional context information from the requestor to the provider

How do I get started in my project?

If you are considering applying the Web service technology to your project, you might wonder which elements of the Web services technology you should apply first and what process to follow.

A good starting point is XML. Try to define an XML representation of your business data. It is even better if such an XML vocabulary already exists. For this step, it is helpful if a metadata repository such as an UML or IDL specification of entities and business processes is already available. An already existing HTTP POST-based proprietary communication protocol is a good foundation as well.

In general, you should familiarize yourself with the technology in a proof of concept project (POC).

In parallel, gain an understanding of your nonfunctional requirements, such as security, performance, and so forth. This is a general recommendation; however, it becomes more important when opening up your systems to the Web.

The IBM jStart team is available to support you during your first steps (see Chapter 16, “IBM’s jStart program” on page 531):

<http://www.ibm.com/software/jStart>

Design

In this section, we discuss some design aspects that developers of real world Web services might be confronted with.

Are there recommended design patterns for developing dynamic Web services?

- ▶ We expect that almost all business applications acting a Web services requestors will bind either statically or provider-dynamically; only generic tools such as IDEs and UDDI browsers will be truly type-dynamic.

How to deal with complex data types (for example, lists)?

- ▶ Note that simple JavaBeans are supported by the SOAP encoding.

However, if the standard SOAP encoding does not support one of the data types that you want to expose as a parameter of a Web service, your design alternatives are the literal XML encoding or custom mappings.

- ▶ Vector and array data types are supported in the final Application Developer product. For more complex data types, either custom mappings or sequential XML elements, can be used.
- ▶ Consider defining your own WSDL and/or SOAP client proxy generation tools only after having gained some experience with the SOA.

How to treat binary data, for example an image?

- ▶ The binary information can be pointed to only; a separate HTTP request can then be used to actually retrieve it. Another alternative would be to define an opaque XML any element. We recommend the first alternative.

How can asynchronous communication be implemented?

- ▶ There are two design alternatives: you can use the message/document SOAP style, or roll your own mechanism residing on top of RPC style.

How do object-oriented concepts map to the service oriented architecture?

- ▶ Web services do provide encapsulation. As a matter of fact, even the implementation language of a service is transparent to the client, not just the data members.
- ▶ The instantiation policy is configurable (*application, request, session*).
- ▶ There is no inheritance concept in WSDL, therefore, Web services can be categorized as *object based*. An existing Java inheritance hierarchy can be flattened before a Web service interface is defined, however. This is a manual step, because the Application Developer Web service wizard does not interrogate any base classes.

- ▶ Regarding associations between instances, Web services are inherently stateless, and SOAP by design does not support *call by reference*. All data members are copies (*call by value*).
- ▶ Support for associations can be implemented in a project specific wrapper layer. References to other Web services are held as URN and network address pairs, the Web service session mode is used on the server side in order to store such references. An alternative solution can transmit application level references, such as abstract representation of EJB handles converted into XML format.
- ▶ Web services are inherently polymorphic; one Java class can implement several WSDL port types and vice versa.

Which development and binding styles fit to my project environment?

- ▶ In most cases, bottom-up static is the best starting point. Even if you start from scratch, code a simple JavaBean as a starting point.
- ▶ Do not write type-dynamic requestors unless you have to.

How do I access transport specific information on the server?

- ▶ For example, you might want to access to the HTTP header in order to inspect user ID, password, and proxy settings.
- ▶ When you use the RPC communication style, there is no equivalent to the client side SOAPHttpConnection. The SOAP headers are not accessible for the service implementation either.
- ▶ The developer of a message oriented Web service has more control over the incoming SOAP parameters including the transport settings.
- ▶ Apache SOAP also offers transport hooks and the pluggable provider interface that can be used to implement an additional SOAP to Java interface.

What if I have a SOAP intermediate, but want to use SSL end-to-end encryption?

- ▶ This is an open issue; we expect standards covering SOAP level security issues to come out in the near future. XML level encryption can be a design alternative.

When should I prefer the message oriented communication style to the RPC mode?

- ▶ Refer to “Advanced SOAP programming” on page 448.

What is the best way to implement a list-details oriented Web application using Web services?

For example, a Web service client might be a client, rather than an EJB or a database connection, in order to access its business data?

- ▶ This design issue is essentially the same as for any other backend connection. You should define two methods in your Web service, and get only the minimal data set you need for the list display in the first place. When the user clicks on a list entry, the second Web service method then retrieves the full data set needed for the details view.
- ▶ Under some circumstances, it can make sense to use session scope and hold the connection handle in there (a pointer to a stateful session bean, for example). Be careful with this approach, though (as with stateful session beans in general): there is no failover support, and the load balancing capabilities are limited.

How do I connect my Java program to a COM object?

- ▶ The SOA gives you two additional design alternatives. You can either use the pluggable COM provider that comes with the Apache SOAP server, or connect a Java SOAP client to a Microsoft SOAP server, which in turn calls the COM object.

Early best practices

During the development of the auto parts application, we identified the following guidelines and recommendations, which we expect apply to many Web service development efforts. Please note that this is a collection of *early* best practices; they are likely to change as standards and implementations evolve.

Do not assume a certain client environment

- ▶ Web services are about openness and interoperability, so try to avoid compromising it.

Carefully decide which method to expose as Web service operations

- ▶ Resist the temptation to Web service enable all your micro-level OO method calls. A lot of programmers making the transition from general OO to CORBA development made that mistake.

Keep the method signatures simple and short

- ▶ The method signatures are exposed as operations, messages, and message parts in the WSDL specification. If small parameter lists are used, the marshalling and unmarshalling overhead is minimized.

- ▶ If throughput is a concern, send small amounts of data per request; fragment larger entities is needed.

Implement custom mappings only if there is no other design alternative

- ▶ Design encoding style and mapping carefully, stay with the defaults as long as possible.
- ▶ Writing and testing custom mappers is a nontrivial effort. Client and server easily become tightly coupled, which decreases the openness and extensibility of your solution.

Keep method names, parameter names, XML tag names, and namespace prefixes short

- ▶ These identifiers travel with the message payload at runtime.

Reuse and/or pool SOAPMappingRegistry objects

- ▶ The creation of SOAPMappingRegistry instances is an expensive operation. These objects can easily be pooled.

Be as stateless as possible

- ▶ Use the *application* style whenever possible. Switch to *request* if you have to allocate certain resources each time a Web service is invoked. Use *session* only if you use case has stateful conversation semantics.

Apply the general Web development best practices

- ▶ All best practices for servlet and JSP development are still applicable. For example, you should keep the data content of your HTTP session small.
- ▶ Regarding a Web service implementation of *session* style, this general recommendation translates into the recommendation that instance variables of a Web service implementation class should only carry small amounts of data.
- ▶ Thread safety is likely to be required because the RPC router is executing as a servlet. Use Java synchronization.

Apply an iterative and incremental development style

- ▶ This is a general recommendation. It not extraordinarily well supported by the quite monolithic Web service wizards, however.
- ▶ You might want to rename your Web service implementation class frequently. This can be useful when you want to preserve the original version and you do not want to go to the versioning system all the time.
- ▶ Do not forget to cleanup any temporary UDDI registrations and other generated artifacts from time to time.

Apply UDDI best practices

- ▶ UDDI Version 1, which is currently in use in Application Developer, WSTK, and the IBM WebSphere UDDI registry), does not support global search operation; you always need to locate the service provider business entity when searching for a certain service.
- ▶ For more information regarding UDDI related best practices, see <http://www.uddi.org>.

Troubleshooting

This section is about potential pitfalls and frequently made mistakes:

- ▶ Configuration errors
 - Invalid URN name
 - Method signature not found by SOAP server
 - No marshalling/unmarshalling (de) serializer found in mapping registry
- ▶ Invalid proxy settings
 - ▶ You need to install a socks client when you are behind a firewall and want to connect to an external service broker or service provider
- ▶ XML namespace conflicts
- ▶ Bean properties violated (runtime errors)
 - No public constructor, getter/setter missing
 - Vector, hash table used (currently not supported)
 - See README as well
- ▶ UDDI find methods return strange results
 - SP/BE accessed first? Global searches do not work.
- ▶ SOAP server faults:
 - Server not running
 - Service not deployed or stopped
 - No matching parameter signature found
 - Namespace conflict
 - Deserializer not found
- ▶ SOAP client faults
 - Wrong URN used
 - Invalid soap router servlet URL
 - No serializer found
 - S0APMappingRegistry not populated right
- ▶ Check Apache FAQ and troubleshooting information.

Tools for troubleshooting

- ▶ TCP/IP tunnel
- ▶ Application Developer debugger
- ▶ SOAP admin GUI



IBM's jStart program

IBM's jStart program consists of a world wide team that is focused on helping customers and partners embrace new and emerging software technologies—and we are ready to help you get started right away! We focus on your business problems and suggest solutions that can be implemented quickly, as we build and grow a unique partnership that delivers concrete proof points utilizing exciting new technology.

Have you heard of Web services, UDDI, SOAP, or WSDL? If so, we are ready to **jump-Start** your companies adoption of dynamic e-business utilizing these key technologies. How about Java, XML, and/or PvC? These technologies have been in our portfolio over the past one to four years, and provide a strong foundation for our current emerging technology focus.

Put the industries most advanced software tools and techniques, the most sophisticated e-business technology, and one of the most capable development teams to work for you. Start today by choosing jStart for your business.

jStart provides the edge

If you want to add the latest, most advanced dynamic e-business technologies to your strategic applications, the IBM jStart program is ready to help. We are an experienced, highly capable group of experts who will work with you to dramatically shorten your time to market and get a jump on your competition.

The jStart team partners with you to help you use software technologies to deliver the high-quality business-critical solutions you need now. We will help you add the power of XML, SOAP, WSDL, pervasive technologies and UDDI - plus IBM's powerful arsenal of software tools such as WebSphere Application Server, the Web services toolkit, MQ Series, and UDDI4J to your existing IT assets.

We deliver complete solutions as we mentor you and your staff in the use of sophisticated tools and technologies. By partnering with jStart to develop and deploy a dynamic e-business project, you and your staff will realize the benefits of our world class program and skills.

About the program

Let us introduce ourselves:

Who we are

We are a highly experienced and uniquely capable team of special purpose experts in emerging technologies. jStart engagement managers are ready to help any organization—from the biggest multi-national to the five-person start-up in any industry ranging from finance and insurance to manufacturing and distribution—gain an edge in today's market, as we help you develop the strategic applications you need now.

How we work

You will notice a big difference when you work with the IBM jStart team. Unlike many other application solutions providers, with jStart there are no ponderous steering committees to deal with. We do not put a lot of layers between the strategic work that needs to get done. We are hands-on professionals who are driven to help you quickly achieve your e-business objectives.

What we do

We focus on helping you create flexible and scalable solutions using emerging software technologies. Tell us your requirements and objectives—then let us help you address your tactical and strategic challenges, as we deploy new applications that meet your real world business needs.

Where competitive advantage begins

Gaining a business advantage today requires more than engineering innovation, manufacturing capability, distribution efficiency, or market share. The edge goes to companies that can leverage and deploy the latest and most advanced e-business technologies faster and more effectively than the competition.

Hours of planning sessions do not make a difference in your project until the solutions you envision are designed, developed, and working for you in the marketplace. You have to move quickly from merely talking about e-business to actually doing e-business.

By engaging with jStart to move quickly, you will realize the benefits from a time-to-market standpoint, as we work with you to rapidly deploy e-business applications. You benefit because your engaged early in the technology cycle and you pave the way for future projects and technology adoption. The jStart engagement model is a proven approach for quickly moving from talking about e-business to doing e-business.

It is time to jump start your technology initiatives. Do so by bringing in a proven, experienced, cutting edge partner: the IBM jStart emerging technologies team.

jStart engagement model

Our five step engagement model is designed for rapid results (Figure 16-1).

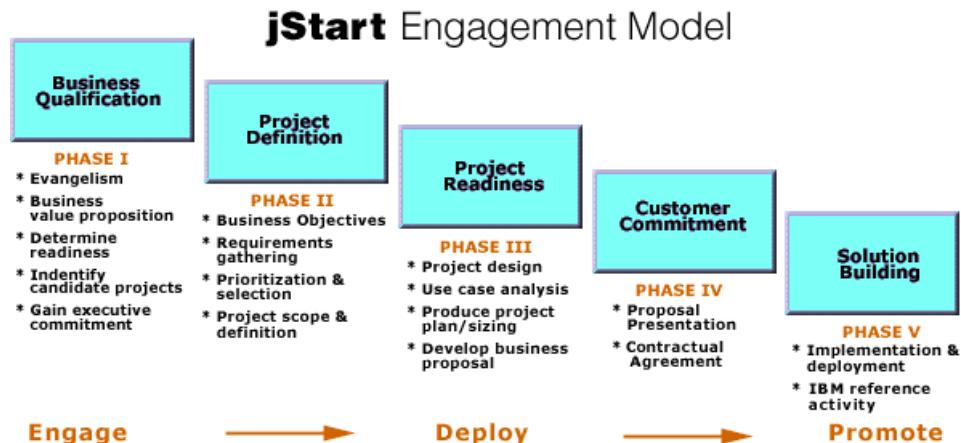


Figure 16-1 jStart engagement model

- ▶ **Business qualification** is the first step. We build a business relationship, identify candidate projects, determine readiness, and establish executive support.
- ▶ Step two is a **project definition workshop**. We gather information and requirements related to potential projects, determine business objectives, validate and prioritize the opportunities with your staff, then move towards selecting an initial project and defining its scope.
- ▶ Our unique **project readiness workshop** is the third step in the process. Working together, we produce a comprehensive plan and a detailed proposal.
- ▶ In the fourth step, we seek your **commitment** as we present a contract for review and approval.
- ▶ The final step is **solution building** which takes you through project implementation and deployment. In many cases, the implementation phase takes just 90 days or less—which means your solution is up and running, not just up and coming.

Take the step

To learn more about the jStart program and to get details on actual adoption scenarios for Web services, please visit the Web site:

<http://www.ibm.com/software/ebusiness/jstart/>

Explore our site to learn more about the range of capabilities we offer, the solutions that we can provide, our experience, and customer examples—then be sure to contact us in order to take the next step. To contact the jStart team directly, send an E-mail to:

<mailto:jstart@us.ibm.com>



Part 4

Appendices



Product installation and configuration

In this appendix, we describe the installation of the products we used throughout this book including:

- ▶ DB2 7.2
- ▶ WebSphere Application Server 4.0 Advanced Edition Single Server (AEs)
- ▶ WebSphere Application Server 4.0 Advanced (AE)
- ▶ WebSphere Studio Application Developer
- ▶ IBM WebSphere UUDI Registry

We installed these products on top of Microsoft Windows 2000 Professional, Service Pack 2. As an alternative, you can install the products on top of Windows NT 4.0 workstation (or server) with Fixpack 6a.

DB2 Version 7.2

The installation procedure for DB2 Enterprise Edition is as follows:

- ▶ Select DB2 Enterprise Edition.
- ▶ Select DB2 Application Development Client.
- ▶ Select custom install with:
 - Communication protocols
 - Stored Procedure Builder
 - Sample Application
 - Documentation (remove Query Patroller, Warehouse, OLAP)
 - Administration/Configuration tools
 - Getting Started
- ▶ Directory: d:\SQLLIB
- ▶ Create DB2 instance.
 - User ID/password: *db2admin/db2admin*

Alternatively, you can install DB2 7.1 FixPack 3 with the same options.

Create sample database

Run *First Steps* after reboot, and create the SAMPLE database to make sure you correctly installed DB2.

JDBC Version 2

You have to enable JDBC 2.0 because access to DB2 using DataSources only works with JDBC 2.0. To enable JDBC 2.0, do the following:

- ▶ Stop all DB2 processes from the Services list.
- ▶ Change to JDBC 2.0 by running d:\SQLLIB\java12\usejdbc2.bat
- ▶ Restart DB2 processes.

WebSphere Application Server Advanced Single Server

To install WebSphere AEs, choose the following options:

- ▶ Language: English
- ▶ Custom installation, select all options.
- ▶ User ID: use your Windows user ID (with admin authority)
- ▶ Directory: d:\WebSphere\AppServer, d:\IBM HTTP Server

Verification

Check that the lines listed here are added at the end of the IBM HTTP Server configuration file (d:\IBM HTTP Server\conf\http.conf):

```
LoadModule ibm_app_server_http_module
D:/WebSphere/AppServer/bin/mod_ibm_app_server_http.dll
Alias /IBMWebAS/ "D:/WebSphere/AppServer/web/"
Alias /WSsamples "D:/WebSphere/AppServer/WSsamples/"
WebSpherePluginConfig D:/WebSphere/AppServer/config/plugin-cfg.xml
```

WebSphere Application Server Advanced

To install WebSphere AE, choose the following options:

- ▶ Language: English
- ▶ Custom installation, select all options.
- ▶ User ID: use your Windows user ID (with admin authority)
- ▶ User ID for DB2 database access: *db2admin* (or whatever ID was used for DB2 installation)
- ▶ Directory: d:\WebSphere\AppServer, d:\IBM HTTP Server

Attention: If you want to install WebSphere AE and AEs on the same machine, you have to specify different directories, for example:

d:\WebSphere\AppServer	<== Advanced Edition
d:\WebSphere\AppServerAEs	<== Single Server edition

WebSphere AE and AEs cannot share the same HTTP configuration file (httpd.conf), so you have to keep a separate copy for AE and AEs (such as httpdae.conf and httpdaes.conf). You then have to start the HTTP Server on the command line, specifying the right configuration file:

```
apache -file httpdae.conf
```

WebSphere Studio Application Developer

Install the beta code of the WebSphere Studio Application Developer with the following installation procedure:

- ▶ Directory d:\WSAD
- ▶ Select any primary user role you want. This is not really important because you can change it afterwards using the *Window -> Preferences* menu.

Attention: You have to select if you want to use CVS or ClearCase LT (or nothing) as the team repository.

We suggest that you select CVS for single workstation usage.

IBM Agent Controller

If you want to install the IBM Agent Controller separately on a target machine, run the file IBM Agent Controller.msi located in the installation directory of Application Developer (or run the setup.exe in the RAC_install subdirectory).

IBM WebSphere UDDI Registry

IBM provides a stand-alone UDDI registry product. Download the IBM WebSphere UDDI Registry code from:

<http://www7b.boulder.ibm.com/wsdd/downloads/UDDIregistry.html>

You have to register to download the code.

The IBM WebSphere UDDI Registry can be used instead of the IBM Test Registry for Web services publishing. This is especially useful for testing on a machine that is not connected to the Internet.

The IBM WebSphere UDDI Registry is not a complete product at the time of the writing of this book:

- ▶ Originally we used the IBM WebSphere UDDI Registry Preview code on Windows, which worked reasonably well with the Application Developer.
- ▶ When we updated the book in February 2002, the preview code was not available any more for Windows, and the beta code for Windows did not work good enough with the Application Developer.
- ▶ In March 2002, IBM WebSphere Registry Version 1.1 became available. It supports UDDI Version 2, whereas Application Developer is based on UDDI Version 1, and not all interfaces are working.
- ▶ You can try to download whatever code is available for the Windows platform and try it out with the Application Developer.
 - Follow the installation instructions that come with the downloaded code.
 - Connect to the registry from the UDDI Explorer and try out find and publish functions.

See also the comments in “IBM WebSphere UDDI Registry” on page 496.

Define and load the ITSOWSAD database

A command stream to define and load the ITSOWSAD database is provided in the sampcode\Setup\DDL directory of the sample code (see “Using the Web material” on page 570).

Open a DB2 command window and run this command to load the database:

```
db2 -tf itsowsad.ddl
```

Sample data

The ITSOWSAD database has four sets of two tables, PARTS and INVENTORY:

- ▶ The Almaden Autos tables are AAPARTS and AAINVENTORY
- ▶ The Mighty Motors tables are MMPARTS and MMINVENTORY
- ▶ The Plenty Parts tables are PPPARTS and PPINVENTORY
- ▶ The Santa Cruz Sports Cars tables are SSPART ad SSINVENTORY

AAPARTS and MMPARTS tables

PARTNUMBER	NAME	WEIGHT	DESCRIPTION
M100000001	CR-MIRROR-L-01	10.50	Large drivers side mirror for Cruiser
M100000002	CR-MIRROR-R-01	10.80	Large passenger side mirror for Cruiser
M100000003	CR-MIRROR-R-01	4.60	Large rear view mirror for Cruiser
W111111111	WIPER-BLADE	0.90	Wiper blade for any car
B222222222	BRAKE-CYLINDER	2.20	Brake master cylinder
X333333333	TIMING-BELT	0.60	Timing belt
T0	Team	100.00	International WSAD Groupies
T1	Olaf	100.11	German Turbo Engine
T2	Wouter	100.22	Belgium Chocolate Steering Wheel
T3	Denise	100.33	US Spark Plug
T4	Mark	100.44	British Muffler
T5	Ueli	100.55	Swiss Cheese Cylinder
L1	License	0.30	Personalized license plate

PPPARTS and SSPARTS tables

PARTNUMBER	NAME	WEIGHT	DESCRIPTION
M100000001	CR-MIRROR-L-01	10.50	Large drivers side mirror for Cruiser
M100000002	CR-MIRROR-R-01	10.80	Large passenger side mirror for Cruiser
M100000003	CR-MIRROR-R-01	4.60	Large rear view mirror for Cruiser
W111111111	WIPER-BLADE	0.90	Wiper blade for any car
B222222222	BRAKE-CYLINDER	2.20	Brake master cylinder
X333333333	TIMING-BELT	0.60	Timing belt

AAINVENTORY table

ITEMNUMBER	PARTNUMBER	QUANTITY	COST	SHELF	LOCATION
21000001	M100000001	10	89.99	2A	AA - Almaden
21000002	M100000002	5	99.99	2B	AA - Almaden
800	T0	1	99.00	A0	AA - Almaden
801	T1	1	11.00	A1	AA - Almaden
802	T2	1	22.00	A2	AA - Almaden
803	T3	1	33.00	A3	AA - Almaden
804	T4	1	44.00	A4	AA - Almaden
805	T5	1	55.00	A5	AA - Almaden
810	L1	1	30.00	A6	AA - Almaden

MMINVENTORY table

ITEMNUMBER	PARTNUMBER	QUANTITY	COST	SHELF	LOCATION
21000003	M100000003	10	59.99	L8	MM - San Francisco
21000004	M100000003	12	57.99	B7	MM - New York
31000005	W111111111	2	12.34	H8	MM - Los Angeles
900	T0	1	99.00	M0	MM - San Jose
901	T1	1	11.00	M1	MM - Heidelberg
902	T2	1	22.00	M2	MM - Brussels
903	T3	1	33.00	M3	MM - Raleigh
904	T4	1	44.00	M4	MM - London
905	T5	1	55.00	M5	MM - Zurich

PPINVENTORY table

ITEMNUMBER	PARTNUMBER	QUANTITY	COST	SHELF	LOCATION
21000001	M100000001	99	99.99	X1	PP - Heidelberg
21000003	M100000003	42	49.99	01	PP - Heidelberg
31000006	B222222222	13	123.45	E5	PP - Frankfurt

SSINVENTORY table

ITEMNUMBER	PARTNUMBER	QUANTITY	COST	SHELF	LOCATION
21000001	M100000001	99	89.99	2A	SS - Santa Cruz
21000002	X333333333	7	12.34	2D	SS - Santa Cruz



B

State transitions for resources in Application Developer

In this appendix, we describe how resources transition between states when using the team development environment described in Chapter 7, “Working in a team” on page 207:

- ▶ State transitions for a stream
- ▶ State transitions for a project
- ▶ State transitions for a file

Stream lifecycle

In this section we illustrate the lifecycle of a stream in the repository, and how to transition between states.

State transition diagram

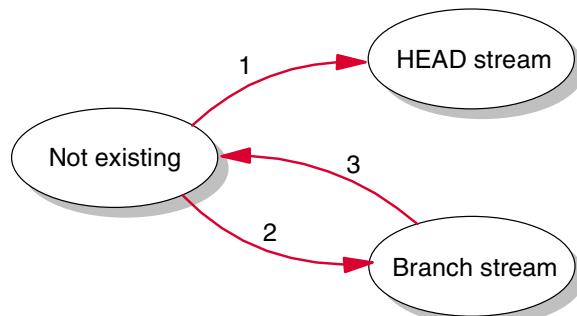


Figure 16-2 Valid state transitions for a stream

These transitions are described below, and described in the subsequent sessions:

1. Not existing -> HEAD stream
2. Not existing -> Branch stream
3. Branch stream -> Not existing

Not existing -> HEAD stream (1)

Situation	A new repository is required with a HEAD stream.
Prerequisites	The repository currently does not exist.
Mechanism	Create a new repository using the <code>cvs init</code> command as described in, “Downloading and installing CVS” on page 222, and the HEAD stream is created automatically.

The HEAD stream is displayed by default when new connection to the repository is created.

Not existing -> Branch stream (2)

Situation	A new stream is required in order to perform parallel development to the HEAD stream.
Prerequisites	The repository must already be defined in the workspace.

Mechanism	Select <i>File -> New -> Other</i> from the main workbench. From the <i>CVS</i> category select <i>Stream</i> . Select the repository where the new stream is to be created, and enter a new, unique name for the stream. Click <i>Finish</i> . Note that the currently selected stream in the <i>Repositories</i> view is not always the repository defined by default in this wizard. If a non-unique name is typed, a dialog will display stating that the stream already exists.
	To populate the stream with a project, select the <i>Copy Version to Stream</i> menu.

Branch stream -> Not existing (3)

Situation	Development has completed on the stream, all changes have been merged back into the HEAD stream, or into a product version, and there is now a requirement to remove the stream from the repository.
Prerequisites	This must be a branch stream. The HEAD stream can never be removed.
Mechanism	From the <i>Repositories</i> view, select the stream you wish to remove, right-click, and select <i>Remove</i> from its context menu. All assigned projects in the workspace to this stream must be manually reassigned to another stream before the next synchronization.

Project lifecycle

In this section, we will illustrate the lifecycle of a project in the team environment and how to transition between states.

State transition diagram

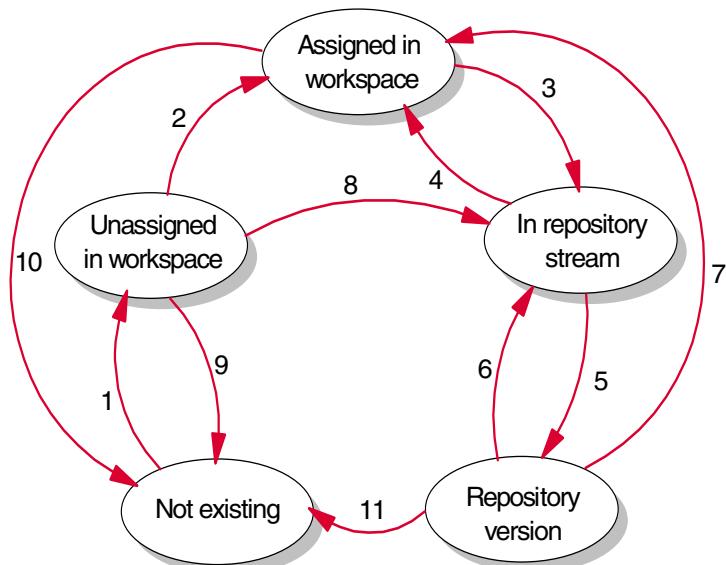


Figure 16-3 Valid state transitions for a project

These transitions are listed below and described in the subsequent sections:

1. Not existing → Unassigned in workspace
2. Unassigned in workspace → Assigned in workspace
3. Assigned in workspace → In repository stream
4. In repository stream → Assigned in workspace
5. In repository stream → Repository version
6. Repository version → In repository stream
7. Repository version → Assigned in workspace
8. Unassigned in workspace → In repository stream
9. Unassigned in workspace → Not existing
10. Assigned in workspace → Not existing
11. Repository version → Not existing

Not existing -> Unassigned in workspace (1)

Situation	This is how brand new projects are created.
Prerequisites	Any user can create a new project in their workspace.
Mechanism	From the main menu of the Workbench, select <i>File -> New -> Project</i> . This invokes the new project wizard. Select the category and type of project required and click <i>Next</i> . Complete the following project wizard and click <i>Finish</i> . The transition applies to any project type.

Unassigned in workspace -> Assigned in workspace (2)

Situation	In order to be able to add source to a repository, a project must first be assigned to a connection and a stream.
Prerequisites	The project must exist.
Mechanism	From the <i>Properties</i> dialog of the project, select the Team category. Click on the <i>Change</i> button and select a defined repository connection and existing stream in that repository.

Assigned in workspace -> In repository stream (3)

Situation	Once the project has been assigned to a repository connection and stream, the project and its source files can be added to the selected stream. This is known as <i>releasing</i> .
Prerequisites	The project must exist and be already assigned to a valid connection and stream.
Mechanism	Select the project to place in the stream. Select <i>Team -> Synchronize with Stream</i> to open the <i>Synchronize</i> view. Select the project in this view and click on the <i>Release</i> item in its context menu. While releasing, a prompt will appear for a comment that will be used to label any of the contained files which are also released to the stream.

In repository stream -> Assigned in workspace (4)

Situation	A project is contained in an existing stream and there is a requirement to add this to the workspace.
Prerequisites	The project cannot exist in the workspace beforehand.
Mechanism	Select the project contained in the required stream from the <i>Repositories</i> view. Open the context menu for the project and click on <i>Add to workspace</i> . The current state of the project in the stream will be added to the workspace and the contained files made available for editing.

The project will also be assigned to the source stream in its *Properties* dialog by default.

In repository stream -> Repository version (5)

- Situation A version of the project is required. The version can be created from the contained elements in the current stream or those in the workspace.
- Prerequisites The project must already be in a repository stream. All components included in the version need to be released.
- Mechanism To create a version of the project based on the most recently released contents of the project stream, select the project from the *Repositories* view, right-click and select *Version from Stream*.

To create a version of the project based on the current contents of the workspace, select the project from the *Navigator* view, right-click, and select *Version from Workspace*.
- On both occasions, a prompt will appear for a version name. Click *OK* when finished.

Repository version -> In repository stream (6)

- Situation The contents of a stream need to be replaced with those of a version.
- Prerequisites The stream cannot be the HEAD stream.
- Mechanism From the stream entry in the *Repositories* view, right-click and select *Copy Version to Stream*.

A dialog will appear called *Choose Versions*. This allows the selection of versions of projects, which are to be added to the stream. Click *OK* when finished. To see the changes in the *Repositories* view, perform a refresh of the view.

Note that if there is a requirement to replace the current contents of the HEAD stream with those in a version of the project, first you must add the files to the workspace, and then release the changes from the workspace back into the stream as described in, “Assigned in workspace -> In repository stream (3)” on page 549.

Repository version -> Assigned in workspace (7)

- Situation A project version exists in the repository, and it needs to be added to the workspace.

Prerequisites	None
Mechanism	Select the required project version from the <i>Project versions</i> branch of the repository tree in the <i>Repositories</i> view. From its context menu, select <i>Add to Workspace</i> . If the project already exists in the workspace, a prompt will appear asking you if the current contents should be over-written.
Unassigned in workspace -> In repository stream (8)	

Situation	A project has not been assigned to a stream, but there is a requirement to add its contents to that stream. This involves <i>releasing</i> the project.
Prerequisites	The project must exist in the workspace, but not in the stream.
Mechanism	Select the project from the <i>Navigator</i> view. Click on its <i>Team -> Synchronize with Stream</i> menu. A prompt will be displayed asking you which repository connection and stream the project should be added to. When in the <i>Synchronize</i> view, perform the steps as described in, “Assigned in workspace -> In repository stream (3)” on page 549.
Unassigned in workspace -> Not existing (9)	

Situation	There is a requirement to remove a project from the workspace that has not been assigned to a repository connection.
Prerequisites	None
Mechanism	Select the project you wish to remove from the workspace in the <i>Navigator</i> view and select <i>Delete</i> from its context menu. A prompt will appear asking if the IDE should also remove all of the contents under the file system.

Assigned in workspace -> Not existing (10)	
Situation	A project needs to be removed from the workspace that has been assigned to a repository connection and stream.
Prerequisites	None
Mechanism	Ensure all of the changes that must be included in the stream have been released first, then repeat the approach as described in, “Unassigned in workspace -> Not existing (9)” on page 551.

Repository version -> Not existing (11)

Situation	A version needs to be removed from a repository, either because it is incorrect, or to reduce the amount of disk space used by the repository on the server.
Prerequisites	You are absolutely sure that this version will never be needed again.
Mechanism	There is no mechanism to perform this option in the IDE, however, versions can be removed using the <code>cvs -o admin</code> command from the machine where the repository has been installed. This command allows the specification of either a single version number or a range of version numbers to remove.

File lifecycle

In this section we will illustrate the lifecycle of a file in the team environment and how to transition between states.

State transition diagram

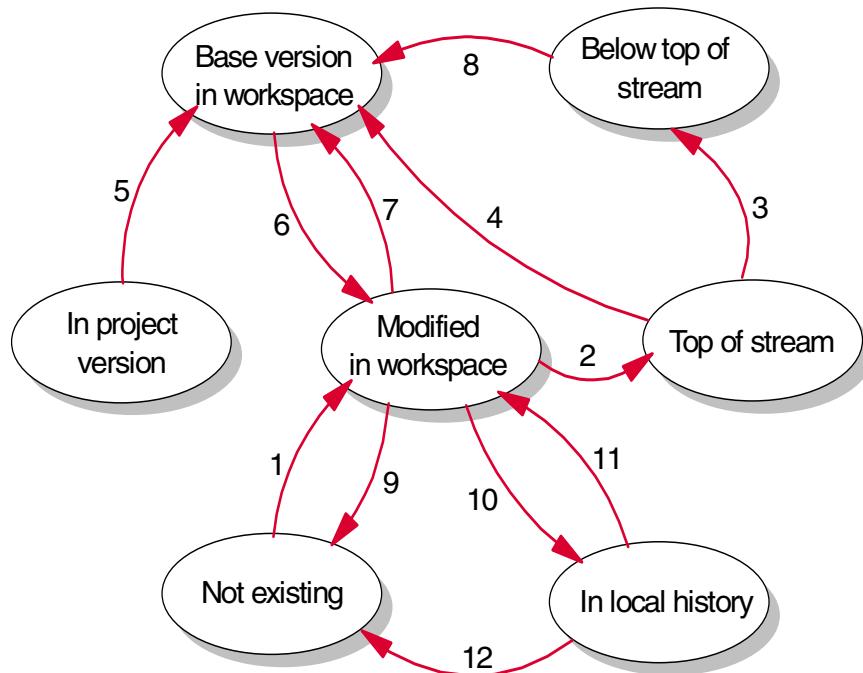


Figure 16-4 Valid state transitions for a file

These transitions are listed below and described in the subsequent sections:

1. Not existing -> Modified in workspace
2. Modified in workspace -> Top of stream
3. Top of stream -> Below top of stream
4. Top of stream -> Base version in workspace
5. In project version -> Base version in workspace
6. Base version in workspace -> Modified in workspace
7. Modified in workspace -> Base version in workspace
8. Below top of stream -> Base version in workspace
9. Modified in workspace -> Not existing

10. Modified in workspace -> In local history

11. In local history -> Modified in workspace

12. In local history -> Not existing

Not existing -> Modified in workspace (1)

Situation	<p>There are three scenarios which potentially provide this transition:</p> <ul style="list-style-type: none">• Creating a new file from scratch• Importing an existing file from the file system• Creating the file outside the Workbench and refreshing the workspace
Prerequisites	A project must be defined in the workspace.
Mechanism	<p>To create a new file from scratch, select the <i>File -> New -> Other</i> menu from the main Workbench toolbar, and select the appropriate wizard from the categories available. The most generic of these is the <i>File</i> item from the <i>Simple</i> category.</p> <p>To import an existing file from the file system, simply select the <i>File -> Import</i> menu from the IDE and select a suitable source type.</p> <p>Finally, to add a file to the workspace, which has been defined externally in the workspace folder, simply select the project in the <i>Navigator</i> view, right-click and select the <i>Refresh from Local</i> menu. This picks up any changes made in the underlying file system and updates the workspace accordingly.</p>

Modified in workspace -> Top of stream (2)

Situation	<p>The project stream contains a stack of revisions to the file, with the most recent at the top. The process in which a file in the workspace is placed at the top of the stream is known as <i>releasing</i>. Releasing a file <i>implicitly</i> creates a version number for the modification.</p>
Prerequisites	The file must be in the workspace and its containing project must be assigned to a repository connection and stream.
Mechanism	Select the file from the <i>Navigator</i> view and click on the <i>Team -> Synchronize with Stream</i> menu. This opens the <i>Structure Compare</i> view. If changes have been made to the file in the stream since the base version was created then a conflict will be displayed and the changes must be merged into the workspace

copy before releasing. Otherwise, check the changes are correct from the *Release Mode* view and select *Release* from the components context menu.

Once the workspace version has been released into the repository stream, open the Resource History for the component, and notice that it has automatically been assigned a version number for that release. These version numbers are generated automatically and cannot be edited by the end user.

Top of stream -> Below top of stream (3)

Situation	When a component is released from the workspace into the stream, the version previously at the top of the stream is moved down the stack automatically.
Prerequisites	A release operation has been performed on the file.
Mechanism	No interaction with the repository is required as this process is automated.

Top of stream -> Base version in workspace (4)

Situation	There are two scenarios where this is possible. Firstly, if a new file has been added to the stream by another developer, this can be added to the workspace. If the file is already in the workspace, it is possible to <i>catch up</i> with the latest changes made by others. The base version represents the state where it was retrieved from the repository before any changes are made locally.
Prerequisites	The project must be assigned to a repository connection and stream.
Mechanism	To add a file not currently in the workspace, select it from the stream in the <i>Repositories</i> view, and click on the <i>Add to workspace</i> menu.

To replace a file currently in the workspace with that at the top of the stream without comparing any changes, also select *Add to workspace* from the *Repositories* view or *Replace with -> Stream Contents* in the Navigator view. To compare the changes first, select the file in the Navigator view and select *Team -> Synchronize with Stream*. In the *Structure Compare* view, select the *Catch up Mode* view, and right-click on the file, selecting its *Catch Up* menu.

This operation starts a new local history for the file.

In project version -> Base version in workspace (5)

Situation	A file stored in a versioned project is retrieved in order to later replace the version in the current development stream.
Prerequisites	The required file must be in a versioned project.
Mechanism	From the Repositories view, select the project version containing the file to import, expand it and select the required file. Right-click on the file and select <i>Add to workspace</i> .

Base version in workspace -> Modified in workspace (6)

Situation	Once a base version has been added to the workspace, its contents are edited.
Prerequisites	The file has recently been added to the workspace from either a stream or project version.
Mechanism	This is performed automatically when the file is modified with any of the editors in the IDE and the changes saved.

Modified in workspace -> Base version in workspace (7)

Situation	The file has been edited in the workspace, but the changes are no longer required and the file needs to be replaced with the base version.
Prerequisites	The file is associated with a stream and has a base version.
Mechanism	Select the file from the <i>Navigator</i> view. Click on the <i>Replace With -> Base version</i> context menu.

Below top of stream -> Base version in workspace (8)

Situation	Occasionally, old releases of the file contained in the repository stream need to be added back to the workspace.
Prerequisites	The file must be defined in the stream.
Mechanism	This is performed by the <i>Show in Resource History</i> menu which is available either from the <i>Team</i> context menu in the <i>Navigator</i> view or directly from the context menu for the file in the <i>Repositories</i> view.

Select the implicitly generated file version required in the *Resource History* view and select *Add to workspace* from its context menu.

Modified in workspace -> Not existing (9)

Situation	A file has been created locally, which requires removing.
-----------	---

Prerequisites	The file must reside in the workspace.
Mechanism	Select the file from the local workspace and select <i>Delete</i> from its menu.

Modified in workspace -> In local history (10)

Situation	Each time a change is made to a file and the file is saved locally, an entry is added to the local history which stores all of the revisions made in that file since a base version was created.
Prerequisites	The file must exist in the workspace.
Mechanism	This transition is performed automatically each time the file is saved in the workspace.

In local history -> Edited in workspace (11)

Situation	An edit has been made to the local file which is no longer required and the developer wishes to return to a modification of that file which was made since the base version was created.
Prerequisites	The file must have been modified at least once since the base version was added to the workspace.
Mechanism	Select the <i>Replace With -> Local version</i> context menu from the file in the <i>Navigator</i> view.

In local history -> Not existing (12)

Situation	On regular occurrences, editions of the file are removed from the local history.
Prerequisites	One of the following conditions must be true: <ul style="list-style-type: none"> • The file has just been replaced with a new base version from a stream or project version. • The history entry is older than the specified time in the Workbench preferences. • More revisions have been made since the base version than defined in the Workbench preferences. • The size of the local history file has exceeded that specified in the Workbench preferences.
Mechanism	This operation is performed automatically by the IDE.



C

Quiz answers

This appendix provides answers to the quiz questions at the end of each chapter of the book.

Introduction

We have thrown a number of terms at you in the introduction. Most of these terms will only be explained in detail in Chapter 8, “Web services overview and architecture” on page 241. Here are the short answers:

1. What is stored in WSDL?

The most important information is where the Web service is located and how it is accessed, that is, the parameters that must be passed and the result that is returned.

2. What is SOAP used for?

SOAP is the (transport independent) XML messaging mechanism that is used between the service requester and the service provider.

3. What is stored in a UDDI registry?

The UDDI registry holds information about providers of Web services (business entities) and the services themselves.

4. What brand of cars does Almaden Autos sell?

This is a trick question - we have not specified that at all. Even scanning the database for inventory parts does not help; the parts are located all over the US and Europe.

5. What is the name of the manufacturer of the cars?

Mighty Motors, a fictive manufacturer of cars and redbooks.

Application Developer overview

1. Which of the following perspectives does not exist? Web, XML, Java, team, Web services, J2EE.

Web services. If you want to run the Web service wizards you have to create a Web project first and work in the Web perspective.

2. Which file format is used to store the IBM extensions to the deployment descriptors?

XMI (XML Metadata Interchange Format).

For instance ibm-application-ext.xmi contains the IBM extensions to the EAR deployment descriptor.

3. Name at least 3 different types of projects in Application Developer?

Java, Web, EJB, EAR, Server

4. How can you add a JAR file to the class path for a Java/Web/EJB project?

You can add a JAR file to the build path through the Java build path menu option in the Properties context menu of the project.

5. What is the difference between a task and a problem?

Tasks you define yourself (for example, to do items) whereas problems are automatically added by Application Developer, such as, when there are compilation problems.

Web development with Application Developer

1. Which file contains the information which links Web and EJB projects to the EAR file?

application.xml

2. To add an external parameter to an SQL query, which of the following options should be selected in the expression builder?

b.) Constant

3. Where is the information stored which links the servlets and JSPs together that are generated from the database Web pages wizard?

The initialization parameters defined in web.xml under the Servlets tab.

4. Where is the Web application context root stored?

application.xml

5. What are the class names for the two JDBC 2.0 DB2 drivers?

COM.ibm.db2.jdbc.DB2ConnectionPoolDataSource and

COM.ibm.db2.jdbc.DB2XADataSource.

(Give yourself ten bonus points if you got the capitalization right!)

XML support in Application Developer

1. Name at least four different types of artifacts that can legally appear in an XML document.

There are six kinds of such artifacts (markup): elements, attributes, entity references, comments, processing instructions, and CDATA.

2. Outline an XML schema for the XML file listed in “XML file” on page 93.

There would be one root element called welcome, containing a complex type with the content model sequence. The other elements containing other elements, such as message, book information, and so forth, would be such complex types, too.

Day, month, and year would either use built in simple types or define custom simple types with restrictions such as day values must be between 1 and 31, and so forth.

3. What is the difference between DTD validation and XSD validation?

XSDs allow to define structure and constraints more precisely. An XSD validator can therefore perform more checks.

4. Name at least four WSAS XML tools. Is there an XML project? Is there an XML perspective?

XML editor, XSD editor, DTD editor, XML to XML mapping editor, RDB to XML mapping editor, XSL trace editor, xerces, xalan, JavaBean generator, other generators.

Yes, there is an XML perspective. No, there is no XML project.

5. In which namespace does the schemaLocation attribute reside, and which local name is typically defined for it (hint: take a look at Figure 4-10 on page 114)?

The local name xsi, which is assigned to the namespace http://www.w3.org/2001/XMLSchema-instance, prefixes the schemaLocation attribute.

() In Figure 4-23 on page 126, how would the schema location attribute look like if the XSD file would be moved from the local file system to a Web server, assuming that the new location of the file is:*

http://localhost:8080/ITSOMighty/allInventoryItems.xsd

That is a tough one. It would contain two URI then, one for the target namespace, just like before, and one for the location:

```
xsi:schemaLocation=
  "http://www.redbooks.ibm.com/ITSOADS/schemas/InquireResults
  http://localhost:8080/ITSOMighty/allInventoryItems.xsd">
```

EJB development with Application Developer

1. What are the three EJB to RDBMS mapping approaches supported by Application Developer?

Top down, bottom up and meet in the middle

2. Which file contains the generated metadata for the mapping?

Map.mapxmi

3. How do you bind an EJB to a given datasource?

In the Binding tab of the EJB Extension editor, change the datasource JNDI name for either the EJB or its EJB module.

4. What are the names of the two supplied project validators?

EJB Validator and Map Validator

5. In which files are the EJB references defined?

In ejb-jar.xml, which needs editing by both the EJB editor and the EJB Extension editor, under the EJB Reference and Binding tabs respectively.

Deployment of Web and EJB applications

1. What are the two different ways to install an enterprise application in AEs?

You can use the Web based administrator's console or you can use the command line install tool SEAppInstall.

2. Name at least two big differences between AE and AEs.

See Table 6-1 on page 170.

3. What protocol is used to communicate between the Web server plug-in and the application server?

In WebSphere 4.0 there is an embedded Web server that talks with the Web server plug-in through HTTP.

4. On what port is the embedded Web server listening for AEs and for AE?

AEs: 9080. AE: The first application server you create has an associated embedded Web server listening on 9080; each time you create another application server the port number is increased by one.

5. In what file is the Web server plug-in configuration stored?

The file is WASROOT/config/plugin-cfg.xml.

Working in a team

1. Are file deletions from the workspace permanent?

Yes, unless you are working with either a local or shared repository.

2. What is a stream?

A stream maintains a teams shared configuration of one or more related projects and their folders and files.

3. What are the two possible operations when synchronizing with a stream?

Release and catch-up

4. What is the most common protocol used with CVS servers?
pserver
5. How do you define the version number for a released resource?
Trick question - you can't!

Web services architecture and overview

1. Name the service roles and operations comprising the service oriented architecture, as well as at least two development strategies each for the client and the server.

Roles: Service provider (server), service requestor (client), service broker (registry).

Operations: publish, find, bind (bind also called invoke).

Server development: bottom up, top down, meet in the middle.

Client development: static, provider-dynamic, type-dynamic.

2. Which SOAP, WSDL, and/or UDDI files contain the URN of a Web service?

The SOAP server side deployment descriptor (dds.xml), the client side SOAP proxy, and the WSDL interface file. The UDDI registry itself does not keep the URN, but pointers to the WSDL files.

3. What are the two parts of a WSDL specification? Which of them contains protocol binding information such as the encoding to be used?

The service interface file, containing the bindings, and the service implementation file.

4. Is any one of the WSDL files involved in the SOAP server deployment process?

Not directly, the SOAP layer is not aware of the specification language. However, a SOAP deployment descriptor can be generated from WSDL files.

5. Explain the difference between a UDDI tModel and a WSDL interface file.

That is a tough one. The WSDL interface file contains a formal specification for a Web service that can be analyzed by tools. The tModel, or service type, is just a technical fingerprint containing supporting informal information about a service and a URL pointer to a formal specification (such as a WSDL file).

Product support for Web services

1. Which servlet invokes a Web service?
The Apache SOAP RPC router or message router servlet.
2. Does the Web service client wizard generate a skeleton client application?
The wizard generates a test client but not a skeleton client application.
3. Name at least two functions of the UDDI explorer?
The UDDI explorer can find business entities, publish business entities, find business services, and publish business services.
4. Where can you find the Web services toolkit?
At IBM's alphaWorks Web site.
5. What Web service wizards does WebSphere Studio classic provide?
Web service creation wizard and Web service consumption wizard.

Static Web services

1. What are the three types of Web service scope, and where are they defined?
Request, session and application. They are defined in the deployment descriptor, dds.xml.
2. What is the difference between Literal XML encoding and SOAP encoding?
Literal XML encoding is only used for parameters of type org.w3.dom.Element and copy the XML information into the payload of the SOAP message. SOAP encoding should be used for all other data types, where serializers are provided for all types defined in the XML schema specification.
3. Which of the following is the service implementation file, and which is the service interface file?
 - *InquireParts-service.wsdl contains the service implementation.*
 - *InquireParts-binding.wsdl contains the service interface.*
4. What are the ISD files used for?
One is generated by the wizard for each Web service, containing information regarding how it should be deployed. They are all concatenated into dds.xml, which is picked up from the SOAP router servlet.
5. Which of the two WSDL files are used as the basis of the client proxy generation?
InquireParts-service.wsdl, the service implementation.

Dynamic Web services

1. Name the two different types of dynamic Web service clients, and explain the difference between them. Which type is likely to be implemented more often?

Type dynamic and provider dynamic service requestors. Type dynamic requestors discover both service interface and implementation at runtime. Provider dynamic requestors are coded against a fixed API and discover service access information such as the RPC router URL at runtime. There will be more provider dynamic applications.

2. List at least three different ways in which you can inspect the content of a UDDI registry.

Standard browser interface, UDDI explorer built into Application Developer, UDDI4J API.

3. Which two HTTP operations must a service provider support after publishing a service to the UDDI registry?

HTTP GET so that the service provider can retrieve the WSDL file.

HTTP POST for the Web service invocation.

4. Name at least three artifacts in the UDDI object model.

businessEntity, businessService, bindingTemplate, tModelInstance, tModel. Other names are used in UDDI GUIs.

5. Which WSDL file is the implementation file and which file is the interface file?

Implementation file == Xxxxx-service.wsdl

Interface file == Xxxx-binding.wsdl

Composed Web services

1. True or false: if you have the service interface and service implementation files of a Web service, you can generate a JavaBean that implements the Web service?

False, the JavaBean skeleton wizard only generates the method signatures, you still have to implement the different methods.

2. What JAR file contains the WASMessageRouterServlet?

SOAPCFG.jar in WSAD_ROOT\plugins\com.ibm.etools.webservice\runtime.

3. From which WSDL file do you start to generate the JavaBean skeleton: interface file or implementation file?

You can start from either one of the WSDL files.

4. Which WSDL file is generated when you create the JavaBean skeleton: interface file or implementation file?

The implementation file is generated (or modified). The interface file is copied if you did not start with it.

5. Which file is the SOAP deployment descriptor?

dds.xml

Deployment of Web services

1. What WebSphere tool can be used to assemble an EAR file?

The application assembly tool (AAT).

2. What command line deployment tool is provided with WebSphere AEs?

The SEApplInstall command.

3. What is the use of the SOAP admin client?

Display, start, and stop Web services.

4. Which application server, AEs or AE, can automatically update the Web server plug-in?

Only AE can automatically update the plugin, in AEs it is a manual process.

5. What tool can enable SOAP services in an EAR file?

The SOAPEARENabler tool.

Web services advanced topics

1. Name the two encodings supported by the Application Developer Web service wizard. Do they support `java.lang.boolean`?

The two mappings are the SOAP encoding and Literal XML. The SOAP encoding does support Boolean, Literal XML does not. Hence, there is no need to implement a custom mapping.

Information about supported binding can be found in the Application Developer help perspective, in the Apache SOAP users's documentation (and in the open source code, of course).

2. Assign the following class names to the two styles (SOAP RPC verses SOAP message-oriented): Message, Envelope, Body vs. Call, Parameter, SOAPMappingRegistry.

*Call, Parameter, SOAPMappingRegistry: SOAP RPC API. Message, Envelope,
Body: SOAP message oriented API.*

3. Decide which SOAP communication style to use for the following two scenarios a news portal provider is confronted with:
 - When processing incoming HTTP requests, the news portal wants to translate the HTTP request parameters into a Web service invocation to one of its content providers. The request parameters are fixed and known at build time.
 - The interface between the content management system and the news portal is supposed to be re implemented; each night, a set of XML files needs to be transferred from the content management system to the portal.

There is no fixed boundary, both requirements could be implemented with both styles. The first requirement appears to be more RPC oriented, the second one could be an example for message oriented communication. Analyze the requirements in more detail before making a decision!

4. Name at least three possible scenarios for UDDI registries.
UDDI operator cloud, UDDI e-Marketplace, UDDI portal, UDDI partner catalog, internal enterprise application integration.
5. What does WSFL stand for and what is it used for?
WSFL stands for Web services flow language. It defines software workflow processes within the framework of a Web services architecture with two aspects: the sequence in which the services should be invoked and the information flow between the services.



D

Additional material

This redbook refers to additional material that can be downloaded from the Internet as described below.

Locating the Web material

The Web material associated with this redbook is available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser to:

<ftp://www.redbooks.ibm.com/redbooks/SG246292>

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, SG24-6292.

Using the Web material

The additional Web material that accompanies this redbook includes the following files:

- ▶ `readme.txt`—short instructions
- ▶ `sg246292code.zip`—all sample code in ZIP format

System requirements for downloading the Web material

The following system configuration is recommended:

Hard disk space	3 GB
Operating System	Windows NT or Windows 2000
Processor	550 MHz or better
Memory	384 MB, recommended 512 MB

How to use the Web material

Create a subdirectory on your workstation and download the `sg246292code.zip` file into this folder. Unzip the file onto a hard drive to create this directory structure and files:

```
sg246292
  sampcode           - master directory
  Setup              - DDL for ITSOVSAD DB, images, Server project
  WSADWeb            - WSAD Web Development
    solution
  WSADXML            - WSAD XML Development
    solution
  WSADEJB             - WSAD EJB Development
    solution
  WSADDDeploy          - Deployment of Web Applications and EJBs
    solution
  WSStatic             - WSAD Static Web Service development
    solution
  WSDynamic            - WSAD Dynamic Web Service development
    solution
  WSCompose             - WSAD Composed Web Service Development
    solution
  WSDeploy              - WSAD Deployment of Web Service Development
    solution
  WSEnhanced            - WS Advanced Topics
    Advanced Soap        - Custom Mappings
    solution
  DADX                 - Web service from SQL
    solution
```

Directories

Setup

- ▶ **DDL** (ITSOWSAD.ddl—DDL to create the ITSOWSAD database)
- ▶ **ITSOWSAD** (ITSOWSADServer.zip—server definition)
- ▶ **images** (*.gif—auto parts base images for each of the Web applications)

WSADxxx and WSxxx

These directories match the chapters of the redbook and contain code to assist in building the chapter example. This code is provided to assist in the typing effort when building the applications. The code can be used to cut and paste or to import when called from in the process of building the application.

The **solution** subdirectory in each chapter directory contains the ZIP, JAR, or EAR files for the completed chapter.

- ▶ Once you have completed the setup instructions and the configuration of the WebSphere instance, you can import these EAR files for a complete working solution.
- ▶ The ZIP, JAR, or EAR files are cumulative and contain the complete solution to the specified chapter in the book.
- ▶ The instructions following the EAR file import instructions for each chapter are cumulative. They assume that the previous instructions have been completed. This mostly affects the definition of the Java build path for the Web projects and the addition of EAR files to the server instance.
- ▶ For more detailed instructions on building the chapter solution, see the referring chapter.

Important: The Application Developer does not allow you to import an EAR file into an existing EAR project. Therefore:

- ▶ Delete the existing EAR project.
- ▶ Import the EAR file into the specified EAR project. This will create the EAR project and import the WAR files into the existing Web projects. This overwrites the existing files in the Web project but leaves the Web project definitions in place. This is important because deleting the Web projects along with the EAR project and allowing the EAR import to create the contained Web projects will reset your Web project definitions back to the default... most importantly resetting the Java build path for the project.
- ▶ However, EJB projects must be deleted before importing the EAR file.

Instructions

Here are short instructions on how to use the sample code for each chapter of the book.

Setup directory

From a DB2 command window, from the sampcode\Setup\DDL directory containing the DDL run:

```
db2 -tf ITSOWSAD.ddl
```

See “Define and load the ITSOWSAD database” on page 542 for more information.

For testing of Web and EJB applications in the Application Developer we use an internal WebSphere server:

- ▶ Create a server project called ITSOWSADServer
- ▶ Import the zip file into the ITSOWSADServer project.
- ▶ Update the paths for the server configuration.
 - Open the server-cfg.xml file.
 - Click Yes when presented with the message in Figure 16-5.
 - Save the configuration file.



Figure 16-5 Automatic updating of configuration files

Define the following Java build variables in Application Developer. Select *Window* -> *Preferences* -> *Java* -> *Classpath variables*, and click on *New*:

UDDI4JJAR <WSAD_INSTALL_DIR>plugins/com.ibm.etools.websphere.runtime/runtime/uddi4j.jar

MAILJAR <WSAD_INSTALL_DIR>plugins/com.ibm.etools.servletengine/lib/mail.jar

WSADWeb directory

These instructions refer to Chapter 3, “Web development with Application Developer”:

- ▶ Import the ITS0A1ma.ear file from the **solution** directory into the ITS0A1maEAR project. This creates or overwrites the ITS0A1maWeb project.

WSADXMLE directory

These instructions refer to Chapter 4, “XML support in Application Developer”:

- ▶ Create a Java project called ITS0MightyXML.
- ▶ Import the zip file from the **solution** directory into the ITS0MightyXML project, overwriting existing files.
- ▶ Place the InquireParts.xml file from the **solution** directory in the root install directory (d:\WSAD, for example).

WSADEJB directory

These instructions refer to Chapter 5, “EJB development with Application Developer”:

- ▶ Delete the ITS0MightyEAR and the ITS0MightyEJB projects.
- ▶ Import the ITS0Mighty.ear file from the **solution** directory into the ITS0MightyEAR project.
- ▶ Rebuild the ITS0MightyEJB project.
- ▶ Add the ITS0MightyEAR project to the ITS0SADWebSphere server configuration.

WSADDdeploy directory

These instructions refer to Chapter 6, “Deployment of Web and EJB applications to WebSphere”:

- ▶ Import the ITS0SADRemoteServer.zip file from the **solution** directory into the ITS0SADServer project, overwriting files.
- ▶ Edit the ITS0SADRemote server instance:
 - Change the hostname to the name of the machine running your remote WAS instance.
 - Make sure the installation directory and deployment directory exist on your machine or change them to directories that exist on your machine.

- ▶ Edit the ITS0WSADRemote server configuration:
 - Make sure the class path for the Db2JdbcDriver points to your install path for DB2.
- ▶ Use the supplied EAR files to follow the instructions for deploying the ITS0A1ma and ITS0Mighty enterprise applications to WebSphere AEs and WebSphere AE.

WSStatic directory

These instructions refer to Chapter 10, “Static Web services”:

- ▶ Delete the ITS0A1maEAR project.
- ▶ Import the ITS0A1ma.ear file from the **solution** directory into the ITS0A1maEAR project.
- ▶ Add the variables WAS_XALAN and SOAPJAR to the ITS0A1maWeb project Java build path.
- ▶ Delete the ITS0MightyEAR and the ITS0MightyEJB projects.
- ▶ Import the ITS0Mighty.ear file from the **solution** directory into the ITS0MightyEAR project.
- ▶ Add the variables WAS_XALAN, and SOAPJAR to the ITS0MightyWeb project Java build path.
- ▶ Add project ITS0MightyEJB to the ITS0MightyWeb project Java build path.

WSDynamic directory

These instructions refer to Chapter 11, “Dynamic Web services”:

- ▶ Import the ITS0Plenty.ear file into the ITS0PlentyEAR project.
 - Add the ITS0PlentyEAR project to the ITS0WSADWebSphere server instance.
 - Add the variables WAS_XERCES and SOAPJAR to the ITS0PlentyWeb Java build path.
- ▶ Delete the ITS0A1maEAR project.
- ▶ Import the ITS0A1ma.ear file from the **solution** directory into the ITS0A1maEAR project.
 - Add the variables WAS_XERCES, WAS_XALAN, SOAPJAR, UDDI4JJAR and MAILJAR to the ITS0A1maWeb project Java build path
- ▶ Delete the ITS0MightyEAR and ITS0MightyEJB projects.

- ▶ Import the ITSOMighty.ear file from the **solution** directory into the ITSOMightyEAR project.
- ▶ Follow the instructions in “Working with the Application Developer UDDI browser” on page 386 for publishing the ITSOMighty and ITSOPlenty Web services to the UDDI registry.

WSComposed directory

These instructions refer to Chapter 12, “Composed Web services”:

- ▶ Delete the ITSOAlmaEAR project.
- ▶ Import the ITSOAlma.ear file from the **solution** directory into the ITSOAlmaEAR project.
 - Add JAR soapcfg.jar to the ITSOAlmaWeb project Java build path from /ITSOAlmaWeb/webApplication/WEB-INF/lib
 - Rebuild the ITSOAlmaWeb project.
- ▶ Import the ITSOSanta.ear file from the **solution** directory into the ITSOSantaEAR project.
 - Add JAR dbbeans.jar, ibmSerializers.jar and xsdbeans.jar to the ITSOSantaWeb project Java build path from /ITSOSantaWeb/webApplication/WEB-INF/lib
 - Add the variables WAS_XALAN, WAS_XERCES and SOAPJAR to the ITSOSantaWeb project Java build path.
 - Rebuild the ITSOSantaWeb project.
 - Add the ITSOSantaEAR project to the ITSOWSADWebSphere server instance.

WSDeploy directory

These instructions refer to Chapter 13, “Deployment of Web services to WebSphere”:

- ▶ Use the supplied EAR files to follow the instructions for deploying the ITSOAlma, ITSOMighty, ITSOPlenty, and ITSOSanta enterprise applications to WebSphere AEs and WebSphere AE.

WSEnhanced directory

These instructions refer to Chapter 14, “Web services advanced topics”.

Advanced SOAP programming

- ▶ Import the ITSOAdvanced.ear file from the **solution** directory into the ITSOAdvancedEAR project.
- ▶ Add variables WAS_XERCES, MAILJAR, SOAPCFGJAR and SOAPJAR to the ITSOAdvancedWeb project Java build path.
- ▶ Make sure the port used in the proxies is set to your port (8080 by default, 8081 for the TCP/IP Monitor).
- ▶ Rebuild the ITSOAdvancedWeb project.
- ▶ Add the ITSOAdvancedEAR project to the ITSOWSADWebSphere server instance.
- ▶ Copy the catalog.xml file from the solutions directory to the <WSAD_INSTALL_ROOT>.

Creating a Web service from a DADX file

- ▶ Import the ITSODadx.ear file from the **solution** directory into the ITSODadxEAR project.
- ▶ Add JAR xsdbeans.jar, ibmSerializers.jar, soapcfg.jar, worf.jar and worf-servlets.jar to the ITSDadxAjax project Java build path from /ITSODadxWeb/webApplication/WEB-INF/lib.
- ▶ Add the variables WAS_XERCES and SOAPJAR to the ITSDadxAjax project Java build path.
- ▶ Add the ITSDadxAjaxEAR project to the ITSOWSADWebSphere server instance.
- ▶ Rebuild the ITSDadxAjax project.

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

IBM Redbooks

For information on ordering these publications, see “How to get IBM Redbooks” on page 581.

- ▶ *Self-Study Guide: WebSphere Studio Application Developer and Web Services*, SG24-6407
- ▶ *WebSphere Version 4 Application Development Handbook*, SG24-6134
- ▶ *WebSphere V4.0 Advanced Edition Handbook*, SG24-6176
- ▶ *Programming J2EE APIs with WebSphere Advanced*, SG24-6124
- ▶ *Enterprise JavaBeans for z/OS and OS/390 CICS Transaction Server V2.1*, SG24-6284
- ▶ *EJB Development with VisualAge for Java for WebSphere Application Server*, SG24-6144
- ▶ *Design and Implement Servlets, JSPs, and EJBs for IBM WebSphere Application Server*, SG24-5754
- ▶ *Programming with VisualAge for Java Version 3.5*, SG24-5264
- ▶ *WebSphere V3.5 Handbook*, SG24-6161
- ▶ *Version 3.5 Self Study Guide: VisualAge for Java and WebSphere Studio*, SG24-6136
- ▶ *How about Version 3.5? VisualAge for Java and WebSphere Studio Provide Great New Function*, SG24-6131
- ▶ *Servlet and JSP Programming with IBM WebSphere Studio and VisualAge for Java*, SG24-5755
- ▶ *Revealed! Architecting Web Access to CICS*, SG24-5466
- ▶ *IMS Version 7 and Java Application Programming*, SG24-6123
- ▶ *Migrating WebLogic Applications to WebSphere Advanced Edition*, SG24-5956
- ▶ *WebSphere Personalization Solutions Guide*, SG24-6214

- ▶ *Self-Service Applications Using IBM WebSphere V4.0 and IBM MQSeries Integrator*, SG24-6160
- ▶ *WebSphere Scalability: WLM and Clustering Using WebSphere Application Server Advanced Edition*, SG24-6153
- ▶ *Patterns for e-business: User-to-Business Patterns for Topology 1 and 2 using WebSphere Advanced Edition*, SG24-5864
- ▶ *The XML Files: Using XML for Business-to-Business and Business-to-Consumer Applications*, SG24-6104
- ▶ *The XML Files: Using XML and XSL with IBM WebSphere 3.0*, SG24-5479
- ▶ *CCF Connectors and Database Connections Using WebSphere Advanced Edition Connecting Enterprise Information Systems to the Web*, SG24-5514
- ▶ *WebSphere V3 Performance Tuning Guide*, SG24-5657
- ▶ *WebSphere Application Servers: Standard and Advanced Editions*, SG24-5460
- ▶ *VisualAge for Java Version 3: Persistence Builder with GUIs, Servlets, and Java Server Pages*, SG24-5426
- ▶ *IBM WebSphere and VisualAge for Java Database Integration with DB2, Oracle, and SQL Server*, SG24-5471
- ▶ *Developing an e-business Application for the IBM WebSphere Application Server*, SG24-5423
- ▶ *The Front of IBM WebSphere Building e-business User Interfaces*, SG24-5488
- ▶ *VisualAge for Java Enterprise Version 2: Data Access Beans - Servlets - CICS Connector*, SG24-5265
- ▶ *VisualAge for Java Enterprise Version 2 Team Support*, SG24-5245
- ▶ *Creating Java Applications Using NetRexx*, SG24-2216

Other resources

These publications are also relevant as further information sources:

- ▶ *Sections on Web services in IBM WebSphere Application Developer beta, Help perspective, August 2001 (contained in Application Developer beta code)*
- ▶ *Web Services Conceptual Architecture, Version 1.0, Web Services Architecture Team, June 2001, contained in IBM WSTK 2.3 (currently no ISDN number or URL available)*

- ▶ *Web Services Development Concepts (WSDC 1.0), Peter Brittenham, IBM Software Group, May 2001, contained in IBM WSTK 2.3 (currently no ISDN number or URL available)*
- ▶ *Web Services Flow Language (WSFL 1.0), Prof. Frank Leymann, IBM Software Group, May 2001*
- ▶ *Combine business process management technology and business services to implement complex Web services, Peter Lambros, Marc-Thomas Schmidt, Claudia Zenter, IBM Software Group*
- ▶ *The Web services insider, Parts 4 to 7, James Snell, IBM Emerging Technologies, June 2001*
- ▶ *Eclipse platform technical overview, Object Technology International, July 2001*
- ▶ *CVS for the developer or amateur, Daniel Robbins, Gentoo technologies Inc., March 2001*
- ▶ *Versata Business Logic Automation Product and the Web services Paradigm, J. Liddle and K. Yousaf, Versata, July 2001*
- ▶ *The role of private UDDI notes, Parts 1 and 2, Steve Graham, IBM Emerging Technologies, May 2001*
- ▶ *API Documentation for SOAP, UDDI4J, IBM WSTK, WSDL APIs, Javadoc format (contained in WSTK)*
- ▶ Articles on IBM developerWorks:
 - Myths and misunderstandings surrounding SOAP, Frank Cohen:
<http://www-106.ibm.com/developerworks/webservices/library/ws-spmhyths>
 - Understanding WSDL in a UDDI registry, Peter Brittenham (2 parts):
<http://www-106.ibm.com/developerworks/webservices/library/ws-wsdl1>
<http://www-106.ibm.com/developerworks/webservices/library/ws-wsdl12>
- ▶ *Professional Java XML*, various authors, Wrox Press 2001, ISBN 1-861004-01-X
- ▶ *Java & XML*, Brett McLaughlin, O'Reilly, September 2001, ISBN 0-596-00197-5

- ▶ UDDI
 - <http://www.uddi.org>
 - <http://www.ibm.com/services/uddi>
- ▶ XMethods Web service repository
 - <http://www.xmethods.com/>
- ▶ SOAP
 - <http://www.w3.org/TR/SOAP/>
- ▶ XML protocol
 - <http://www.w3.org/2000/xp/>
- ▶ WSDL specification
 - <http://www.w3.org/TR/WSDL>
- ▶ Component based development and integration (CBDI) forum
 - <http://www.cbdiforum.com>

How to get IBM Redbooks

Search for additional Redbooks or Redpieces, view, download, or order hardcopy from the Redbooks Web site:

ibm.com/redbooks

Also download additional materials (code samples or diskette/CD-ROM images) from this Redbooks site.

Redpieces are Redbooks in progress; not all Redbooks become Redpieces and sometimes just a few chapters will be published this way. The intent is to get the information out much quicker than the formal publishing process allows.

IBM Redbooks collections

Redbooks are also available on CD-ROMs. Click the CD-ROMs button on the Redbooks Web site for information about all the CD-ROMs offered, as well as updates and formats.

Special notices

References in this publication to IBM products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent program that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program or service.

Information in this book was developed in conjunction with use of the equipment specified, and is limited in application to those specific hardware and software products and levels.

IBM may have patents or pending patent applications covering subject matter 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 the IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM Corporation, Dept. 600A, Mail Drop 1329, Somers, NY 10589 USA.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

Any pointers in this publication to external Web sites are provided for convenience only and do not in any manner serve as an endorsement of these Web sites.

The following terms are trademarks of other companies:

Tivoli, Manage. Anything. Anywhere., The Power To Manage., Anything. Anywhere., TME, NetView, Cross-Site, Tivoli Ready, Tivoli Certified, Planet Tivoli, and Tivoli Enterprise are trademarks or registered trademarks of Tivoli Systems Inc., an IBM company, in the United States, other countries, or both. In Denmark, Tivoli is a trademark licensed from Kjøbenhavns Sommer - Tivoli A/S.

C-bus is a trademark of Corollary, Inc. in the United States and/or other countries.

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

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

PC Direct is a trademark of Ziff Communications Company in the United States and/or other countries and is used by IBM Corporation under license.

ActionMedia, LANDesk, MMX, Pentium and ProShare are trademarks of Intel Corporation in the United States and/or other countries.

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

SET, SET Secure Electronic Transaction, and the SET Logo are trademarks owned by SET Secure Electronic Transaction LLC.

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

Abbreviations and acronyms

AAT	application assembly tool	HTTP	Hypertext Transfer Protocol
ACL	access control list	IBM	International Business Machines Corporation
API	application programming interface	IDE	integrated development environment
BLOB	binary large object	IDL	Interface Definition Language
BMP	bean-managed persistence	IOP	Internet Inter-ORB Protocol
CCF	Common Connector Framework	IMS	Information Management System
CICS	Customer Information Control System	ITSO	International Technical Support Organization
CMP	container-managed persistence	J2EE	Java 2 Enterprise Edition
CORBA	Component Object Request Broker Architecture	J2SE	Java 2 Standard Edition
DBMS	database management system	JAF	Java Activation Framework
DCOM	Distributed Component Object Model	JAR	Java archive
DDL	data definition language	JDBC	Java Database Connectivity
DLL	dynamic link library	JDK	Java Developer's Kit
DML	data manipulation language	JFC	Java Foundation Classes
DOM	document object model	JMS	Java Messaging Service
DTD	document type description	JNDI	Java Naming and Directory Interface
EAB	Enterprise Access Builder	JSDK	Java Servlet Development Kit
EAI	Enterprise Application Registration	JSP	JavaServer Page
EAR	enterprise archive	JTA	Java Transaction API
EIS	Enterprise Information System	JTS	Java Transaction Service
EJB	Enterprise JavaBeans	JVM	Java Virtual Machine
EJS	Enterprise Java Server	LDAP	Lightweight Directory Access Protocol
FTP	File Transfer Protocol	MFS	message format services
GUI	graphical user interface	MVC	model-view-controller
HTML	Hypertext Markup Language	OLT	object level trace
		OMG	Object Management Group
		OO	object oriented

OTS	object transaction service	WS	Web service
RAD	rapid application development	WSBCC	WebSphere Business Components Composer
RDBMS	relational database management system	WSDL	Web Service Description Language
RMI	Remote Method Invocation	WSTK	Web Service Development Kit
SAX	Simple API for XML	WTE	WebSphere Test Environment
SCCI	source control control interface	WWW	World Wide Web
SCM	software configuration management	XMI	XML metadata interchange
SCMS	source code management systems	XML	eXtensible Markup Language
SDK	Software Development Kit	XSD	XML schema definition
SMR	Service Mapping Registry		
SOAP	Simple Object Access Protocol (a.k.a. Service Oriented Architecture Protocol)		
SPB	Stored Procedure Builder		
SQL	structured query language		
SRP	Service Registry Proxy		
SSL	secure socket layer		
TCP/IP	Transmission Control Protocol/Internet Protocol		
UCM	Unified Change Management		
UDB	Universal Database		
UDDI	Universal Description, Discovery, and Integration		
UML	Unified Modeling Language		
UOW	unit of work		
URL	uniform resource locator		
VCE	visual composition editor		
VXML	voice extensible markup language		
WAR	Web application archive		
WAS	WebSphere Application Server		
WML	Wireless Markup Language		

Index

Symbols

.NET 253, 523

A

access point 399, 402
admin client
 SOAP 352
Administrator's Console
 WebSphere AE 190
 WebSphere AEs 172, 173, 439
Administrator's Console
 WebSphere AE 443
Agent Controller 47, 200
 installation 540
Almaden Autos 7, 322
 Web service 418
 client 363
alphaWorks 252, 496
alternate JRE 37
Apache
 AXIS 252, 261, 276
 SOAP server 270, 350
 Struts 79
 Tomcat 83
 Xalan 104, 105
 Xerces 103, 105
 XML project 252
application
 assembly tool 436
 client 39
 editor 39
 extension editor 39
Application Developer 17
 EJB development 133
 features 18
 installation 540
 preferences 22
 product 18
 project 33
 remote testing 200
 search 80
 system files 52
 team 207

UDDI browser 386

workspace 58

XML

 support 91
 tools 105

Application Installation wizard 178

application.xml 39, 61, 145

architecture

 team support 208
 Web service 514

Ariba 277

asynchronous communication 525

auto parts

 data model 12
 database 12
 EJB 133
 products 13
 sample application 3
 stage 1 7, 56
 stage 2 8, 134
 stage 2b 322
 stage 3 9, 380
 stage 4 10, 414
 system design 6
 system diagram 11
 use cases 4

Auto Parts Association 9

AXIS 276

B

background color 75

best practices 527

binding

 HTTP 289
 SOAP 287
 template 293
 WSDL 278, 286

body 265

BPML 498

branch stream 213, 546

breakpoint

 enabling 86
 set 37

view 32
browser
 embedded 86
build
 path 37
 scripts 236
 settings 34
builder 33
business
 entity 293, 387
 logic 161
process management 497
service 293

C

catch-up 212, 226
categorization 297
CDATA section 95
ClearCase 209
ClearCase LT 20
client
 application 366
 proxy 342, 354
code
 assist 26, 35
 refactoring 36
commands
 CVS 238
comment
 XML 94
communication style 267
compare versions 218
complex data type 99, 525
composed Web service 413
compound type 267
concurrency 212
Concurrent Versions System
 see CVS
configuration
 server 27
 WebSphere AEs 181
conflict 229
connection pool 68
console
 view 30
container-managed entity EJB 135
context root 40, 59, 325
controller servlet 69, 72

cross dealership inquiry 6, 10
custom mapping 449, 458
customize perspective 24
CVS 20, 209
 commands 238
 ignore 235
 installation 222
 macros 238
 repository 214, 215
 security 236
 Windows service 223

D

DAD 312
extension
 see DADX
DADX 312
DB2 317
document 312
group 312, 482
Web service 325, 479
data
 model 12
 SOAP 266
 perspective 28
 source 49, 174
 ITSOWSAD 68
 view 28
database
 connection 65
 ITSOWSAD 12
DB explorer
 view 28
DB2
 installation 538
 JDBC
 datasource 85
 ZIP file 35
 JDBC 2.0 62, 85
 XML Extender 317
db2java.zip 35, 62
dbbeans.jar 73
DDL 58, 572
dds.xml 310, 350, 422, 454
debug
 Java 37
 perspective 31
 view 31

deployed code 166
deployment 169
descriptor
 EJB 44, 145
 Web 42, 60
 Web application 73
programmatic 448
SOAP 255
 Web application 81
 Web service 433
WebSphere AE 190
 WebSphere AEs 170
deserializer 455
design 525
developerWorks 290
document
 access definition
 see DAD
 object model
 see DOM
 type definition
 see DTD
DOM 93, 102
 API 131
 tree 129
DTD 29, 96
 editor 105
dynamic
 image 78
 inquiry 5, 8
 service 257
 Web service 244, 379

E

EAR
 deployment descriptor 39
 export 171, 435
 file 40
 import 571
 project 38
ebXML 498
eclipse 18
editor 21, 22
 application 39
 extension 39
 DTD 105
 EJB 45
 extension 46, 149

mapping 145
Java 26, 35
script 77
server configuration 30
table 28
XML 29, 105, 113
XSD 114
XSL 124
trace 105
EJB 44
 1.1 specification 160
 bindings 46
 classes 144
 client 134
 deployed code 151
 deployment descriptor 27, 44, 145
 editor 45
 extension editor 46, 149
 inventory 135
 JAR file 39
 JDBC data source 149
 key class 145
 mapping 137, 145
 editor 145
 metadata 145
 module 27, 45
 part 135
 project 44
 reference 43, 165
 test client 153, 167
 deployment 183, 197
 Web service 332
 testing 152
 validate 150
 Web service 325
EJB to RDB Mapping wizard 138
ejbdeploy tool 152
ejb-jar.xml 39, 44, 145
element
 XML 94
e-marketplace 490
encoding 262
 SOAP 268
 style 338, 356
enterprise
 application
 export 435
 install 176, 443
 integration 493, 514

archive file
 see EAR
Enterprise JavaBeans
 see EJB
entity
 EJB 138
 create 138
 reference 93
envelope 262, 264
error handling 266
export
 EAR file 171
 WSDL 390
eXtensible style sheet language
 see XSL
extension point 19
external JAR 35

F

file
 life cycle 553
firewall 518
FTP 42

G

generated types
 Web application 71
getter method 147
global JNDI name 162
graphics editing framework 20

H

head stream 213, 546
header 265
help
 hover 36
 perspective 32
hierarchy view 26
history of changes 210
home interface 144, 162
host variable 67
hover help 36
HTML
 input page 72
 markup 92
 style sheet 41
HTTP

binding 289
port 242
server 189, 199
transport 440, 444
HTTPR 316

I

IBM Agent Controller
 see Agent Controller
ibm-application-ext.xmi 237
ibm-application-ext.xml 61
ibm-ejb-jar-bnd.xmi 44, 145
ibm-ejb-jar-ext.xmi 44, 145
ibm-web-bnd.xml 60
ibm-web-ext.xml 60
ignore 235
image
 display 75
 import 76
 URL 77
implementation
 WSDL 277
import
 EAR 571
 images 76
 WAR file 42
 Web site 41
 WSDL 394
initialization parameters 74
inspector view 32
install
 enterprise application 176
installation
 Agent Controller 540
 Application Developer 540
 CVS 222
 DB2 538
 WebSphere
 AE 539
 AEs 539
 UDDI Registry 541
integrated development environment 19
interface 277
 WSDL 277
interoperability 244, 523
inventory
 EJB 135
 table 12

ISD file 308, 337
isolation level 46
ITSOWSAD
 database 12, 61, 542
ITSOWSADServer 82

J

J2EE
 packaging 39
 perspective 27, 39
 specification 60
 view 39
Jakarta 217
JAR
 files 34
Java
 build settings 34
 builder 33
 debugging 37
 editor 26, 35
 Message Service 509
 package 35
 perspective 25
 project 33
 run 36
 runtime 37
 XML mapping 341
JavaBean
 skeleton 419, 422
 Web service 325, 327
JavaScript 76
JDBC
 2.0 62
 data source 84, 174, 191
 EJB 149
 driver 49, 174, 191
JDOM 103
JMS 318
JNDI
 explorer 154
 name 43, 68
 namespace 155
join 66
JRE
 alternate 37
 installed 37
JSP
 detail view 70

expression 76
master view 69
precompile 177
scriptlet 370
tag libraries 43
useBean 76
jStart 531

K

key class 145

L

life cycle
 file 553
lifecycle 258, 553
 project 548
literal XML 268, 309, 339
load balancing 518
local
 dealership inquiry 4, 7
 history 210
 JNDI name 162

M

macro
 CVS 238
management 521
Map.mapxmi 145
mapping 269
 custom 458
 EJB 137, 145
 XML 120
marshall 459
Master.css 61, 75
memory considerations 33
Merant 20
merge streams 233
message
 oriented service 466
 oriented style 267
 router servlet 309
 service 469
 SOAP 262
messagerouter 351
meta object framework 19
metadata 142
META-INF folder 44

method
 promote 148, 164
 read-only 148
Mighty Motors 8, 106, 134
mime type 49
model-view-controller 71
modular
 Web services 243
module
 dependencies 41
 visibility 193
monitor server 48
MQSeries 261, 493
 Integrator 493, 507
 Workflow 503
MQSI 507

N

NAICS 298, 389
namespaces
 UDDI 297
 WSDL 282
 XML 97
naming convention 59
 CVS 223
navigator view 22

O

open source 18, 215, 523
optimistic concurrency 212
outline view 23

P

package
 Java 35
packages view 26
page designer 55, 78, 370
parallel development 227
parser 101, 102, 128
part EJB 135
partner catalog 492
parts table 12
peer-to-peer computing 517
performance 33, 522
perspective 20
 customize 24
 data 28

debug 31
help 32
J2EE 27
Java 25
server 30
team 215
toolbar 21
views 21
Web 21, 25
XML 29

Plenty Parts 8
 Web service 384

plug-in 19
 Web server 186, 193
 WebSphere AEs 441

port 49
 type 284
 WSDL 283

portal UDDI 491

precompile JSP 177

preferences 22, 35
 server 50

problem 23

processes view 30

processing
 instructions 95
 XML 101

project 33
 configuration 58
 EAR 38
 EJB 44
 Java 33
 lifecycle 548
 naming convention 59
 server 47, 82
 type 221
 version 550
 Web 40

promote 164
 method 148

properties view 23

protocol 246

proxy
 testing 427
 Web service 342

publishing 51

Q

quality of service 521
quiz answers 559

R

Rational 20, 209
RDB to XML mapping editor 105
read-only method 148
Redbooks Web site 581
 Contact us xx
refactoring code 36
reference
 EJB 165
relational database design 28
release 212, 225, 554
remote
 file transfer instance 203
 interface 144
 procedure call
 see RPC
 server instance 202
 testing 200
repository
 creation 222
 management 236
resource
 history 217
RMI/IOP 242
root element 93
RPC 249, 262, 267, 275
 router servlet 282, 309
 style 288
rpcrouter 351

S

sample client 344, 356, 425
Santa Cruz Sports Cars 10, 414
SAX 102, 103
scenario
 multiple streams 231
 parallel development 227
 sequential development 224
schema
 editor 105
 XML 98
scope 271
 Web service 338, 421
script editor 77

scriptlet 370
SEAppInstall 182, 438
search 80
 dialog 27
 icon 27
secure socket layer 49
security
 CVS 236
 SOAP 520
self-contained
 Web services 243
self-describing
 Web services 243
serializable 160
serializer 455
server
 configuration 27, 49
 assign project 83
 editor 30
 view 30
 instance 27, 47
 monitor 48
 perspective 30
 preference 50
 project 47, 82
 remote 47
 start 153
 template 51
server-cfg.xml 49, 181, 237
servers view 30
service
 broker 245
 development 257
 implementation document 278, 283
 interface document 278, 284
 life cycle 258
 oriented architecture 241
 provider 245
 development 255
 proxy
 generator tool 314
 requestor 245
 development 256
service oriented architecture 239, 245
servlet
 deployment descriptor 43
 engine 243
 initialization 74
 SOAP router 351

UDDI 405
wizard 368

session
 EJB
 create 158
 Web service 374
 facade 135, 158

SGML 92

simple API for XML
 see SAX

simple object access protocol
 see SOAP

simple type 99

SOAP 6, 246
 1.1 specification 276
 admin client 352, 435
 administration 422
 binding 281, 287, 344
 body 265
 Call object 273
 client
 API 272
 proxy 397, 455
 communication style 267
 data model 266, 267
 deployment 422
 descriptor 255, 271, 350
 encoding 262, 268, 339
 envelope 262
 error handling 266
 header 265
 introduction 261
 Lite 253
 mapping 269
 message 262, 456
 communication 517
 service 471
 programming 448
 remote procedure call 262
 router servlet 347, 351
 RPC client 249
 security 338
 server 250
 implementation 270

soap.xml 350

SOAPEnabler tool 445

SQL
 query 67
 statement 66, 110, 480

DADX 483
wizard 110

state transitions 545

static service 256

stream 212, 545
 branch 546
 head 546
 merge 233
 multiple 231

style sheet 407
 HTML 59

synchronize 212

syntax highlighting 26

system files 52

T

table
 editor 28

tag library 43, 63

tasks view 23

taxonomy
 UDDI 298

TCP/IP
 Monitor view 359, 456
 Monitoring Server 48, 358, 408, 456

team
 development 224
 perspective 215
 terminology 212

template 51

terminology
 matrix 214
 team support 212

test
 client
 EJB 153
 registry 300
 Web service 332

tips
 performance 33
 variables 35

tModel 251, 293, 404

Tomcat 48, 83

toolbar 21

transformation
 XML 125

troubleshooting 530

type

complex 99
hierarchy 26
mapping 448
simple 99

U

UDDI 6, 246
 advanced 489
 API 298, 398
 browser 313, 386
 Business
 Registry 252, 300
 Test Registry 252
 categorization 297
 data model 295
 e-marketplace 490
 enterprise application integration 493
 entities 302
 entity 386
 explorer 313, 386
 export 255
 import 255
 lookup servlet 405
 namespaces 297
 node 495
 operator cloud 489
 organization 303
 overview 293
 partner catalog 492
 portal 491
 programming 399
 publishing 387
 registration 251
 registry 381
 anatomy 300
 structure 293
 taxonomy 298
 test bed 494
 Test Registry 387
 universal unique identifier 297

UDDI4J 252, 299, 398
UDDIAccessor 400
UML collaboration diagram 254
UN/SPSC 298
unified resource name
 see URN
universal description, discovery, and integration
 see UDDI

universal resource identifier
 see URI
universal test client 153, 332
universal unique identifier 297
unmarshall 459
UPES 504
URI 263
URL
 Web service 325, 354
URN 263, 272, 281
useBean 76
usejdbc2.bat 62, 538
user interface 19
UTC 153

V

validate
 CVS connection 216
 EJB project 150
 XML 115
validation
 XML 96
variable
 external JAR files 35
 host 67
variables view 31
vehicle manufacturer system 5, 7
Versata 317
 Logic Suite 317
 Transaction Logic Engine 317
version 214
 file 555
 project 550
view 21
 bean 63, 72
 breakpoint 32
 console 30
 data 28
 DB explorer 28
 debug 31
 hierarchy 26
 icons 24
 inspector 32
 navigator 22
 outline 23
 packages 26
 processes 30
 properties 23

server configuration 30
servers 30
tasks 23
TCP/IP Monitor 359
variables 31
virtual host 195, 200, 440
visibility 184, 193
VisualAge for Java 18

W
WAR file 40, 41
Web
 application 40
 archive 41
 deployment 81
 development 55
 execution 86
 deployment descriptor 41, 42
 module 27, 60
 perspective 21, 25
 project 40
 configuration 58
 files 60
service
 client 364, 373, 426
 client wizard 310
 composed 413
 consumption wizard 317
 creation 334
 creation wizard 316
 DADX 479
 DADX group configuration wizard 312, 482
 deployment 433
 design 525
 development 254
 dynamic 379
 generated files 345
 Mighty Motors 334
 proxy 342, 354
 publishing 345
 sample client 356
 scope 271, 421
 session EJB 374
 skeleton JavaBean wizard 311
 standards 304
 static 321
 test client 344

testing 332, 425
type 325
URI 337
wizard 308, 336, 420
WSDL files 347

services
 architecture 514
 conversation language 498
 description language
 see WSDL
 endpoint language 498
 flow language
 see WSFL
 inspection language 302
 invocation framework 292
 object runtime framework 487
 overview 241
 process management toolkit 504
 protocol stack 246
 terminology 305
 toolkit
 see WSTK
 tools 307

site
 import 41

web.xml 41, 42, 60, 73

WebSphere
 Administration Server 190
 AE 170, 443
 installation 539
 AEs 170, 438
 installation 539
 remote server 47
 Studio (classic) 316
 Studio Application Developer
 see Application Developer
 Studio Workbench
 see Workbench
 test environment 47, 82
 UDDI Registry 496
 installation 541

well-formed XML document 95

wizard
 Application Installation 178
 Database Web Pages 62
 EJB to RDB Mapping 138
 servlet 368
 Web service 308, 336
 XML from SQL Query 109

worf 485
Workbench 18
 architecture 19
 perspective 20
 team architecture 208
 toolbar 27
 user interface 19
 window 20
workspace 58, 210
 add project 220
 multiple 211
 synchronize 212
World Wide Web Consortium 92, 242
WSCL 498
WSDL 6, 246, 277
 1.1 specification 289
 API 290
 binding 278, 286
 document 277, 279
 export 390
 HTTP binding 289
 implementation 277
 import 394
 Java API 290
 message 285
 namespaces 282
 operation 284
 port 283
 primer 277
 schema document 337
 type 285
WSDL4J 252, 290
WSFL 247, 497
 flow model 499
WSPMTK 504
WSTK 292, 313
 architecture 313
 client runtime 315
 configuration tool 315

X

Xalan 104, 105
Xerces 103, 105, 128, 328
XLANG 498
XMethods 248, 301
XMI 269
 files 142
XML

attribute 93
comment 94
concepts 93
development 91
document 94
editor 29, 105, 113
element 94
file 113
from Java 131
from SQL Query wizard 111
Java mapping 340, 343
mapping 120
markup 94
metadata interchange
 see XMI
namespaces 97, 282
parser 101, 102, 128
perspective 29
processing 101, 125
processor 95, 101
schema 29
 see XSD
schema editor 105
tag 93
to HTML 127
to XML mapping 120
tools 105
transformation 125
validate 115
validation 96
XML to XML mapping editor 105
XML4J 103
XPath 104
XSD 29, 98
 editor 114
 type 267
XSL 101, 104
 editor 124, 366
 style sheet 366, 407, 427
 trace editor 105, 125, 127
XSL-FO 104
XSL-T 104
XSLT
 generation 123
 script 123

IBM



Redbooks

Web Services Wizardry with WebSphere Studio Application Developer

(1.0" spine)
0.875" <-> 1.498"
460 <-> 788 pages



Web Services Wizardry with WebSphere Studio Application Developer



Redbooks

Creating dynamic e-business with Web services

This IBM Redbook explores the new WebSphere Studio Application Developer for J2EE application development and WebServices. The WebSphere Studio Application Developer basic tooling and team environment is presented along with the development and deployment of Web Applications (JSPs and servlets), XML, data, EJBs and Web services.

Using the IBM toolset for Web services

WebSphere Studio Application Developer is the new IBM tool for Java development for client and server applications. It provides a Java integrated development environment (IDE) that is designed to provide rapid development for J2EE-based applications. It is well integrated with WebSphere Application Server Version 4 and provides a built-in single server that can be used for testing of J2EE applications.

Introducing WebSphere Studio Application Developer

Web services are a new breed of Web applications. Web services are self-contained, self-describing, modular applications that can be published, located, and invoked across the Web. Web services perform callable functions that can be anything from a simple request to complicated business processes. Once a Web service is deployed and registered, other applications can discover and invoke the deployed service. The foundation for Web services are the simple object access protocol (SOAP), the Web services description language (WSDL), and the Universal Description, Discovery, and Integration (UDDI) registry.

This redbook consists of three parts: an introduction of the sample auto parts application that is used throughout the book, J2EE development and deployment with WebSphere Studio Application Developer, and Web services technology along with the development and deployment of Web services in WebSphere Studio Application Developer.

**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**