

WebSphere for z/OS V6 Connectivity Handbook

Learn how to use Web services for connectivity

Acquire the latest technical details on JCA connectors and messaging

Follow sample code to get started quickly



Alex Louwe Kooijmans
Sonpreet Bhatia
Chris Backhouse
Michael Daubman
Denis Gäbler
Mitch Johnson
Edward McCarthy
Bas Otten
Rajesh Ramachandran



International Technical Support Organization

WebSphere for z/OS V6 Connectivity Handbook

December 2005

Archived

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

Third Edition (December 2005)

This edition applies to the following products:

- ▶ WebSphere Application Server V6.01 for z/OS, build level cf10515.05 Release 040
- ▶ WebSphere MQ Version 5.3.1 for z/OS
- ▶ CICS Transaction Server Version 3.1
- ▶ IMS Version 9
- ▶ UDB DB2 for z/OS Version 8.1
- ▶ Rational Application developer Version 6.0.1.0, Build 20050628

© Copyright International Business Machines Corporation 2004, 2005. All rights reserved.

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

Contents

| | |
|---|-------|
| Notices | xiii |
| Trademarks | xiv |
| Preface | xv |
| The team that wrote this redbook | xv |
| Become a published author | xviii |
| Comments welcome | xviii |
| Summary of changes | xix |
| December 2005, Third Edition | xix |
| Chapter 1. Accessing DB2 using JDBC | 1 |
| 1.1 Topology | 2 |
| 1.2 Introduction to JDBC | 2 |
| 1.2.1 JDBC Driver architecture types | 3 |
| 1.2.2 JDBC with WebSphere Application Server for z/OS Version 6.01 ... | 4 |
| 1.2.3 Configuring the IBM DB2 Universal Driver for JDBC and SQLJ | 4 |
| 1.2.4 Define the DB2 Universal JDBC driver provider | 8 |
| 1.2.5 Defining the DB2 Universal JDBC driver provider data source | 9 |
| 1.3 JDBC application programming | 11 |
| 1.3.1 Environment setup for communication with DB2 on z/OS | 11 |
| 1.3.2 Data sources in the deployment descriptors | 16 |
| 1.4 The TraderDB application | 20 |
| 1.4.1 Deploying the TraderDB Application | 20 |
| 1.4.2 Testing TraderDB with JDBC | 23 |
| Chapter 2. Accessing DB2 using SQLJ | 25 |
| 2.1 Introduction to SQLJ | 26 |
| 2.1.1 JDBC versus SQLJ | 26 |
| 2.1.2 Application connection concepts | 27 |
| 2.1.3 Connection coding differences | 28 |
| 2.1.4 SQLJ coding | 29 |
| 2.1.5 General application flow | 32 |
| 2.2 Creating an SQLJ program in RAD | 33 |
| 2.2.1 Supplied material | 33 |
| 2.2.2 Migrating the Version 5 application | 34 |
| 2.2.3 Setting up the SQLJ environment | 36 |
| 2.2.4 Adding SQLJ support to our project | 38 |
| 2.2.5 Adding an SQLJ file to the project | 42 |

| | |
|---|-----------|
| 2.2.6 Customization | 50 |
| 2.3 Testing SQLJ in TraderDB | 58 |
| 2.3.1 Running the application | 59 |
| 2.3.2 Debugging potential problems | 60 |
| 2.4 SQLJ and QoS | 61 |
| 2.4.1 Performance | 61 |
| 2.4.2 Security | 62 |
| 2.4.3 Dynamic SQL is possible in SQLJ | 62 |
| Chapter 3. Accessing IMS databases using JDBC | 63 |
| 3.1 IMS JDBC topology | 64 |
| 3.2 Configuring the servant region for ODBA | 64 |
| 3.3 Installing the IMS JDBC resource adapter | 65 |
| 3.4 Configuring connection factories | 69 |
| 3.4.1 Configuring a J2C connection factory resource | 70 |
| 3.4.2 Installing a custom service in the application server | 73 |
| 3.5 Installation verification with IMS DealerShip application | 76 |
| 3.5.1 Installing the IMS DealerShip application | 76 |
| 3.5.2 Using the IMS Dealership application | 78 |
| 3.6 Problem determination | 80 |
| Chapter 4. Messaging | 83 |
| 4.1 Messaging within J2EE/JMS technology | 85 |
| 4.1.1 MQ concepts | 85 |
| 4.1.2 JMS concepts | 86 |
| 4.2 General overview of JMS provider types | 88 |
| 4.2.1 WebSphere Default messaging provider | 89 |
| 4.2.2 WebSphere MQ JMS provider | 95 |
| 4.2.3 Generic JMS provider | 96 |
| 4.2.4 V5 Default messaging JMS Provider | 97 |
| 4.2.5 Pros and cons of the four types of JMS providers | 97 |
| 4.3 Message Driven Beans | 100 |
| 4.3.1 MDB characteristics | 101 |
| 4.3.2 Listener service | 101 |
| 4.3.3 Activation specification | 103 |
| 4.3.4 MDB programming directives | 105 |
| 4.4 Messaging security | 106 |
| 4.4.1 Identity authentication | 106 |
| 4.4.2 Message Driven Bean identity | 107 |
| 4.4.3 Resource authorization | 108 |
| 4.4.4 Confidentiality and integrity | 113 |
| 4.5 Using the WebSphere MQ JMS provider and JMS 1.0 | 114 |
| 4.5.1 Preparing the TraderMQ application | 114 |

| | | |
|-------------------|---|------------|
| 4.5.2 | Configuring the WebSphere MQ JMS provider | 123 |
| 4.5.3 | Running TraderMQ on z/OS | 132 |
| 4.6 | Using the WebSphere MQ JMS provider and JMS 1.1 | 133 |
| 4.6.1 | Converting the TRADERMQ application from JMS 1.0 to JMS 1.1 | 133 |
| 4.6.2 | Configuring an MQ (unified) connection factory | 143 |
| 4.7 | Using the Default messaging JMS provider | 148 |
| 4.7.1 | Application changes for using Default messaging | 148 |
| 4.7.2 | Configuring the Default messaging JMS provider | 149 |
| 4.8 | The TraderMQ application environment | 166 |
| 4.8.1 | JMS 1.0 TraderMQ using WebSphere MQ | 167 |
| 4.8.2 | JMS 1.1 TraderMQ using WebSphere MQ | 169 |
| 4.8.3 | JMS 1.1 TraderMQ using Default messaging | 169 |
| Chapter 5. | Introduction to J2EE Connector Architecture | 173 |
| 5.1 | Connector components | 175 |
| 5.2 | The Common Client Interface | 176 |
| 5.2.1 | Establishing a connection to a resource | 177 |
| 5.2.2 | Interacting with the resource | 178 |
| 5.3 | Enabling Rational Application Developer for J2C | 179 |
| 5.3.1 | Installing the J2EE Connector Tools feature | 179 |
| 5.3.2 | Enabling the J2C feature in RAD | 180 |
| Chapter 6. | Developing J2C applications accessing CICS | 183 |
| 6.1 | CICS Transaction Gateway topologies | 184 |
| 6.2 | CICS J2C application development | 185 |
| 6.2.1 | Backward compatibility | 185 |
| 6.2.2 | Version 6 J2C CICS support | 185 |
| 6.2.3 | Import TraderCICS ear | 186 |
| 6.2.4 | Delete TraderCICSEJB2 | 188 |
| 6.2.5 | Generating J2C CICS Connector Code | 190 |
| 6.2.6 | Rational Application Developer Version 6 and meta-data | 204 |
| 6.2.7 | Deprecated use of direct JNDI lookup | 205 |
| 6.2.8 | Update the ejb-jar.xml | 206 |
| Chapter 7. | Developing J2C applications accessing IMS | 211 |
| 7.1 | Preparing the project | 212 |
| 7.2 | Create J2C Java Data Beans | 214 |
| 7.3 | Creating a J2C Java Bean | 218 |
| 7.4 | Creating an EJB J2C Java Bean | 227 |
| 7.4.1 | Exposing connection and interaction properties | 233 |
| 7.5 | Generating SOA J2C Java Beans | 236 |
| 7.5.1 | Create J2C Java Data Beans | 241 |
| 7.5.2 | Create a J2C Java Bean | 245 |
| 7.5.3 | Creating a SOA J2C Java Bean | 254 |

| | |
|---|-----|
| 7.5.4 Testing the SOA J2C Java Bean | 256 |
| Chapter 8. Configuring J2C for CTG | 265 |
| 8.1 Installing the CICS resource adapter | 266 |
| 8.2 Configuring connection factories | 269 |
| 8.2.1 Configuring a local J2C connection factory resource | 270 |
| 8.2.2 Configuring a remote CICS J2C factory resource | 272 |
| 8.3 Cell scope J2C connection factories | 275 |
| 8.4 Command-line CICS J2C definition | 276 |
| 8.5 Application deployment | 278 |
| 8.6 TranName and TPNName | 279 |
| 8.6.1 TranName | 279 |
| 8.6.2 TPNName | 279 |
| 8.6.3 Setting TranName and TPNName | 280 |
| 8.7 Security and CICS J2C Connector | 280 |
| 8.7.1 Defining an authentication alias | 280 |
| 8.7.2 Container-managed security | 282 |
| 8.7.3 Application-managed security | 284 |
| 8.7.4 CICS J2C security decision tree | 287 |
| 8.8 Performance and availability | 288 |
| 8.9 Problem determination | 290 |
| 8.9.1 Common errors | 291 |
| 8.10 Transaction management | 292 |
| Chapter 9. Configuring J2C for IMS Connect | 295 |
| 9.1 IMS Connect topologies | 297 |
| 9.2 Configuring IMS Connect and IMS | 298 |
| 9.2.1 IMS Connect setup | 299 |
| 9.2.2 OTMA setup | 300 |
| 9.3 Installing the IMS Connect Resource Adapter | 301 |
| 9.4 Configuring connection factories | 304 |
| 9.4.1 Configuring a local J2C connection factory resource | 305 |
| 9.4.2 Configuring a remote IMS J2C factory resource | 310 |
| 9.5 Cell scope J2C connection factories | 314 |
| 9.6 Application deployment | 316 |
| 9.7 Implementing application security | 317 |
| 9.7.1 Connection properties | 318 |
| 9.7.2 Interaction properties | 318 |
| 9.8 Implementing infrastructure security | 319 |
| 9.8.1 RACF keyrings | 320 |
| 9.8.2 JSSE keyrings | 323 |
| 9.8.3 IMS Connect and SSL | 325 |
| 9.9 Configuration automation using Jython scripts | 326 |

| | |
|---|-----|
| 9.10 Performance and availability | 330 |
| 9.11 Problem determination | 331 |
| 9.11.1 Common errors | 332 |
| 9.11.2 IMS Connect authorization and authentication support | 333 |
| 9.12 Transaction management | 335 |
| Chapter 10. Introduction to Web services. | 337 |
| 10.1 Introduction | 338 |
| 10.2 Service-Oriented Architecture | 338 |
| 10.2.1 Characteristics | 340 |
| 10.2.2 Web services verses Service-Oriented Architectures | 340 |
| 10.3 Web services | 341 |
| 10.3.1 Properties of a Web service | 341 |
| 10.3.2 Core standards | 343 |
| 10.3.3 Additional standards | 344 |
| 10.4 Simple Open Access Protocol (SOAP) | 345 |
| 10.4.1 Overview | 345 |
| 10.4.2 The three pillars of SOAP | 346 |
| 10.4.3 SOAP elements | 347 |
| 10.5 Web Services Definition Language (WSDL) | 353 |
| 10.5.1 WSDL document | 353 |
| 10.5.2 WSDL document anatomy | 354 |
| 10.5.3 WSDL definition | 358 |
| Chapter 11. Web services using SOAP over HTTP | 361 |
| 11.1 An end-to-end Web service example | 362 |
| 11.2 Simple Example Web Services (SEWS) application | 363 |
| 11.2.1 WebSphere part of SEWS | 363 |
| 11.2.2 CICS part of SEWS | 365 |
| 11.2.3 Our system layout | 367 |
| 11.2.4 Supplied material | 368 |
| 11.3 Web service enabling SEWS for WebSphere | 369 |
| 11.3.1 Overview | 369 |
| 11.3.2 Web service-enable SnoopInfo Java Bean | 370 |
| 11.3.3 Test SEWS in Rational Application Developer Version 6 | 380 |
| 11.3.4 Export Web service client code | 382 |
| 11.4 Development of Web service invoker | 383 |
| 11.4.1 Java Code that performs the WebSphere Web service call | 384 |
| 11.4.2 Java code that calls the CICS Web service | 384 |
| 11.4.3 Import the SnoopInfoBean client WAR file | 385 |
| 11.5 CICS - Web service-enable business logic | 385 |
| 11.5.1 Generate WSDL from a CICS business logic program | 386 |
| 11.5.2 Import WSDL into project | 388 |

| | |
|---|------------|
| 11.6 CICS - Performing a Web service request | 391 |
| 11.6.1 Defining SnoopInfo Web service to CICS | 393 |
| 11.7 Web service definitions required in CICS | 396 |
| 11.7.1 DFHPIPE | 396 |
| 11.7.2 TCPIP and TCPIP Service | 397 |
| 11.7.3 URIMAP | 397 |
| 11.7.4 Pipeline | 398 |
| 11.7.5 WEBS (Webservice) | 401 |
| 11.8 Testing the SEWS application | 402 |
| 11.8.1 Running the Web service requestor application | 403 |
| 11.8.2 Test Web service call to SnoopInfo Bean in WebSphere | 403 |
| 11.8.3 Test Web service call to SnoopInfo in CICS | 404 |
| 11.8.4 Test Web service call from CICS to WebSphere | 405 |
| 11.9 Summary | 406 |
| Chapter 12. Web services and SOAP over JMS | 407 |
| 12.1 Using SOAP over JMS | 408 |
| 12.1.1 Our system layout | 408 |
| 12.1.2 Supplied material | 410 |
| 12.2 Enabling SnoopEjb for SOAP over JMS | 410 |
| 12.2.1 Web service-enable the SnoopEJB | 411 |
| 12.2.2 Enabling SOAP over JMS support | 412 |
| 12.2.3 Exploring the new ejb-jar.xml | 414 |
| 12.2.4 Update the application.xml | 415 |
| 12.2.5 Update sewsWsDemo to invoke SnoopEjb Web service | 415 |
| 12.2.6 Update the web.xml | 417 |
| 12.3 Web service using SOAP over JMS in one cell | 419 |
| 12.3.1 Overview of definitions required | 419 |
| 12.3.2 Defining a SIB in WebSphere | 420 |
| 12.3.3 Define authentication alias | 424 |
| 12.3.4 Define queue | 424 |
| 12.3.5 Define JMS queue | 425 |
| 12.3.6 Define JMS queue connection factory | 426 |
| 12.3.7 Define JMS activation specification | 428 |
| 12.3.8 Definitions for the Web service driver application | 430 |
| 12.3.9 Testing in one server | 431 |
| 12.3.10 Preventing deprecated JNDI lookup | 436 |
| 12.3.11 Security errors without authentication alias | 437 |
| 12.3.12 Checking queue status | 440 |
| 12.4 Web service using SOAP over JMS between cells | 441 |
| 12.4.1 WebSphere servers used example | 441 |
| 12.4.2 Second server configuration | 442 |
| 12.4.3 Define authentication alias | 442 |

| | | |
|-------------------|---|------------|
| 12.4.4 | Define SIB in ws6483 | 442 |
| 12.4.5 | Define foreign bus in ws6483 | 443 |
| 12.4.6 | Define foreign bus in ws6481 | 444 |
| 12.4.7 | Verify inter-bus connectivity | 445 |
| 12.4.8 | Define sewsQueue in ws6483 | 446 |
| 12.4.9 | Define JMS queue connection factory | 447 |
| 12.4.10 | Define JMS queue | 447 |
| 12.4.11 | Test using two WebSphere cells | 448 |
| 12.5 | CICS to WebSphere using SOAP over JMS | 448 |
| 12.5.1 | CICS prereqs | 448 |
| 12.5.2 | Define foreign bus | 449 |
| 12.5.3 | Define WebSphere MQ Link | 449 |
| 12.5.4 | MQ definitions | 452 |
| 12.5.5 | Define SnoopEjb Web service to CICS | 453 |
| 12.5.6 | Test Web service - CICS to WebSphere | 455 |
| 12.6 | WebSphere to CICS using SOAP over JMS | 456 |
| 12.6.1 | WebSphere definitions | 456 |
| 12.6.2 | MQSeries definitions | 458 |
| 12.6.3 | CICS Web service | 459 |
| 12.6.4 | Test Web service - WebSphere to CICS | 459 |
| 12.7 | Summary | 460 |
| Chapter 13 | Integrating WebSphere and DB2 using Web services | 461 |
| 13.1 | Calling a DB2 stored procedure as a Web service | 462 |
| 13.1.1 | Why stored procedures | 463 |
| 13.1.2 | When not to use stored procedures | 463 |
| 13.1.3 | Developing the Trader sample application | 464 |
| 13.1.4 | Enabling WebSphere to execute DB2 Web services | 483 |
| 13.1.5 | Deployment and testing | 485 |
| 13.2 | Calling a Web service from DB2 | 486 |
| 13.2.1 | Prerequisites for development and deployment | 486 |
| 13.2.2 | Invoking the Web service consumer | 486 |
| 13.2.3 | The Trader sample | 487 |
| 13.3 | Summary | 489 |
| Chapter 14 | Connecting to WebSphere from IMS using SOAP | 491 |
| 14.1 | Setup | 493 |
| 14.1.1 | WebSphere requirements | 493 |
| 14.1.2 | z/OS requirements | 493 |
| 14.1.3 | IMS requirements | 494 |
| 14.1.4 | Creating a default OMVS segment | 495 |
| 14.2 | Generating Java code to access the Web service | 496 |
| 14.2.1 | Case of missing Web services menu | 507 |

| | |
|--|------------|
| 14.3 Test the Web service. | 510 |
| 14.4 Calling Web service from within IMS Java transaction | 511 |
| 14.4.1 Preparing RAD for use of ANT scripting | 512 |
| 14.4.2 Importing the IMS Java classes into the project classpath | 513 |
| 14.4.3 Build the IMS Java transaction to call the Web service | 523 |
| 14.4.4 Required WAS classes for calling a Web service | 529 |
| 14.4.5 Application settings in IMS PROCLIB | 530 |
| 14.4.6 Testing the IMS Java transaction from a 3270 terminal | 533 |
| 14.4.7 Test IMS Java transaction. | 535 |
| 14.4.8 Automatic class reloading for testing | 536 |
| Chapter 15. Using the RMI-IIOP protocol | 537 |
| 15.1 Calling EJBs and IIOP. | 539 |
| 15.1.1 Basics of remote EJB invocation. | 539 |
| 15.1.2 Supplied material | 539 |
| 15.1.3 ejbMagic application | 540 |
| 15.1.4 SnoopMagicEjb methods | 541 |
| 15.1.5 Understanding logical and direct JNDI lookups | 542 |
| 15.1.6 ejbMagic application - Logical EJB references | 546 |
| 15.1.7 Our system layout | 546 |
| 15.2 Running ejbMagic in one server | 547 |
| 15.2.1 URL to invoke the servlet | 547 |
| 15.2.2 Testing the ejbMagic application. | 549 |
| 15.3 Running ejbMagic between two cells | 551 |
| 15.3.1 Call remote EJB test | 551 |
| 15.3.2 Handling corbanames | 554 |
| 15.3.3 Name space bindings | 555 |
| 15.4 CSI | 559 |
| Appendix A. The Trader application explained. | 561 |
| Overview of Trader application. | 562 |
| Trader IMS and CICS applications and data stores | 563 |
| Trader Web front-end user interface | 564 |
| Trader Web front-end and back-end interface | 566 |
| Packaging | 570 |
| Deploying Trader applications to WebSphere for z/OS | 574 |
| Running the Trader applications | 575 |
| Additional configuration information | 575 |
| Trader DB2 table definitions | 575 |
| Trader VSAM file definitions. | 577 |
| Appendix B. Additional material | 579 |
| Locating the Web material | 579 |
| Using the Web material | 579 |

| | |
|--|-----|
| System requirements for downloading the Web material | 580 |
| How to use the Web material | 580 |
| Related publications | 581 |
| IBM Redbooks | 581 |
| Other publications | 581 |
| Online resources | 582 |
| How to get IBM Redbooks | 582 |
| Help from IBM | 583 |
| Index | 585 |

Notices

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

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

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

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

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

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

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

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


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

COPYRIGHT LICENSE:

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

Trademarks

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

| | | |
|---|-------------------------|------------|
| @server® | CICSplex® | OS/390® |
| @server® | DB2 Connect™ | Rational® |
| Redbooks (logo)  ™ | DB2 Universal Database™ | Redbooks™ |
| developerWorks® | DB2® | RACF® |
| z/OS® | DFS™ | S/390® |
| zSeries® | Everyplace® | System z9™ |
| z9™ | IBM® | TXSeries® |
| Cloudscape™ | IMS™ | VTAM® |
| CICS Connection® | MQSeries® | WebSphere® |
| CICS® | MVS™ | |

The following terms are trademarks of other companies:

Enterprise JavaBeans, EJB, Forte, Java, Java Naming and Directory Interface, JavaBeans, JavaMail, JavaServer, JavaServer Pages, JDBC, JDK, JSP, JVM, J2EE, Sun, and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

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

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

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

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

Preface

This IBM® Redbook is an update to the *WebSphere for z/OS V5 Connectivity Handbook*, SG24-7064-01, and helps you to create integration solutions with WebSphere Application Server for z/OS Version 6.01.

This publication covers development, deployment, and configuration aspects of the most common integration and connectivity scenarios. Most scenarios are explained using a sample application, which is available for download as additional material.

This publication specifically focuses on the usage of WebSphere® Application Server on the z/OS® platform, but a large part of the information is useful for other platforms, too.

This publication does *not* fully replace the *WebSphere for z/OS V5 Connectivity Handbook*, SG24-7064-01. Some chapters have been updated according to new versions of products, but there are also chapters that have not been updated and not included in this book this time.

- ▶ If you are a WebSphere Version 5 user, you should keep on using the *WebSphere for z/OS V5 Connectivity Handbook*, SG24-7064-01, as a reference book. You may use this publication for those topics that have not been covered in *WebSphere for z/OS V5 Connectivity Handbook*, SG24-7064-01, but you should note, however, that some information in this book may not be valid for Version 5.
- ▶ If you are a WebSphere Version 6 user, you should use this publication as a reference book. You may use the *WebSphere for z/OS V5 Connectivity Handbook*, SG24-7064-01, for those topics that we have not covered, but you should note, however, that some information in *WebSphere for z/OS V5 Connectivity Handbook*, SG24-7064-01, may not be valid anymore for Version 6.

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, Poughkeepsie Center.

Alex Louwe Kooijmans is a Project Leader at the International Technical Support Organization, Poughkeepsie Center. He leads residencies to create

Redbooks™ in the areas of Java™ and WebSphere on z/OS. He also instructs workshops in this area. Alex is on assignment to the ITSO for the second time. Before joining the ITSO, he worked in various other roles, such as an IT Specialist supporting customers in Europe getting started with WebSphere on various platforms, and a Client IT Architect in the Financial Services Sector in The Netherlands. He has much experience with Java and WebSphere on z/OS, and integrating WebSphere with DB2®, MQ, CICS®, and IMS™. He has worked for IBM since 1986.

Chris Backhouse is a staff Software Engineer in the CICS Transaction Server development organization. He has a degree in Computer Engineering from the University of Southampton. Chris's areas of expertise include CICS TS, enterprise connectivity with a focus on Web services, J2EE™, and enterprise application development. Prior to his development role he spent four years working as a designer on enterprise integration scenario projects.

Sonpreet Bhatia is an IT Architect for IBM, USA, with the IBM @server Client Technology Center. She holds a Masters of Science (Information Systems) degree from Marist College, Poughkeepsie, NY. Her area of expertise is Enterprise Integration and Service Oriented Architecture using the WebSphere Business Integration suite. She is IBM Certified for WBI Message Broker V5 (IBM Certified Solution Designer) and WebSphere MQ (IBM Certified Systems Administrator).

Michael Daubman is a Staff Software Engineer for the zSeries® and System z9™ Client Technology Center (CTC), where he specializes in enterprise security with a focus on RACF®, PKI, and WebSphere security. Before his services role, Michael worked as a developer and function tester in the z/OS Security Server team. Michael holds a Bachelor's degree in Computer Science from the Rochester Institute of Technology and an MBA from Marist College. He was also a contributing author for the IBM Redbook *WebSphere Application Server for z/OS V5 and J2EE 1.3 Security Handbook*, SG24-6086.

Denis Gäbler is a Technical Sales Specialist at IBM in Germany. He holds a degree in business with a specialization in computer science from Staatliche Studienakademie Dresden. He has been working in the areas of IMS and IMS Web enablement since 1997. His areas of expertise include WebSphere on z/OS, IMS DB and TM, Service Oriented Architectures and Enterprise Application Integration. Lately he is working extensively with IMS Java, IMS connectivity solutions, COBOL and Java Integration, application servers, and Eclipse-based application development tools. If you have any questions in the mentioned areas feel free to contact him at gaebler@de.ibm.com.

Mitch Johnson is a Consulting IT Specialist in IBM Software Services for WebSphere (ISSW) at the IBM Research Triangle Laboratory. His areas of expertise include enterprise connectivity and installation, configuration, and

administration of WebSphere for z/OS and IMS as well as CICS, TXSeries®, and DB2 on various platforms. He holds a Bachelor of Science degree in computer science from North Carolina State University in Raleigh, North Carolina.

Edward McCarthy currently works on the e-business Enablement Services team for IBM Global Services Australia. The team is responsible for covering all aspects of WebSphere Application Server and WebSphere MQ across all platforms. For the last three years, he has specialized in supporting the WebSphere range of products. Previously, he worked as a senior CICS and WebSphere MQ systems programmer for over eight years with a large IBM client. He also participated in writing the IBM Redbooks *IBM WebSphere Everyplace® Server, Enterprise Wireless Applications*, SG24-6519, and *WebSphere Business Integration Server Foundation V5.1 for z/OS*, SG24-6382.

Bas Otten got involved with mainframe technology when he participated in an OS/390® traineeship in 1997, after studying physics. Since then he has been working at the Belastingdienst/Centrum voor ICT in the Netherlands in a number of different z/OS-related roles. He first specialized in the area of capacity and performance, and later on he worked for two years as a z/OS systems programmer. Then he turned his attention towards MQ Workflow and became more acquainted with the WebSphere suite. By the end of 2004 he joined the project that started with the installation and configuration of WebSphere Application Server 5.1 on z/OS, where he is mainly working on connector configuration, design issues, and documentation.

Rajesh Ramachandran is an Advisory Software Engineer in IBM zSeries e-Business Services. He has 11 years of experience in application development in various platforms, including mainframe, UNIX®, and Linux®. He used COBOL, Java, CICS, and Forte™ on his assignments. Recently he was working in DB2 tools development, where he was a lead developer of DB2 Data Archive Expert.

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

The team that wrote *WebSphere for z/OS V5 Connectivity Handbook*, SG24-7064-01:

Tamas Vilaghy, Robert Cronin, Hidenori Fujioka, Mitch Johnson, Wilbert Kho, Rajesh Ramachandran, Allen Schmutzler and Claus Schroeder-Hansen

Rich Conway
International Technical Support Organization, Poughkeepsie Center

The editing team in the International Technical Support Organization, Poughkeepsie Center:

Terry Barthel, Allison Chandler, Al Schwab, and Julie Czubik

Ivey Ho and Leho Nigul
IBM Toronto lab

Patsy Yu, Vivek prasad and Dirk Wollscheid
IBM Santa Theresa lab, USA

Mark Cocker, Brian De Pradine
IBM Hursley lab

Become a published author

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

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

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

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

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

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

ibm.com/redbooks

- Send your comments in an email to:

redbook@us.ibm.com

- Mail your comments to:

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

Summary of changes

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

Summary of Changes for SG24-7064-02 for WebSphere for z/OS V6 Connectivity Handbook as created or updated on December 28, 2005.

December 2005, Third Edition

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

New information

- ▶ Access to DB2 data using JDBC™ and SQLJ technology
- ▶ Access to DB2 Stored Procedures using Web services
- ▶ Using WebSphere Default Messaging
- ▶ Using Web services to/from WebSphere Application Server for z/OS Version 6.01
- ▶ Using RMI/IIOP as communication protocol between WebSphere servers
- ▶ Using Web services from IMS to WebSphere Application Server for z/OS Version 6.01

Changed information

- ▶ The J2C connector information (CICS Transaction Gateway and IMS Connect) has been updated to new product levels.
- ▶ All development-related information has been based on the usage of Rational Application Developer Version 6, instead of WebSphere Studio Application Developer Integration Edition.
- ▶ Accessing WebSphere MQ using the new JMS 1.1 specification.

Deleted information

- ▶ Some of the security-related information has been left out of this publication, but will be incorporated in a security redbook.

Accessing DB2 using JDBC

There are various ways of connecting to DB2 from WebSphere Application Server for z/OS Version 6.01 on z/OS, including JDBC Type 2 and Type 4 technology. In this chapter we explain how to configure a connection-pooled data source using WebSphere for z/OS V6 with the IBM DB2 Universal Driver for JDBC and SQLJ in Type 2 mode. Since this is a Type 2 connection, DB2 will reside on the same z/OS system image as the application.

Java Database Connectivity (JDBC) is an Application Programming Interface (API) that Java applications can use to access any relational database. JDBC is similar to the Open Database Connection (ODBC) and is based on the X/Open Structured Query Language (SQL) Call Level Interface (CLI) specification.

1.1 Topology

In order to demonstrate connection to DB2, this chapter utilizes a version of the TraderDB application. The TraderDB application has been used in previous Redbooks on WebSphere for z/OS. The version updated for this book includes three access methods for connecting to DB2. These are demonstrated by Figure 1-1.

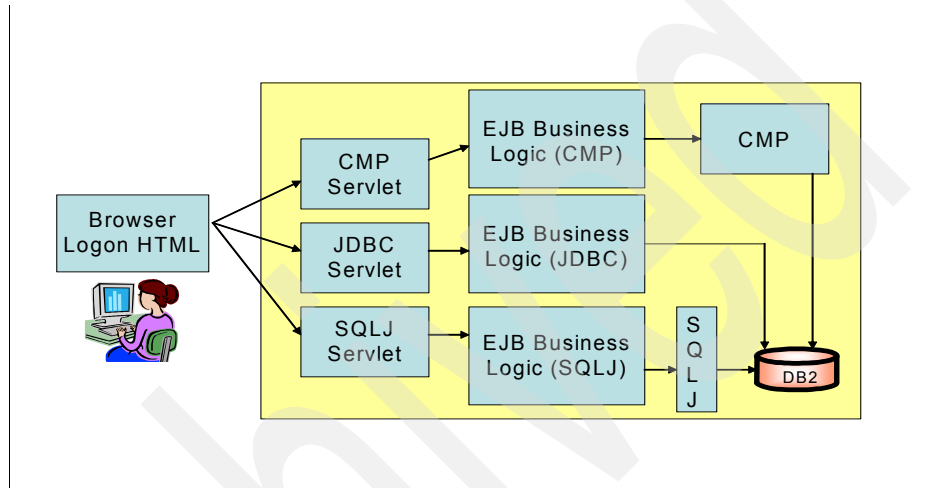


Figure 1-1 TraderDB high level view

1.2 Introduction to JDBC

To enable JDBC applications for DB2, an implementation of the various Java classes and interfaces, as defined in the standard, is required. This implementation is known as a JDBC Driver. DB2 UDB offers a complete set of JDBC Drivers for this purpose. The JDBC Drivers are categorized as the legacy/CLI drivers, offering Type 2 connectivity, and the new Universal JDBC Drivers, offering both Type 2 and Type 4 connectivity.

DB2 UDB's Java support includes support for JDBC, a vendor-neutral dynamic SQL interface that provides data access to your application through standardized Java methods. JDBC is similar to DB2 CLI in that you do not have to pre-compile the application code or bind packages to a DB2 database. As a vendor-neutral standard, JDBC applications offer increased portability—a required benefit in today's heterogeneous business infrastructure. During the execution of a JDBC application the driver will validate SQL statements against the currently connected DB2 database server. Any problems during access will be reported to

the application as a Java exception along with the corresponding SQLSTATE and SQLCODE.

The IBM DB2 Universal Driver for JDBC and SQLJ also supports access from Java using SQLJ. This topic is covered in considerable detail in Chapter 2, “Accessing DB2 using SQLJ” on page 25.

1.2.1 JDBC Driver architecture types

There are several different driver architecture types to choose from when connecting to a database, but note that not all of them may be available for a certain combination of database system, application server, and the platform on which they are running. Always check out the specifications of a certain JDBC technology to find out whether it is supported for the combination of application server and database system you will be using. The JDBC technology offered by DB2 UDB for z/OS Version 8 in combination with WebSphere Application Server for z/OS Version 6.01 only provides Type 2 and Type 4 connectivity. For completeness, the different types are explained in detail in Table 1-1.

Table 1-1 Driver types

| Driver type | Description | IBM supported type |
|-------------|---|--|
| Type 1 | Based on the JDBC-ODBC bridge. JDBC Driver provided by Sun™ can be used with an ODBC driver. | No. This is not a recommended environment. |
| Type 2 | OS-specific communication with an RDBMS. In z/OS this driver type allows for quick and direct communication with DB2. JNI-based connectivity. | Yes. In our example application, we used a Type 2 driver. |
| Type 3 | Pure Java implementation that communicates with a DB2 JDBC Applet server to access DB2 data. | Not in future releases. While DB2 can use a type 3 driver, WebSphere does not support it. Also, the type 3 driver will no longer be enhanced. The Type 4 driver is considered the replacement. |

| Driver type | Description | IBM supported type |
|-------------|---|--|
| Type 4 | Java driver that connects directly to DB2. All-java connectivity. | Yes. Along with Type 2 support, the Universal JDBC Driver supports Type 4 connections. |

If the Java application is executing on the same machine as the DB2 server, then the best performance is with the Type 2 driver because TCP/IP overhead can be avoided and therefore throughput improves. You should always consider writing more efficient SQL statements, or use SQLJ or stored procedures if you are attempting to improve overall Java application performance.

1.2.2 JDBC with WebSphere Application Server for z/OS Version 6.01

Below are the steps involved with using the DB2 Universal JDBC Driver for z/OS with WebSphere Application Server for z/OS Version 6.01. More information about each step can be found following the list of instruction.

1. Verify that the IBM DB2 Universal Driver for JDBC and SQLJ is installed and available in an HFS directory.

Check the system or ask a system administrator to detail the DB2 paths on the system. In the ITSO system used in this book, the path was `/usr/lpp/db2/<db2 group attach name>/jcc/classes`. Most installations should have a similar path. After finding and verifying the location of the IBM DB2 Universal Driver for JDBC and SQLJ, make note of this directory because it will be needed in a future setup step.

2. Configure the IBM DB2 Universal Driver for JDBC and SQLJ. This step is further explained in “Configuring the IBM DB2 Universal Driver for JDBC and SQLJ” on page 4.
3. Define an IBM DB2 Universal Driver for JDBC and SQLJ for WebSphere Application Server for z/OS Version 6.01. This step is further explained in “Define the DB2 Universal JDBC driver provider” on page 8.
4. Define a IBM DB2 Universal Driver for JDBC and SQLJ provider data source. This step is further explained in “Defining the DB2 Universal JDBC driver provider data source” on page 9.

1.2.3 Configuring the IBM DB2 Universal Driver for JDBC and SQLJ

Environment variables need to be added to WebSphere to indicate where the installed IBM DB2 Universal Driver for JDBC and SQLJ is located, where the license file is, and where the location of the native files are.

1. Log on to the WebSphere admin console.

2. Expand **Environment** and select **WebSphere Variables**.
3. Update the variables as listed below. In the event that they do not exist, create them with a *node* scope.

Note: The path used for the environment variable values is the path where the IBM DB2 Universal Driver for JDBC and SQLJ was found in “JDBC with WebSphere Application Server for z/OS Version 6.01” on page 4.

| | |
|--------------|-------------------------------------|
| Name | DB2UNIVERSAL_JDBC_DRIVER_NATIVEPATH |
| Value | /usr/lpp/db2/d8ig/jcc/lib |

| | |
|--------------|-------------------------------|
| Name | DB2UNIVERSAL_JDBC_DRIVER_PATH |
| Value | /usr/lpp/db2/d8ig/jcc/classes |

4. Bind the required DB2 packages.

As with any application that executes SQL statements in DB2 for z/OS, the IBM DB2 Universal Driver for JDBC and SQLJ must first bind with DB2 the packages that represent the SQL statements to be executed. The IBM DB2 Universal Driver for JDBC and SQLJ does not use the same packages used by the legacy JDBC Driver, and uses a different process for binding its packages.

The specific details of the bind utility and bind process are described by the readme provided with the installed IBM DB2 Universal Driver for JDBC and SQLJ. Refer to this readme for details on how to set up and perform the required binding.

You must perform the bind process for each target DB2 that will be accessed using the DB2 Universal JDBC Driver.

The BIND job in Example 1-1 binds the JDBC default packages and plan and permits authorization to use the plan DSNJDBC.

Example 1-1 Setting the properties for the DSNJDBC plan

```
//DSNTJJCL JOB (999,P0K), 'DB4B INSTALL', CLASS=A, MSGCLASS=T,
// NOTIFY=xxxxx, TIME=1440, REGION=OM
/*JOBPARM L=9999, SYSAFF=your system here
//*****
/** JOB NAME = DSNJJCL                               */
/**                               */
/** DEPENDENCIES = JDBC MUST BE INSTALLED             */
/**                               */
//*****
//JOBLIB DD DISP=SHR,
//          DSN=DSN710.SDSNLOAD
```

```

//BINDJDBC EXEC PGM=IKJEFT01,DYNAMNBR=20
//DBRMLIB DD DISP=SHR,
//          DSN=DSN710.SDSNDBRM
//SYSTSPRT DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//SYSTSIN DD *
DSN SYSTEM(DB4B)
  BIND PACKAGE (DSNJDBC) MEMBER(DSNJDBC1) ISOLATION(UR)
  BIND PACKAGE (DSNJDBC) MEMBER(DSNJDBC2) ISOLATION(CS)
  BIND PACKAGE (DSNJDBC) MEMBER(DSNJDBC3) ISOLATION(RS)
  BIND PACKAGE (DSNJDBC) MEMBER(DSNJDBC4) ISOLATION(RR)
  BIND PLAN(DSNJDBC) -
    PKLIST(DSNJDBC.DSNJDBC1, -
           DSNJDBC.DSNJDBC2, -
           DSNJDBC.DSNJDBC3, -
           DSNJDBC.DSNJDBC4)
END
/*
//GRANT EXEC PGM=IKJEFT01,DYNAMNBR=20
//SYSTSPRT DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//SYSTSIN DD *
DSN SYSTEM(DB4B)
  RUN PROGRAM(DSNTIAD) PLAN(DSNTIA71) -
    LIBRARY('DB2V714B.RUNLIB.LOAD')
END
//SYSIN DD *
GRANT EXECUTE ON PLAN DSNJDBC TO PUBLIC;
/*

```

5. Define a db2.jcc.properties file.

This step is not really required if default settings are good for a particular environment. In our case, we wanted to specify a specific DB2 subsystem identifier for the driver to use. Since the default will pull this value from the DSNHDECP load module, we used the properties file to specify our choice.

Create the properties file. A logical place is in the installation directory tree of your WebSphere server in the properties subfolder. For the ITSO installation, this was /WebSphereAL1/V6R0/BS01/AppServer/properties.

Within this directory, we created a file called db2jcc.properties, as shown below.

Example 1-2 db2jcc.properties file

```
db2.jcc.ssid=D8I1
```

Note: This option specifies the subsystem name of the DB2 system that we want to access from this server. Change the value D8I1 as appropriate for your installation. The above entry is required only if DB2 datasharing is being used. However, it is recommend that you add the entry for documentation purposes even if DB2 datasharing is not being used.

Driver-generic properties are specified at the server level. To set those properties, navigate in the Administrative Console to:

- a. **Servers** and select **Application Servers**.
- b. Click the server that needs to use the JDBC Driver.
- c. Under the Server Infrastructure section, expand **Java and Process Management**.
- d. Click **Process Definition**.
- e. Select the **Servant** region.
- f. Under the Additional Properties section click **Java Virtual Machine**.
- g. Under the Additional Properties section click **Custom Properties**.
- h. Click **New** and define the property pointing to a file that was created earlier in this step (see Figure 1-2 on page 8).

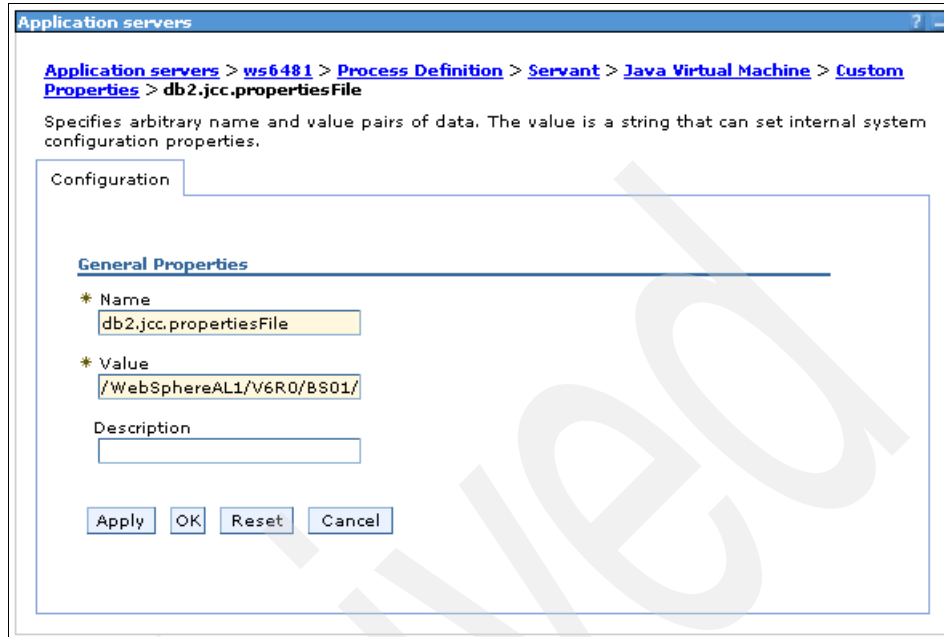


Figure 1-2 DB2 JCC properties file

1.2.4 Define the DB2 Universal JDBC driver provider

It is time now to set up a server to use the configured IBM DB2 Universal Driver for JDBC and SQLJ. IBM suggests that the DB2 Universal JDBC driver provider be set up at the server level. This helps to avoid conflicts with the DB2 for z/OS local JDBC provider (using RRS) that uses the DB2 Legacy Driver.

In order to define a DB2 Universal JDBC driver provider:

1. From the WebSphere Administrative Console, expand the **Resources** section.
2. Click **JDBC Providers** in the expanded section.
3. Change the scope to server. To do this, click the radio button next to **server** and click **apply**.
4. If there is not already a valid DB2 Universal JDBC driver provider defined, click **New**. If this is one defined, verify the setting are correct.
5. After clicking on **New**, the JDBC provider's configuration is started. Make your selection for the DB2 Universal JDBC driver provider (see Figure 1-3 on page 9). Your setting may be different if you are using XA connectivity (not used in our example).

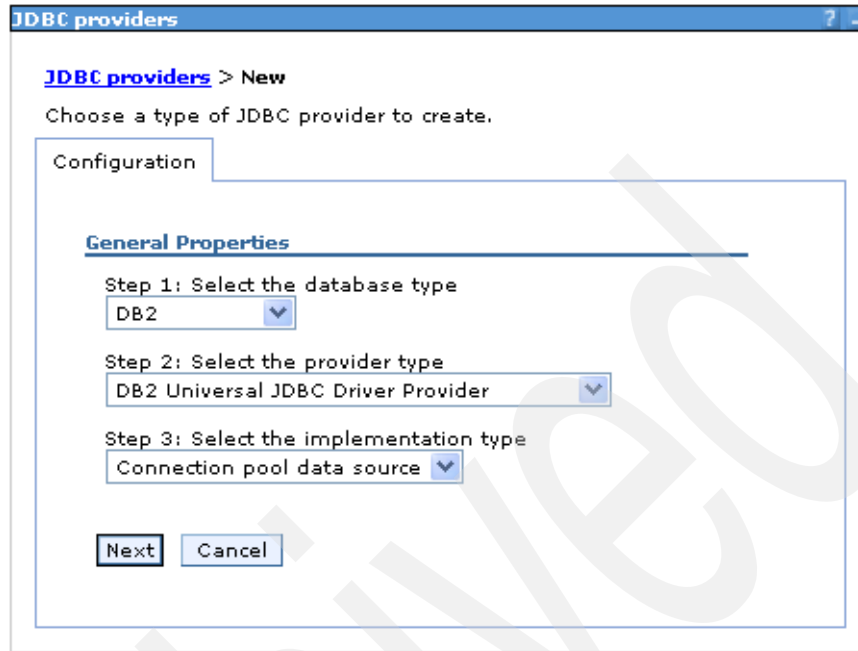


Figure 1-3 JDBC provider's configuration

6. Click **Next** and verify that the settings are correct. No additional changes should be needed.
7. Click **Apply** to define this provider.
8. Save the new setting to the configuration.

1.2.5 Defining the DB2 Universal JDBC driver provider data source

A data source must be set up for the provider. A data source details the access information for this particular DB2 subsystem. To configure a data source, follow the following steps:

1. From the WebSphere Administrative Console, expand the **Resources** section.
2. Click **JDBC Providers**.
3. If not already set, change the scope to **server**.
4. Click the previously defined DB2 Universal JDBC driver provider.
5. Under Additional Properties select **Data sources**.

6. Click **New** to add a data source. We will define a Type 2 data source because the DB2 system that we are using is on the same physical server. This offers better performance over a Type 4 connection.
7. Change the name as desired, but most importantly, set the database name to the DB2 location name¹ and the driver type to 2 (if Type 2 is desired). If using a Type 4 driver, the server name and port number will also be needed (see Figure 1-4).

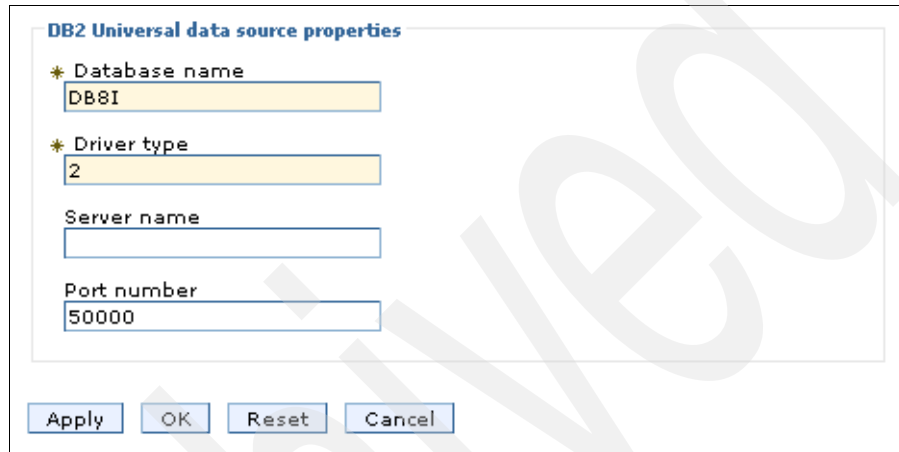


Figure 1-4 Bottom, required section of data source

8. Click **Apply** and notice a new link called Custom Properties is now available.
9. Click the **Custom Properties** link to set up any other needed data source properties. We will need one specific property set up. The currentSQLID should be set to TRADER. Setting this property is equivalent to the SQL command SET CURRENT SQLID. Because of this, connection IDs using this data source will need RACF authority enabling this command to be executed.
10. Save the changes to the configuration.

A deployed application should now be able to use this provider to access DB2 through the IBM DB2 Universal Driver for JDBC and SQLJ.

¹ Note that the DB2 location name is not the same as the DB2 Subsystem ID (SSID). They may have different values.

1.3 JDBC application programming

Developing an application that uses JDBC to speak with DB2 can be a challenge. Having DB2 on a z/OS system can offer its own set of challenges. This chapter explains:

- ▶ Setting up the development environment to enable communication with DB2 on z/OS, explained further in “Environment setup for communication with DB2 on z/OS” on page 11.
- ▶ Deployment descriptor data source information, explained further in “Data sources in the deployment descriptors” on page 16.

Our explanation is based on the TraderDB application, which is part of the additional material for this book.

1.3.1 Environment setup for communication with DB2 on z/OS

When programming on a workstation, it is often desired to test with a local deployment of the application connecting to the z/OS system for DB2 access and data. This is possible in a variety of ways. We focus on the required setup for two of these methods. The first will be the creation of a simple Type 4 connection. The second will be the use of IBM DB2 Connect™ to connect to our DB2 on a z/OS system and have our application use this connection.

Local deployment with a Type 4 connection

To do this:

1. Deploy the application locally by clicking the **Server** tab, right-clicking the desired server, and clicking **Add and remove projects**.

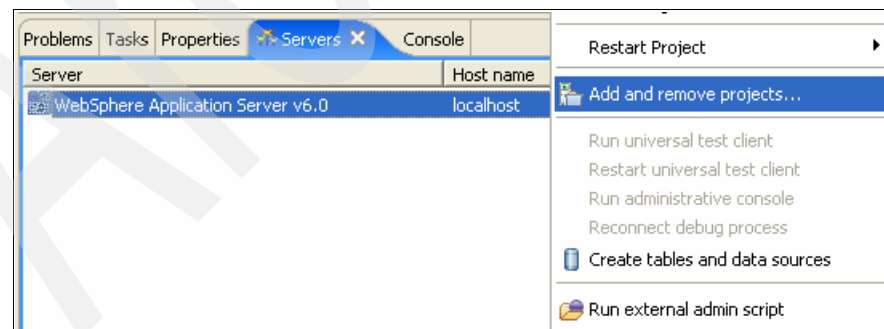
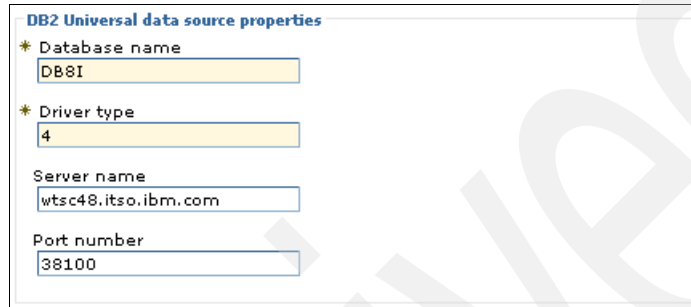


Figure 1-5 Local deployment

2. Add the TraderDB2004 project and click **Finish**. You can find this application in the additional material for this book.

3. The application is now deployed. Since the code specifies a JNDI name for a DB2 connection, the code will not work yet. We will now set up a Type 4 connection on our local server.
4. Right-click the server again and click the **Run administrative console** link.
5. Follow the same process as described in 1.2.4, “Define the DB2 Universal JDBC driver provider” on page 8, and 1.2.5, “Defining the DB2 Universal JDBC driver provider data source” on page 9, except that this time we want a Type 4 data source so we can get to the remote DB2 system. See Figure 1-6.



The screenshot shows a dialog box titled "DB2 Universal data source properties". It contains four input fields: "Database name" with the value "DB81", "Driver type" with the value "4", "Server name" with the value "wtsc48.itso.ibm.com", and "Port number" with the value "38100".

Figure 1-6 Type 4 connection

For this connection, the server name and port number of the DB2 on z/OS system are important.

Using DB2 Connect for the connection

For this book, we had DB2 Connect Version 8.1 installed. Sometimes developers prefer using DB2 Connect because a Type 2 connection can be configured locally and also, the tooling can be used to test SQL commands and look directly at the database on z/OS. To set up the development environment to use a DB2 Connect connection, there are a few simple steps that must be followed.

1. Install IBM DB2 Connect.
2. Start the Configuration Assistant (usually **Start** → **All Programs** → **IBM DB2** → **Set-up Tools** → **Configuration Assistant**).
3. Choose **Selected** → **Add Database Using Wizard**, as shown in Figure 1-7 on page 13.

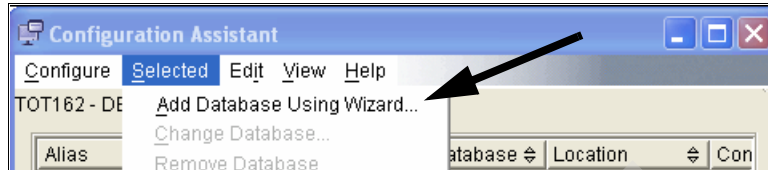


Figure 1-7 Add database

4. Choose the **Manually configure a connection to a database** radio button.
5. Click **Next**.
6. Choose **TCP/IP** for this connection and check the box indicating that this database resides on a host system.

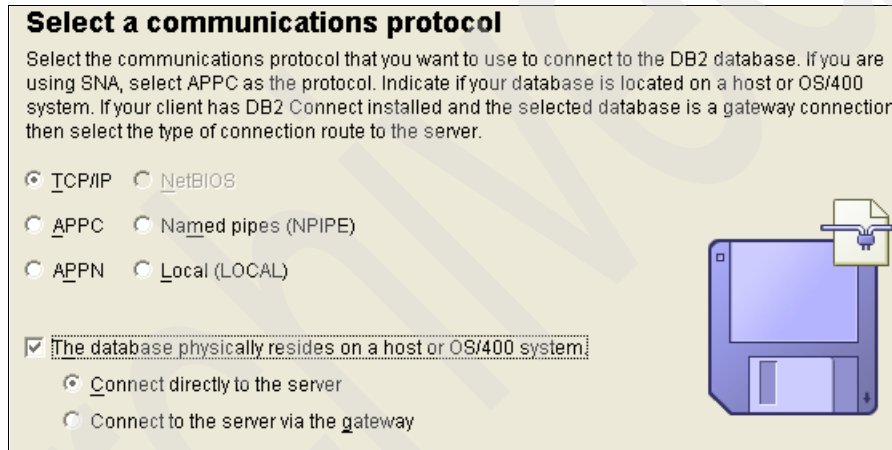


Figure 1-8 Selecting the communications protocol for DB2 Connect

7. Click **Next**.
8. Specify the proper TCP/IP hostname and port number, as shown in Figure 1-9 on page 14.

Specify TCP/IP communication parameters

You must provide the communication information required to connect to the database that you want to add. Your database administrator can provide the information necessary to configure communications for a database connection. If you specify a Service name only, there must be an existing service name entry in the TCP/IP services file.

Host name

Service name

Port number

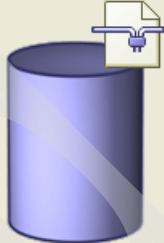


Figure 1-9 Communication parameters for DB2 Connect

9. Click **Next**.
10. Enter the database location name where the wizard asks for a database name, as shown in Figure 1-10.

Specify the name of the database to which you want to connect

You must identify the database to which you are connecting. The database name is dependent on the type of server to which you are connecting. For OS/390 and z/OS databases specify the Location name. For OS/400 databases use the RDB name. For VM/VSE specify the DBNAME. Otherwise use the name of the database on the server.

Database name

Database alias

Comment

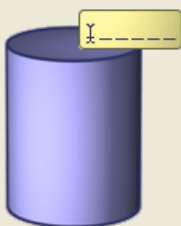


Figure 1-10 Database specification for DB2 Connect

11. Click **Next**.
12. No changes should be needed in the data source registration panel. Click **Next**.
13. Choose **OS/390 or Z/OS** as the operating system, as shown in Figure 1-11 on page 15.

Specify the node options

You must provide the following information in order to configure the node for the database connection.

Operating system: OS/390 or z/OS

Remote instance name:

Comment:




Figure 1-11 Specifying the node options for DB2 Connect

14. Click **Finish**. No other changes are needed in the wizard.
15. The Test Connection window will open. Test the standard and CLI connections by entering a valid RACF user name and password, as shown in Figure 1-12.

Test Connection - DB81

Connections | Results

Select connection type

☒ Standard ☐ OLEDB

☒ CLI ☐ JDBC

☐ ODBC ☐ ADO

User ID:

Password:

Figure 1-12 Test Connection

16. If clicking **Test Connection** displays successful connection, the window can be closed and the DB2 Connection side of this setup is complete.

The next step is to configure the application server similar to how we did this in 1.2.4, “Define the DB2 Universal JDBC driver provider” on page 8, and 1.2.5, “Defining the DB2 Universal JDBC driver provider data source” on page 9, with a couple of very important differences:

1. Follow the instructions in “Local deployment with a Type 4 connection” on page 11, up to instruction number 4.
2. At this point the WebSphere Administrative Console will be running and you will be ready to create a provider. Follow the same process as 1.2.4, “Define the DB2 Universal JDBC driver provider” on page 8, except choose the **DB2 Legacy CLI-based Type 2 JDBC Driver**.
3. Similar to how there was a data source defined in 1.2.5, “Defining the DB2 Universal JDBC driver provider data source” on page 9, define a data source for this provider.
4. Enter the relevant information to point to the DB2 Connect connection. This should only mean changing the JNDI name to the one used by the application and entering the proper database name (this would be the alias name that was created in DB2 Connect).
5. Click the test connection and ensure that it is successful, as shown in Figure 1-13. If so, the application can be run. A connection to DB2 using DB2 Connect will be used.

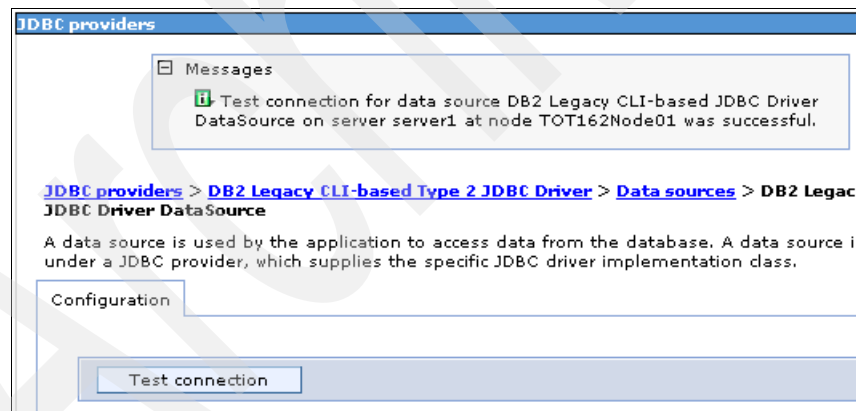


Figure 1-13 JDBC provider test

1.3.2 Data sources in the deployment descriptors

It is also possible to create providers and data sources in the application. They will be deployed and used by the application. This enables the developer to code the connection without the need for any special setup to be done at deployment time.

Coding process

The process will be similar for any type of provider desired. For this example we add a Type 4 data source similar to the one used earlier in the chapter. Since these providers are used by the deployed application, it makes sense that we will be adding them to the deployment descriptor of the enterprise application.

1. Open the deployment descriptor by double-clicking it under the expanded TraderDB2004 section of the Enterprise Applications. The changes that need to be done are under the Deployment tab that appears after the descriptor is opened. This is shown in Figure 1-14.

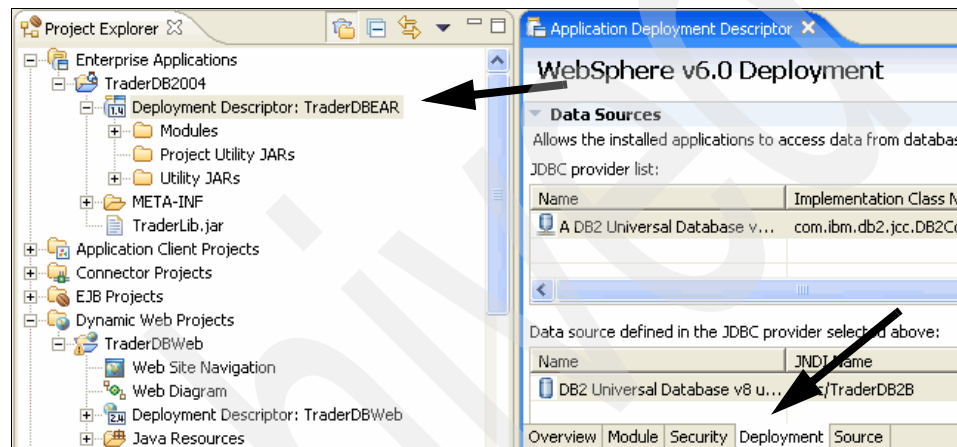


Figure 1-14 JDBC providers in the application

2. Using the updated version of the TraderDB application, there will not be any providers in the list. Clicking the **Add** button next to the JDBC provider list will open the Create a JDBC Provider wizard.
3. In this wizard we can choose IBM DB2 and then select the DB2 Universal JDBC driver provider, as we have done in the WebSphere Administrative Console. Refer to Figure 1-15 on page 18.

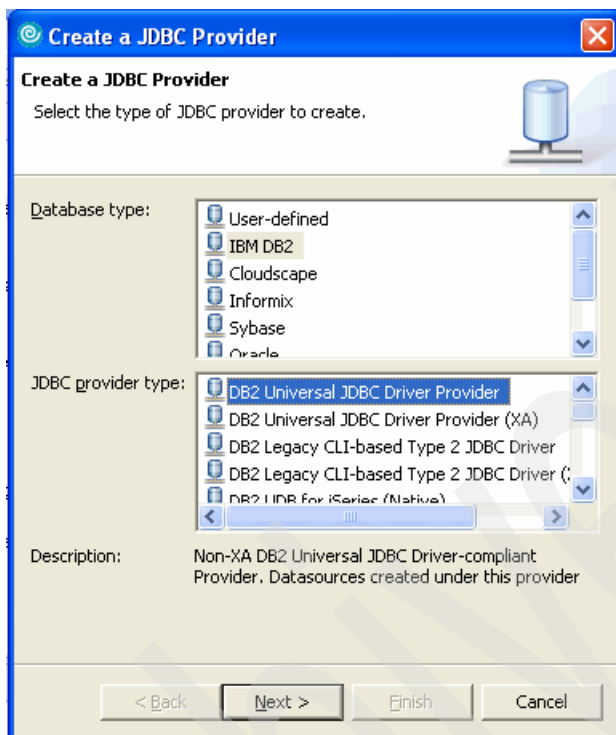


Figure 1-15 Selecting the universal driver

4. Click **Next**, enter a name for the provider, and click **Finish** to create the provider.
5. Now, click the provider created that is now in the JDBC provider list. Below this list are the data sources that are defined to the provider. It will be empty. We need to click the **Add** button next to the list to add a new data source. This will open the Create a Data Source wizard.
6. In this wizard, select the **DB2 Universal JDBC Provider** and click **Next**.
7. In this panel, enter the name of the data source and set the JNDI name. Our choice was jdbc/TraderDB2 to remain consistent.

Modify Data Source

Create a Data Source
Select the type of data source to create.

Name: * Data source 1

JNDI name: * jdbc/TraderDB2

Description: DB2 Universal Driver Datasource

Category:

Statement cache size: 10

Figure 1-16 Defining a data source

8. Click **Next** to display the Create Resource Properties panel.
9. In this panel, select **databaseName**, **driverType**, **serverName**, and **port number**, and enter their values separately. The choices that we made can be found in Table 1-2.

Table 1-2 Data source properties

| Resource property | Value |
|-------------------|---------------------|
| databaseName | DB8I |
| driverType | 4 |
| serverName | wtsc48.itso.ibm.com |
| portNumber | 38100 |

10. Click **Finish**. Now, saving this deployment descriptor and deploying this application would allow for this provider and data source to be used automatically.

No additional setup should be required on the application server. This allows a way for a developer to be fully in charge of setting up providers and data sources without having to bother the server administrators. A negative effect is that this also removes some of the visibility to the providers and data sources and can lead to a loss of flexibility in the future. For example, one of the advantages of using a JNDI name in an application is to allow different environments to set up the data sources differently, potentially pointing to different databases. This method does allow for this flexibility.

Required changes in TraderDB

The original TraderDB application from the previous book had a few of these providers defined. Unfortunately, one of the providers was pointing to a database called SAMPLE and had the JNDI name that we were using for the application. The simplest solution was to remove these providers from the application. Once removed and deployed, the application could run JDBC calls without further changes.

1.4 The TraderDB application

The TraderDB Application has been updated to include an SQLJ example. Before any changes were made, we deployed and tested the TraderDB application from the previous version of this book. Without modification, the JDBC connection functioned as expected, but only intermittently. Changes were needed for the deployment descriptors, as explained in 1.3.2, “Data sources in the deployment descriptors” on page 16.

This section focuses on describing:

- ▶ How to deploy the TraderDB application on WebSphere Application Server for z/OS Version 6.01
- ▶ How to test the DB2 connections with the application

1.4.1 Deploying the TraderDB Application

Deploying the TraderDB Application is trivial. We have created the source to allow for the defaults to be accepted and the application to function fine when this is done.

The defaults can be changed to match the desired setting for the application. For our example we have used container resource authorization with no authentication method specified. The deployment process can be changed to meet your needs as desired.

1. In the WebSphere administrative console expand the **Applications** section and click **Install New Application**.
2. Enter the path to the updated TraderDB2005.ear file shipped with this book. See Figure 1-17 on page 21.

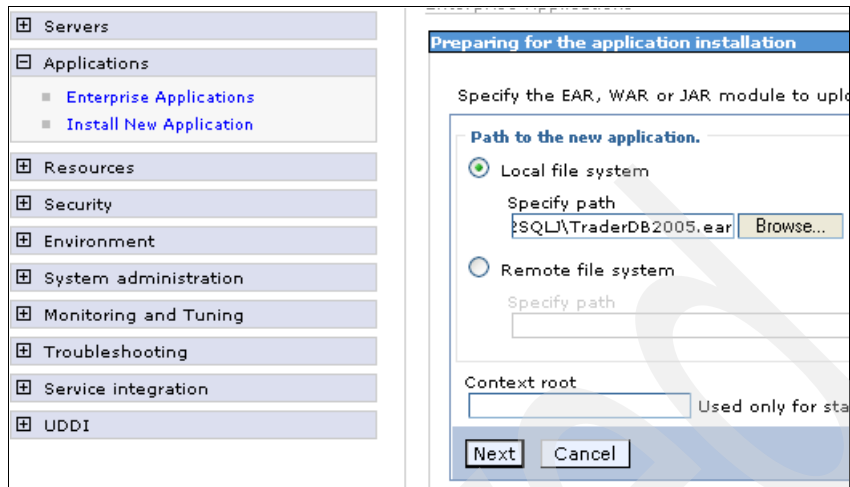


Figure 1-17 Deploying TraderDB

3. Click **Next**.
4. At this point, the deployment descriptor contains enough information to deploy the application so you can click **Next** through the deployment process or skip to step 11 for the summary, as shown in Figure 1-18 on page 22.

Install New Application

Specify options for installing enterprise applications and modules.

Step 1

Select installation options

Step 2

Map modules to servers

Step 3

Select current backend ID.

Step 4

Provide JNDI Names for Beans

Step 5

Provide default data source mapping for modules containing 2.x entity beans

Step 6

Map data sources for all 2.x CMP beans

Step 7

Map EJB

Summary

Summary of installation options

| Options | Values |
|----------------------------------|----------------------------|
| Use Binary Configuration | No |
| Create MBeans for resources | Yes |
| Cell/Node/Server | Click here |
| Reload interval in seconds | |
| Enable class reloading | No |
| Process embedded configuration | Yes |
| Application name | TraderDBEAR |
| Validate Input off/warn/fail | warn |
| Application Scoped Resources | Yes |
| Directory to install application | |
| Distribute application | Yes |
| Deploy Web services | No |
| Pre-compile JSP | No |
| Deploy enterprise beans | No |

Figure 1-18 Deployment step 11: Summary

- Clicking **Finish** on this screen will deploy the application. A successful deployment is illustrated by messages similar to those shown in Figure 1-19.

Application TraderDBEAR installed successfully.

To start the application, first save changes to the master configuration.

[Save to Master Configuration](#)

To work with installed applications, click the "Manage Applications" button.

[Manage Applications](#)

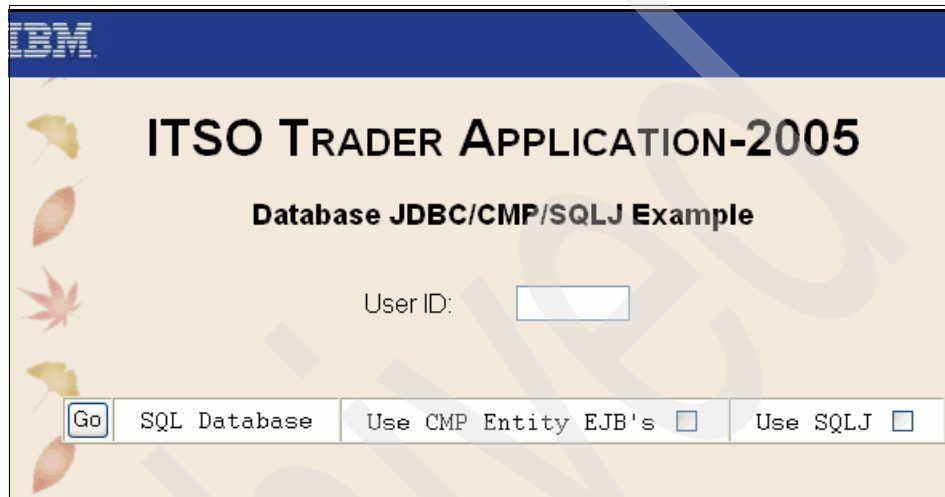
Figure 1-19 Deployment successful

- With a successful deployment, click the **Save to Master Configuration** link and save this deployment.
- The application can be started and it is ready to test.

1.4.2 Testing TraderDB with JDBC

Assuming nothing has been changed in the deployment descriptors, the URL for the deployed Trader application will be `http://server:port/TraderDBWeb/`.

Navigating to this link will display the page shown in Figure 1-20.



The screenshot shows a web application interface with a blue header bar containing the IBM logo. Below the header, the title "ITSO TRADER APPLICATION-2005" is displayed in large, bold, black letters. Underneath the title, the subtitle "Database JDBC/CMP/SQLJ Example" is shown in a smaller, bold, black font. To the left of the main content area, there is a vertical decorative element consisting of several stylized autumn leaves in shades of yellow, orange, and red. The main content area has a light beige background. In the center, there is a label "User ID:" followed by a small, empty text input field. Below this, there is a horizontal row of four buttons: "Go", "SQL Database", "Use CMP Entity EJB's", and "Use SQLJ". The "Go" button is highlighted with a blue border. The "Use CMP Entity EJB's" and "Use SQLJ" buttons each have a small, empty checkbox to its right.

Figure 1-20 TraderDB application

Table 1-3 on page 24 describes how these choices on this Web page effect how the application is run.

Table 1-3 Application key

| Item in Figure 1-20 | Purpose |
|----------------------|--|
| UserID | This ID is used to look up information in DB2 for the particular user. Specifically, it is used to see ownership of shares in each company. |
| Go | After entering a user ID, this button will start the trading application using JDBC. |
| Use CMP Entity EJB™s | Check this box to use the application with CMP Entity EJBs. No additional setup is required to try this. |
| Use SQLJ | This will use SQLJ for database queries rather than JDBC. If this is desired, read Chapter 2, “Accessing DB2 using SQLJ” on page 25. Additional setup and knowledge is required for this option. |

Accessing DB2 using SQLJ

This chapter explains the process for developing, deploying, and testing an application that uses SQLJ to connect to DB2. A small portion of this chapter also discusses the benefits and issues with using SQLJ rather than JDBC.

The version of the TraderDB application updated for this book has the added ability to use SQLJ to access the DB2 backend. Rational Application Developer was used for its SQLJ tooling, which can accomplish setup and customizing of SQLJ and provide a test environment to ensure functionality.

We encountered some unique migration problems that will also be mentioned in this chapter to help others avoid the confusion that can occur.

2.1 Introduction to SQLJ

Although for many Java developers SQLJ might not be the preference to access databases, opposed to JDBC, SQLJ does provide significant value for applications accessing DB2 on z/OS. In most cases SQLJ is faster than JDBC and provides better tuning and security control, as each Java application can have its own plan. In general, static SQL fits better in most existing z/OS environments than dynamic SQL.

SQLJ is simply an embedded SQL in Java. Like JDBC, SQLJ is used by the application to execute SQL statements against a database. This can lead to some confusion as to the differences and benefits of one technology over the other.

The following topics will be discussed in this chapter:

- ▶ The differences between JDBC and SQLJ in 2.1.1, “JDBC versus SQLJ” on page 26.
- ▶ Application connection concepts in 2.1.2, “Application connection concepts” on page 27.
- ▶ Connection coding differences in 2.1.3, “Connection coding differences” on page 28.
- ▶ SQLJ coding principles in 2.1.4, “SQLJ coding” on page 29.
- ▶ In 2.2, “Creating an SQLJ program in RAD” on page 33, we describe everything that is needed to get started with SQLJ in Rational Application Developer.
- ▶ In 2.3, “Testing SQLJ in TraderDB” on page 58, we show how you can test the TraderDB application using SQLJ.
- ▶ Finally, in 2.4, “SQLJ and QoS” on page 61, we show the QoS advantages of SQLJ as opposed to JDBC.

2.1.1 JDBC versus SQLJ

SQLJ is fundamentally different than JDBC because it is a Java language extension. JDBC is an API (ODBC). As a result of this difference SQLJ programs have to be preprocessed before execution. SQLJ is run through a translator (called *sqlj*), which takes the SQLJ statements and replaces them with Java methods. The result is two or more files:

- ▶ `ClassName_SJProfile0.ser` is created to store the SQL statements.
- ▶ The Java code is placed in standard Java source files named as `ClassName.java`. After the translation, the code can be compiled with a standard compiler and finally run.

Additionally, SQLJ goes through an additional step when connecting to DB2. DB2 allows for a customization step. Customization binds the SQL statements in the .ser files into packages to allow them to be executed as static SQL. Without a customization step, the SQL will not fail, but be issued dynamically through JDBC. In case customization is not done or just forgotten, there is no benefit in using SQLJ for this reason.

2.1.2 Application connection concepts

With both JDBC and SQLJ, a connection is needed before any SQL commands can be issued. SQLJ uses a different syntax than JDBC, but interestingly, it is actually invoking JDBC to create a connection.

There are two different methods for obtaining a connection in Java:

- ▶ The DriverManager interface
- ▶ The DataSource interface

The DriverManager interface requires the names of the driver and database in the calls. The DataSource removes this reliance on the application for coding the database. WebSphere Application Server can create a data source (demonstrated in 1.2, “Introduction to JDBC” on page 2) to store access information. The data source exposes a JNDI lookup name that the application can use when accessing the backend. A simplified view of this can be seen in Figure 2-1 on page 28.

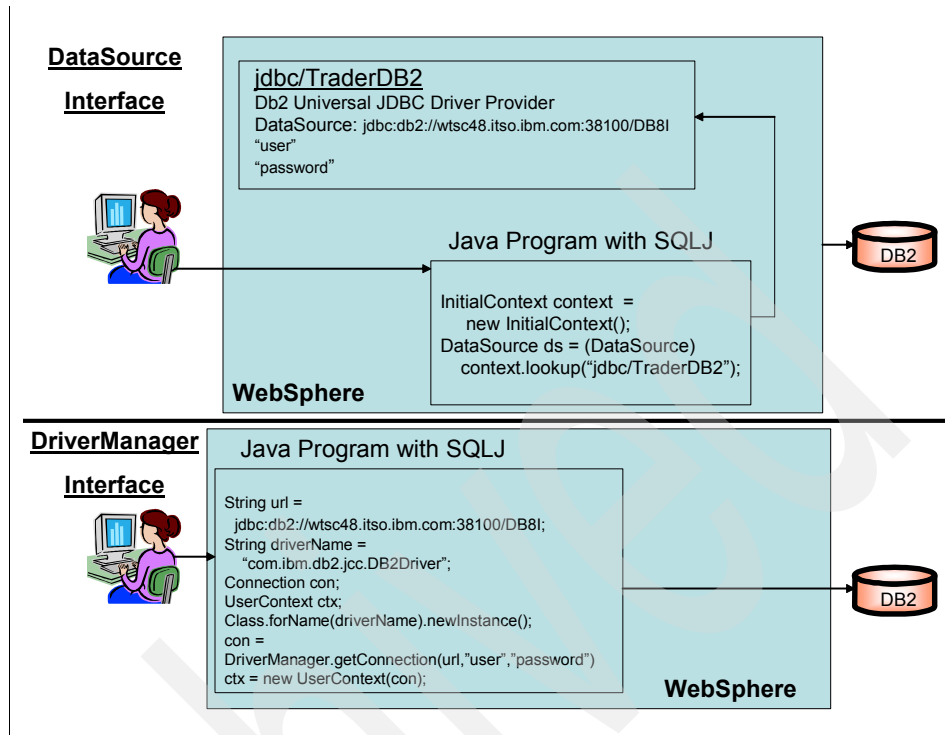


Figure 2-1 DriverManager versus DataSource

Figure 2-1 demonstrates the differences in where database definitions are defined. It also helps to demonstrate how a connection is established in SQLJ. For the remainder of this document we focus on the DataSource interface. This interface is recommended for a variety of reasons. The DataSource interface allows for connection pooling, which allows for robustness and scalability. Also, a DataSource interface allows the application programmer to define a JNDI name to connect to. The application can be deployed in different environments, each with its own implementation of the connection. Furthermore, the same data source can be updated to point to different databases without impact to the application and downtime is reduced because the application does not need to be edited and deployed.

2.1.3 Connection coding differences

It has already been mentioned that SQLJ uses JDBC to create its connections. As a result, there is not a huge amount of difference in the coding. We consider the most basic connection here, using the DataSource interface. A connection

coded in JDBC is shown in Example 2-1, and a connection coded in SQLJ is shown in Example 2-2

Example 2-1 JDBC connection

```
import javax.naming.InitialContext;
import javax.sql.DataSource;
import java.sql.Connection;

InitialContext context = new InitialContext();
DataSource ds = (DataSource) context.lookup("java:comp/env/jdbc/TraderDB2");
Connection conn = ds.getConnection();
```

Example 2-2 SQLJ connection

```
import javax.naming.InitialContext;
import javax.sql.DataSource;
import java.sql.Connection;

// Create a connection context
#sql static context MyConnectionContext;

//Look up the datasource
InitialContext context = new InitialContext();
DataSource ds = (DataSource) context.lookup("java:comp/env/jdbc/TraderDB2");
Connection conn = ds.getConnection();

// Create and instance of the context class
MyConnectionContext myContext = new MyConnectionContext(conn);
```

Showing these examples close to each other helps to demonstrate the minor difference in their syntax. SQLJ is exactly the same as a regular JDBC connection except that it contains a class that wraps the connection. This connection context is actually created by the SQLJ preprocessor, but we define it here.

2.1.4 SQLJ coding

This section provides only the highest level of information about coding in SQLJ. In most cases this is enough to get started. For further information about the SQLJ language, refer to *DB2 for z/OS and OS/390: Ready for Java*, SG24-6435.

There are four different forms of SQLJ statements:

- ▶ Connection contexts
- ▶ Executable statements
- ▶ Assignment statements

► Iterator declarations

Common for all SQLJ statements is that they start with the token `#sql` and end with a semicolon `(;)`.

We have already seen the first one, namely creating the context, in Example 2-2 on page 29. We now look briefly at the executable statement, which is the simplest SQLJ construct. You use an executable statement to perform database operations such as:

- Selecting, inserting, modifying, and deleting data
- Creating database objects
- Controlling transactions (commit and rollback)
- Calling stored procedures

We take a closer look at the executable statement. It can appear anywhere a Java statement can appear, and it looks like this:

```
#sql [context] { SQL-statement};
```

Important: Even though the context is optional, we highly recommend that you specify it. If the context is not specified, an implicit context is created, namely the *default context*. The default context is a static variable and therefore shared by every thread in the same JVM™. This can lead to a throughput bottleneck, but worse, another thread can change the default context while we are using it.

Conclusion: We always use a specific context.

Let us now have a look at the `COMPANYS` table used by the TraderDB application. A list of the columns is found in Table 2-1.

Table 2-1 *COMPANYS table used in TraderDB*

| Column name | Description |
|------------------|--------------------------|
| COMPANY | Company name |
| SHARE_PRICE | Current price of a share |
| UNIT_VALUE_7DAYS | Share price 7 days ago |
| UNIT_VALUE_6DAYS | Share price 6 days ago |
| UNIT_VALUE_5DAYS | Share price 5 days ago |
| UNIT_VALUE_4DAYS | Share price 4 days ago |
| UNIT_VALUE_3DAYS | Share price 3 days ago |

| Column name | Description |
|------------------|-------------------------|
| UNIT_VALUE_2DAYS | Share price 2 days ago |
| UNIT_VALUE_1DAYS | Share price 1 days ago |
| COMM_COST_SELL | Commission cost to sell |
| COMM_COST_BUY | Commission cost to buy |

We want to retrieve these columns into a Java application. Apart from creating the connection context, we need to define some host variables that we can select the values into. With the context at hand, along with the host variables, we are ready to write our SELECT statement, as shown in Example 2-3.

Example 2-3 Writing a SELECT statement using SQLJ

```
String company= "IBM";
// Define the connection context
#sql public static context MyContext with (dataSource="jdbc/TraderDB2");

// Setup a Cursor to store values
#sql iterator cursor1 (String, float, float, float, float, float, float,
float, float, int, int);

// Create an instance of the context class
MyContext context = new MyContext();

// The SQLJ Select statement
#sql [context] cursor1 {
    SELECT * FROM TRADER.COMPANY
    WHERE COMPANY = :company
}
// Now, cursor1 can be used to iterate through the returned data
// Alternately, results can be stored in variable
// Use the INTO sqlj statment and specify variables to store into.
// Tell the Transaction Manager that we are done with the connection for
// now
context.close();
```

Assuming we are running within the scope of a transaction in WebSphere Application Server, closing the context will not close the underlying connection. We just put the connection in a valid state for commit, leaving to the Transaction Manager to commit, or roll back, when appropriate.

The SQL statement can be almost any DB2 statement that can be statically prepared. Additionally, there is a special statement to set the isolation level for the current transaction:

```
#sql [context] { SET TRANSACTION ISOLATION LEVEL level};
```

Where *level* is one of the values listed in Table 2-2. Please be aware that both JDBC/SQLJ and DB2 use the term *repeatable read*, but for different purposes.

Table 2-2 SQLJ isolation levels and their corresponding DB2 counterparts

| SQLJ isolation level | DB2 isolation level |
|----------------------|-----------------------|
| READ UNCOMMITTED | UR (uncommitted read) |
| READ COMMITTED | CS (cursor stability) |
| REPEATABLE READ | RS (read stability) |
| SERIALIZABLE | RR (repeatable read) |

2.1.5 General application flow

Figure 2-2 demonstrates a basic, high-level flow of a properly coded SQLJ program. We generally followed this model when programming TraderDB, so this should help as a guide to understanding our program.

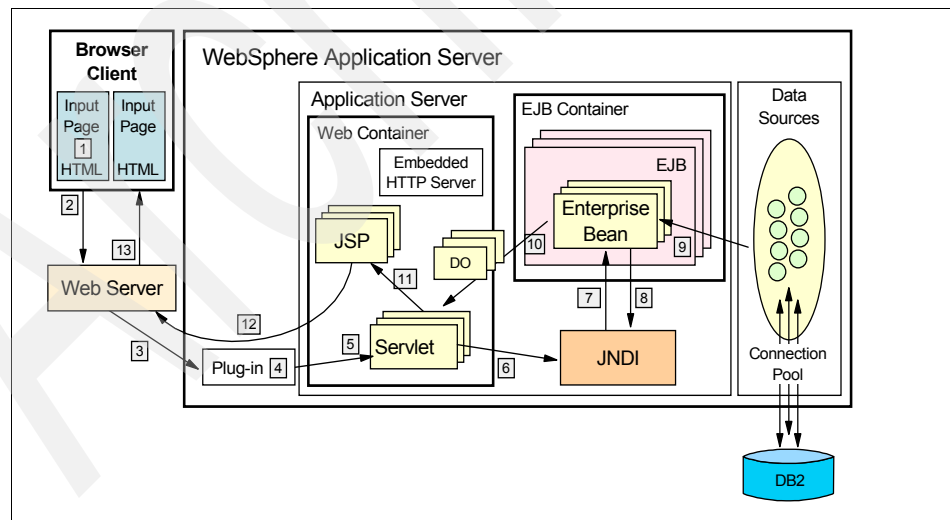


Figure 2-2 Application flow

The flow is:

1. A Web client requests a URL in the browser (input page).
2. The request is routed to the Web server over the Internet.
3. The Web server immediately passes the request to the Web server plug-in. All requests go to the WebSphere plug-in first.
4. The Web server plug-in examines the URL, verifies the list of hostname aliases from which it will accept traffic, and chooses a server to handle the request.
5. A stream is created. The stream is the connection to the Web container. The Web container receives the request and, based on the URL, dispatches it to the proper servlet.
6. JNDI is now used to look up the session Bean requested by the servlet.
7. JNDI will direct the servlet to the corresponding EJB container, which then instantiates the EJB that is requested.
8. The session Bean will now go back into the JNDI either to look up an entity Bean, or to look up the data source. If an entity Bean is used, the entity Bean will do the look up of the data source.
9. The SQLJ statement is executed, and data is sent back to the enterprise Bean.
10. Data Objects (DOs), also known as Data Transfer Objects, are created and handed back to the servlet.
11. The servlet sends the DOs to the JSP™.
12. The JSP generates the HTML that is sent back through the WebSphere plug-in to the Web server.
13. The Web server sends the HTML to the browser.

2.2 Creating an SQLJ program in RAD

In this section, we focus on using Rational Application Developer to update our TraderDB application to add some SQLJ functionality. This section can be used in two ways. Each section can be read independently to provide information about a specific topic, or they can be read concurrently to understand the process of enabling the use of SQLJ.

2.2.1 Supplied material

For this book we decided to supply two versions of the TraderDB application. The first version contains the exact code that came with the previous version of

the book. Also provided is the updated code with SQLJ functionality built in. The reason for this is to allow access to an application that does not have SQLJ support for experimentation purposes.

- ▶ TraderDB2004.ear: Version 5, non-SQLJ code
- ▶ TraderDB2005.ear: Version 6 with SQLJ support

A high level overview of the newest version of TraderDB can be found in 1.1, “Topology” on page 2.

2.2.2 Migrating the Version 5 application

The Version 5 application will function when deployed on WebSphere Application Server for z/OS Version 6.01; however, when updating the application in Rational Application Developer and starting work at the J2EE 1.4 level, a migration step is required first.

The code needs to be migrated to J2EE 1.4. Rational Application Developer contains a migration wizard that will handle migrations from the J2EE 1.2 or J2EE 1.3 level.

At this point we assume that Version 5 of TraderDB has been imported into RAD. If this is the case, there will immediately be some errors in the application. They will be similar to those shown in Figure 2-3.

| | Description | Resource | In Folder | Location |
|---|---|-------------------|--|----------|
| ✖ | Class must implement the inherited abstract method EJB... | TraderDB2CMPCo... | Trader_DB/ejbModule/itso/db2/cmp/j2ee... | line 12 |
| ✖ | Class must implement the inherited abstract method EJB... | TraderDB2CMPCu... | Trader_DB/ejbModule/itso/db2/cmp/j2ee... | line 12 |

Figure 2-3 Errors requiring migration

A migration needs to be performed to bring the generated code to the correct level and remove these errors. To perform this migration:

1. Right-click the enterprise application name (**TraderDB2004** in this case).
2. Choose **migrate**.
3. Select the **J2EE Migration Wizard**, as shown in Figure 2-4 on page 35.

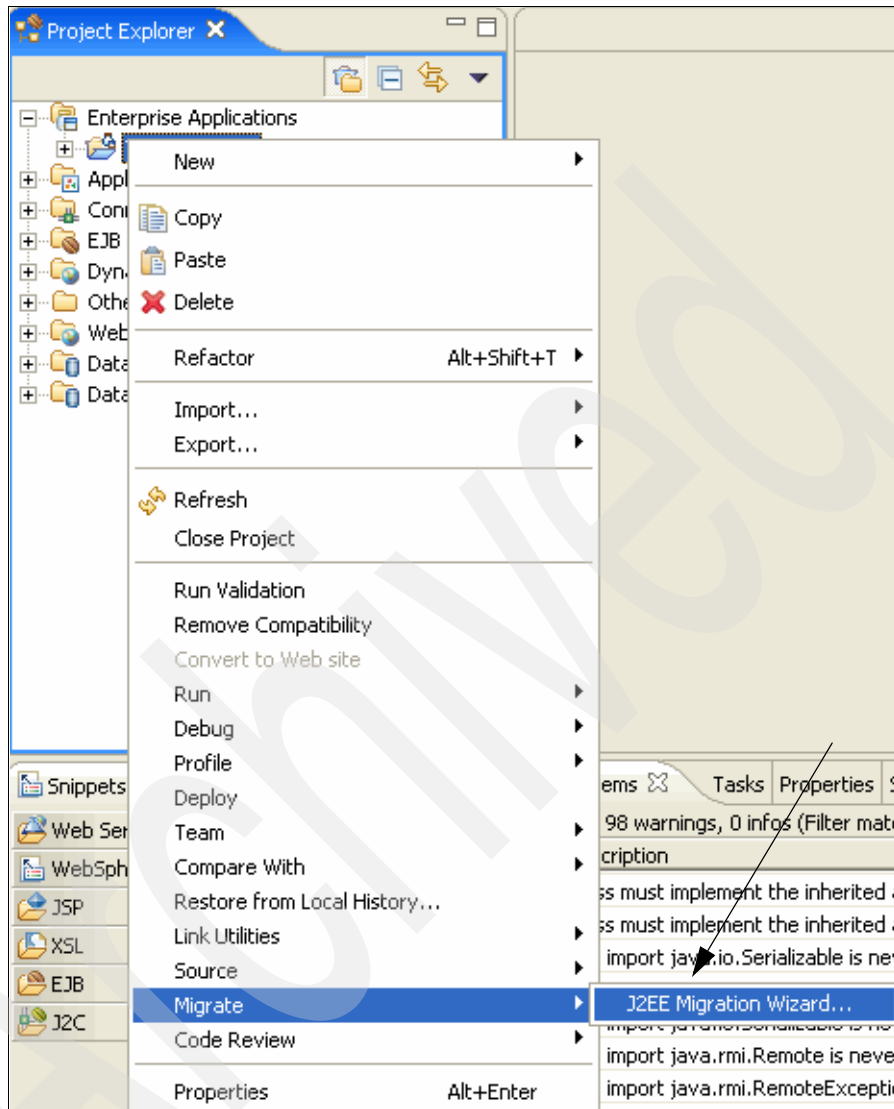


Figure 2-4 Selecting the Migration Wizard

4. Click **Next** and ensure the project structure and all module projects are being migrated that are contained in the enterprise application. Also be sure the J2EE version selected is 1.4 targeted at a Version 6 server. See Figure 2-5 on page 36.

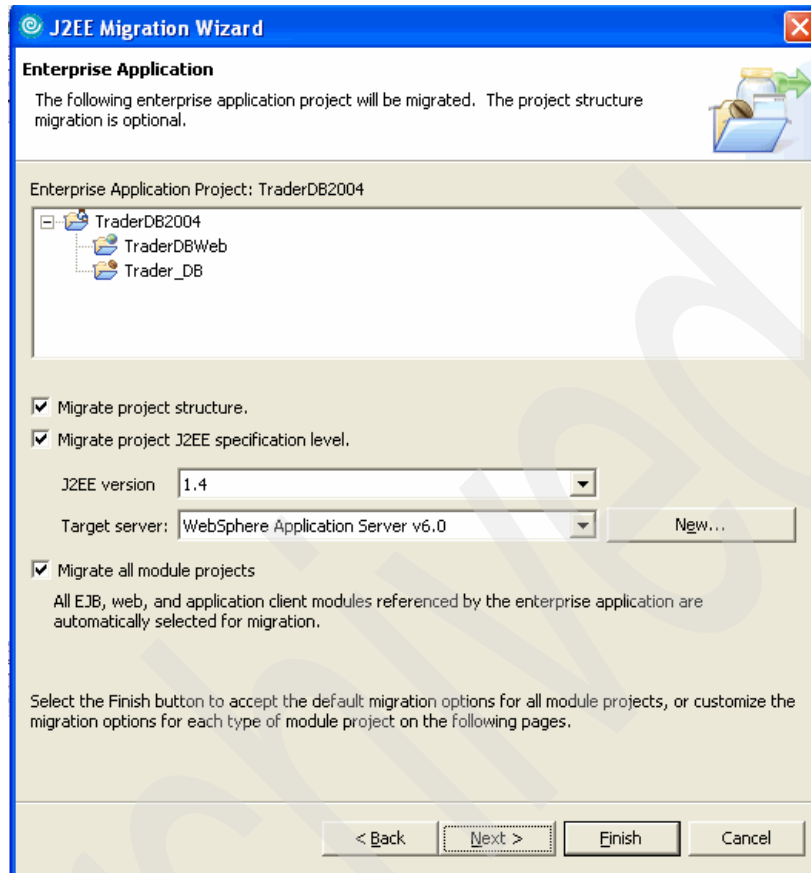


Figure 2-5 Enterprise Application Migration in RAD

5. Click **Next**.
6. Verify that the EJB Module Migration and Web Projects also migrate the project structure and J2EE specification level.
7. Click **Finish**.

The errors should now be gone. The application has now been migrated to the J2EE 1.4 specification. After this step, SQLJ can be enabled.

2.2.3 Setting up the SQLJ environment

Some setup is required to allow for a Java application to include SQLJ. The first step is to set up RAD so that it is aware of the location of the SQLJ translator. Before we can even set up SQLJ in RAD, we need to add this capability. To do

this click **Window** → **Preferences** → **Capabilities**. Expand the **Database Developer** section and choose **SQLJ Development**. Click **OK** to enable this capability and close the window.

We are now ready to set up the SQLJ translator. The setup window can be accessed from **Window** → **Preferences** → **Data** → **SQLJ**, as shown in Figure 2-6.

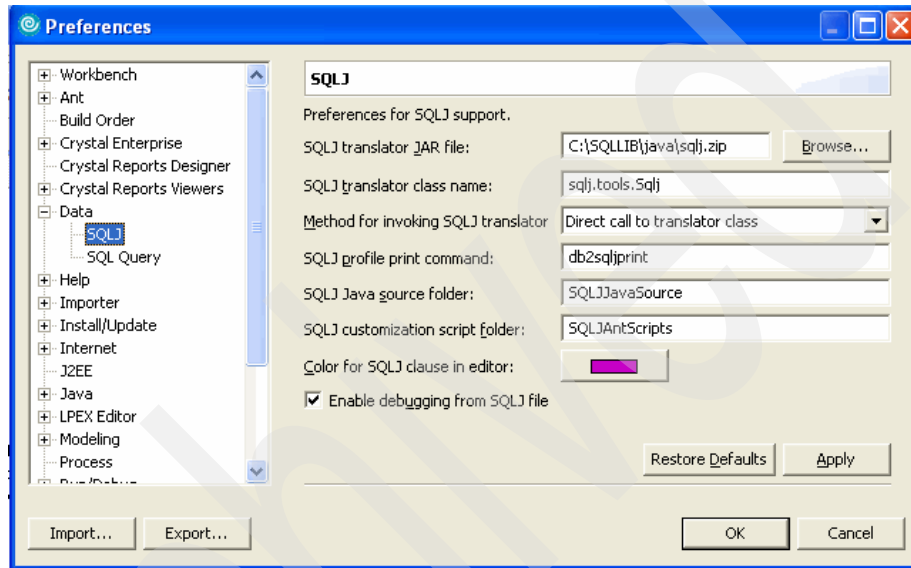


Figure 2-6 SQLJ settings in RAD

The fields are:

- ▶ **SQLJ translator JAR file**
Full path and file name of the JAR file containing the SQLJ class library that implements SQLJ translation support. If DB2 is installed on your workstation, the file is called <db2home>/java/sqlj.zip, where <db2home> is where DB2 is installed, by default, C:\Program Files\IBM\SQLLIB. This file can also be downloaded from the host where DB2 is installed. For our installation the location was /usr/lpp/db2/d8ig/jcc/classes. For most installations, the path should be similar.
- ▶ **SQLJ translator class name**
Fully package-qualified name of the SQLJ class used for translating SQLJ statements. The default is sqlj.tools.Sqlj.
- ▶ **Method for invoking SQLJ translator**

The default is a direct call to the translator class. The alternate choice would be to create a new process for every translation.

- ▶ SQLJ profile print command

This field specifies the command used for SQLJ profile printing. The default of db2sqljprint is fine.

- ▶ SQLJ Java source folder

Name to use for the folder containing SQLJ Java source files. This preference is only used if we choose to create a specific source folder when we define our SQLJ project. If the source folder and the output folder are the same, then generated Java files are put in the same folder as the SQLJ files.

- ▶ SQLJ customization script folder

Name to use for the folder containing SQLJ Ant scripts.

- ▶ Color for SQL clause in editor

Color in which SQLJ statements are displayed in the Java editor.

- ▶ Enable debugging from SQLJ file

If selected, debugging will occur in SQLJ files, instead of the generated Java files. If you change this preference, you must rebuild the project for the change to take effect.

2.2.4 Adding SQLJ support to our project

The procedure is the same at this point whether we are adding SQLJ support to a new project, or a project that has already been created (like TraderDB). This step is usually trivial, but we did find a problem executing this step for our application. This section explains how to solve the problem in case a similar situation is encountered with other applications.

The actual application now needs to be given SQLJ support. For the TraderDB application we want to create a new EJB that is called to perform database actions using SQLJ. To enable this support, right-click the **Trader_DB** EJB project, as shown in Figure 2-7 on page 39.

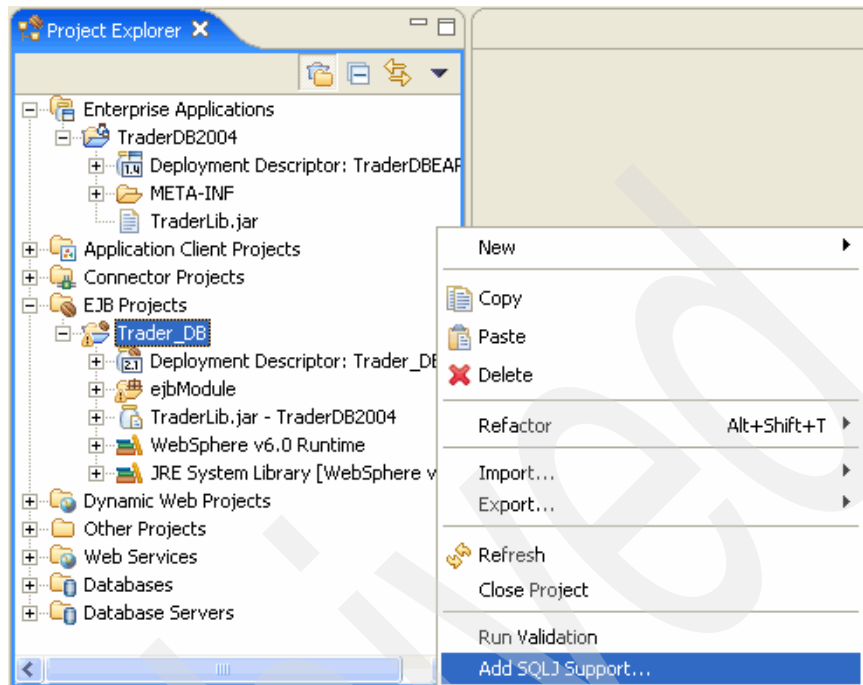


Figure 2-7 Adding SQLJ support to an EJB project in RAD

This will start the wizard that adds SQLJ support. We have the option to add support to the Web project and the EJB project. Since we only want to add a new EJB with SQLJ support, we only select the EJB project, as shown in Figure 2-8 on page 40.

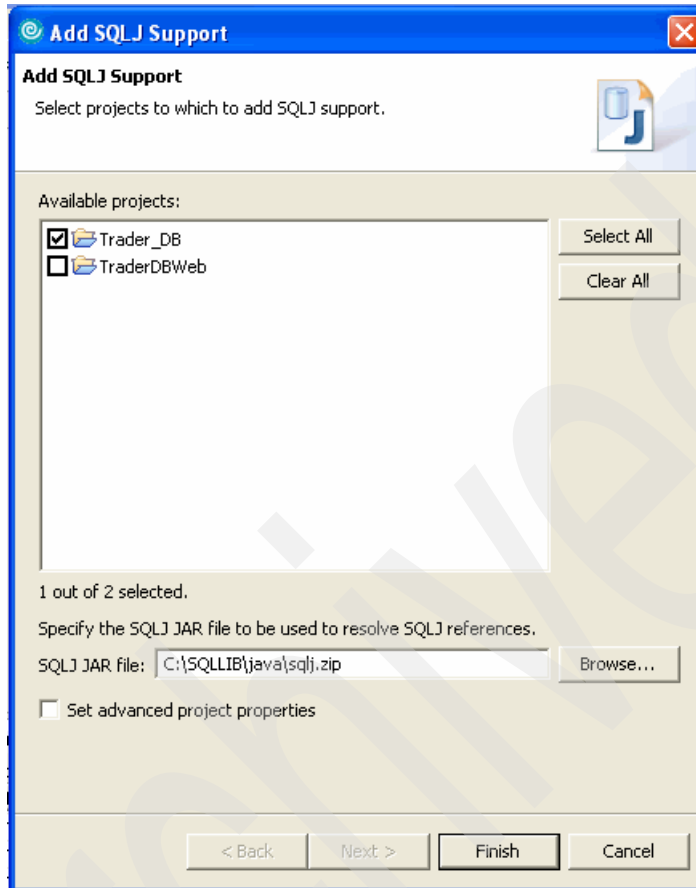


Figure 2-8 Selecting the projects to add SQLJ support to in RAD

At this point, clicking the **Finish** button should complete the process. However, with our application we received the error, as shown in Figure 2-9 on page 41.

Important: The error we received is *not* caused by adding SQLJ support in RAD. This was just where it surfaced. The text following Figure 2-9 on page 41 describes the real cause of this error in more detail.

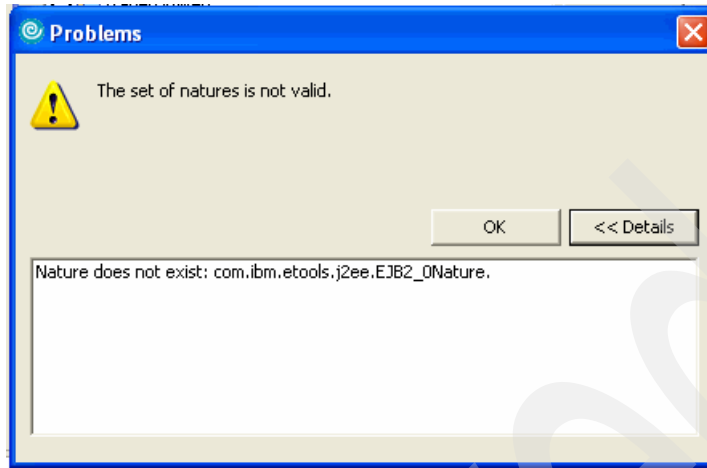


Figure 2-9 Nature does not exist error

So, what happened?

This error demonstrates a possible error that can be seen when updating an imported ear that was created with different tooling. The previous version of TraderDB was created using WebSphere Studio Application Developer. Somewhere in the development cycle a nature was setup called EJB2_0Nature. A nature defines java-specific behavior for a plug-in. WebSphere Studio Application Developer must have needed this nature. If similar errors occur, it is a good idea to verify the nature is not needed and remove it or find a replacement.

To remove this nature we need to find the .project file in the active workspace and edit it. For our system, we called our workspace trader1, and the Trader_DB .project file was found in C:\Documents and Settings\TOT162\IBM\rationalsdp6.0\trader1\Trader_DB. Open this file in notepad. Figure 2-4 shows the section of the .project file causing the error. Closing Rational Application Developer, removing the nature, and restarting the workspace will get rid of the problem.

Example 2-4 .project excerpt

```
... previous lines removed
<natures>
  <nature>org.eclipse.jem.workbench.JavaEMFNature</nature>
  <nature>com.ibm.etools.j2ee.EJB2_0Nature</nature>
  <nature>com.ibm.wtp.ejb.EJBNature</nature>
  <nature>org.eclipse.jdt.core.javanature</nature>
  <nature>org.eclipse.jem.beaninfo.BeanInfoNature</nature>
</natures>
```

... lines following removed

While it is interesting that we see this error now, it is not totally unexpected. When we were enabling SQLJ support, we were installing a new nature. This causes all natures to be re-checked and this error to show itself.

At this point, adding SQLJ support should finish without errors and the program is ready for an sqlj file.

2.2.5 Adding an SQLJ file to the project

After RAD has been set up to use SQLJ and a project has been enabled for SQLJ support, the actual SQLJ code can be added. There is a very good wizard built into RAD that can create a good starting point for developing SQLJ further. This section focuses on how to use this wizard. We take a brief look at the results and create a plan for changes that are needed. The finished product can be seen with the additional material shipped with this book. This documented process is exactly what was done to create that code.

To begin the process of creating an SQLJ file, right-click the EJB project and select **New** → **Other**, as shown in Figure 2-10.

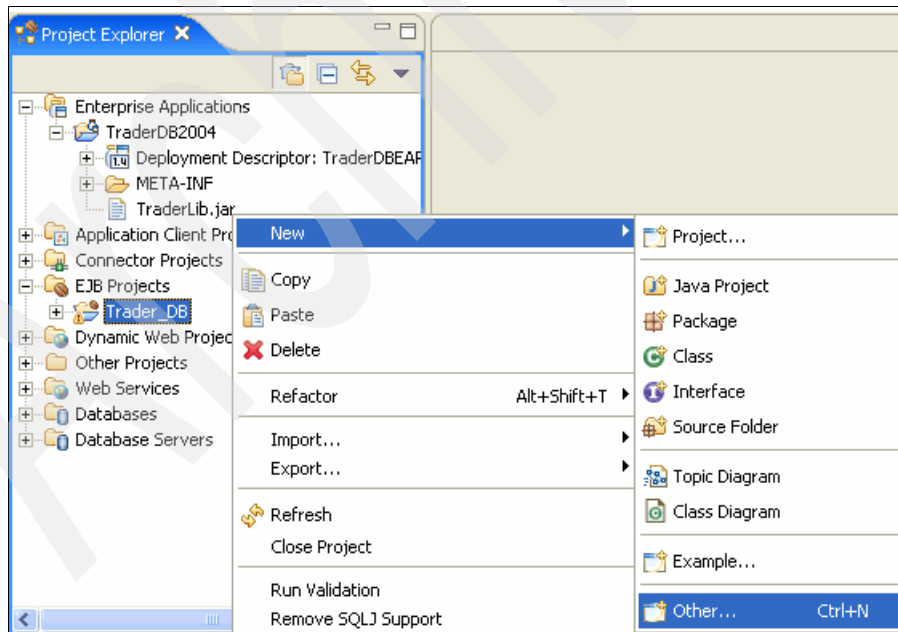


Figure 2-10 Creating a new SQLJ file

Expand the data section and then expand the SQLJ section. If SQLJ does not appear in the list, click **Show All Wizards** and try again. Now choose **SQLJ File**, as shown in Figure 2-11.

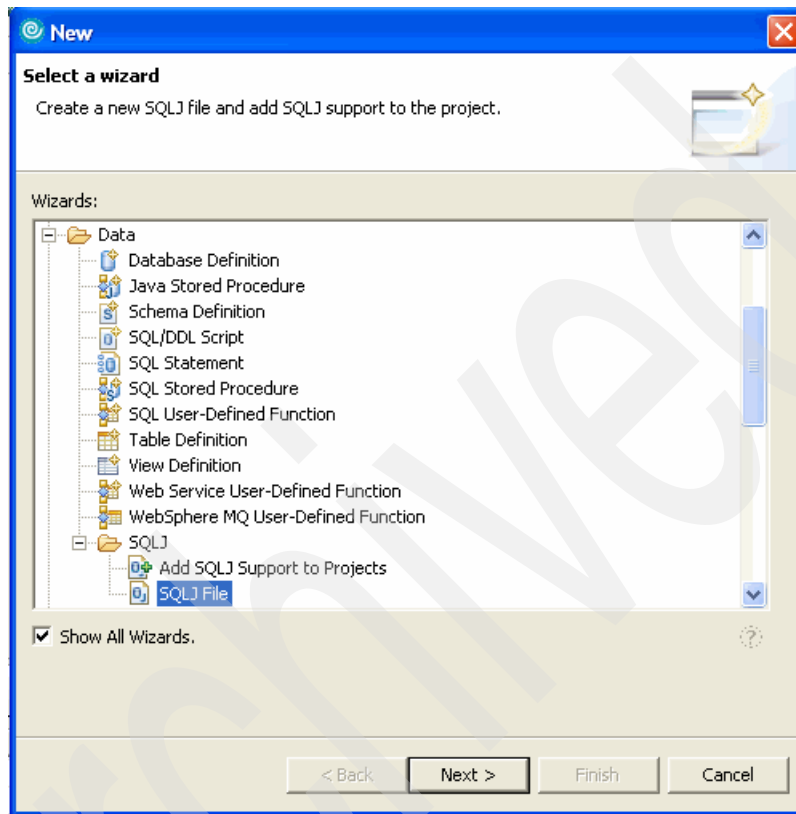


Figure 2-11 Starting the SQLJ wizard in RAD

You will see two choices for implementing SQLJ support here:

- ▶ **Add SQLJ support to projects**
This allows the project to use imported or manually created SQLJ files. To experiment with this, it would be possible to import the SQLJ file from the finished TraderDB application.
- ▶ **SQLJ File**
This starts the SQLJ creation wizard. In simple cases, this wizard can implement all the code needed for retrieving data from DB2.

Clicking **Next** will start the wizard. There are many choices when moving through this wizard. In order to show the power of the wizard, we may select options that

are not needed for most programs. This section details the purpose of each choice to allow for evaluation in any environment.

Figure 2-12 shows the first screen in the SQLJ file wizard.

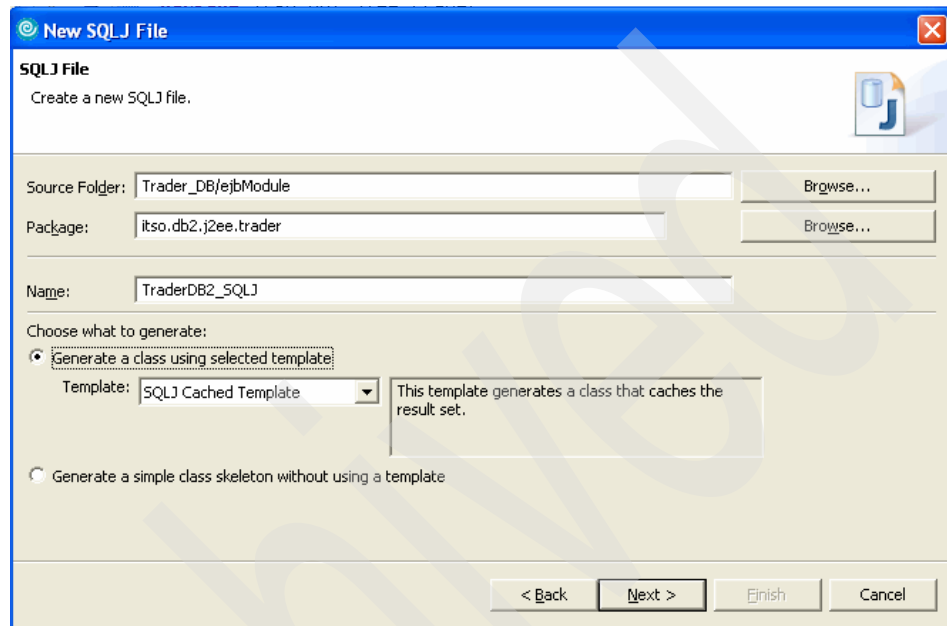


Figure 2-12 SQLJ File Wizard in RAD

The fields are:

- ▶ **Source Folder**
Name of the source folder for this file. In most cases this will already be filled in with the correct value.
- ▶ **Package**
The package that will include this SQLJ and generated Java file. We chose itso.db2.j2ee.trader package because it is the same package that contains the session beans that will be calling the functions of the SQLJ file.
- ▶ **Name**
The name of the SQLJ class. For consistency we chose a name that matches the naming convention used in TraderDB.
- ▶ **Choose what to generate**
There are three choices for the final outcome of the wizard:
 - SQLJ Cached Template

The SQLJ file created will cache results. This was our choice simply because it demonstrated the most with SQLJ.

- SQLJ Template

The SQLJ file will be created as in the previous template except that the results of the SQLJ statements will *not* be cached.

- Generate a simple class skeleton without using a template

A skeleton is created without the use of a template. For code that is not generated properly with a template, this is a good place to start.

Figure 2-13 shows the second panel of the wizard. Here we need to choose how we want the SQL statements created and which database to connect to.

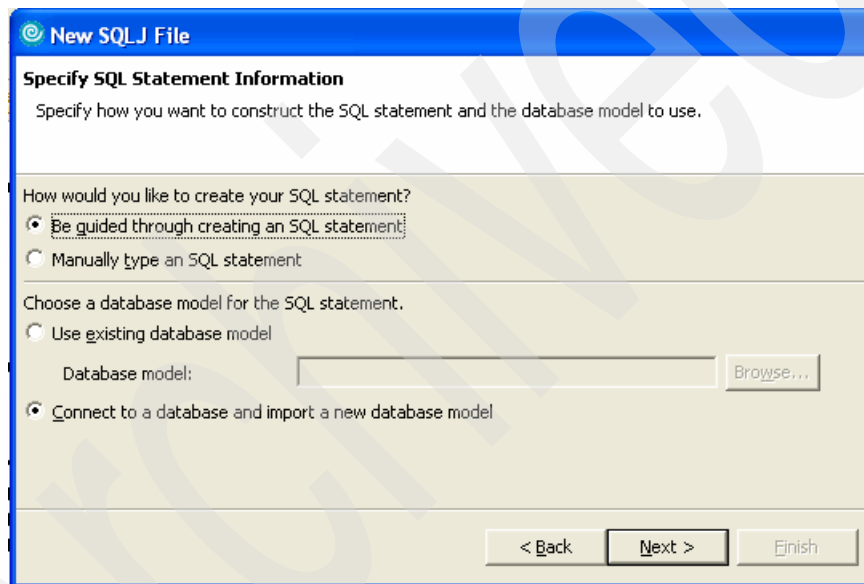


Figure 2-13 SQLJ File wizard: SQL and Database

The fields and options are:

- How would you like to Create your SQL statement?

- Be guided through creating an SQL statement

This choice adds functionality to the wizard to generate the SQL commands automatically. We had the most success with this method so use it in our demonstration.

- Manually type an SQL statement

SQL statements can be manually typed and tested.

- Choose a database model for the SQL statement.

- Use existing database model

If we had created a database model in RAD for our project, we could point the wizard to it for our connection.

- Connect to a database and import a new model

We create our connection to the database and use the wizard to extract the tables and other information to create our model.

At this point the wizard moves into the Database Connection panel. We need to create a connection to our z/OS DB2 system. Figure 2-14 shows the choices that we made for our server. As described in Chapter 1, “Accessing DB2 using JDBC” on page 1, we use the DB2 Universal JDBC Driver to access our backend system.

New SQLJ File

Database Connection
Establish a JDBC connection to a database.

Connection name: SQLJ test connection

Database: DB8I

User ID: daubman

Password: *****

Database vendor type: DB2 Universal Database for z/OS V8

JDBC driver: IBM DB2 UNIVERSAL DRIVER

Host: wtsc48.itso.ibm.com

(Optional) Port number: 38100

Server name:

Database Location: Browse...

JDBC driver class: com.ibm.db2.jcc.DB2Driver

Class location: C:\SQLJ\db2jcc_license_cisuz.jar;C:\SQLJ\db2jcc.jar Browse...

Connection URL: jdbc:db2://wtsc48.itso.ibm.com:38100/DB8I

Filters...

Connect to Database

< Back Next > Finish Cancel

Figure 2-14 Database Connection panel for SQLJ in RAD

Notice that this panel also requires the JDBC Driver connection jars. These jars are located in the same place that the sqlj.zip file was found. If DB2 is not found on your development workstation, the jars can be obtained from the host. See

2.2.3, “Setting up the SQLJ environment” on page 36, for a description of where these are found. Notice that in order to use the host db2jcc.jar, a license file called db2jcc_license_cisuz.jar is required. This file is also located on the host. The db2jcc.jar found in a local DB2 installation should not require this file.

At the time this book was written we were also able to uncover an unexpected error on this panel. When including the licensing jar we should receive a cryptic error if this jar did not appear first on the class location field. This error can be seen in Figure 2-15, and occurs when the db2jcc.jar is first in the path and the Connect to Database button is clicked.

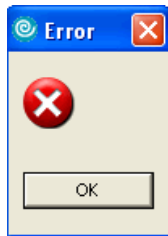


Figure 2-15 Unexpected error

Verify that the order is correct and the error will not occur. At this point, clicking Connect to Database will result in the opening of the SQL construction wizard, which can be seen in Figure 2-16 on page 48.

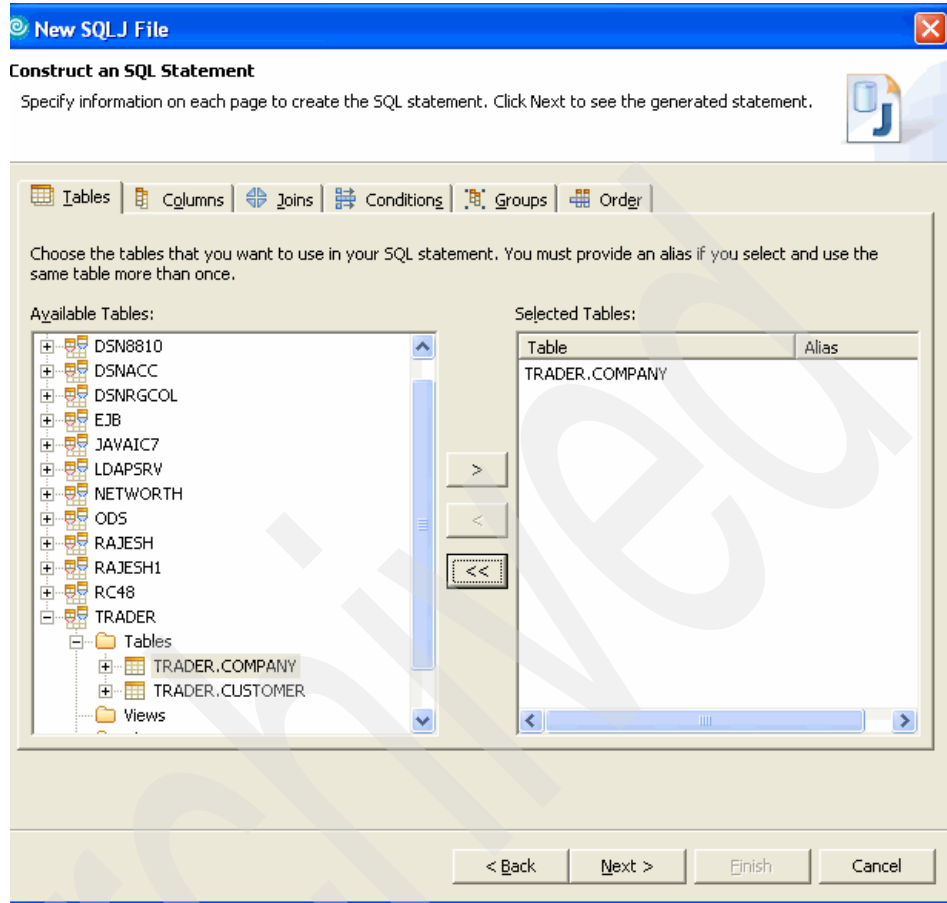


Figure 2-16 Construct an SQL Statement wizard in RAD

Adding a table (TRADER.COMPANY) and clicking **Next** will generate the SQL `SELECT * FROM TRADER.COMPANY`.

This will be a good start for our application. Going too much further would only require more changes in the application later. At this point, the wizard will generate an SQLJ file to perform this SQL statement, as well as generate accessor functions for all the columns in this table.

Often, the Next button will not be clickable when on the SQL statement screen. Clicking Reset will usually fix this problem. This seems to be an issue with the current version of Rational Application Developer.

Figure 2-17 on page 49 defines the connection information used by the wizard for setting up the connection to DB2. The different methods are discussed in

2.1.2, “Application connection concepts” on page 27. If testing a connection from within RAD, it may be advantageous to use a DriverManaged connection and switch to a data source connection when ready to deploy.

New SQLJ File

Specify Runtime Database Connection Information
Enter information for establishing a database connection at runtime

☒ Use DataSource connection
DataSource/JNDI name: jdbc/TraderDB2

☐ Use DriverManager connection
Driver name: com.ibm.db2.jcc.DB2Driver
URL: jdbc:db2://wtsc48.itso.ibm.com:38100/DB81

How will user authentication be provided?
☒ Passed as method parameters
☐ Variables inside of method

User ID: daubman
Password: *****
Reenter password: *****

< Back Next > Finish Cancel

Figure 2-17 Define the connection for SQLJ in RAD

Clicking **Finish** will now create the `TraderDB2_SQLJ.sqlj` program that includes SQL to retrieve all the columns of the `TRADER.COMPANY` table. This provides a good example of generating SQLJ, but it does not give us exactly what we need for our program. The goal for us was to create a new session bean that could call this SQLJ in a very similar fashion to the JDBC calls we issued in the JDBC bean. To do this, the following changes were required.

To bring the SQL into the state where it can be used, the following changes will be needed:

- ▶ Other execute functions are needed to issue other SQL commands:
 - `SELECT * FROM TRADER.COMPANY WHERE COMPANY = :company`
Get information for a specific company.
 - `SELECT * FROM TRADER.CUSTOMER WHERE CUSTOMER = :customer AND COMPANY = :company`
Get the information from the customer table for a specific customer and specific company.
 - `INSERT INTO TRADER.CUSTOMER VALUES (:customer , :company, :numShare)`
Insert the customer, company, and shares for specific trade.
 - `UPDATE TRADER.CUSTOMER SET NO_SHARES = :numShare WHERE CUSTOMER = :customer AND COMPANY = :company`
Update the information in the customer table based on a trade.
 - Accessors for the additional information retrieved

The finished code is part of the additional material for this book and shows these changes in their final state. Also done in this code was the addition of the SQLJ session bean (TraderDB2_UseSQLJ) and the servlet that calls it (TraderSQLJServlet.java). Finally, the Logon.html was updated to use this servlet. The specifics of this work are beyond the scope of this book, but are interesting to look at.

2.2.6 Customization

There would not be a problem if we wanted to deploy the code now. The SQLJ would run as JDBC and our program would work. However, this chapter has already mentioned the performance impact of this. Customization is required and the packages created must be bound to our DB2 system. We can then instruct our data source to use this package and execute the static SQL. Until this is done, we are not using the real power of SQLJ.

There are two possible ways to perform the customization and binding. The first is to copy the SQLJ serialized profile onto the host and run the commands from there. This approach would require that the USS environment variables be set up properly. Usually this is best done through a shell script. The command **db2sqljcustomize** can, and by default will, create the packages and perform the bind to DB2. More information can be found in *DB2 Universal Database™ for z/OS: Application Programming Guide and Reference for Java*, SC18-7414-02.

We have chosen not to run the commands on z/OS, but use the RAD tooling, which can perform the customization and binding, too. Follow these steps:

1. Start the Customize/Bind SQL Profiles wizard by right-clicking the project with the SQLJ code in it. This brings up the wizard, as shown in Figure 2-18.

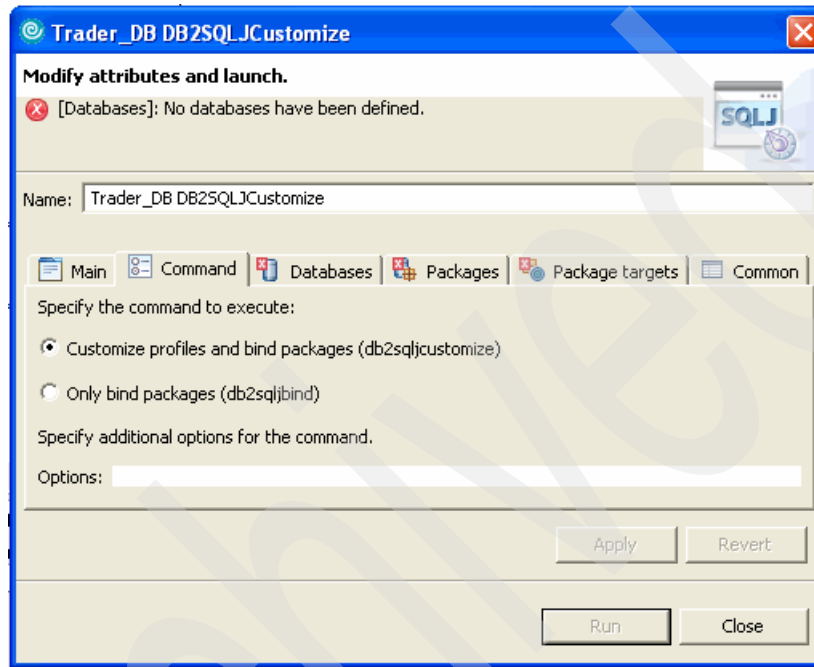


Figure 2-18 SQLJ customize wizard

2. As the error in the window explains, we need to define a database for the wizard to use in order to properly customize and bind. Click the **Database** tab and fill in the database connection information that was asked for. The information will be similar to what was asked for in Figure 2-14 on page 46. As with other places in RAD where the database name is requested, use the DDF location name.

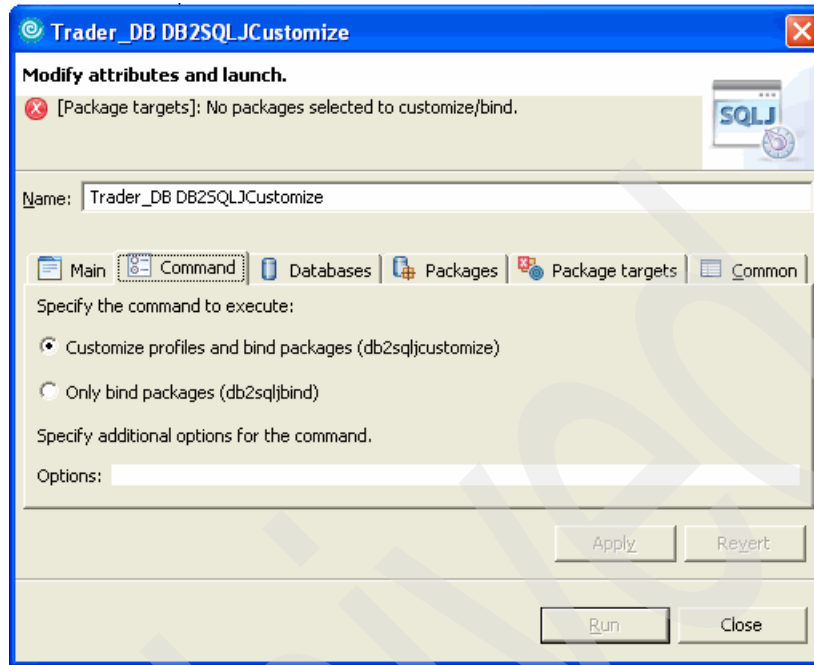


Figure 2-19 SQLJ customize and bind: Database panel

3. A new error presents itself now. A package should be chosen. Click the **Package targets** tab to choose a package, as shown in Figure 2-20 on page 53.

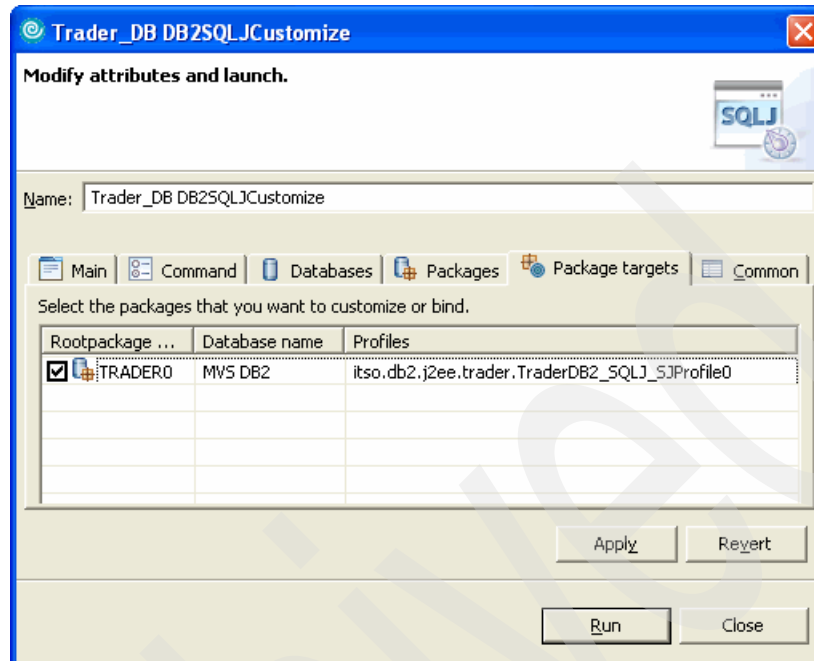


Figure 2-20 SQLJ customize and bind: Package targets panel

- Now enter the command options in the Command tab, as shown in Figure 2-21 on page 54.

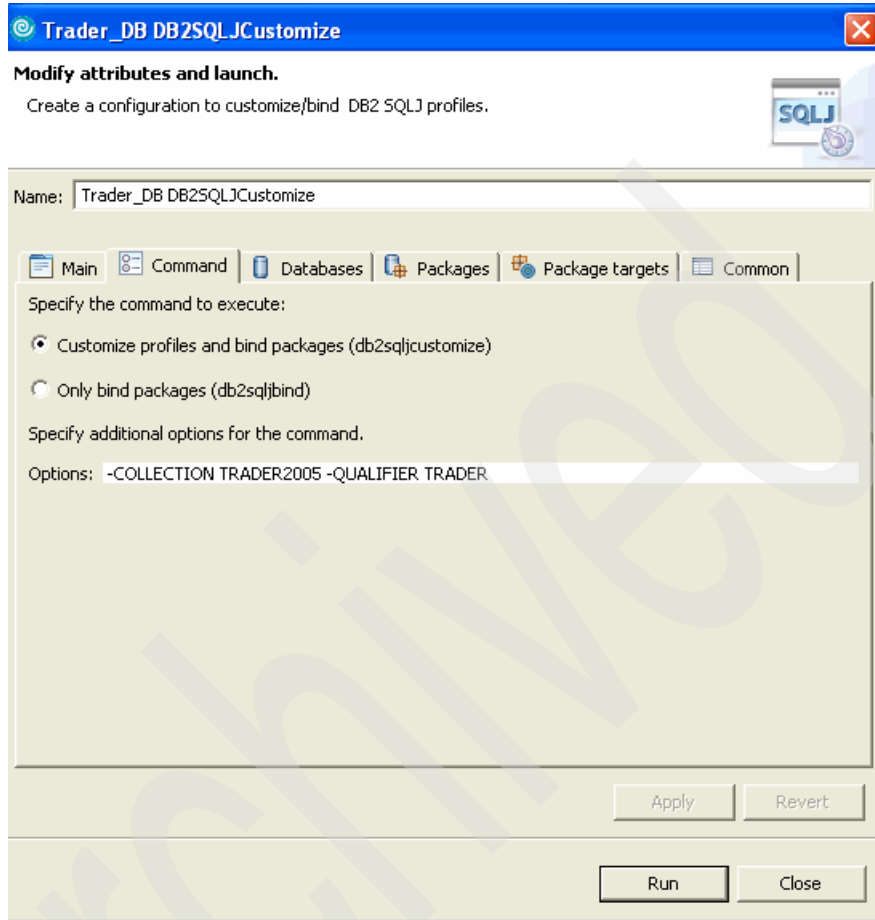


Figure 2-21 SQLJ customize and bind: Command panel

The options specified are:

- **COLLECTION:** This option defines the name of the collection we will build and bind to DB2. In our WebSphere data source, we will specify this collection so the static SQL will be used.
- **QUALIFIER:** The choice here is used to qualify unqualified objects during the online checking stage of customization.
- **bindoptions:** These do not exist in the above example. It is possible to include the `-bindoptions` keyword followed by additional bind options supported by the database. The most common example is the `"QUALIFIER(some_qualifier)"` option, which can be used to perform a

successful bind of a package when using unqualified SQL (where the object qualifier is different from the package owner).

5. Click **Run** now and check the console for error messages. At the time this book was written there were classpath errors with SQLJ customization, resulting in the error seen in Example 2-5.

Example 2-5 Customization error

```
Buildfile: C:\Documents and
Settings\TOT162\IBM\rational\sdp6.0\traderold\Trader_DB\SQLJAntScripts\sqlj.customize.xml
Trader_DB:
[java] java.lang.ClassNotFoundException: com.ibm.db2.jcc.sqlj.Customizer
[java] at java.net.URLClassLoader.findClass(URLClassLoader.java:375)
[java] at java.lang.ClassLoader.loadClass(ClassLoader.java:562)
[java] at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:442)
[java] at java.lang.ClassLoader.loadClass(ClassLoader.java:494)
[java] at java.lang.Class.forName1(Native Method)
[java] at java.lang.Class.forName(Class.java:180)
... Additional text removed for simplicity.
BUILD SUCCESSFUL
Total time: 1 second
```

Do not be fooled by the success message; the command failed. The easiest way around this problem is to update the classpath in the ant script. Navigate to the folder specified as the location of the ant scripts in “Setting up the SQLJ environment” on page 36. Double-click the ant script (**sql.customize.xml**) to edit it, as shown in Figure 2-22 on page 56.

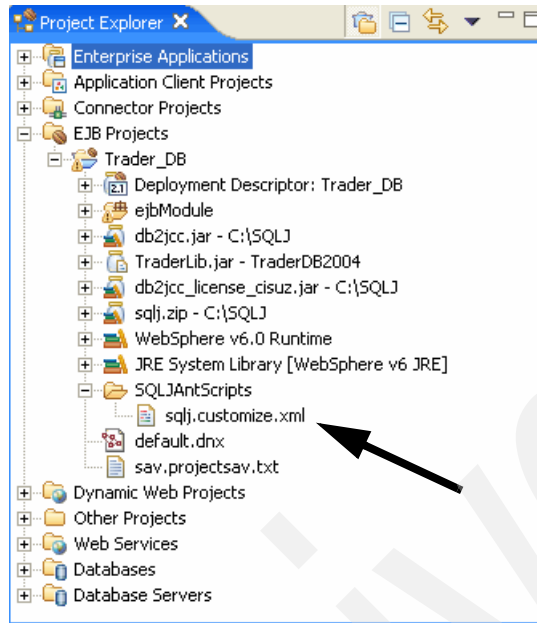


Figure 2-22 SQLJ ant script

6. Edit the classpath to include the db2jcc jar and license file. In our case they were both located in C:/SQLJ and our ant script appeared as in Example 2-6.

Example 2-6 Corrected ant script

```
<project name="Profile L/Trader_DB/SQLJAntScripts/sqlj.customize.xml" default="Trader_DB"
basedir=".">

<property environment="env"/>
<property name="cp" value="${env.CLASSPATH}"/>
<property name="lowercp" value="${env.classpath}"/>
<target name="Trader_DB">
<java fork="true" failonerror="true"
classname="com.ibm.etools.sqlj.customize.script.DB2SQLJCustomize"
classpath="..\ejbModule;..\ejbModule;C:/ProgramFiles/IBM/Rational/SDP/6.0/rwd/eclipse/plugins/c
om.ibm.etools.sqlj_6.0.0.1/sqlj.jar;C:/SQLJ/db2jcc_license_cisuz.jar;C:/SQLJ/db2jcc.jar;${cp};$
{lowercp};">

<arg value="C:/Documents and
Settings/TOT162/IBM/rational sdp6.0/trader1/.metadata/.plugins/org.eclipse.debug.core/.launches/
Trader_DB DB2SQLJCustomize.launch"/>
</java>
</target>
```

</project>

7. Do not run the customization wizard again to run this job. The wizard will overwrite the values we just entered. Instead, run this script directly by right-clicking the script and selecting **Run** → **Ant Build**, as shown in Figure 2-23.

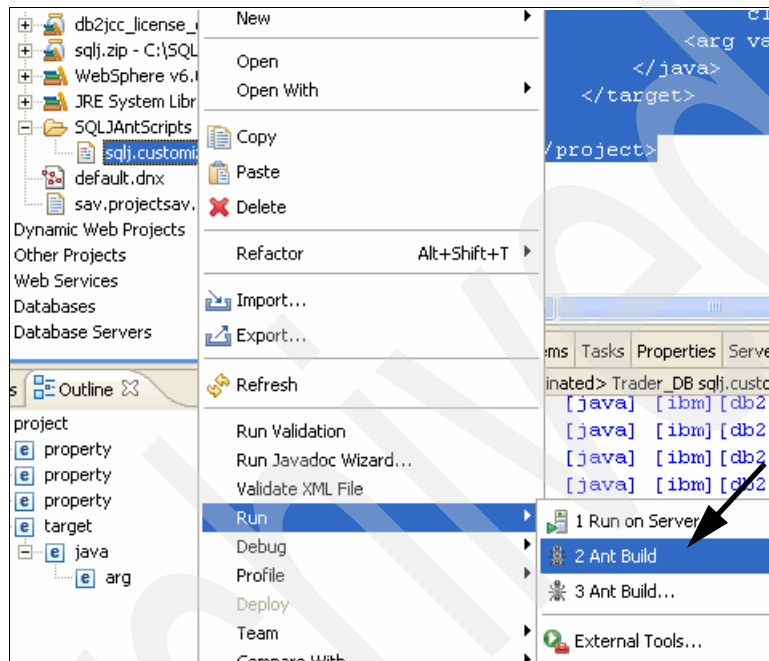


Figure 2-23 Run the ant script

The Console will now show a successful online checking, customization, and binding, similar to Example 2-7.

Important: Most users will want to ensure that they are indeed running with customized SQLJ. There are a variety of ways to do this. One of the simplest is to remove all authorizations to the plan. If a failure occurs as expected, the user can be sure that they are indeed using their customized, static SQL.

Example 2-7 Successful customization and binding

```
Buildfile: C:\Documents and
Settings\TOT162\IBM\rational\sdp6.0\trader1\Trader_DB\SQLJAntScripts\sqlj.customize.xml
Trader_DB:
[java] Selected package target=TRADER0
[java] [ibm] [db2] [jcc] [sqlj]
```

```

[java] [ibm][db2][jcc][sqlj] Begin Customization
[java] [ibm][db2][jcc][sqlj] Set qualifier for online checking to SCHEMA: TRADER
[java] [ibm][db2][jcc][sqlj] Loading profile:
itso.db2.j2ee.trader.TraderDB_SQLJ_SJProfile0
[java] [ibm][db2][jcc][sqlj] Customization complete for profile
itso\db2\j2ee\trader\TraderDB_SQLJ_SJProfile0.ser
[java] [ibm][db2][jcc][sqlj] Begin Bind
[java] [ibm][db2][jcc][sqlj] Loading profile:
itso.db2.j2ee.trader.TraderDB_SQLJ_SJProfile0
[java] [ibm][db2][jcc][sqlj] Driver defaults(user may override): BLOCKING ALL VALIDATE
BIND
[java] [ibm][db2][jcc][sqlj] Fixed driver options: DATETIME ISO DYNAMICRULES BIND
[java] [ibm][db2][jcc][sqlj] Binding package TRADER01 at isolation level UR
[java] [ibm][db2][jcc][sqlj] Binding package TRADER02 at isolation level CS
[java] [ibm][db2][jcc][sqlj] Binding package TRADER03 at isolation level RS
[java] [ibm][db2][jcc][sqlj] Binding package TRADER04 at isolation level RR
[java] [ibm][db2][jcc][sqlj] Bind complete for
itso.db2.j2ee.trader.TraderDB_SQLJ_SJProfile0
BUILD SUCCESSFUL
Total time: 4 seconds

```

Important: In order to use the TraderDB application with SQLJ support, the customization step will need to either be redone *or* the binding must be done from the host. This is because the version of the code that we have shipped in this sample has already been customized. For more information about binding packages without customization, refer to *DB2 Universal Database for z/OS: Application Programming Guide and Reference for Java*, SC18-7414-02.

2.3 Testing SQLJ in TraderDB

Testing the SQLJ support with the updated TraderDB application is as simple as clicking a new button on the main client's page. There is a small exception to this. As described earlier, it is best to customize and bind the application yourself. Also, the code will require a small change for your environment. Right now the user ID and password are hard-coded to *daubman*. Obviously, this is not ideal, and a better system of storing the passwords would be used for production, but for the case of this exercise, these values should be changed to a user ID/password that makes sense in your environment. This single required change is in the file `TraderDB_SQLJ.sqlj` in the `establishConnection()` function.

After the changes have been made and customization/binding has been re-done, the application can be deployed exactly as in 1.4.1, “Deploying the TraderDB Application” on page 20. Since this is exactly the same application, it should function fine as long as the environment was set up as described in Chapter 1, “Accessing DB2 using JDBC” on page 1.

2.3.1 Running the application

The URL for the application will depend on your environment, but have the format `http://wtsc48.itso.ibm.com:29080/TraderDBWeb/`. Pointing a browser to this URL will display the front end, as shown in Figure 2-24.

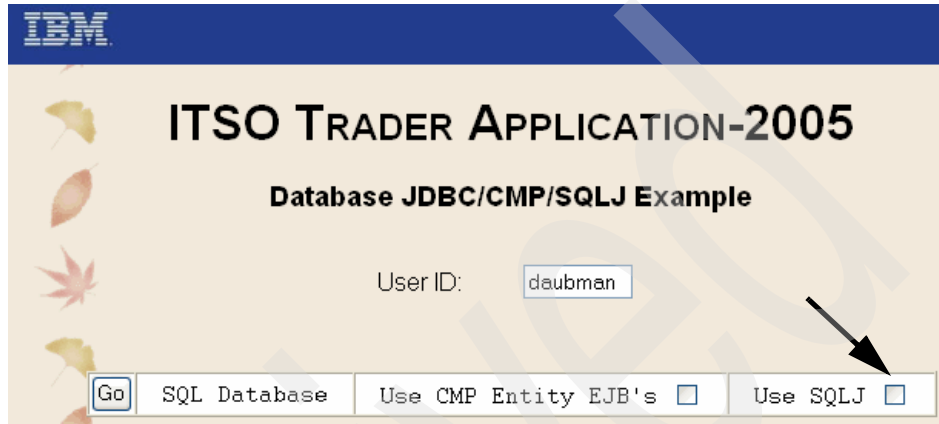


Figure 2-24 TraderDB with SQLJ support

Checking the box indicated by the arrow will force SQLJ to be used rather than JDBC. A simple test will be run by clicking this box, entering any name for the ID, and clicking Go. If Figure 2-25 on page 60 is displayed, an SQLJ command ran successfully. The application can be explored to attempt more SQLJ commands.

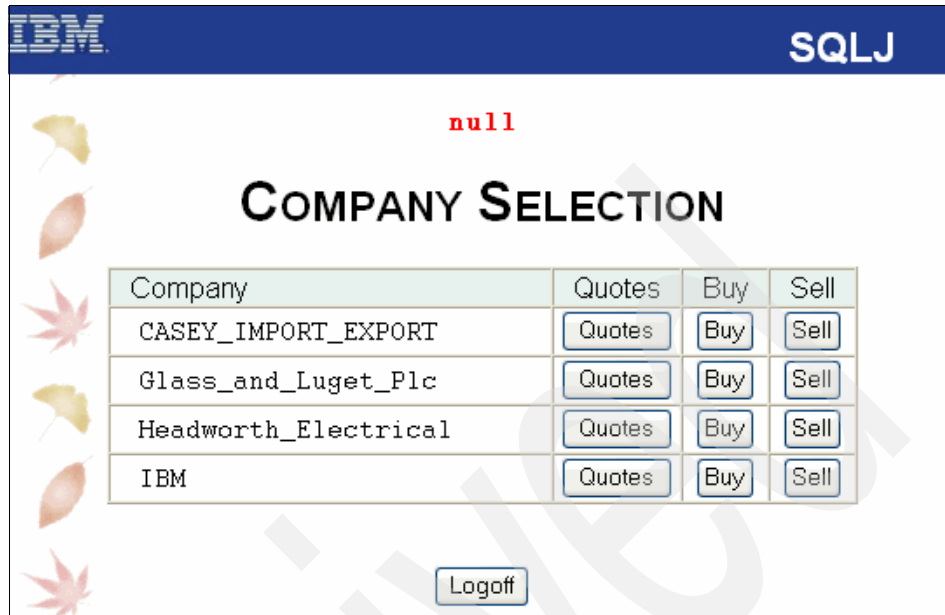


Figure 2-25 Company selection screen in TraderDB

2.3.2 Debugging potential problems

Some additional problems we found and the associated solutions are:

- Packages not bound, as shown in Figure 2-26.

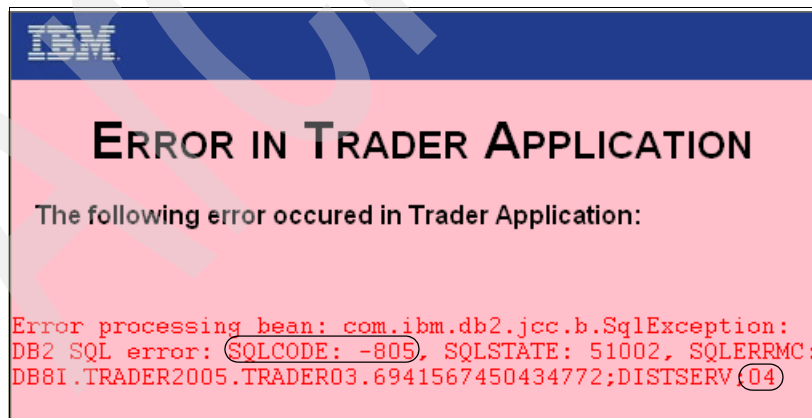


Figure 2-26 SQLCODE -805

DB2 Universal Database for z/OS: Messages and Codes, GC26-9011-02, describes the -805 SQLCODE as “PACKAGE NOT FOUND IN PLAN” error. The information in the exception gives us a specific code of 04, which translates into an error meaning that the package does not exist.

TraderDB must be customized and the packages must be successfully bound to function properly. Instructions can be found in 2.2.6, “Customization” on page 50.

- Other SQL codes can also be found in *DB2 Universal Database for z/OS: Messages and Codes*, GC26-9011-02. The book also describes solutions to these problems. In general, the SQL being run by this application is fairly basic and should not present too much trouble to DB2.

2.4 SQLJ and QoS

SQLJ can arguably offer many advantages in quality of service when compared with JDBC. This section focuses on the major advantages of SQLJ and their effect on creating a robust and secure enterprise information systems environment.

2.4.1 Performance

SQLJ is essentially different because it uses static SQL statements. This creates a performance benefit because static SQL has considerably less overhead than dynamic statements. In general, as long as the majority of the SQL statements allow for static issuing, SQLJ will demonstrate a performance advantage. Tests have shown performance gains well over 30 percent when using SQLJ. There are a couple of reasons for this. The first is the reduction of overhead in an SQLJ application. The second is that most installations do not have an optimally tuned DB2 implementation and the reduction of Java overhead has an exaggerated effect.

This assumes, of course, that customization is done for every SQLJ program. It is always recommended to perform customization as long as the database supports this. DB2 supports customization, and thus makes SQLJ a viable option.

There are other areas where SQLJ can create what are considered performance gains in code. For example, it generally requires considerably fewer SQLJ statements than JDBC to create the same functionality. For example, SQLJ has the ability to imbed variables into the code. By comparison, JDBC would require separate get and sets. On large database access intensive applications, this can amount to a measurable difference in performance.

SQLJ can also increase the performance of the developer. The SQLJ translator checks the syntax of SQL statements during translation. Since this is happening at compile time and not runtime, it saves the developer the hassle of deploying an application to check SQL syntax. JDBC does not perform these checks.

2.4.2 Security

In JDBC applications, the code is executed with the privileges of the person who connects to the database and executes the application. As a result, these users require access to the tables. SQLJ's use of static SQL allows for a more secure environment. Security privileges are assigned to the package creator and stored in the DB2 package. The statements are executed with this owner's privileges. As a result, the person running an SQLJ program does not need to have access to the tables at all. All they will need is EXECUTE privileges on the package. Now, users can be separated from the data in an ideal way.

2.4.3 Dynamic SQL is possible in SQLJ

It is often the case that dynamic SQL is required in an application. For example, it is possible that an application passes a table name as input to the program. Since the table is only known at runtime, dynamic SQL is needed. SQLJ allows for JDBC calls to be executed and also provides a means to cast JDBC result sets into SQLJ iterators. This is demonstrated in Example 2-8.

Example 2-8 JDBC result set casting

```
#sql public static iterator Info (String user, double ssn);  
// ---- Code removed for simplicity  
// JDBC with SQLJ playing together  
Statement stmt = con.createStatement();  
String query = "SELECT user, ssn FROM ";  
query += argv[0];  
ResultSet rs = stmt.executeQuery(query);  
Info theInfo;  
  
//Case the result into an iterator  
#sql [ctx] theInfo = {cast :rs };
```

Accessing IMS databases using JDBC

This chapter explains the steps we followed to configure WebSphere for z/OS V6.01 and IMS Java Database Connection (JDBC). We used these steps to install and execute the sample IMS JDBC AutoDealer application that ships with IMS.

We cover the following topics:

- ▶ IMS JDBC Topology, discussed in “IMS JDBC topology” on page 64
- ▶ Servant Region configuration, explained in “Configuring the servant region for ODBA” on page 64
- ▶ Installing the IMS JDBC Resource Adapter, explained in “Installing the IMS JDBC resource adapter” on page 65
- ▶ Configuring the IMS JDBC J2C connection factory, explained in “Configuring connection factories” on page 69
- ▶ Installation verification with the IMS DealerShip sample application, explained in “Installation verification with IMS DealerShip application” on page 76
- ▶ And, finally, some problem determination, as explained in “Problem determination” on page 80

3.1 IMS JDBC topology

IMS JDBC uses Open Database Access (ODBA) to access an IMS region directly. Therefore, the IMS region and WebSphere have to coexist on the same LPAR, as illustrated in Figure 3-1.

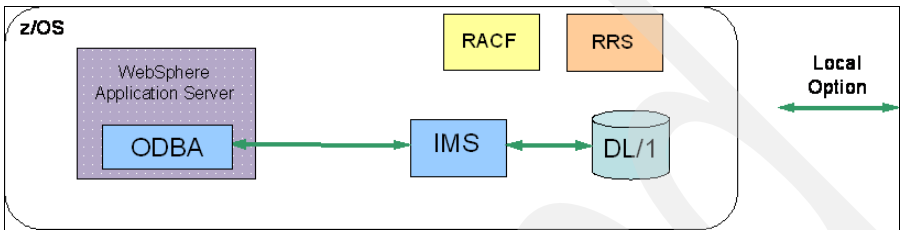


Figure 3-1 IMS JDBC configuration

3.2 Configuring the servant region for ODBA

Access to IMS databases from Java applications running in an IBM WebSphere Application Server for z/OS servant region uses IMS Open Database Access (ODBA). The ODBA interface between IMS JDBC in a WebSphere on z/OS servant region and an IMS control region is configured in a Database Resource Adapter (DRA) load module. An ODBA DRA load module is created by assembling a DFSPRP macro (see Example 3-1), which is configured with the interface details, and creating a load module with a name of DFSxxxx0, where xxxx is the same as the DRAName specified in an IMS JDBC J2C connection factory.

1. We configured and deployed a DRA module, which contained the characteristics of our ODBA connection. We created our DRA Startup Table (see Example 3-1) by assembling a DFSPRP macro into a DRA load module with a name of DFSIMSA0.

This process is described in the topics “Accessing IMS Databases via the ODBA Interface” and “The DRA Startup Table” in *IMS Version 9: Installation Volume 2: System Definition and Tailoring*, GC18-7823-00.

Example 3-1 DFSIMSA0 DRA source

```
DFSIMSA0 CSECT
          DFSPRP DSECT=NO,                                X
              FUNCLV=1,                                    CCTL FUNCTION LEVEL      X
              DDNAME=CCTLDD,                               DDN FOR CCTL RESLIB DYNALOC X
              DSNAME=IMS910A.SDFSRESL,                     DSN FOR CCTL RESLIB      X
              DBCTLID=IM4B,                                NAME OF DBCTL REGION     X
              USERID=,                                     NAME OF USER REGION     X
```

| | | |
|--------------|-----------------------------|---|
| MINTHRD=001, | MINIMUM THREADS | X |
| MAXTHRD=005, | MAXIMUM THREADS | X |
| TIMER=60, | IDENTIFY TIMER VALUE - SECS | X |
| FPBUF=010, | FP FIXED BFRS PER THREAD | X |
| FPBOF=010, | FP OVFLW BFRS PER THREAD | X |
| CNBA=010, | FP FIXED NBA BFRS PER CCTL | X |
| SOD=T, | SNAP DUMP CLASS | X |
| AGN=IVP | APPLICATION GROUP NAME | |
| END | | |

2. We updated the JCL for the IBM WebSphere Application Server for z/OS servant region by adding the load library that contains the DRA load module, the ODBA runtime code (SDFSRESL), and IMS Java native libraries (SDFSJLIB) to the STEPLIB.

Example 3-2 IMS JDBC data sets added in servant region STEPLIB

```

/*
/* Included DataSets for IMS JDBC J2C Connector
/*
//STEPLIB DD DISP=SHR,DSN=IMS910A.SDFSJLIB
//        DD DISP=SHR,DSN=IMS910A.SDFSRESL

```

3.3 Installing the IMS JDBC resource adapter

We first installed the IMS JDBC resource adapter (imsjava91.rar), as explained here:

1. On the WebSphere Administrative Console (Figure 3-2 on page 66), we expanded **Resources** and clicked **Resource Adapters** (see 1 and 2 in Figure 3-2 on page 66).
2. On the Resource Adapters pane, we selected the node on which we wanted to install the resource adapter. We clicked **Install RAR** to start the installation process (see 3 and 4 in Figure 3-2 on page 66).

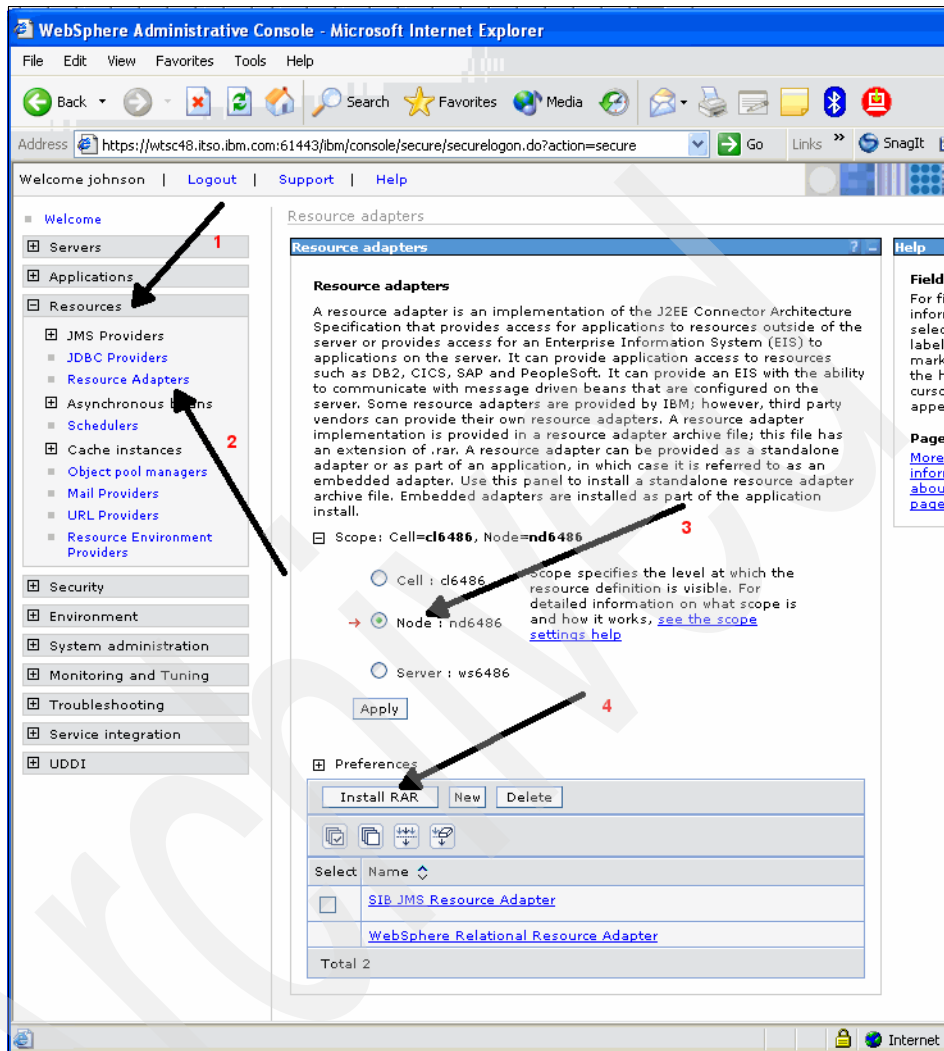


Figure 3-2 WebSphere for z/OS Administrative Console

3. On the next pane (Figure 3-3 on page 67), we traversed to the IMS JDBC RAR file (imsjava91.rar) and then clicked **Next**.

Note: The target node *cannot* be the deployment manager node.

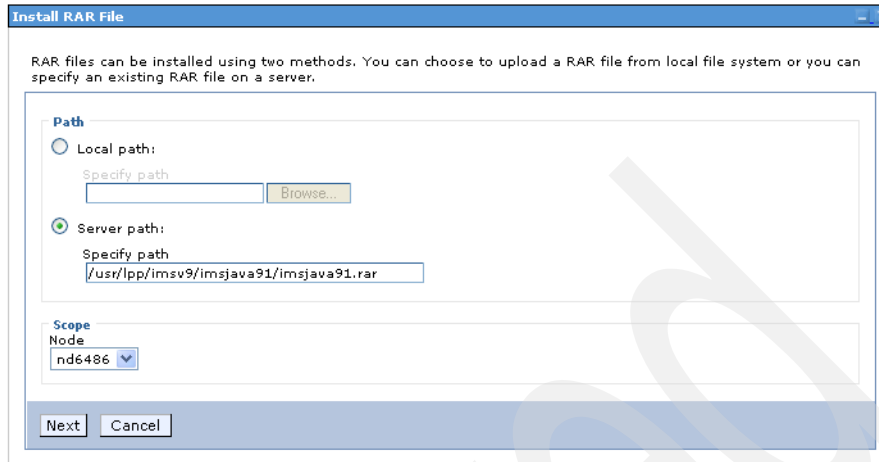


Figure 3-3 Identifying the RAR file

4. A Resource Adapter configuration pane is displayed (Figure 3-4 on page 68), and we then clicked **OK**.

Configuration

General Properties

* Scope
cells:cl6486:nodes:nd6486

Name
[Empty]

Description
[Empty]

Archive path
[Empty]

Class path
[Empty]

Native path
[Empty]

OK Reset Cancel

Figure 3-4 Resource adapter configuration

This adapter is now displayed as being installed on this node (Figure 3-5).

| Install RAR New Delete | |
|---|---|
| <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> | |
| Select | Name |
| <input type="checkbox"/> | IMS JDBC Resource Adapter |
| <input type="checkbox"/> | SIB JMS Resource Adapter |
| | WebSphere Relational Resource Adapter |
| Total 3 | |

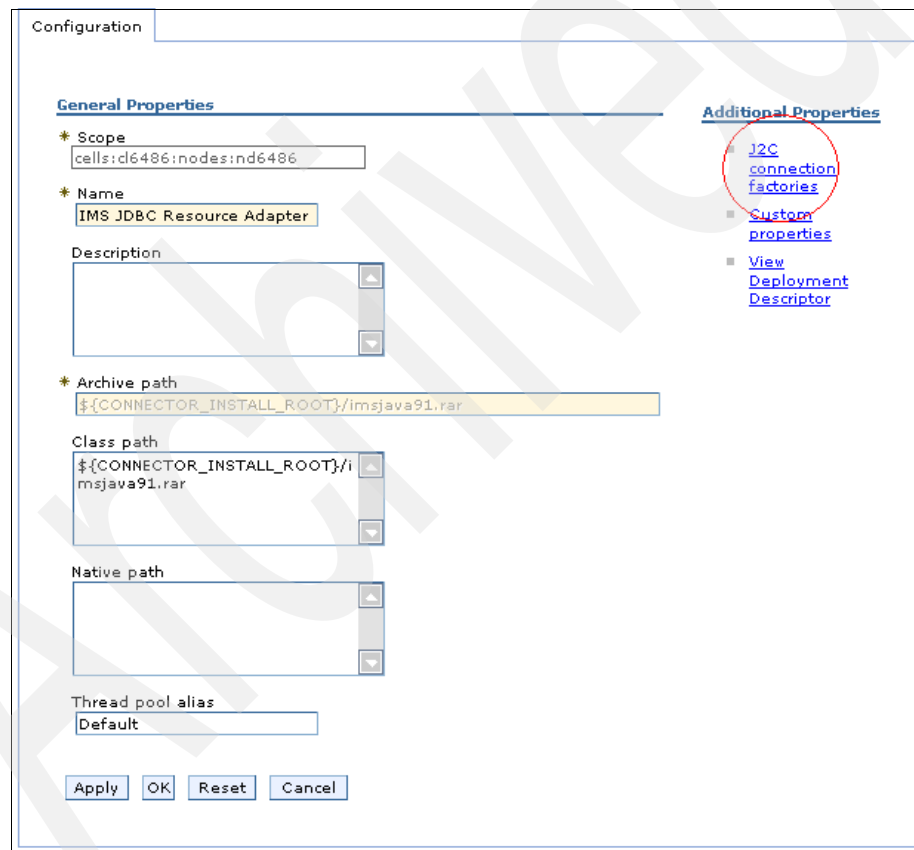
Figure 3-5 List of installed resource adapters

We could have saved the configuration changes at this time, but we wanted to configure a J2EE Connector (J2C) connection factory for the IMS JDBC adapter.

3.4 Configuring connection factories

To configure connection factories, we followed these steps:

1. We clicked the IMS JDBC Resource Adapter that we just installed to display its Configuration pane (Figure 3-6).
2. On the IMS JDBC Resource Adapter - Configuration pane, we clicked **J2C connection factories**.



The screenshot shows the 'Configuration' window for the 'IMS JDBC Resource Adapter'. The 'General Properties' tab is selected, displaying the following fields:

- Scope:** cells:cl6486:nodes:nd6486
- Name:** IMS JDBC Resource Adapter
- Description:** (empty text area)
- Archive path:** \${CONNECTOR_INSTALL_ROOT}/imsjava91.rar
- Class path:** \${CONNECTOR_INSTALL_ROOT}/imsjava91.rar
- Native path:** (empty text area)
- Thread pool alias:** Default

At the bottom are buttons for 'Apply', 'OK', 'Reset', and 'Cancel'. On the right, the 'Additional Properties' tab is visible, with 'J2C connection factories' highlighted by a red circle. Other options in this tab include 'Custom properties' and 'View Deployment Descriptor'.

Figure 3-6 IMS JDBC Configuration pane

3. The Resource Adapter → IMS JDBC Resource Adapter → J2C connection factories pane (not shown) is displayed. We clicked **New**.

Now the Resource Adapter → IMS JDBC Resource Adapter → J2C connection factories - New Configuration pane opens.

3.4.1 Configuring a J2C connection factory resource

Next, we configured a J2C connection factory.

1. On the Resource Adapter → IMS JDBC Resource Adapter → J2C connection factories - New pane (Figure 3-7 on page 71), we performed the following tasks:
 - a. For Name, we entered DealershipSample for the name for the factory.
 - b. For the Java Naming and Directory Interface™ (JNDI) name, we entered jdbc/DealerShipSample.
 - c. We then clicked **Apply**.

Configuration

General Properties

* Scope

cells:d6486:nodes:nd6486

* Name

DealerShipSample

JNDI name

jdbc/DealerShipSample

Description

* Connection factory interface

javax.sql.DataSource

Category

Component-managed authentication alias

Component-managed authentication alias

(none)

Container-managed authentication

Container-managed authentication alias
(deprecated in V6.0, use resource reference authentication settings instead)

(none)

Authentication preference (deprecated in V6.0, use resource reference authentication settings instead)

None

Mapping-configuration alias (deprecated in V6.0, use resource reference authentication settings instead)

(none)

Apply

OK

Reset

Cancel

The additional properties will not be available until the general properties for this item are saved.

Additional Properties

■ Connection pool properties

■ Advanced connection factory properties

■ Custom properties

Related Items

Figure 3-7 IMS JDBC general properties

2. Additional Properties (Figure 3-8) are now available at the right of the pane. Under Additional Properties, we clicked **Custom Properties**.

Configuration

General Properties

- * Scope: cells:cl6486:nodes:nd6486
- * Name: DealerShipSample
- JNDI name: jdbc/DealerShipSample
- Description: [Empty text area]
- * Connection factory interface: javax.sql.DataSource

Additional Properties

- Connection pool properties
- Advanced connection factory properties
- Custom properties** (circled in red)

Related Items

Figure 3-8 IMS JDBC J2C connection factory additional properties

3. Now the Custom Properties pane (Figure 3-9 on page 73) is displayed for this J2C connection factory. We entered the DRAName (IMSA is the subsystem ID) provided by the IMS system programmer and the DatabaseViewName (samples.dealership.AUTPSB11DatabaseView) provided by the application developer.

Notice the following explanations from the pane:

DRAName The name of the target IMS data store. It must match the ID parameter of the DataStore statement that is specified in the IMS Connect configuration member.

DatabaseViewName The fully qualified DLIDatabaseView subclass name.

4. We saved our configuration changes.

| Name | Value | Description | Required |
|---|---|--|----------|
| DatabaseViewName | samples.dealership.AUTPSB11DatabaseView | Fully qualified name of the database view subclass | false |
| DRAName | IMSA | The DRA name of the IMS to connect to | false |
| TransactionResourceRegistration | dynamic | Type of transaction resource registration (enlistment). This value must be "dynamic" (deferred) for this resource adapter. | false |
| Total 3 | | | |

Figure 3-9 IMS JDBC J2C custom properties

3.4.2 Installing a custom service in the application server

IMS JDBC requires the initialization of the IMS JDBC runtime environment on each application server in which an IMS JDBC application will run. We did this by adding a custom service to the server that will be executed at startup time.

1. On the WebSphere Administrative Console, we expanded **Servers** to display the available application servers. We clicked the name of the desired server to open its Configuration pane.
2. Under Server Infrastructure - Administration, we clicked **Custom Services**.
3. The Custom Services pane opens. We clicked **New**.
4. On the Custom Services Configuration pane (Figure 3-10 on page 74), we followed these steps:
 - a. We ensured that **Enable Service at startup** was selected.
 - b. For Classname, we entered `com.ibm.connector2.ims.db.IMSJdbcCustomService`.
 - c. For the fully specified classpath, we entered `/usr/lpp/imsv9/imsjava91/imsjava.jar` (see Figure 3-10 on page 74).
 - d. We clicked **OK**.

Configuration

General Properties

☒ Enable service at server startup

External Configuration URL
IMS JDBC Custom Service

* Classname
com.ibm.connector2.ims.db.I

* Display Name
IMS JDBC Custom Service

Description

* Classpath
/usr/lpp/imsv9/imsjava91/ir

Additional Properties

■ [Custom Properties](#)

Apply OK Reset Cancel

Figure 3-10 IMS JDBC Custom Services configuration

5. We returned to the server configuration pane.
6. Under Server Infrastructure - Java and Process Management, we clicked **Process Definition**.
7. We clicked the servant region to open its configuration pane.
8. On the Servant configuration pane we clicked **Environment Entries** (see Figure 3-11 on page 75).

Application servers

[Application servers](#) > [ws6486](#) > [Process Definition](#) > [Servant](#)

A process definition defines the command line information necessary to start/initialize a process.

Configuration

General Properties

Executable name

Executable arguments

startCommand

startCommandArgs

Additional Properties

- [Java Virtual Machine](#)
- [Environment Entries](#)
- [Process Execution](#)
- [Process Logs](#)

Figure 3-11 Servant configuration - Process definition

- We added an new variable for LIBPATH with a value of /usr/lpp/imsv9/imsjava91 (see Figure 3-12). This provides access to native code required to access IMS to the IMS JDBC custom service.

| <input type="button" value="New"/> <input type="button" value="Delete"/> | | | |
|---|-------------------------|--------------------------|-------------|
| <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> | | | |
| Select | Name | Value | Description |
| <input type="checkbox"/> | LIBPATH | /usr/lpp/imsv9/imsjava91 | |
| Total 1 | | | |

Figure 3-12 LIBPATH variable

- We then saved the configuration and restarted the application server.

Deployment strategies for IMS JDBC Resource Adapter

When configuring an instance of the IMS JDBC resource adapter, you can optionally set the `DLIDatabaseView` subclass name.

If you do set the subclass name, you must either create a new instance of the IMS JDBC resource adapter for every PSB an EJB accesses, or you must override the `DLIDatabaseView` subclass name (set in step 12) in the `DataSource` object by calling the `setDatabaseView` method and providing the fully qualified name of the subclass.

If you do not set the subclass name, you only need to deploy an instance of the IMS JDBC resource adapter for each DRA startup table. In the EJB, define the `DLIDatabaseView` subclass name (set in step 12) in the `DataSource` object by calling the `setDatabaseView` method and providing the fully qualified name of the subclass.

3.5 Installation verification with IMS DealerShip application

The IMS Installation DealerShip application allowed us to validate our installation and configuration.

3.5.1 Installing the IMS DealerShip application

We followed these steps to install the IMS DealerShip application:

1. From the WebSphere Application Server for z/OS Administrative Console, we clicked **Applications** → **Install New Application** to display the dialog for installing new applications.
2. We entered the path on the remote file system server to the DealerShip EAR file `/usr/lpp/ims/imsjava91/samples/dealership/was/imsjavaDealership.ear`.
3. We clicked **Next** and continued past the security warning message until step 3 Provide JNDI Names for Beans was displayed.
4. Here we changed the JNDI name for the DealershipSession Enterprise JavaBean (EJB) from the default name to `samples.dealership.was.DealershipSessionHome` (see Figure 3-13 on page 77).

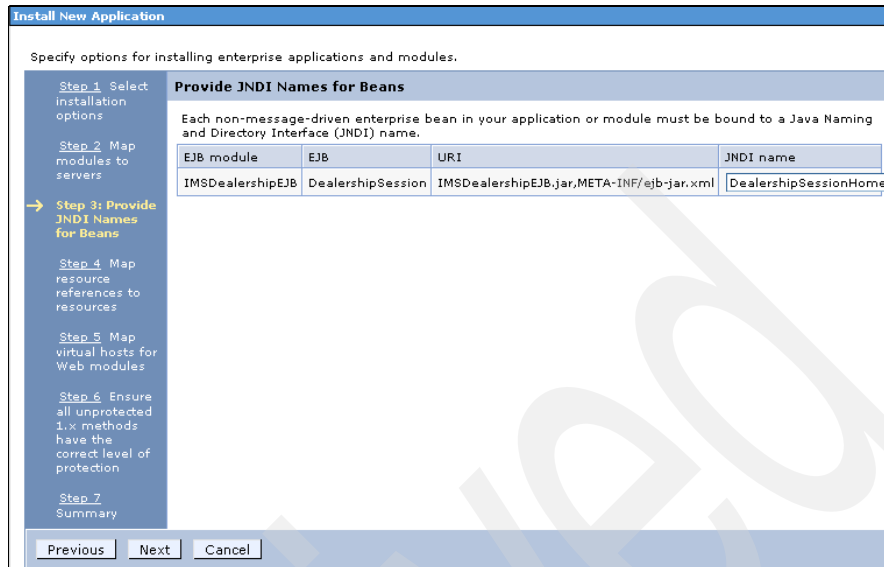


Figure 3-13 Providing a JNDI name for the EJB

- We clicked **Next** to display Map resource references to resources. IMS JDBC connection factory is not displayed when mapping the resource to resource references so we had to explicitly specify the JNDI name (jdbc/DealerShipSample) of the IMS JDBC connection factory we previously configured (see Figure 3-14 on page 78).

Step 1
Select installation options

Step 2
Map modules to servers

Step 3
Provide JNDI Names for beans

→ Step 4: Map resource references to resources

Step 5
Map virtual hosts for Web modules

Step 6
Ensure all unprotected 1.x methods have the correct level of protection

Step 7
Summary

Map resource references to resources

Each resource reference that is defined in your application must be mapped to a resource.

javax.sql.DataSource

To set multiple existing resource JNDI names:

1. Select one or more checkboxes in the table
2. Select existing resource JNDI name
3. Click Apply

Specify existing Resource JNDI name:

Select... Apply

To modify Resource Authentication method (if Authorization type is 'container'):

1. Select one or more checkboxes in the table
2. Select either 'none', 'default', or 'custom login configuration'
 - if 'none' is selected:
 - a. Select one or more checkboxes in the table
 - if 'default' is selected:
 - a. select an authentication data entry from the dropdown menu
 - b. Click Apply
 - if 'custom login configuration' is selected:
 - a. select a custom login configuration from the dropdown menu
 - b. Click Apply
 - c. To edit the properties of the custom login configuration, click Mapping Properties in the table

Specify authentication method:

☐ none

☒ Use default method

Select authentication data entry

Select... Apply

☐ Use custom login configuration

Select application login configuration

Select... Apply

| Select | Module | EJB | URI | Reference binding | JNDI name | Login configuration |
|--------------------------|------------------|-------------------|---|-----------------------|-----------------------|--|
| <input type="checkbox"/> | IMSDealershipEJB | DealershipSession | IMSDealershipEJB.jar,META-INF/ejb-jar.xml | jdbc/DealershipSample | jdbc/DealerShipSample | Resource authorization: Container Authentication method: none |

Previous Next Cancel

Figure 3-14 Mapping resource references to an application resource

6. We clicked **Next** and continued past the warning message until the Summary pane was displayed.
7. We then clicked **Finish** and saved our changes to the Master Configuration.

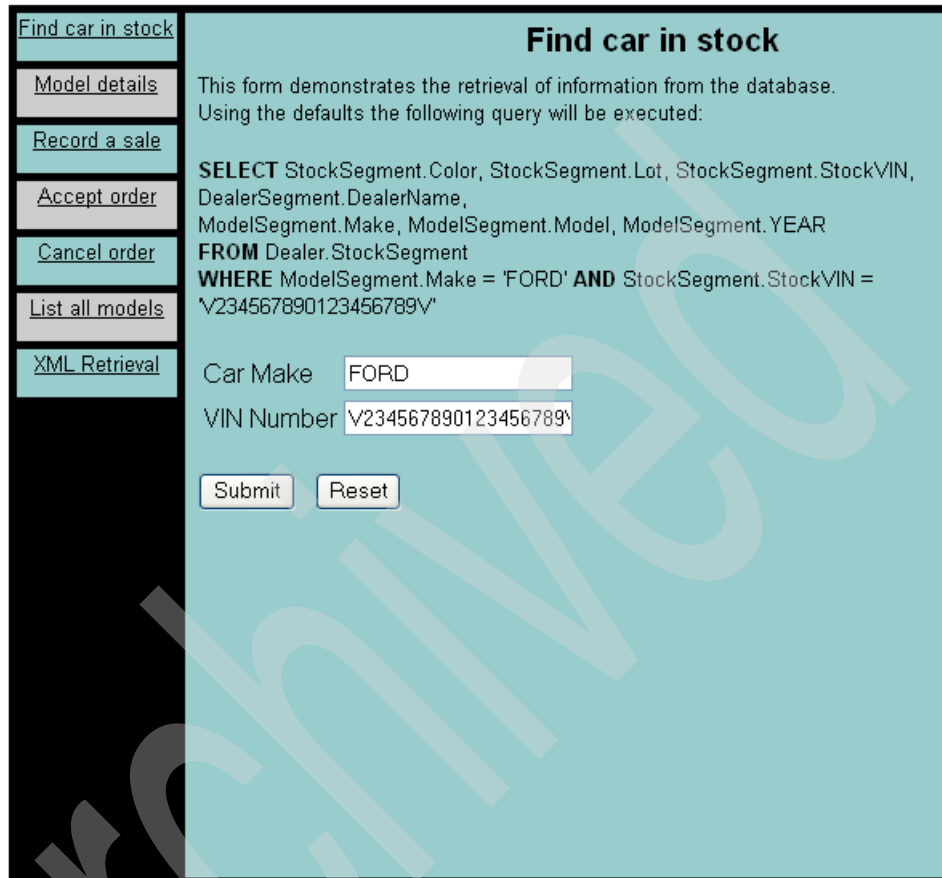
3.5.2 Using the IMS Dealership application

To use the DealerShip application, we completed the following steps.

1. From the IBM WebSphere Application Server for z/OS Administrative Console, we clicked **Applications** → **Enterprise Application**.
2. We started the application by selecting **IMSDealererShipEAR** and clicking **Start**.
3. We opened a Web browser and entered a URL of:

http://host_IP_address.ibm.com:port/IMSDealershipWeb/dealership.html

The following was displayed (Figure 3-15).



The screenshot shows a web application interface for a car dealership. On the left is a vertical navigation menu with buttons: 'Find car in stock' (highlighted), 'Model details', 'Record a sale', 'Accept order', 'Cancel order', 'List all models', and 'XML Retrieval'. The main content area has a title 'Find car in stock' and a description: 'This form demonstrates the retrieval of information from the database. Using the defaults the following query will be executed:'. Below this is a SQL query: `SELECT StockSegment.Color, StockSegment.Lot, StockSegment.StockVIN, DealerSegment.DealerName, ModelSegment.Make, ModelSegment.Model, ModelSegment.YEAR FROM Dealer.StockSegment WHERE ModelSegment.Make = 'FORD' AND StockSegment.StockVIN = 'V234567890123456789V'`. The form contains two input fields: 'Car Make' with the value 'FORD' and 'VIN Number' with the value 'V234567890123456789V'. At the bottom are 'Submit' and 'Reset' buttons.

Figure 3-15 DealerShip sample application

We clicked the **Submit** button and received the following results (see Figure 3-16 on page 80).

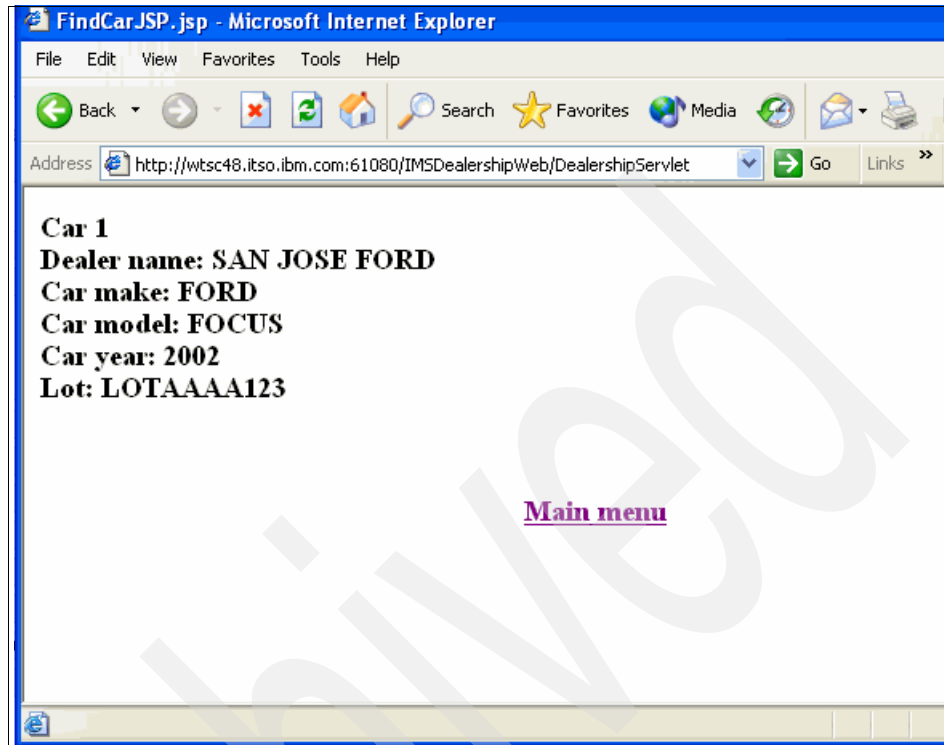


Figure 3-16 Find car results

3.6 Problem determination

To perform problem determination, we recommend that you enable the following WebSphere traces, as shown in Figure 3-17 on page 81:

- ▶ `com.ibm.ejs.j2c.*=all=enabled`
- ▶ `com.ibm.connector2.*=all=enabled`

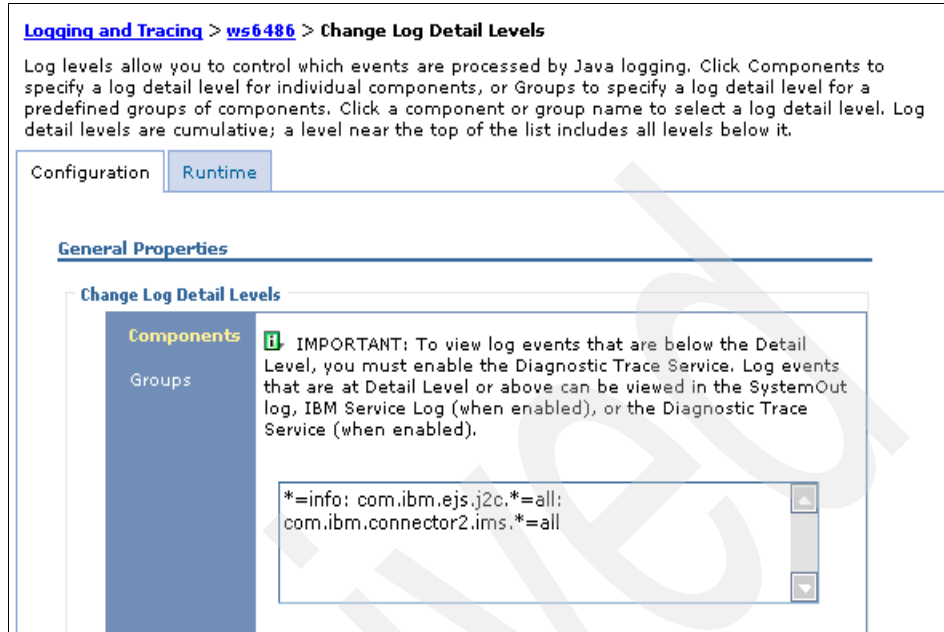


Figure 3-17 Log detail levels for IMS JDBC

Common errors

Some common errors that you may encounter are:

- ▶ Exception: java.rmi.ServerException: RemoteException occurred in server thread; nested exception is: java.rmi.RemoteException: ; nested exception is: java.lang.NoClassDefFoundError: com/ibm/ims/base/IMSTrace
 - Explanation: WebSphere was unable to locate the Java code required to access the IMS JDBC functions.
 - Usual cause: The IMS JDBC custom service was not properly configured and enabled at server startup.
- ▶ Exception: java.rmi.ServerException: RemoteException occurred in server thread; nested exception is: java.rmi.RemoteException: ; nested exception is: javax.ejb.EJBException: nested exception is: java.sql.SQLException: javax.resource.spi.ResourceAllocationException: Exception occurred during initializing the IMS JDBC Custom Service: com.ibm.ims.base.IMSException: Error in loading JavTDLI. libJavTDLI.so not found in LIBPATH
 - Explanation: WebSphere was unable to load the native code required to access the IMS JDBC functions.

- Usual cause: Either the libJavTDLI.so was not specified in the LIBPATH or IMS load module DFSCLIB was not present in the servant's regions STEPLIB.
- ▶ J2CA0021E: An exception occurred while trying to make a connection from Managed Connection resource jdbc/DealerShip :
com.ibm.connector2.ims.db. IMSJdbcEISSystemException:
com.ibm.ims.base.RuntimeExceptionTrace:
com.ibm.ims.base.RuntimeExceptionTrace: An invalid environment was detected.
 - Explanation: A proper IMS JDBC environment does not exist in the servant region's runtime.
 - Usual cause: The custom service class was not run at server startup.
- ▶ J2CA0021E: An exception occurred while trying to make a connection from the Managed Connection resource jdbc/IMSJDBC:
com.ibm.connector2.ims.db. IMSJdbcEISSystemException:
com.ibm.ims.base.RuntimeExceptionTrace: Unknown IMS Status code
 Function: APSB
 Status Code Hex: 4040
 AIB Return Code Hex: 108
 AIB Reason Code Hex: 540
 AIB Error Code Extension Decimal: 0
 - Explanation: IMS JDBC was unable to contact an active IMS control region.
 - Usual cause: The IMS subsystem specified in the DRA table is not active.
- ▶ J2CA0021E: An exception occurred while trying to make a connection from Managed Connection resource jdbc/IMSJDBC : com.ibm.connector2.ims.db. IMSJdbcEISSystemException: com.ibm.ims.base.RuntimeExceptionTrace: Unknown IMS Status code
 Function: APSB
 Status Code Hex: 4040
 AIB Return Code Hex: 108
 AIB Reason Code Hex: 20
 AIB Error Code Extension Hex: 4
 - Explanation: The load of the DRA module failed.
 - Usual cause: The DRA module was not linked with the expected name, or the load library containing the DRA module is not in the STEPLIB concatenation list.

Messaging

This chapter describes how to configure and use the Java Message Service (JMS) providers for WebSphere Application Server for z/OS Version 6.01. For ease of reference, it is divided into eight main sections:

- ▶ “Messaging within J2EE/JMS technology” on page 85 discusses messaging from a general J2EE/JMS perspective, introducing concepts, domains, and standards.
- ▶ “General overview of JMS provider types” on page 88 provides an overview of the four types of JMS providers available in WebSphere Application Server for z/OS Version 6.01, complemented by the pros and cons of each provider.
- ▶ “Message Driven Beans” on page 100 deals with the concept of Message Driven Beans (MDB) and their role in the messaging context.
- ▶ “Messaging security” on page 106 covers a number of security issues specifically related to messaging.
- ▶ “Using the WebSphere MQ JMS provider and JMS 1.0” on page 114 describes the configuration of the WebSphere MQ JMS provider using JMS 1.0.
- ▶ “Using the WebSphere MQ JMS provider and JMS 1.1” on page 133 describes the configuration of a WebSphere MQ (unified) connection factory, including JMS 1.1 application migration issues.
- ▶ “Using the Default messaging JMS provider” on page 148 describes the configuration of the Default messaging JMS provider, including an MQ link.

- ▶ “The TraderMQ application environment” on page 166 shows how to set up the environmental specifics for use with our sample TraderMQ application.

4.1 Messaging within J2EE/JMS technology

Messaging has been used for many years and has become one of the prevailing means of exchanging information between applications through communication channels. Messaging is a way of loosely coupling applications by having them exchange messages in an asynchronous way. The main advantage of messaging is the possibility to easily and reliably integrate heterogeneous application frameworks, with fewer dependencies than need to be addressed in synchronous communication protocols.

The principal objective of messaging is to exchange information:

- ▶ In an asynchronous way
- ▶ Offering assured delivery
- ▶ Offering failure independence of applications
- ▶ Offering a triggering mechanism on the reception of messages
- ▶ Using various qualities of service such as message persistence, transactional integrity, and security
- ▶ Using a common set of APIs, independent of platform or network complexity

4.1.1 MQ concepts

The messaging middleware currently proposed by IBM is WebSphere MQ, which provides a messaging service with a common set of APIs available for over forty different platforms.

The conceptual implementation of WebSphere MQ is shown in Figure 4-1 on page 86. The sending and receiving programs interface with a middleware layer, consisting of queue managers, in this case residing in system 1 and system 2, through the Message Queue Interface (MQI), and using local queues (such as Q1 for applications in system 1) or remote queues (such as Q2, managed by a remote queue manager, for applications in system 1) via a transmission queue (XQ).

A channel initiator address space in z/OS, controlled by the queue manager address space, hosts as many *Message Channel Agents (MCA)* as necessary. An MCA is in charge of sending or receiving messages to or from a queue, after establishing the communication link. The message *channel* is the one-way path consisting of the sending MCA, the communication link, and the receiving MCA.

Note there are two types of access to an MQ queue: Shared access and exclusive access. Shared access means that any number of applications can

access the queue simultaneously. With exclusive access, only one application can access the queue for a specific operation at any one time.

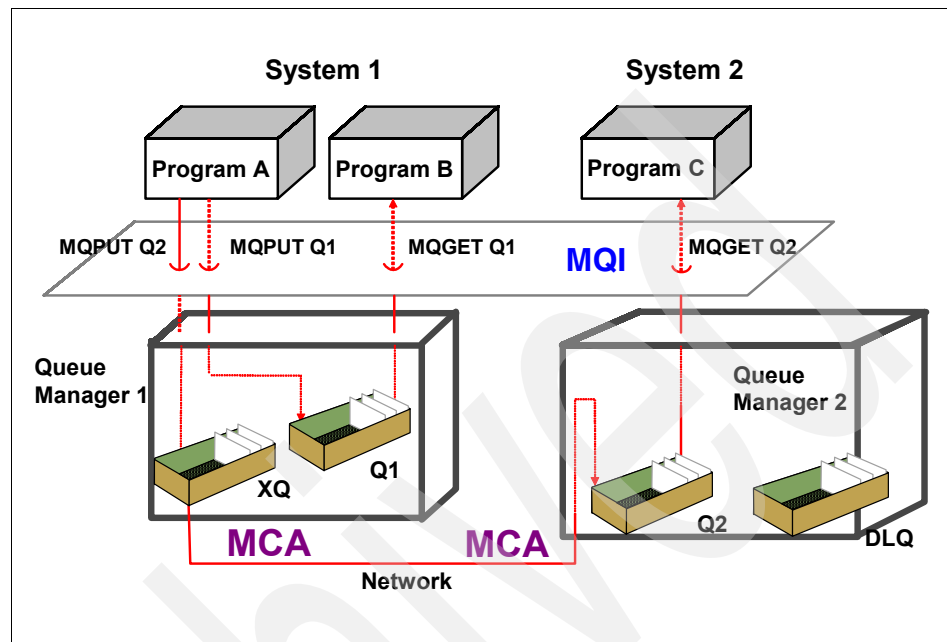


Figure 4-1 Conceptual implementation of WebSphere MQ

4.1.2 JMS concepts

Conceptually, JMS requires that a J2EE client application be able to use a set of APIs to connect to a JMS provider, in order to send and receive messages to and from other applications. The JMS API set should be available from all containers.

As shown in Figure 4-2 on page 87, a JMS client application generates a message using the JMS APIs that invoke the JMS provider. The JMS provider represents the actual code providing the JMS implementation.

The JMS client connects to the JMS provider by issuing a *createConnection* statement, and then opens a session using that connection. After the session has been opened, the client can send and receive messages. Sending messages is done using an object called a *producer*, and receiving messages is done using a *consumer*.

JMS achieves provider-independence, which allows applications to be ported across different messaging products without change.

There are two kinds of objects that together hold the parameters needed for the provider:

- ▶ The *connection factory*, which is used to generate the connection and hold the properties of the target queue manager
- ▶ The *destination*, which holds the properties of the target queue

Usually there will be many connection factories and destinations defined for a provider.

The JMS client retrieves the connection factory and destination home interfaces from the directory using the Java Naming and Directory Interface (JNDI) API.

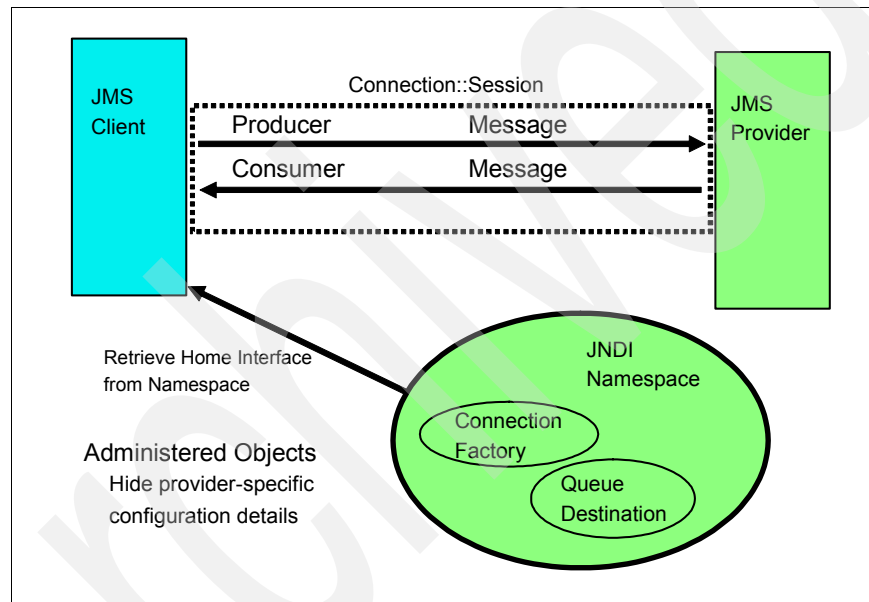


Figure 4-2 Key JMS concepts

JMS domains

Besides the *queue* type connection factories and destinations, there are also *topic* type connection factories and destinations. This relates to the two JMS domains prescribed by the JMS specification, being:

- ▶ Point-to-point
- ▶ Publish/subscribe

In a point-to-point scenario, the sending application knows the exact destination queue of the message. Point-to-point applications can be *send & forget*, where a reply to the message is not required, or *request & response*, where the

application specifies the destination for the response message with the request message and expects to receive a response.

In a publish/subscribe scenario, publisher applications publish messages to subscriber applications using a *message broker*. The messages published contain application data and one or more topic strings that describe the data. Subscribing applications register subscriptions informing the broker which topics they are interested in. When the broker receives a published message, it forwards the message to all subscribing applications for which a topic in the message matches a topic in the subscription.

JMS 1.1 specification

WebSphere Application Server for z/OS Version 6.01 implements J2EE 1.4, which includes the Java Message Service 1.1 specification.

The most significant change to JMS 1.1, compared to previous versions, is that it specifies the *unification* of domains. As a consequence, JMS clients should be able to use a common set of APIs for both the point-to-point and publish/subscribe messaging domains.

In WebSphere Application Server for z/OS Version 6.01, this is reflected by the introduction of (unified) connection factories, as opposed to queue and topic connection factories, which are still available as well.

4.2 General overview of JMS provider types

In WebSphere Application Server for z/OS Version 6.01, there are four types of JMS providers available. Two of these are actually shipped with the WebSphere product itself—the WebSphere Default messaging and V5 Default messaging. The *WebSphere MQ* JMS provider is shipped with IBM WebSphere MQ. The *Generic* provider can effectively be any other third-party JMS provider. Their purpose is to provide WebSphere applications with the possibility to easily send and retrieve messages to and from other applications using a prescribed set of JMS 1.1 APIs. Figure 4-3 on page 89 indicates the four available types of JMS Providers after selecting **Resources** → **JMS providers** from the WebSphere administrative console.



Figure 4-3 WebSphere JMS provider overview

All types of JMS providers have their own purpose, in order to suit a variety of environments. The next sections go into details about the specifics and advantages of each provider.

4.2.1 WebSphere Default messaging provider

WebSphere Application Server for z/OS Version 6.01 includes a pure Java, integrated JMS provider, called *Default messaging*. This messaging system was not available in WebSphere Application Server for z/OS Version 5.1 and is based on a new concept called the *Service Integration Bus (SIB)*. All necessary components of the Default messaging provider are shipped with and fully integrated in the WebSphere product. Once configured correctly, existing JMS clients can take advantage of the new Service Integration Bus technology.

Note: That the Default messaging provider is shipped with WebSphere can be noticed by the fact that no classpath or native library path needs to be specified on the provider configuration panel in the WebSphere administrative console.

The following sections discuss the various components of the Service Integration Bus architecture and how they work together to provide the messaging service.

Service Integration Bus

Figure 4-4 on page 90 and Figure 4-5 on page 90 depict a simple messaging infrastructure using WebSphere Application Server for z/OS Version 6.01, Default messaging, incorporating the Service Integration Bus, in a single-server and in a multi-server network deployment configuration.

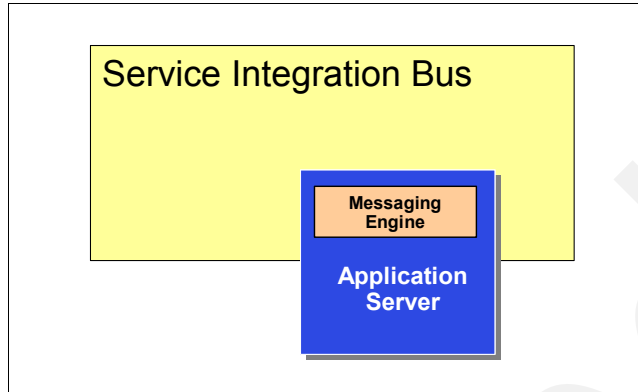


Figure 4-4 Single-server Default messaging

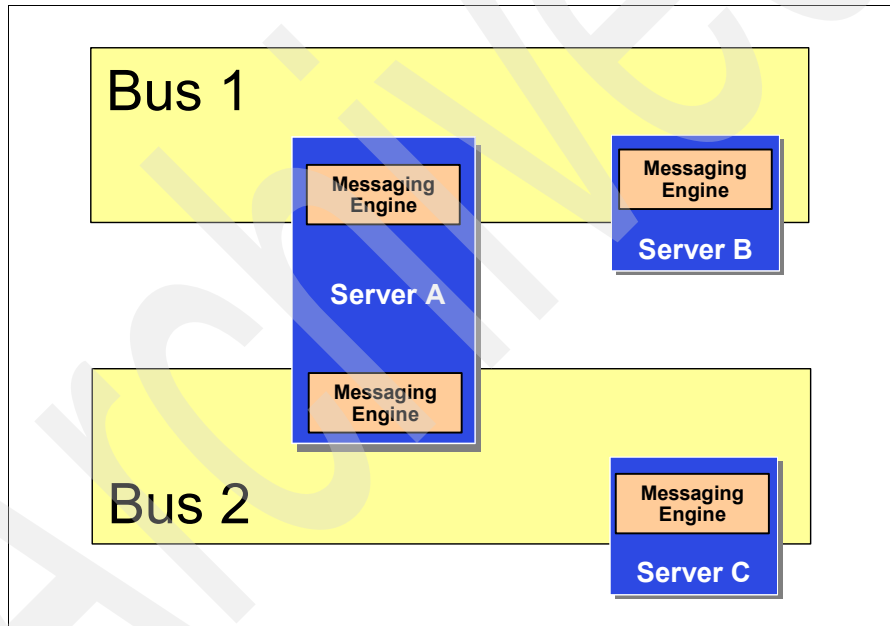


Figure 4-5 Multi-server Default messaging

The Service Integration Bus can be defined as a shared communication channel for a group of interconnected *Messaging Engines (MEs)*.

Note: The local Service Integration Bus within the cell can be managed from the WebSphere administrative console by selecting **Service integration** → **Buses**. As a consequence, buses cannot expand beyond cell boundary.

Note: There can be several buses within the same cell and application servers can be members of more than one bus.

The bus provides the basic framework for WebSphere Application Server for z/OS Version 6.01, to participate in a Service-Oriented Architecture (SOA) setup by providing messaging services to applications.

From an architectural standpoint it is considered an important component of the *Enterprise Service Bus (ESB)* concept, illustrated in Figure 4-6.

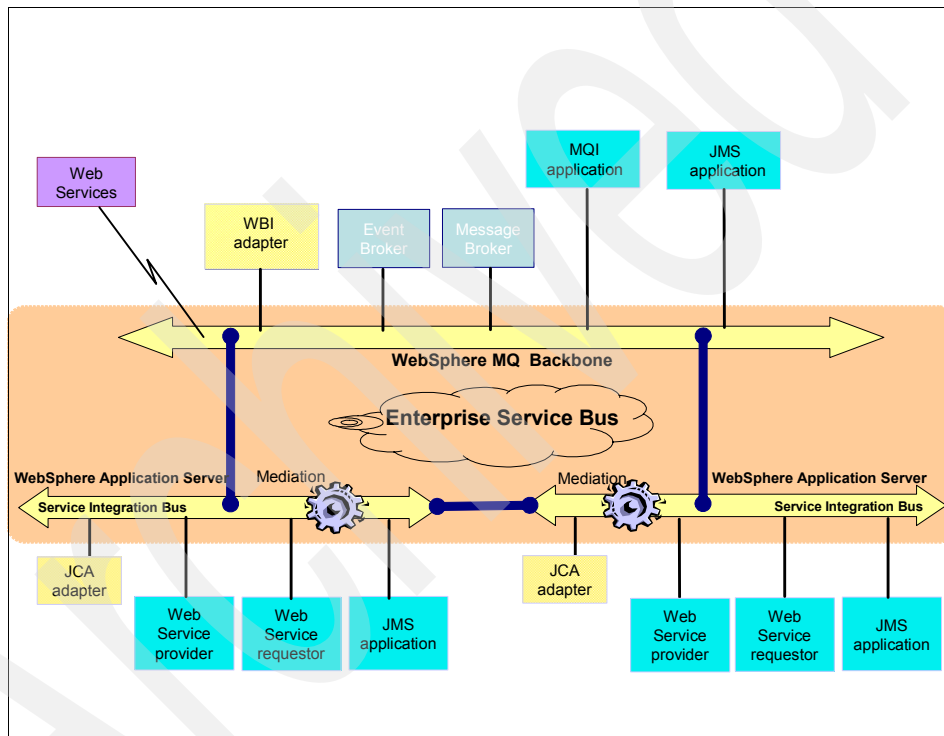


Figure 4-6 The Enterprise Service Bus concept

As the previous picture shows, the Service Integration Bus also supports the attachment of Web services requestors and providers.

Bus members

Application servers that are connected to the bus are called *bus members*. When application servers have been clustered, then effectively the cluster becomes the bus member.

Messaging Engine

A Messaging Engine (ME) represents the JMS provider. It runs in a separate *Control Region Adjunct (CRA)* address space, which is automatically started as soon as the first Messaging Engine is created. In our server we got a WS6481A address space, as shown in Example 4-1.

Example 4-1 Adjunct address space in SDSF

| SDSF STATUS DISPLAY ALL CLASSES | | | | | | | | | | LINE 1-21 (35) | |
|---------------------------------|----------------|-----------------|---------------|-----------|------------------|---|-----|-------------|-------------|------------------|--|
| COMMAND INPUT ==>> | | | | | | | | | | SCROLL ==>> PAGE | |
| NP | JOBNAME | JobID | Owner | Prty | Queue | C | Pos | SAff | ASys | Status | |
| | WS6481 | STC09410 | ASCR1 | 15 | EXECUTION | | | SC48 | SC48 | ARMELEM | |
| | WS6481D | STC09415 | WSDMNCR1 | 15 | EXECUTION | | | SC48 | SC48 | ARMELEM | |
| | WS6481A | STC09417 | ASCRA1 | 15 | EXECUTION | | | SC48 | SC48 | | |
| | WS6481S | STC09450 | ASSR1 | 15 | EXECUTION | | | SC48 | SC48 | | |

The Messaging Engine hosts the bus destinations and thus provides a connection point for clients producing and consuming messages. Messages sent to the Messaging Engine are then routed over the bus to the appropriate bus members where the message destination resides. An application server has one Messaging Engine per bus of which it is a member.

Note: When a server or cluster member is added to the bus, a Messaging Engine is automatically created for it.

Tip: It is possible to configure the Messaging Engine to look like another queue manager to WebSphere MQ.

Bus destination

A *bus destination* is an endpoint within the Service Integration Bus that applications can send messages to or receive messages from. Destinations can be either permanent or temporary. Temporary destinations live only as long as the application's session. Destinations are either local to the service integration bus or located on a foreign bus. The main types of destinations are:

- Queue** Used for point-to-point messaging.
- Topic space** Used for publish/subscribe messaging.
- Alias** An alternate name for a destination; the actual destination that the alias maps to can either be on the local bus or on a foreign bus.
- Foreign** Used to identify a destination on a foreign bus.

Note: An application can only receive messages from a destination on the bus to which it is connected. An application that subscribes to a local topic space can receive messages from a foreign topic space if topic space names have been mapped between the buses.

Tip: By the use of the bus Web services enablement, it is possible to:

- ▶ Make an internal service that is already available at a destination available as a Web service.
- ▶ Make an external Web service available at a destination.

Foreign bus

A *foreign bus* is just another Service Integration Bus that the local bus can be configured to communicate with. A foreign bus can be another Service Integration Bus within or outside the particular cell. The connection to a foreign bus can be directly by a Service Integration Bus link or by an *indirect link*, in which case messages are routed to the foreign bus through one or more intermediate buses. By interconnecting buses, messages can be sent across servers and clusters that are not directly connected to the local service integration bus. Therefore these are called foreign buses.

Note: Though it is possible to connect buses, keep in mind that a *local* Service Integration Bus can never expand beyond cell boundary.

MQ connectivity

Apart from interconnecting buses, it is also possible to configure a *WebSphere MQ link* on the bus. By doing so the Messaging Engine, to an MQ queue manager, appears to behave just like another MQ queue manager. Messages can now be exchanged between a JMS client and any application processing messages.

Restriction: A Messaging Engine cannot participate in a WebSphere MQ cluster.

Quality of service

Settings related to QoS aspects can be specified on the bus destination or on the connection factory. There are five available modes of persistence, as shown in Table 4-1 on page 94.

Table 4-1 Default messaging persistence modes

| | |
|---------------------------|--|
| Best effort nonpersistent | Messages are never written to DASD even if memory cache overflows. |
| Express nonpersistent | Messages are written to DASD if memory cache overflows; no acknowledgement on reception. |
| Reliable nonpersistent | Messages are written to DASD if memory cache overflows; acknowledgement on reception. |
| Reliable persistent | Messages are asynchronously written to DASD; in-cache messages are lost on server failure. |
| Assured persistent | Messages are asynchronously written to DASD; highest degree of delivery assurance. |

Data stores

Figure 4-7 is just a reprint of Figure 4-5 on page 90, but now includes additional concepts.

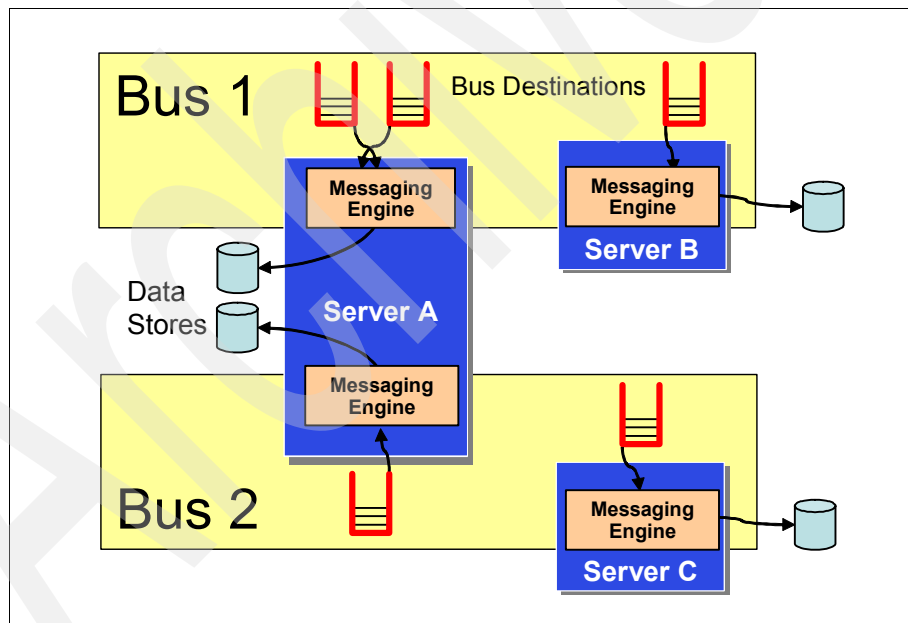


Figure 4-7 Default messaging including bus destinations and data stores

Figure 4-7 shows some example bus destinations, and the *data stores* available to the Messaging Engines. A Messaging Engine needs connectivity to a relational database, in order to store persistent messages. In the Service integration → Buses → <bus name> → Messaging Engines → <engine

name> → Data store, the JNDI name of a corresponding data source may be specified. By default a Cloudscape™-embedded JDBC provider type data source is supplied here and automatically created. However, which RDBMS should be used depends on the topology of your application servers and the capabilities of that RDBMS. If permissions are set correctly, the required tables can automatically be created as well. If not, the <WebSphere installation directory>/bin/sibDDLGenerator command can be used to generate the DDL to create these tables.

Mediation

The term *mediation* refers to the broker type component that is implemented in the Default messaging system. With mediation it is possible to manipulate messages as they traverse the bus. For example, mediation could:

- ▶ Transform a message from one format into another.
- ▶ Route a message to another destination.
- ▶ Distribute a message to multiple target destinations.
- ▶ Augment a message by adding data from a data source.

A mediation is administratively associated with a bus destination. The mediation handler is the Java program that performs the actual function of the mediation. It can be deployed in a mediation handler list, which is a collection of handlers to be invoked in sequence during mediation.

Attention: Existing WebSphere V5 JMS client applications do not need to be migrated to make use of Default messaging. In case MDBs are used, their deployment descriptor may be optionally changed to have them use activation specifications. Optionally, because this may equally well be overruled at deployment time. More information about this is in 4.3.3, “Activation specification” on page 103.

4.2.2 WebSphere MQ JMS provider

The WebSphere MQ JMS provider is the second JMS provider available on the WebSphere administrative console. If you have the IBM WebSphere MQ product installed, it is a valid choice to use this as your messaging system.

The required modules and classes are shipped with IBM WebSphere MQ. The paths to these files can be supplied as variables in the installation dialog, or later on they may be changed using the WebSphere administrative console via Environment → WebSphere variables as the node level MQ_INSTALL_ROOT and MQJMS_LIB_ROOT variables.

The WebSphere administrative console provides full administration of the WebSphere MQ JMS provider resources, being the connection factories and destinations. You need to be at least in the role of configurator in order to create or change these resources.

On the definition of an MQ connection factory there are three *transport types* available:

| | |
|-----------------|--|
| Bindings | This transport type uses cross-memory services; this is typically used when the queue manager is on the same LPAR as the application server. |
| Client | A WebSphere MQ client connection is used to connect to the queue manager; this is typically a TCP/IP-based protocol. |
| Direct | Used only for a topic connection factory; this is a lightweight connection protocol for use with the WebSphere MQ Event Broker. |

The management of WebSphere MQ resources like queue managers and queues must be performed via utilities provided with MQ, along with the definitions of RACF profiles to protect the MQ resources.

Important: Make sure you have at least WebSphere MQ 5.3.1 installed, and check the IBM Web site for prerequisite PTFs. In our project we used CSD level 8.

As explained in “JMS 1.1 specification” on page 88, JMS 1.1 specifies the presence of a *unified connection factory*, which is domain independent. In WebSphere Application Server for z/OS Version 6.01, this has been implemented by the WebSphere MQ connection factory, as opposed to *queue* and *topic* connection factories, though these are still available as well.

In case you want to use *topic* connection factories and destinations, it is also required that a message broker like WebSphere MQ Event Broker or WebSphere MQ Integrator be installed and configured.

4.2.3 Generic JMS provider

The idea of the Generic JMS provider is quite like the WebSphere MQ JMS Provider, except that any third-party JMS-compliant messaging product can be used to provide the messaging service.

In that case, WebSphere resource definitions are indirections to the provider's resources. Refer to the provider's documentation for installing and configuring the provider and managing its resources.

4.2.4 V5 Default messaging JMS Provider

WebSphere Application Server V5 specifically provided JMS 1.0 support with its *Embedded Messaging (EM)* component.

For backward compatibility with WebSphere Application Server for z/OS Version 5.1, WebSphere Application Server for z/OS Version 6.01 also includes support for the V5 Default messaging JMS provider to enable you to configure resources identical to the WebSphere Application Server for z/OS Version 5.1 Embedded Messaging system.

Tip: JMS clients developed to be used in WebSphere Application Server for z/OS Version 5.1 can just as well be deployed to make use of the Default messaging described in 4.2.1, "WebSphere Default messaging provider" on page 89. Configuration of the infrastructure is different, but the application need not be changed as long as you do not intend to use the unified connection factory.

The concept behind the supported V5 Default messaging JMS provider is to provide easy access to JMS services for J2EE applications without installing, configuring, and managing the full IBM WebSphere MQ product.

The V5 Default messaging JMS provider needs little administration. It automatically creates a queue for an application if it does not exist in the appropriate EM queue manager, using a sample model queue.

Although the technology under the covers for EM is essentially provided by IBM WebSphere MQ (for point-to-point messaging) and IBM WebSphere MQ Event Broker (for publish/subscribe messaging), this function is completely isolated and not interoperable with WebSphere MQ.

4.2.5 Pros and cons of the four types of JMS providers

As it stands today for JMS implementations in WebSphere on z/OS, there is a choice between four JMS providers (Default messaging, WebSphere MQ messaging, Generic messaging, or V5 Default messaging).

We do not give any direct recommendations, since each JMS provider has its own advantages and disadvantages, depending on your environment. Instead, to

help you decide which JMS provider suits your environment best, the following tables provide an overview of the pros and cons:

Table 4-2 Default messaging pros and cons overview

| Default messaging | |
|--|---|
| Pros | Cons |
| It comes shipped with WebSphere Application Server for z/OS Version 6.01. | You need to get familiar with the new concepts. |
| It fully supports the JMS 1.1 API specification. | It is only available on platforms with WebSphere Application Server Version 6.01. |
| It uses a single point of administration by the WebSphere administrative console. | The Messaging Engine cannot participate in an MQ cluster, although it can connect to a queue manager that is part of a cluster. |
| It provides a broker type functionality by the concept of mediation. | |
| It can coexist with WebSphere MQ on the same LPAR and be connected to a queue manager via an MQ link. | |
| It offers a high granularity of qualities of service regarding message persistence. | |
| It includes a fail-over mechanism for Messaging Engines in a cluster. | |
| Existing V5 JMS clients can make use of it without any migration changes made to them. | |
| Default messaging is a strategic messaging mechanism (as well as WebSphere MQ messaging) focused on the J2EE market. | |

Table 4-3 WebSphere MQ messaging pros and cons overview

| WebSphere MQ messaging | |
|--|--|
| Pros | Cons |
| It fully supports the JMS 1.1 API specification and the MQI API set. | IBM WebSphere MQ needs to be purchased separately. |
| It provides direct connectivity to all (over forty) platforms where WebSphere MQ is available. | No single point of administration. |
| It provides direct connectivity to an MQ cluster. | In case of publish/subscribe messaging, a broker product must be purchased separately. |
| It uses the connection pooling mechanism to improve performance. | |
| WebSphere MQ messaging is a strategic messaging mechanism (as well as Default messaging) focused on the predominantly non-J2EE market. | |

Table 4-4 Generic messaging pros and cons overview

| Generic messaging | |
|--|--|
| Pros | Cons |
| It allows you to install and configure any third-party JMS messaging product of your own choice. | A third-party messaging provider needs to be purchased separately. |
| Refer to the product's documentation for the specific advantages the provider offers. | No single point of administration. |

Table 4-5 V5 Default messaging pros and cons overview

| V5 Default messaging | |
|---|--|
| Pros | Cons |
| It comes shipped with WebSphere Application Server for z/OS Version 6.01. | It does not support the JMS 1.1 API specification, only 1.0, so no unified connection factories. |
| It uses a single point of administration by the administrative console. | Communication is limited to within the WebSphere Application Server cell. |
| It requires very little configuration to make it work. | It cannot coexist with WebSphere MQ on the same LPAR. |
| Despite its limited functionality it is a robust and production quality messaging system. | It will be discontinued in future releases of WebSphere. |

4.3 Message Driven Beans

WebSphere Application Server for z/OS Version 6.01 supports a triggering mechanism based on the arrival of messages onto a destination. This mechanism is implemented by the use of a special type of enterprise bean, called the *Message Driven Bean (MDB)*, as defined in the EJB 2.1 specification.

The triggering of MDBs is accomplished by the *message listener service* for WebSphere MQ and V5 Default messaging, or the definition of a *JMS activation specification* for Default messaging. This difference results from the fact that the Default messaging provider implements the JCA 1.5 resource adapter. For generic messaging, refer to the provider's documentation to see what specification has been implemented.

Figure 4-8 on page 101 shows what the triggering mechanism using the MDB conceptually looks like from the JMS client's perspective.

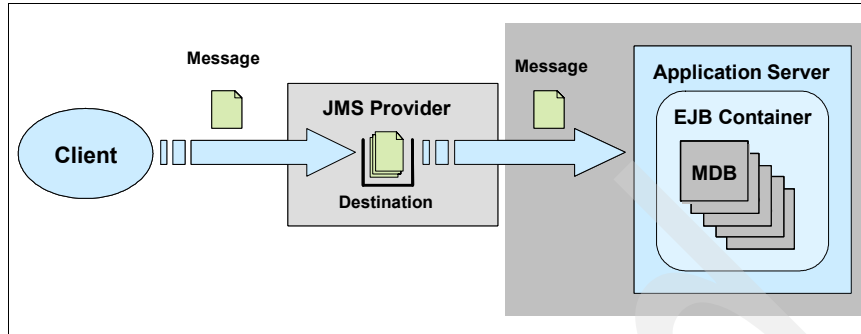


Figure 4-8 Client's perspective of the Message Driven Bean

4.3.1 MDB characteristics

The following lists some of the key characteristics of Message Driven Beans to keep in mind:

- ▶ They are invoked by the arrival of a message on a JMS destination.
- ▶ They support both the point-to-point and publish/subscribe domains.
- ▶ They are modelled after a stateless EJB.
- ▶ They do not expose their home interfaces.
- ▶ The messages they process contain no client credentials.
- ▶ They preserve transactional integrity.

4.3.2 Listener service

As mentioned before, the message listener service is used for the WebSphere MQ and V5 Default messaging. It was already provided with WebSphere Application Server for z/OS Version 5.1. It comprises a *listener manager* that controls one or more *listeners*. Each listener monitors a *listener port* for incoming messages. The listener port can be considered a logical connection between the connection factory, the destination, and the MDB, as illustrated by Figure 4-9 on page 102.

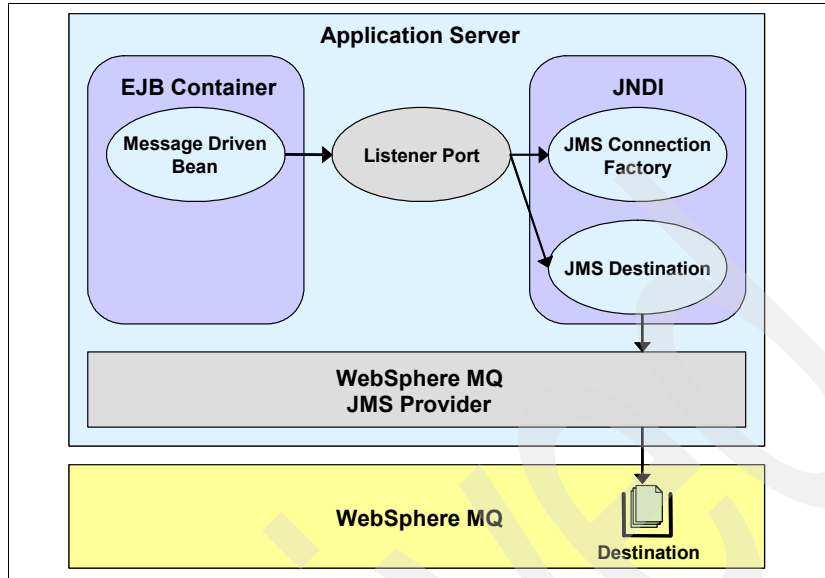


Figure 4-9 Basic listener port configuration

When the listener detects the arrival of a message on the destination, it passes it on to a new instance of the associated MDB for processing. The listener then starts looking for the next incoming message to be processed, without waiting for the bean to return.

A more elaborate scenario is presented in Figure 4-10 on page 103, showing a number of possible interconnections among listeners, listener ports, connection factories, destinations, and Message Driven Beans.

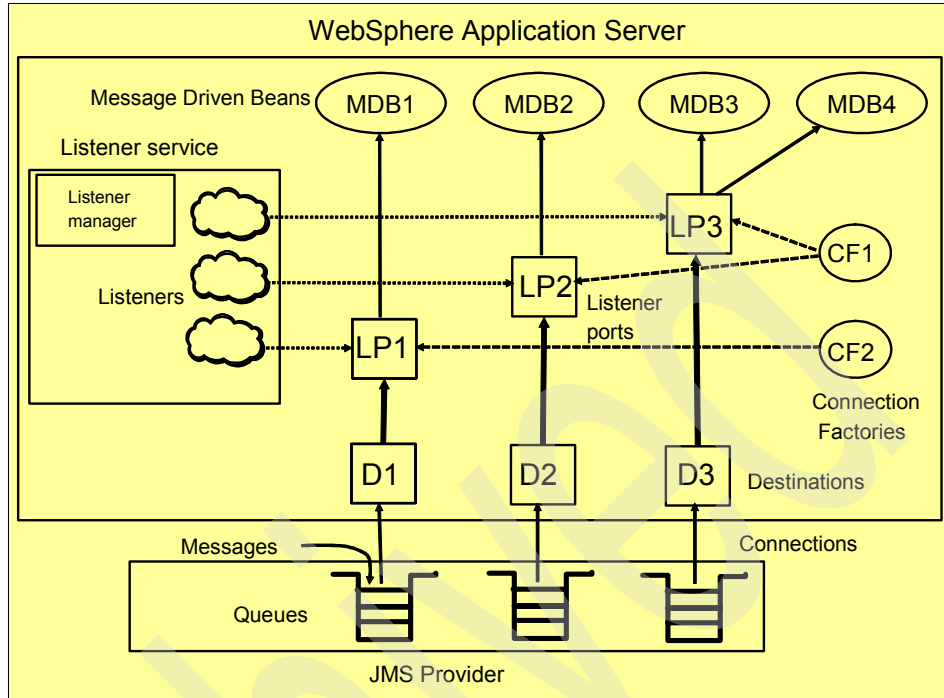


Figure 4-10 Elaborate listener port configuration

Before deployment of the MDB, the required listener ports must be configured. This can be done using the administrative console on the **Servers** → **Application servers** → <server name> → **Messaging** → **Message listener service** → **Listener ports** panel. Then, during deployment, the MDB is associated with the listener port.

4.3.3 Activation specification

As mentioned before, the JMS activation specification is used for Default messaging, since the Default messaging provider implements the JCA 1.5 resource adapter. The activation specification can be considered a logical connection between the destination and the MDB, as illustrated by Figure 4-11 on page 104.

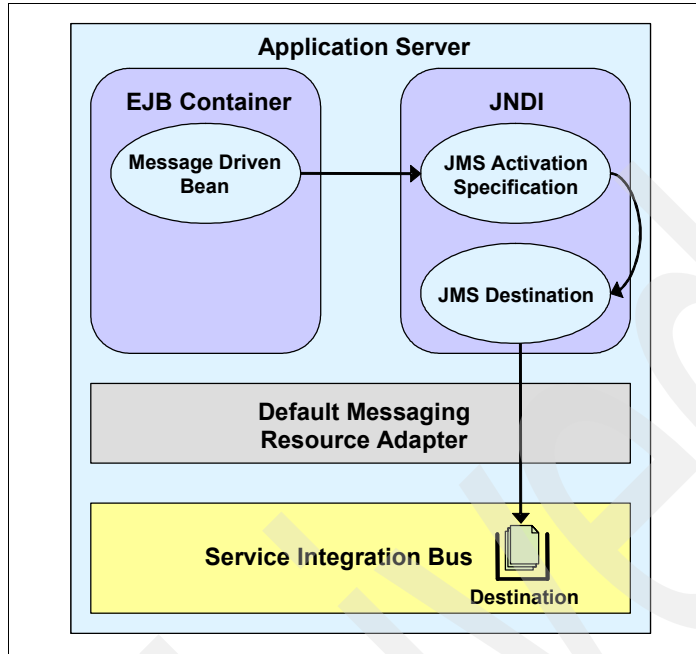


Figure 4-11 Basic JMS activation specification configuration

Before deployment of the MDB, the required activation specifications must be configured. This can be done using the WebSphere administrative console on the Resources → JMS providers → Default messaging provider → JMS activation specification panel. These JMS activation specifications are stored in the JNDI name space by WebSphere Application Server for z/OS Version 6.01. Then, during deployment, the MDB is associated with the activation specification. MDBs can only be triggered from destinations that are local to the Service Integration Bus.

Attention: Existing V5 pure JMS MDBs using listener ports do not need to be migrated to make use of activation specifications. However, you may *optionally* change their deployment descriptors to sue activation specifications. Optionally, because this may equally well be overruled at deployment time.

4.3.4 MDB programming directives

The following list provides some of the best practices to keep in mind for a programming model using Message Driven Beans:

- ▶ Delegate business logic to another EJB to provide clear separation of message handling and business processing.
- ▶ Minimize message processing time for the MDB.
- ▶ Avoid large message bodies as much as possible.
- ▶ Do not maintain any conversational state in the MDB.
- ▶ Avoid dependencies on the order in which messages arrive.
- ▶ Prevent having *poison* messages, exhausting resources as they are rolled back to immediately reinvoke an MDB.

Figure 4-12 illustrates an example programming model using a Message Driven Bean. A company's employee using the browser can invoke a session EJB over a servlet. On the other hand, there is a JMS client involved writing messages to a queue. These messages trigger an MDB, which in turn invokes the same EJB. The EJB can then use an entity bean to retrieve or update entries in a database.

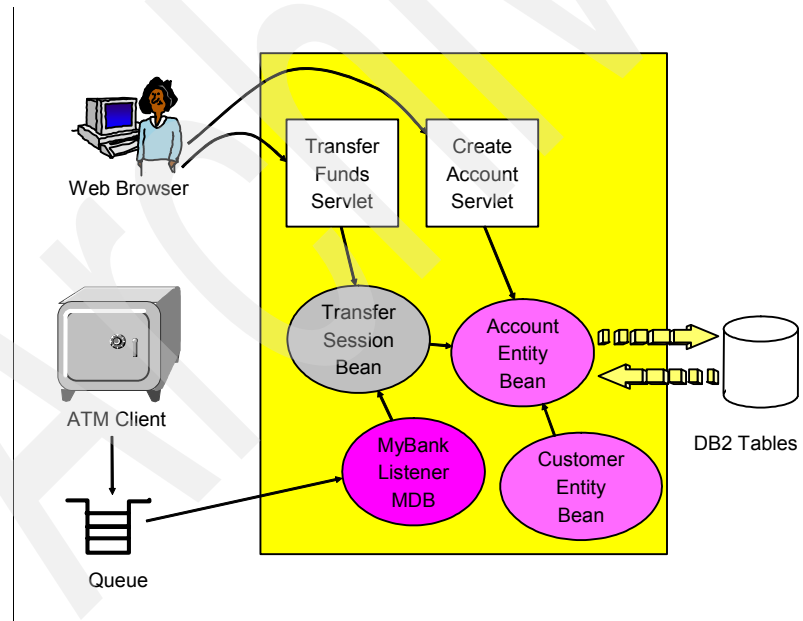


Figure 4-12 MDB programming model example

4.4 Messaging security

The messaging security model includes the following three major areas:

- ▶ Determining and authenticating the user's identity
- ▶ Checking the identity's authorization on the provider's resources
- ▶ Ensuring the message's confidentiality and integrity in transit

The next sections cover these areas.

4.4.1 Identity authentication

This section describes the way WebSphere Application Server for z/OS Version 6.01 determines the user ID to be authenticated to propagate with the message towards the JMS provider. It ends with some security considerations regarding MDBs.

Messaging security operates as part of the WebSphere Application Server global security and is only enabled when global security is enabled. If so, user IDs requesting connections to the JMS provider are authenticated, and can then be used by the JMS provider to control access to its resources like queues.

The user ID that will be used for authentication can be provided by the application or the container and depends on a combination of settings. When the authentication fails, the connection request is rejected.

In fact, the JMS identity decision is based on the standard J2C authentication mechanism. Figure 4-13 on page 107 provides a detailed decision tree explaining which identity will be propagated with the message.

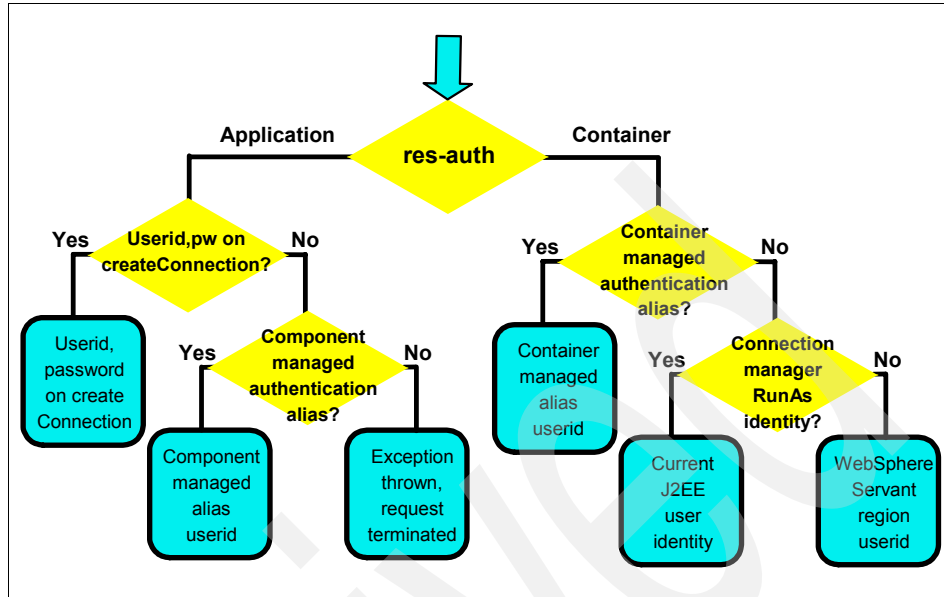


Figure 4-13 JMS identity authentication decision tree

A restriction to this decision tree is that for the WebSphere MQ JMS provider it is valid only when the transport type for the connection factory is specified as bindings, with the transport type set to client thread security not supported. However, on z/OS the bindings transport type is typically used since the queue manager and the application server are likely to be colocated on the same LPAR.

Note: It is important to be aware that due to the asynchronous nature of messaging it is not possible to propagate the full security context, that is, an *authenticated* user identity token. WebSphere Application Server will authenticate the user before accessing the JMS provider and propagate *just the identity* in the message according to the tree. It is then the responsibility of the receiving application or system component whether to re-authenticate the user ID in any way before further processing is done on its behalf.

4.4.2 Message Driven Bean identity

The MDBs are a particular kind of beans in that they are *anonymous*. The messages being processed by an MDB have no client credentials associated with them. As a consequence, the security of an MDB depends on the role specified by the run-as identity for the MDB. This identity can then be propagated to the invoked EJB as recommended in “MDB programming directives” on page 105.

4.4.3 Resource authorization

After the identity to be associated with a message has been determined it is up to the JMS provider to check the identity's authorizations on its resources. The following sections deal with the four available messaging providers and how they handle authorization checking.

Default messaging

Like the Default messaging system itself, the protection of the Service Integration Bus resources is also integrated in WebSphere Application Server for z/OS Version 6.01.

Restriction: It is not possible to delegate the security administration of Default messaging to RACF or another external user registry.

For security checking on the bus, the Secure box on the Bus properties panel of the WebSphere administrative console should be checked. You need to be at least in the role of configurator in order to change this setting.

Note: Security for messaging operates as part of the WebSphere Application Server global security; hence, it is enabled only when global security is enabled. Java 2 Security, however, does not have to be enabled.

With these settings in place, Default messaging security is switched on and all access to the bus and all its destinations must be authorized. This means that all users who want to connect to the bus must have permission to use the bus resources, either directly or as a member of a group.

When a bus is created, an initial set of permissions is created simultaneously that allow all authenticated users to connect to the bus, and grant them full access to all local destinations on the bus.

Note: Messaging security is a bus-wide setting, which means that it is not possible to switch security on for some Messaging Engines in a bus and off for the others.

Messaging security implements a simple role model in which a role is associated with the permission required to perform a certain operation. If messaging security is switched on, all users who connect to the bus to perform the operations they request have to be assigned the appropriate roles. Roles come in six categories:

| | |
|----------------------|---|
| Default roles | Used to define permissions that apply to all resources of a specific kind |
|----------------------|---|

| | |
|-------------------------------|--|
| Bus connector roles | Used to define permissions to connect to a specific local bus |
| Foreign bus roles | Used to define permissions to connect to a specific foreign bus |
| Destination roles | Used to define permissions to perform operations on a specific destination |
| Topic space root roles | Used to define permissions to a specific topic space virtual root |
| Topic roles | Used to define permissions to separate topic items |

The default roles are special in that they do not require the name of, say, a destination. Instead, once defined they apply to all destinations. This type of role is used to create the initial set of permissions when a new bus is added. The following list specifies the available default roles:

| | |
|------------------------|--|
| Connector | Grants permission to connect to local buses |
| Sender | Grants permission to produce messages to destinations |
| Receiver | Grants permission to consume messages from destinations |
| Browser | Grants permission to browse messages on destinations |
| Creator | Grants permission to create temporary destinations |
| IdentityAdopter | Grants permission to send a message under a different user identity, not to be used from JMS |

The administration of Default messaging security is done using the *wsadmin* shell script and issuing the required **\$AdminTask** commands from there. For example, to list the groups with access to a default role, or add a user to a queue type destination role for putting messages onto it, specify what is shown in Example 4-2.

Example 4-2 Example Default messaging security administration commands

```
wsadmin>$AdminTask listGroupsInDefaultRole {-bus MyLocalBus -role Receiver}
or
wsadmin>$AdminTask addUserToDestinationRole {-bus Basbus -type Queue
-destination EdQueue -role Sender -user Otten}
```

Issue **\$AdminTask help -commands** to list all available commands and **\$AdminTask help <commandname>** to see all details for a command. Make sure to **\$AdminConfig save** your changes.

For ease of administration, two predefined special groups of users are available:

| | |
|-------------------------|---|
| AllAuthenticated | Containing all authenticated users. If the <i>AllAuthenticated</i> group is authorized to perform an operation, then all authenticated users are authorized to perform it. When a bus is created, this special group is used to initially grant all authenticated users permission to connect to the bus and access all local destinations. |
| Everyone | Containing all users whether or not they are authenticated. |

If authentication is successful, an access check is performed to see whether the user has adequate permission to connect to the bus. If not, the connection is rejected. If so, further access checks are performed depending on the requested operations.

When the user tries to access a topic, an access check is performed on the topic space, which is the destination that contains the topic. But, if the Topic access check required box on the Topic space properties panel is checked, a second access check is performed on the topic itself. Every topic can be protected separately or inherit its protection from a higher level topic or the virtual root. This option allows a more granular control of access within a topic space hierarchical structure.

To control which Messaging Engines can create a connection to a given Messaging Engine, set the Inter-engine authentication alias on the Bus properties panel to prevent unauthorized clients or Messaging Engines from establishing a connection.

If you are using mediations, set the Mediations authentication alias on the Bus properties panel to prevent unauthorized mediations from establishing a connection.

To control which foreign buses can establish a link to your local bus, set the authentication alias when configuring a link definition on the Messaging Engine Service Integration Bus link panel to be used to authenticate the foreign bus joining your bus.

Note: Beware that permissions to foreign destinations only control whether a user is allowed to send messages to the foreign bus. When a message reaches the foreign bus, permissions set by the administrator of the foreign bus determine whether the message can proceed to its destination.

Access to the data store for a Messaging Engine can be secured using an *authentication alias*. More advanced levels of security can be applied to the data store by exploiting the features of the underlying RDBMS.

For more information about the specifics of how to set up the security definitions using the `wsadmin $AdminTask` commands, refer to the WebSphere Application Server for z/OS Version 6.01 InfoCenter at:

http://publib.boulder.ibm.com/infocenter/wasinfo/v6r0/index.jsp?topic=/com.ibm.websphere.zseries.doc/info/welcome_nd.html

WebSphere MQ messaging

To WebSphere Application Server for z/OS Version 6.01, WebSphere MQ is considered an external messaging provider, so naturally WebSphere does not provide any protection to the MQ resources. Essentially, disabling global security can never prevent protection of the MQ resources.

IBM WebSphere MQ uses the SAF interface and therefore, in our environment, RACF profiles to ensure that its resources are not accessed in any unauthorized way. Note that the WebSphere MQ RACF profiles are specific to a queue manager, since they are prefixed by the queue manager's subsystem identifier.

As a consequence, if you decide to use the WebSphere MQ JMS provider, you have to create the appropriate RACF profiles to control access to the MQ resources, as explained in detail in the WebSphere MQ documentation.

It is interesting to see how the WebSphere MQ JMS provider maps the JMS message to an MQ message and back. In order to achieve this, the provider needs to map the properties from the JMS message header onto corresponding fields in the MQ message. Some of the properties like the `JMSXUserID` are mapped onto the identity context, which is part of the MQ message descriptor (MQMD). This is exactly the value set to the identity, as discussed in 4.4.1, "Identity authentication" on page 106. Other properties are mapped onto the xml-format RFH2 header, which comes immediately after the MQMD. The JMS message body is copied unchanged into the MQ message body.

The mapping is illustrated schematically by Figure 4-14 on page 112.

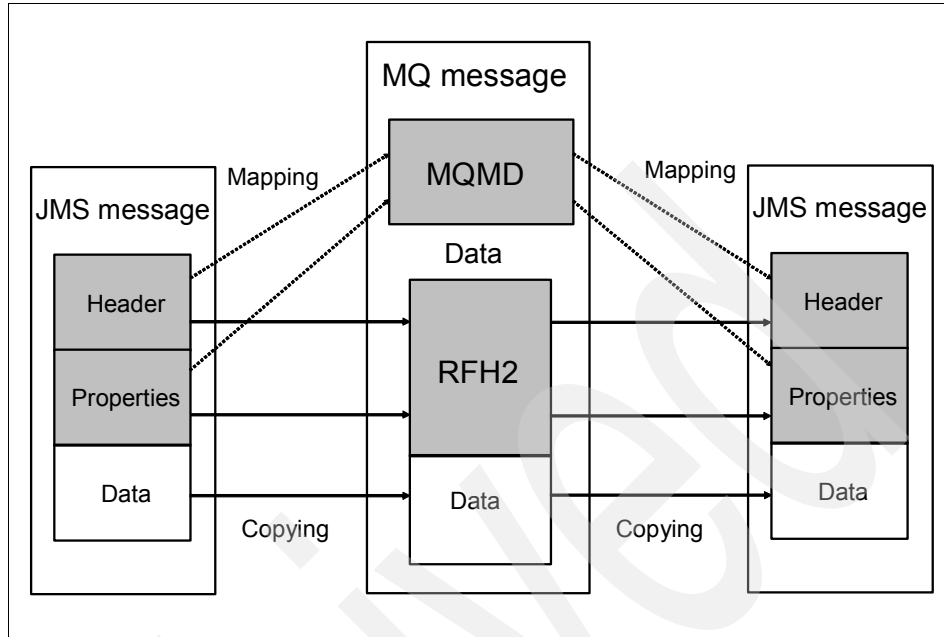


Figure 4-14 JMS to MQ message mapping

Generic messaging

As with WebSphere MQ, to WebSphere Application Server for z/OS Version 6.01, a third-party messaging product is considered an external messaging provider, so naturally WebSphere does not provide any protection to the messaging product's resources. Most likely, the third-party messaging product has implemented its own way of protecting its resources, for example, by using the SAF interface. Refer to the product's documentation to learn how to adequately set up its security configuration.

V5 Default messaging

Like the V5 Default messaging system itself, the protection of EM resources is also integrated in WebSphere Application Server for z/OS Version 6.01.

Security for messaging operates as part of the WebSphere Application Server global security; hence, it is enabled only when global security is enabled. Java 2 Security, however, does not have to be enabled.

All security administration related to the V5 Default messaging system is held in:

```
<embedded messaging server pathname>/config/integral-jms-authorizations.xml
```

For more information about the specifics of how to set up the security definitions in this xml file, refer to the WebSphere Application Server for z/OS Version 6.01 InfoCenter at:

http://publib.boulder.ibm.com/infocenter/wasinfo/v6r0/index.jsp?topic=/com.ibm.websphere.zseries.doc/info/welcome_nd.html

4.4.4 Confidentiality and integrity

The terms confidentiality and integrity refer to the requirement that messages in transit cannot be opened in secret or modified without notice by any unauthorized means. Generally, this is achieved by setting up secure transport network connections. This section touches briefly on the configuration of SSL secure transport connections for the Default messaging and WebSphere MQ JMS providers. V5 Default messaging is, by its embedded nature, not concerned with SSL. For the generic provider, refer to the vendor's documentation on how to set up secure transport connections.

Default messaging

When using the Default messaging system, it is possible to ensure the confidentiality and integrity of messages in transit. To do this you have to configure SSL secure transport connections between clients and Messaging Engines, between Messaging Engines in the same bus, and between buses:

- ▶ For JMS client connections, configure the SSL parameters on the connection factory.
- ▶ For connections between Messaging Engines, set the Inter-engine transport chain property on the bus to use *InboundSecureMessaging*.
- ▶ For connections to foreign buses, set the Target inbound transport chain property on the Service Integration Bus link of the Messaging Engine to use *InboundSecureMessaging*.
- ▶ For connections to WebSphere MQ, set the Transport chain property on the MQ link sender channel to *OutboundSecureMQLink* as to use SSL.

For both Service Integration Bus links and MQ links it is possible to disable any non-secure inbound transport chains to ensure that only secure chains can connect to the Messaging Engines of an application server. To achieve this, select **Servers** → **Application servers** → **<server name>** → **Inbound ME/MQ transports** and disable the *InboundBasicMessaging* and *InboundBasicMQLink* chains.

Note: Unlike WebSphere MQ messaging, Default messaging foreign bus MQ links only use the TCP/IP protocol.

WebSphere MQ messaging

Communication between WebSphere Application Server for z/OS Version 6.01 and WebSphere MQ can be established over TCP/IP, or if the queue manager and application server are on the same LPAR using cross-memory services.

Transport type bindings security is not required since using cross-memory services does not involve a network connection. For transport type client security, configure the SSL parameters on the MQ connection factory.

Attention: The SSL settings on the application server side must match the settings on the queue manager side.

4.5 Using the WebSphere MQ JMS provider and JMS 1.0

In the following sections we explain how we used the TraderMQ application from the *WebSphere for z/OS V5 Connectivity Handbook*, SG24-7064-01, and deployed it to WebSphere Application Server for z/OS Version 6.01.

In 4.5.1, “Preparing the TraderMQ application” on page 114, we describe how you can upgrade the existing V5-compliant TRADERMQ application to Version 6 in your development environment, and in 4.5.2, “Configuring the WebSphere MQ JMS provider” on page 123, we describe the infrastructure part.

Note: Although we recommend that you upgrade your existing Version 5 applications to Version 6 using the Rational Application Developer Version 6 migration wizard, we were able to successfully deploy and start the TraderMQ application, as originally developed for WebSphere Application Server for z/OS Version 5.1, without any migration changes into WebSphere Application Server for z/OS Version 6.01. We only had to re-define the WebSphere MQ JMS queue connection factory, queue destinations, and listener ports.

4.5.1 Preparing the TraderMQ application

In the following sections we describe how we upgraded the existing TraderMQ JMS 1.0-compliant application from the *WebSphere for z/OS V5 Connectivity Handbook*, SG24-7064-01, to be used in WebSphere Application Server for z/OS Version 6.01.

Overview of the Trader application

Trader is an example application that provides different incarnations that show how to use some of the connectivity technologies available in WebSphere

Application Server in a real application. It is a simple application that mimics trading stocks in four different companies.

The Trade application consists of four major components, as shown in Figure 4-15:

- ▶ A back end
- ▶ A data store
- ▶ A middle tier providing access to the back end
- ▶ A front end that is implemented as a Web application

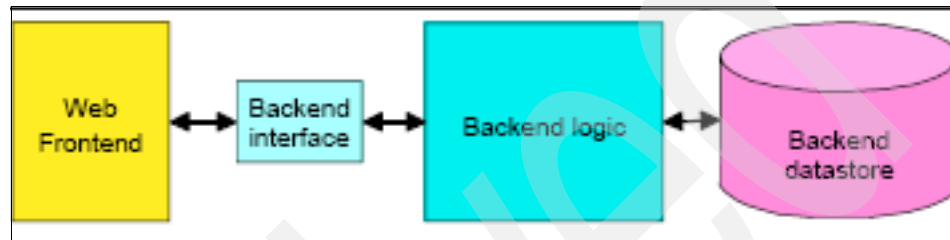


Figure 4-15 High-level architecture of the TRADER application

For more details on the Trader Application, please refer to Chapter 11, “The Trader Application,” in *WebSphere for z/OS V5 Connectivity Handbook*, SG24-7064-01.

Complete the following steps using Rational Application Developer Version 6 to migrate the TraderMQ application from Version 5 to Version 6.

1. Start Rational® Application Developer. Specify the workspace location, as shown in Figure 4-16.

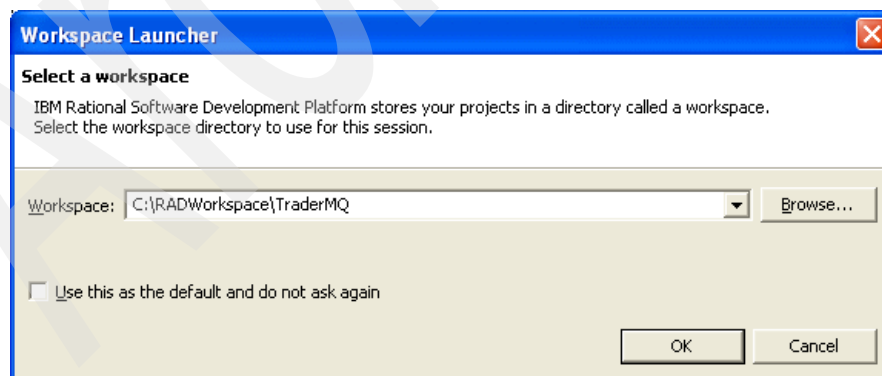


Figure 4-16 Specifying the workspace in RAD

2. An empty workspace appears. Import the TRADERMQ2004 ear file by clicking **File** → **Import**.
3. Select **EAR file** and click **Next**, as shown in Figure 4-17.

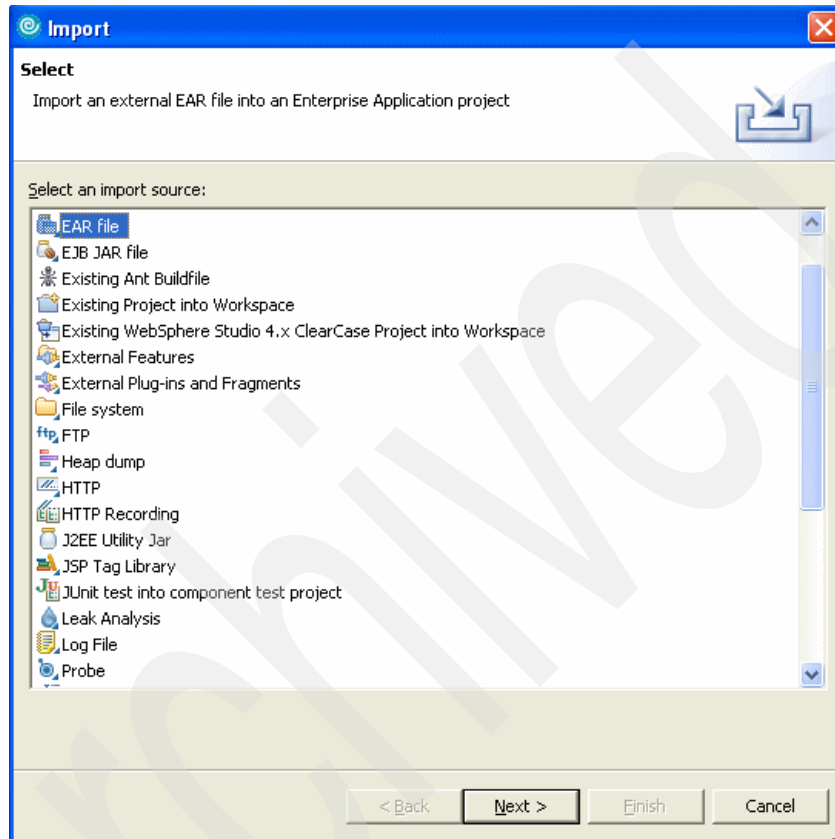


Figure 4-17 Specifying import file format in RAD

4. Specify the location of the ear file. Name the ear project TraderMQ. Click **Next**, as shown in Figure 4-18 on page 117.

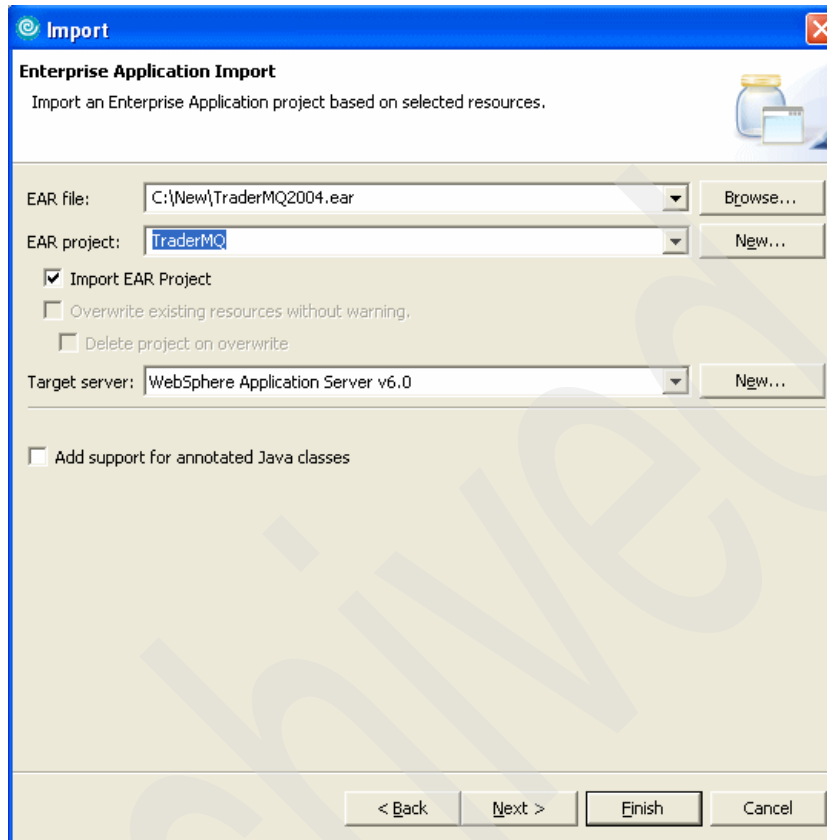


Figure 4-18 Naming the ear project in RAD

5. Select **TraderLib.jar** in the Utility JARs and web libraries section. Click **Next**.

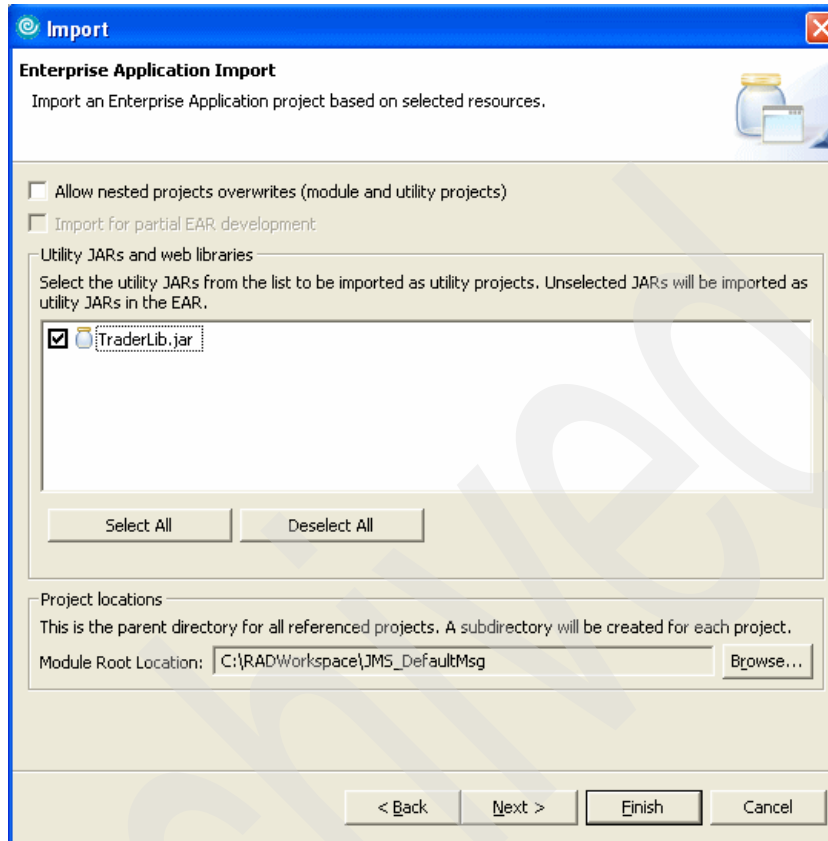


Figure 4-19 Importing the TraderLib utility jar in RAD

6. Select all the modules in the EAR and click **Finish**, as shown in Figure 4-20 on page 119.

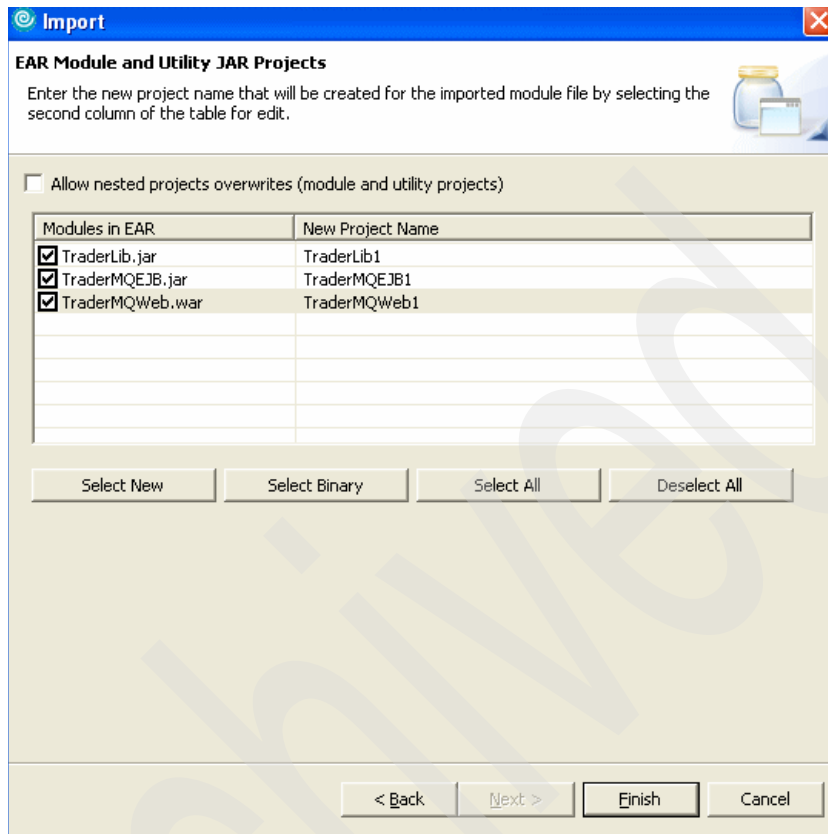


Figure 4-20 Selecting all modules in RAD upon import

7. Migrate the application to J2EE 1.4. In the J2EE perspective, right-click **Enterprise Application TraderMQ**. Select **Migrate** → **J2EE Migration Wizard**. On the first information Screen, click **Next**.
8. Keep the options Migrate project structure, Migrate project J2EE specification level, and Migrate all project modules selected. Click **Next**, as shown in Figure 4-21 on page 120.

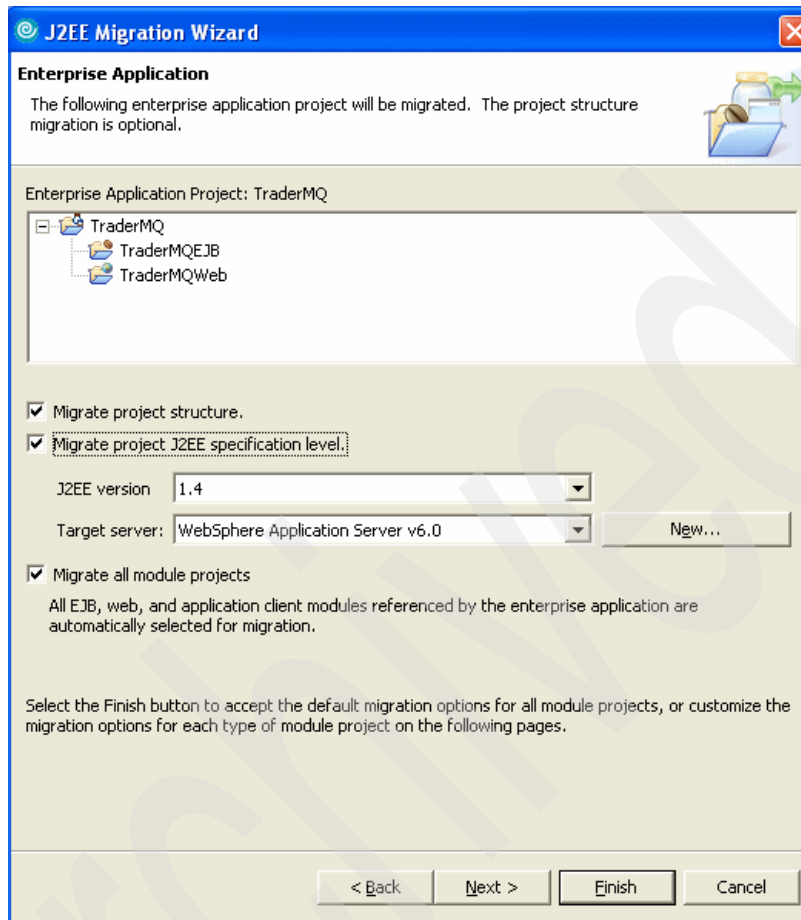


Figure 4-21 J2EE Migration Wizard in RAD

9. Keep the options **Migrate project structure** and **Migrate project J2EE specification level** selected for the EJB module. Click **Next** and you will see the window shown in Figure 4-22 on page 121.

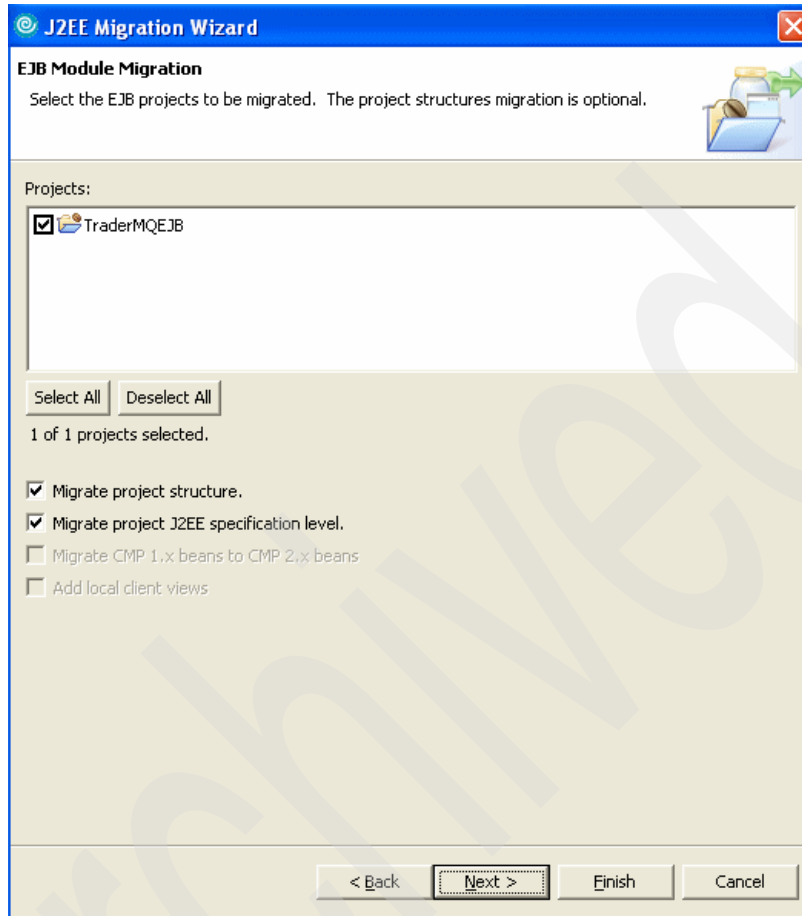


Figure 4-22 EJB Module Migration window in RAD

10. Keep the options Migrate project structure and Migrate project J2EE specification level selected for the Web module. Click **Finish** and you will see the window shown in Figure 4-23 on page 122.

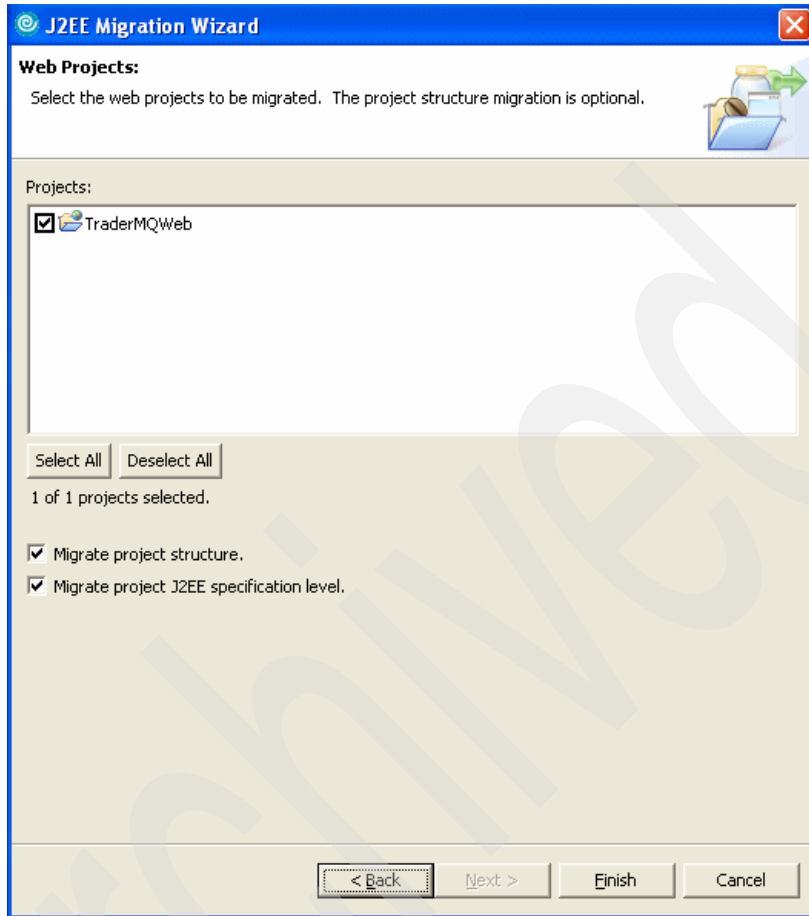


Figure 4-23 Web Projects Migration window in RAD

11. The Information box should display Migration Finished Successfully. Click **OK**.

You now have a version of the TraderMQ application that you can safely deploy to any WebSphere Application Server V6 environment.

Note: You must provide external reference jar files com.ibm.mq.jar and com.ibm.jms.jar as well to your project.

4.5.2 Configuring the WebSphere MQ JMS provider

This chapter describes how to set up the WebSphere MQ JMS queue connection factory, queue destinations, and listener ports required by the TraderMQ application, as originally designed for WebSphere Application Server for z/OS Version 5.1.

Restriction: Although we did define all the WebSphere resources for it, we did not retest the TraderMQ application with IMS.

Accessing WebSphere MQ

Since WebSphere MQ is an external JMS provider there are two node level variables involved that enable WebSphere Application Server for z/OS Version 6.01 to load the required modules and classes. Make sure these variables point to the right directories. In our installation with the latest maintenance on MQ, the two variables needed to be set as indicated in Figure 4-24. Not doing this accurately may result in unexpected failure of sending or receiving messages.

| | | | |
|--------------------------|---------------------------------|------------------------------|---------------------------|
| <input type="checkbox"/> | MQJMS_LIB_ROOT | \${MQ_INSTALL_ROOT}/java/lib | cells:cl6481:nodes:nd6481 |
| <input type="checkbox"/> | MQ_INSTALL_ROOT | /usr/lpp/mqm/V5R3M1 | cells:cl6481:nodes:nd6481 |

Figure 4-24 WebSphere variables pointing to the required MQ directories

Important: Make sure you have at least WebSphere MQ 5.3.1 installed, and check the IBM Web site for prerequisite PTFs. In our system we used CSD level 8.

Next, three of the modules of the WebSphere MQ JMS provider are required to run authorized and program controlled. This is because part of the message listener service runs in the application server control region. The extended attributes can be set using the following commands:

```
extattr +ap /usr/lpp/mqm/V5R3M1/java/lib/libwmqjbatch.so
extattr +ap /usr/lpp/mqm/V5R3M1/java/lib/libwmqjbind.so
extattr +ap /usr/lpp/mqm/V5R3M1/java/lib/libwmqjrrs.so
```

Check the attributes by issuing **ls -lE**.

Also, the following WebSphere MQ libraries need to be added to the STEPLIB DD statements of the control and servant regions, as follows:

```
//STEPLIB DD DISP=SHR,DSN=MQ531.SCSQANLE
//          DD DISP=SHR,DSN=MQ531.SCSQAUTH
```

Configuring a WebSphere MQ queue connection factory

This section describes how to define the WebSphere MQ queue connection factory to be used by the TraderMQ application, as it was originally developed for WebSphere Application Server for z/OS Version 5.1.

Configuration can be done using the WebSphere administrative console. Click **Resources** → **JMS providers** → **WebSphere MQ** and then first select the scope. We recommend defining connection factories and destinations on node level as indicated by Figure 4-25. In that case, applications on all application servers on the node, in particular clusters, can make use of it.

WebSphere MQ messaging provider

A JMS provider enables asynchronous messaging based on the Java Message Service (JMS). It provides J2EE connection factories to create connections for specific JMS queue or topic destinations. WebSphere MQ JMS provider administrative objects are used to manage JMS resources for WebSphere MQ as the JMS provider.

Configuration

☐ Scope: Cell=**cl6481**, Node=**nd6481**

☐ Cell : d6481 Scope specifies the level at which the resource definition is visible. For detailed information on what scope is and how it works, [see the scope settings help](#)

☒ Node : nd6481

☐ Server : ws6481

General Properties

Scope
cells:cl6481:nodes:nd6481

Name
WebSphere MQ JMS Provider

Description
WebSphere MQ Messaging Provider

Class path
\$ {MQJMS_LIB_ROOT}

Native library path
\$ {MQJMS_LIB_ROOT}

Additional Properties

- [WebSphere MQ connection factories](#)
- [WebSphere MQ queue connection factories](#)
- [WebSphere MQ queue destinations](#)
- [WebSphere MQ topic connection factories](#)
- [WebSphere MQ topic destinations](#)

Figure 4-25 Setting the connection factory scope to node level

Attention: After selecting the scope, make sure to click **Apply**, until the red arrow points to the level you intended.

Next, click **WebSphere MQ queue connection factories** → **New**, in order to create a new queue connection factory. For our TraderMQ application the properties had to be configured as indicated by Figure 4-26, Figure 4-27 on page 126, and Figure 4-28 on page 126, which are just scroll-downs of the same panel.

The screenshot shows the 'WebSphere MQ queue connection factories' configuration page for a factory named 'TraderQCF'. The breadcrumb trail is 'WebSphere MQ messaging provider > WebSphere MQ queue connection factories > TraderQCF'. A descriptive paragraph explains that a queue connection factory is used to create connections to the associated JMS provider of JMS queue destinations for point-to-point messaging. The 'Configuration' tab is active, showing two main sections: 'General Properties' and 'Additional Properties'.

General Properties:

- Scope:** cells:cl6481:nodes:nd6481
- Name:** TraderQCF
- JNDI name:** jms/TraderQCF
- Description:** Trader Queue Connection Factory
- Category:** (empty text field)
- Component-managed authentication alias:** (none)
- Container-managed authentication alias:** (none)
- Mapping-configuration alias:** DefaultPrincipalMapping

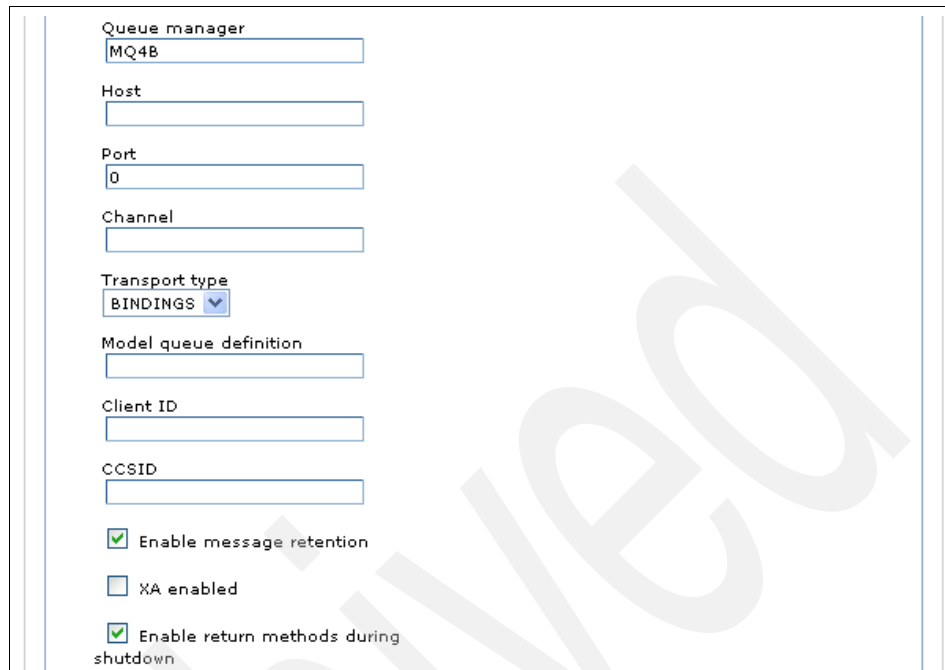
Additional Properties:

- [Custom properties](#)
- [Connection pool](#)
- [Session pools](#)

Related Items:

- [J2EE Connector Architecture \(J2C\) authentication data entries](#)

Figure 4-26 WebSphere MQ queue connection factory properties - 1



Queue manager
MQ4B

Host

Port
0

Channel

Transport type
BINDINGS

Model queue definition

Client ID

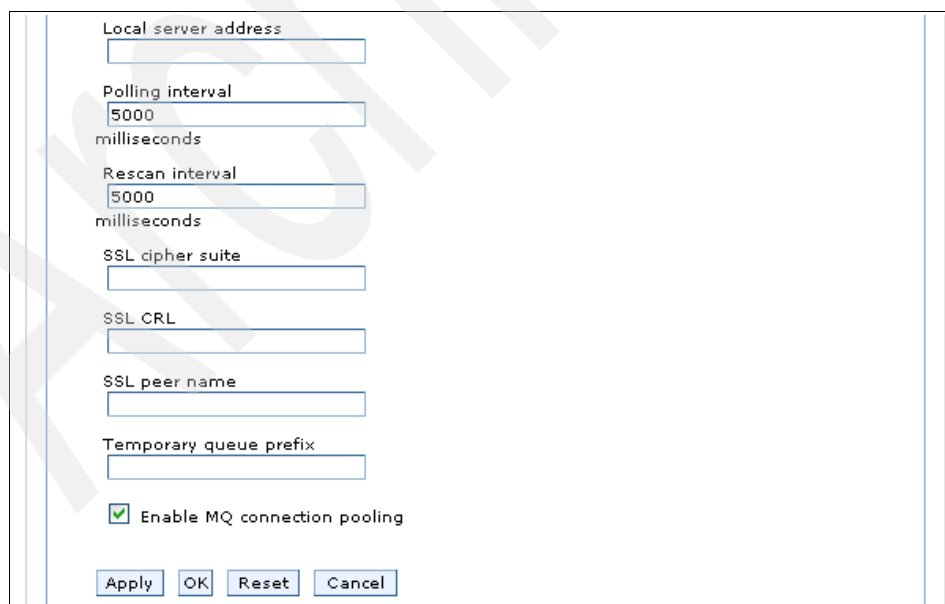
CCSID

☒ Enable message retention

☐ XA enabled

☒ Enable return methods during shutdown

Figure 4-27 WebSphere MQ queue connection factory properties - 2



Local server address

Polling interval
5000
milliseconds

Rescan interval
5000
milliseconds

SSL cipher suite

SSL CRL

SSL peer name

Temporary queue prefix

☒ Enable MQ connection pooling

Apply OK Reset Cancel

Figure 4-28 WebSphere MQ queue connection factory properties - 3

Notice in particular:

- ▶ The convention of the JNDI name being the name prefixed by jms/
- ▶ The absence of component or container managed authentication aliases
- ▶ The specification of the MQ queue manager name
- ▶ The transport type of bindings due to colocation on the same LPAR

Attention: Make sure to save configuration changes and restart the application server (and often also the deployment manager and node agent in a network deployment configuration) for changes to take effect.

Configuring a WebSphere MQ queue destination

This section describes how to define the WebSphere MQ queue destinations to be used by the TraderMQ application, as it was originally developed for WebSphere Application Server for z/OS Version 5.1.

Configuration can be done using the WebSphere administrative console. Although more queue destinations are required for TraderMQ, one definition is presented as an example. For a complete configuration survey, refer to 4.8, “The TraderMQ application environment” on page 166. Click **Resources** → **JMS providers** → **WebSphere MQ** and click **WebSphere MQ queue destinations** → **New**, in order to create a new queue destination. For our TraderMQ application the properties had to be configured as indicated by Figure 4-29 on page 128 and Figure 4-30 on page 129, which are scroll-downs of the same panel.

WebSphere MQ messaging provider

[WebSphere MQ messaging provider](#) > [WebSphere MQ queue destinations](#) > **TraderCICSReqQ**

Queue destinations provided for point-to-point messaging by the WebSphere MQ JMS provider. Use WebSphere MQ queue destination administrative objects to manage queue destinations for the WebSphere MQ JMS provider.

Configuration

General Properties

* Scope

cells:cd6481:nodes:ind6481

* Name

TraderCICSReqQ

* JNDI name

jms/TraderCICSReqQ

Description

TraderMQ Queue Destination

Category

Persistence

QUEUE DEFINED

Priority

QUEUE DEFINED

Specified priority

0

Expiry

UNLIMITED

Additional Properties

Custom properties

MQ Config

Figure 4-29 WebSphere MQ queue destination properties - 1

128 WebSphere for z/OS V6 Connectivity Handbook

* Base queue name
TRADER.CICS.BRIDGEQ

Base queue manager name

CCSID

☐ Use native encoding

Integer encoding
Normal

Decimal encoding
Normal

Floating point encoding
IEEENormal

Target client
MQ

WebSphere MQ Queue Connection Properties

Queue manager host

Queue manager port
0

Server connection channel name

User ID

Password

Apply OK Reset Cancel

Figure 4-30 WebSphere MQ queue destination properties - 2

Notice in particular:

- ▶ The convention of the JNDI name being the name prefixed by jms/
- ▶ The absence of the queue manager name here
- ▶ The specification of the MQ queue name
- ▶ The specification of the target client as MQ

The target client setting can be either JMS or MQ, depending on whether the application is JMS message compliant. If not sure, you can always specify MQ.

To simplify the deployment process, agree on the JNDI names to be used with the application programmer.

Configuring a listener port

This section describes how to define the message listener ports to be used by the TraderMQ application, as it was originally developed for WebSphere Application Server for z/OS Version 5.1.

Configuration can be done using the WebSphere administrative console. Although more listener ports are required for TraderMQ, one definition is presented as an example. Unlike you might expect, the listener ports are not defined on the provider but on the application server. Click **Servers** → **Application servers** → <server name> → **Messaging** → **Message listener service** → **Listener port** → **New** in order to create a new listener port. For our TraderMQ application the properties had to be configured as indicated by the Figure 4-31.

Application servers

[Application servers](#) > [ws6481](#) > [Message Listener Service](#) > [Listener Ports](#) > **TraderMQCICSListener**

Listener ports for Message Driven Beans to listen upon for messages. Each port specifies the JMS Connection Factory and JMS Destination that an MDB, deployed against that port, will listen upon.

Runtime Configuration

General Properties

* Name
TraderMQCICSListener

* Initial State
Started

Description
TraderMQ Listener Port

* Connection factory JNDI name
jms/TraderQCF

* Destination JNDI name
jms/TraderCICSRepQ

Maximum sessions
1

Maximum retries
0

Maximum messages
1

Apply OK Reset Cancel

Figure 4-31 Listener port properties

Notice in particular:

- ▶ The listener port name corresponding to the MDB deployment descriptor
- ▶ The initial state specified as started on start-up of the application server
- ▶ The combination of connection factory and destination the port listens to

Deploying the application

As already mentioned, we were able to successfully deploy and start the TraderMQ application, as originally developed for WebSphere Application Server for z/OS Version 5.1, without any changes into WebSphere Application Server for z/OS Version 6.01, after configuring the correct WebSphere MQ JMS queue connection factory, queue destinations, and listener ports, and referencing them correctly.

However, we did get warnings that the scope of the application resources would not match those of the application server, as illustrated by Figure 4-32.

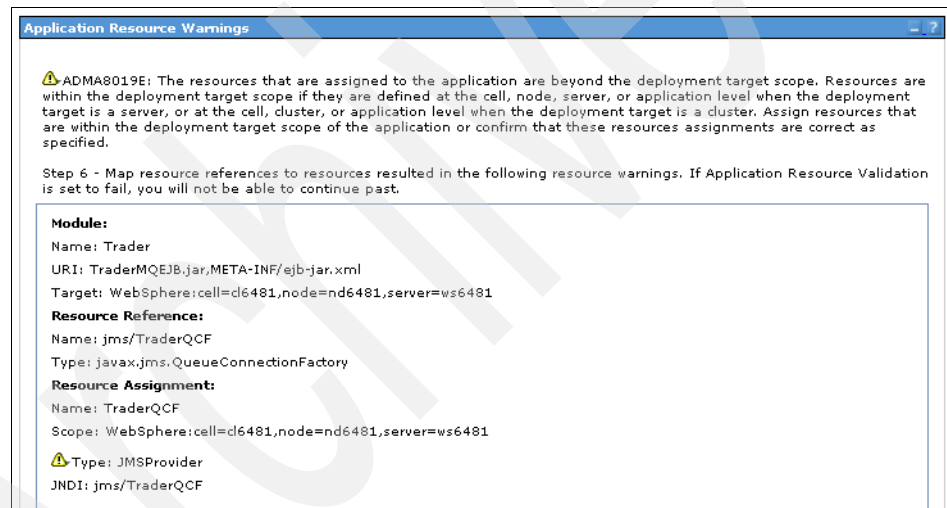


Figure 4-32 Warnings at deployment time

The cause behind this is that the application expects these resources to be on the server level, but we preferred to define them on the node level. Although it is better to avoid this kind of conflict, it is not really harmful.

Figure 4-33 on page 132 shows the deployment summary, after which the application was successfully installed.

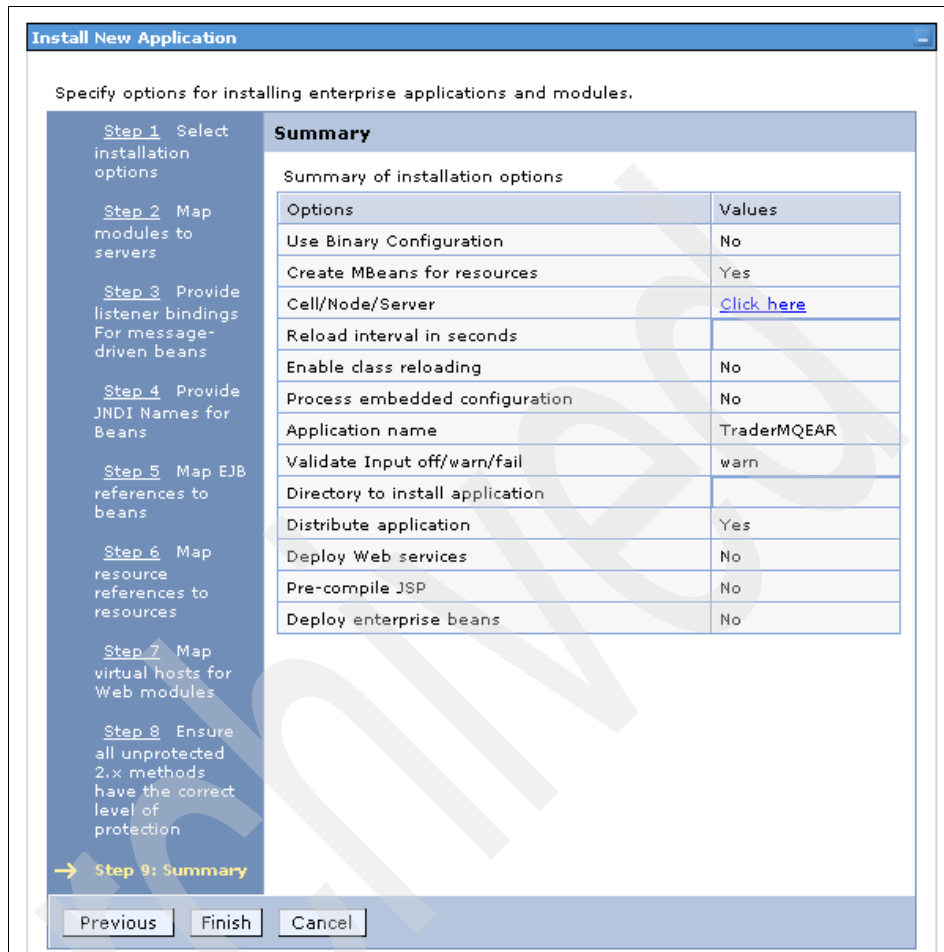


Figure 4-33 Deployment summary

4.5.3 Running TraderMQ on z/OS

To run TraderMQ with the WebSphere MQ JMS provider, enter the following URL:

`http://<server name>:<port number>/TraderMQWeb/Login.html`

You can now select between four options:

- ▶ Using CICS as backend without the use of MDBs
- ▶ Using CICS as backend with the use of MDBs
- ▶ Using IMS as backend without the use of MDBs
- ▶ Using IMS as backend with the use of MDBs

4.6 Using the WebSphere MQ JMS provider and JMS 1.1

In the following sections we explain how we converted the JMS 1.0-compliant TRADERMQ application from the *WebSphere for z/OS V5 Connectivity Handbook*, SG24-7064-01, to a JMS 1.1-complaint application and deployed it to WebSphere Application Server for z/OS Version 6.01 using the new MQ (unified) connection factory.

In 4.6.1, “Converting the TRADERMQ application from JMS 1.0 to JMS 1.1” on page 133, we describe how you can convert a JMS 1.0-compliant application to a JMS 1.1-compliant application, using the existing V5 TRADERMQ application to Version 6 in your development environment; and in 4.5.2, “Configuring the WebSphere MQ JMS provider” on page 123, we describe the infrastructure part.

4.6.1 Converting the TRADERMQ application from JMS 1.0 to JMS 1.1

In the following sections we describe how we upgraded the existing TraderMQ JMS 1.0-compliant application from the *WebSphere for z/OS V5 Connectivity Handbook*, SG24-7064-01, to JMS 1.1.

Overview of JMS 1.1

The JMS 1.1 release addresses unification of the programming interfaces for the Point-to-Point and Pub/Sub messaging domains in the JMS API. In Version 1.0.2b of the JMS specification, there is a strong distinction between the two domains. This domain separation prohibits the use of Point-to-Point domain and the Pub/Sub actions in the same transaction.

In the JMS 1.1 specification, domain unification of the interfaces and methods have been added to support the ability to include Point-to-Point and Pub/Sub messaging in the same transaction. In addition, the domain unification proposed simplifies the client programming model, so that the client programmer can use a simplified set of APIs to create an application.

Table 4-6 shows a comparison between the JMS 1.0.2b objects and the corresponding objects for the JMS 1.1 specification for the Point-to-Point model. Table 4-6 also includes the object comparison for the pub/sub model.

Table 4-6 Object comparison between JMS 1.0 and JMS 1.1

| JMS 1.02b | JMS 1.1 |
|---|-------------------|
| QueueConnectionFactory, TopicConnectionFactory | ConnectionFactory |
| Queue, Topic | Destination |

| JMS 1.02b | JMS 1.1 |
|----------------------------------|-----------------|
| QueueConnection, TopicConnection | Connection |
| QueueSession, TopicSession | Session |
| QueueSender, TopicPublisher | MessageProducer |
| QueueReceiver, TopicSubscriber | MessageConsumer |

Code alterations in the Trader application to support JMS 1.1

The code in the TRADERMQ had to be altered in order to use the unified JMS 1.1 model. The code snippets below describe the required changes.

The JNDI names used in the code should match the JNDI names of the administered objects defined. Refer to “Deployment Descriptor alterations to support JMS 1.1” on page 139.

The code snippet (from TraderMQCICSBean) in Example 4-3 shows the use of the JMS 1.0.2b specification. The QueueConnectionFactory, Queue, QueueConnection, QueueSession, QueueSender, and QueueReceiver objects are used here.

Example 4-3 TraderMQCICSBean using JMS 1.02b code

```
/** <Begin> JMS 1.02 code commented out **/  
/*  
    private QueueConnectionFactory qConnectionFactory;  
    private Queue reqCICSQueue, repCICSQueue, repCICSQueue_UseMDB;  
    private QueueConnection qConnection;  
    private QueueSession reqSession;  
    private QueueSender sender;  
    private QueueReceiver receiver;  
*/  
/** <End> JMS 1.02 code commented out **/  
*/
```

Example 4-4 shows a code snippet (from TraderMQCICSBean) using the JMS 1.1 specification with unified model. The ConnectionFactory, Destination, Connection, Session, MessageProducer, and MessageConsumer objects are used now.

Example 4-4 TraderMQCICSBean using JMS 1.1 unified model

```
/** <Begin> JMS 1.1 Specification code **/  
    private ConnectionFactory connectionFactory;  
    private Destination reqCICSQueue, repCICSQueue, repCICSQueue_UseMDB;  
    private Connection connection;  
    private Session reqSession;
```

```

private MessageProducer messageProducer;
private MessageConsumer messageConsumer;
/*** <End>   JMS 1.1 Specification code ***/

```

In Example 4-5 through Example 4-8 on page 137 we show various sections of the TraderMQCICSBean session Bean of the TraderMQ application using the objects described Example 4-4 on page 134. We highlighted the important lines of code.

Example 4-5 sendTraderMQCICS method in TraderMQCICSBean using JMS 1.1

```

private String sendTraderMQCICS(StringBuffer message) throws Exception {

    try {
        // make connection to Queue
        connectToQ();
        // send msg to the request TraderCICS request Q
        messageProducer = reqSession.createProducer(reqCICSQueue);
        tmessage = reqSession.createTextMessage();

        // build sending message for CICS
        StringBuffer messageToSend =
            new StringBuffer().append("TRADERBL").append(message);

        // send message
        tmessage.setJMSCorrelationIDAsBytes(MQC.MQCI_NEW_SESSION);
        tmessage.setText(messageToSend.toString());
        tmessage.setStringProperty("JMS_IBM_Format", MQC.MQFMT_STRING);
        tmessage.setJMSReplyTo(repCICSQueue);
        messageProducer.send(tmessage);
        messageID = tmessage.getJMSMessageID();

    } finally {
        // close connections
        messageProducer.close();
        disconnectFromQ();
    }

    return messageID;
}

```

Example 4-6 shows the new JMS 1.1 object being used in the receiveTraderMQCICS method of the TraderMQCICSBean, highlighting the important lines of code.

Example 4-6 receiveTraderMQCICS method in TraderMQCICSBean using JMS 1.1

```

/**

```

```

* receiveTraderMQCICS: receive message via JMS
*
* @param corrID correlation ID of JMS
* @param waitTime maximum time for wait till the message arrives
*   (if 0 is specified, waits until message arrives)
* @return message received message
*/
private String receiveTraderMQCICS(String corrID, long waitTime)
    throws Exception {
    String getMessageStr = null;
    try {
        // make connection to Queue
        connectToQ();

        // try to get message
        if (getUseMDB().equals("true")) {

            messageConsumer =
                reqSession.createConsumer(
                    repCICSqueue_UseMDB,
                    "JMSCorrelationID='" + corrID.toUpperCase() + "'");

        } else {

            messageConsumer =
                reqSession.createConsumer(
                    repCICSqueue,
                    "JMSCorrelationID='" + corrID.toUpperCase() + "'");

        }
        Message message;
        if (getUseMDB().equals("true")) {
            message = messageConsumer.receive();
        } else {
            message = messageConsumer.receive(waitTime);
        }

        if (message != null)
            getMessageStr = ((javax.jms.TextMessage) message).getText();

    } finally {
        // disconnect from Q
        messageConsumer.close();
        disconnectFromQ();
    }

    return getMessageStr;
}

```

```
}
```

Example 4-7 shows the new JMS 1.1 objects being used in the `connectToQ` method of the `TraderMQCICSBean`, highlighting the important lines of code.

Example 4-7 connectToQ method in TraderMQCICSBean using JMS 1.1

```
/**
 * connectToQ: make a connection to JMS
 *
 * @param null
 * @return null
 */
public void connectToQ() throws Exception {
    InitialContext context = new InitialContext();
    connectionFactory =
        (ConnectionFactory) context.lookup("java:comp/env/jms/TraderCF");
    reqCICSQueue =
        (Destination) context.lookup("java:comp/env/jms/TraderCICSReqQ");
    repCICSQueue =
        (Destination) context.lookup("java:comp/env/jms/TraderCICSRepQ");
    repCICSQueue_UseMDB =
        (Destination) context.lookup("java:comp/env/jms/TraderProcessQ");

    connection = connectionFactory.createConnection();
    connection.start();
    reqSession =
        connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
}
```

Example 4-8 shows the `disconnectFromQ` method in `TraderMQCICSBean` using the new JMS 1.1 objects, highlighting the important lines of code.

Example 4-8 disconnectFromQ method in TraderMQCICSBean using JMS 1.1

```
/**
 * disconnectFromQ: disconnect from JMS
 *
 * @param null
 * @return null
 */
public void disconnectFromQ() throws Exception {
    // close connections
    reqSession.close();
    connection.close();
}
```

Changes in the Trader MDB

The Message Driven Bean (TraderMQCICS_MDB) of the TraderMQ application also required a few changes to reflect the new JMS 1.1 unified model.

Example 4-9 shows the old MDB using the JMS 1.02b specification and Example 4-10 shows the new MDB using the JMS 1.1 objects, highlighting the important lines of code.

Example 4-9 TraderMQCICS_MDBBean using the JMS 1.02b specification

```
// transfer message to another "Process Queue"
QueueConnectionFactory qConnectionFactory =
    (QueueConnectionFactory) context.lookup(
        "java:comp/env/jms/TraderQCF");
Queue processCICSQueue =
    (Queue) context.lookup("java:comp/env/jms/TraderProcessQ");
QueueConnection qConnection =
    qConnectionFactory.createQueueConnection();

qConnection.start();
QueueSession transferSession =
    qConnection.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);

// designate TraderProcessQ as destination
QueueSender sender = transferSession.createSender(processCICSQueue);

// send message as same as original to the process queue. (corrID is
// passed as-is)
sender.send(msg);

// disconnect from Q
sender.close();
transferSession.close();
qConnection.close();
```

Example 4-10 TraderMQCICS_MDBBean using the JMS 1.1 specification

```
// transfer message to another "Process Queue"

// Use of unified connection factory
ConnectionFactory connectionFactory = (ConnectionFactory)
    context.lookup("java:comp/env/jms/TraderCF");

// Create the Destination Object
Destination processCICSQueue = (Destination)
    context.lookup("java:comp/env/jms/TraderProcessQ");

Connection connection = connectionFactory.createConnection();
connection.start();
```



```

Session transferSession = connection.createSession(false,
    Session.AUTO_ACKNOWLEDGE);

// designate TraderProcessQ as destination
MessageProducer messageProducer =
    transferSession.createProducer(processCICSQueue);
messageProducer.send(msg);
messageProducer.close();
transferSession.close();
connection.close();

```

Deployment Descriptor alterations to support JMS 1.1

Resource references for the used administered objects (ConnectionFactory and Destination) need to be added in the Deployment Descriptors of both the Session EJBs and Message Driven Beans.

This can be done as follows:

1. Switch to the J2EE perspective. In the Project Explorer, double-click the EJB or MDB to be changed, as shown in Figure 4-34.

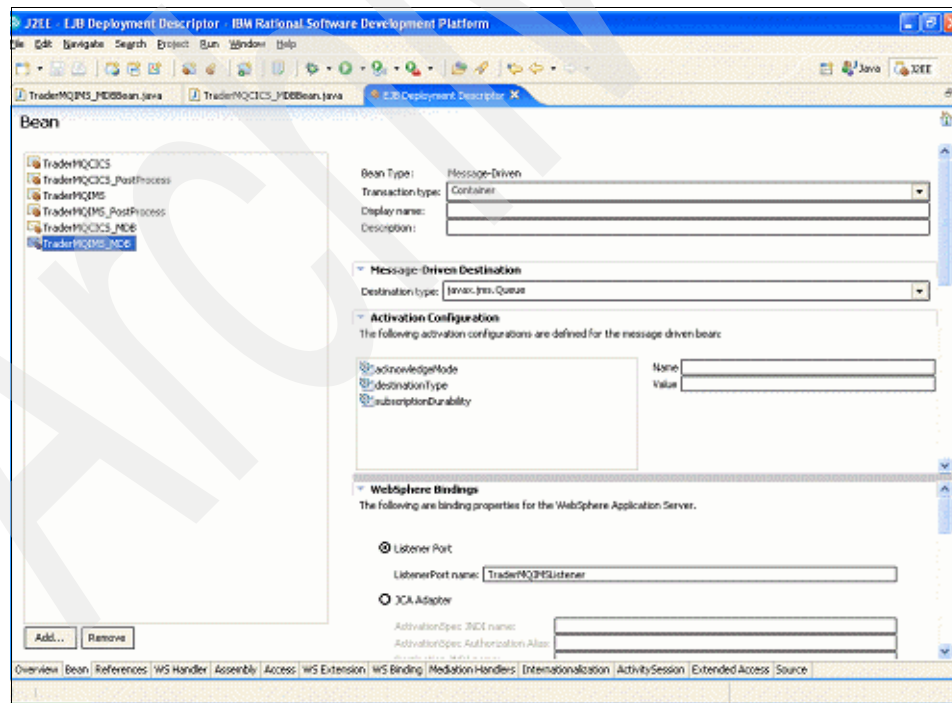


Figure 4-34 Opening the Deployment Descriptor of an EJB or MDB

2. In the Deployment Descriptor Editor go to the References tab. Select the EJB or MDB you want to add the resources for. Click **Add**. This is shown in Figure 4-35.

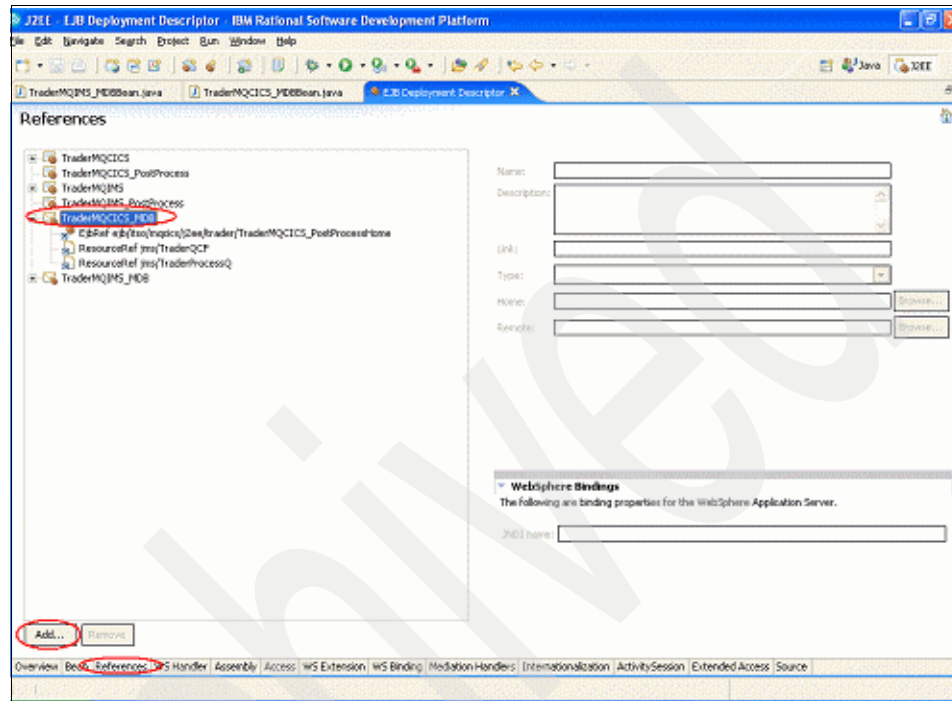


Figure 4-35 References tab in Deployment descriptor of EJB or MDB

3. Select Resource References, as shown in Figure 4-36 on page 141. Click **Next**.

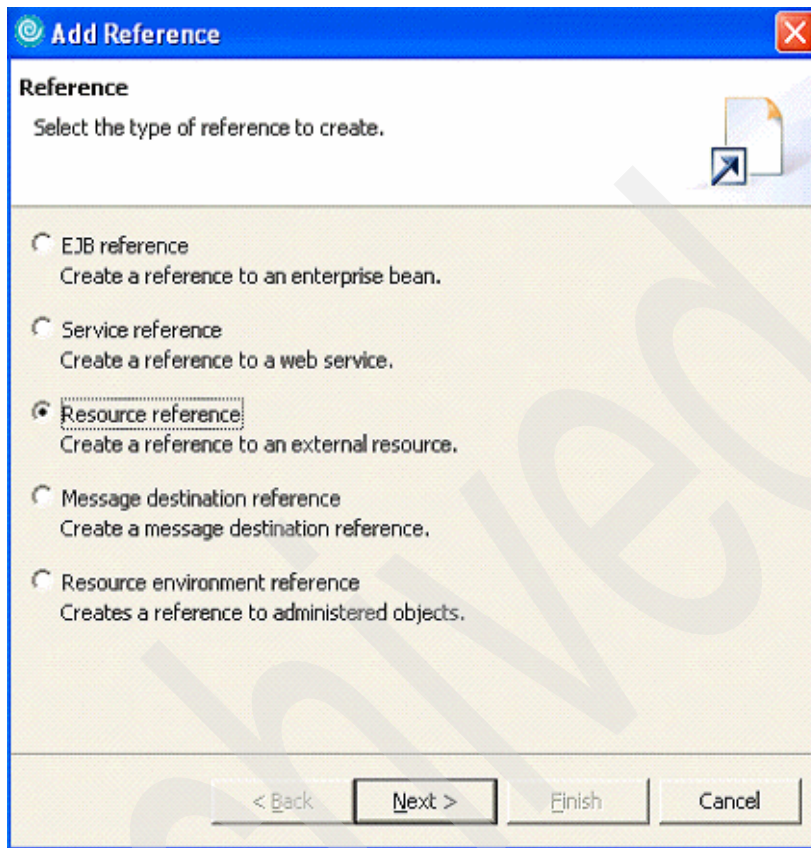
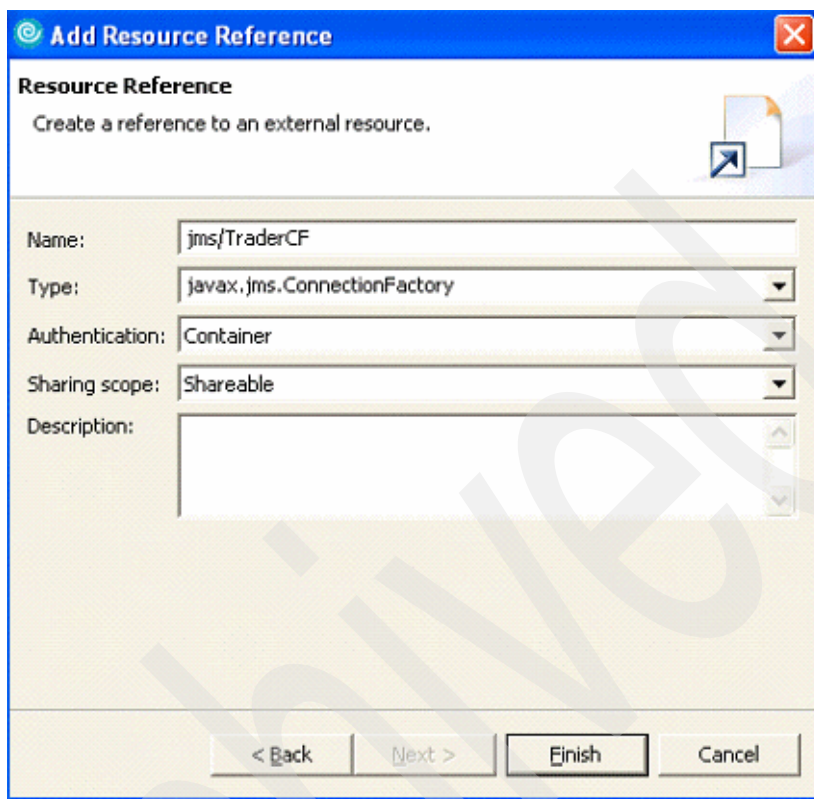


Figure 4-36 Adding a Resource reference to an EJB or MDB

4. Enter the JNDI name in the Name field. Select the required type and authentication. Click **Finish**. This is shown in Figure 4-37 on page 142.



Add Resource Reference

Resource Reference
Create a reference to an external resource.

Name:

Type:

Authentication:

Sharing scope:

Description:

< Back Next > Finish Cancel

Figure 4-37 Specifying the JNDI name of a resource reference

5. In the WebSphere Bindings section of the References editor, fill in the JNDI name of the resource reference just added.

Follow steps 1 through 5 to create the resource references for all EJBs and MDBs in the Trader application.

Listener port

When using the WebSphere MQ JMS provider you must specify the Listener port on the MDB as follows.

Switch to the J2EE perspective. In the Project Explorer, double-click the required EJB. Click the **Bean** tab and go to WebSphere Bindings. Click the **Listener Port** radio button and specify the name of the Listener Port, as shown in Figure 4-38 on page 143.

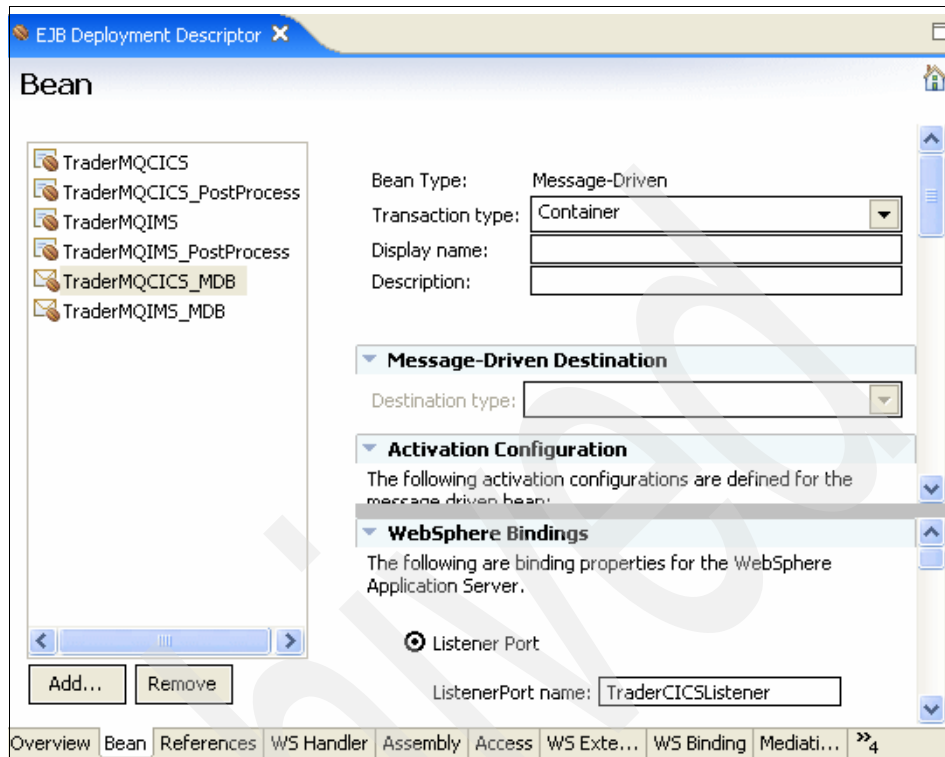


Figure 4-38 Specifying the Listener port name

4.6.2 Configuring an MQ (unified) connection factory

After being able to successfully deploy the V5 TraderMQ application, we intended to migrate the application to make use of the point-to-point or publish/subscribe domain independent API set, as prescribed by the JMS 1.1 specification explained in “JMS 1.1 specification” on page 88, and define a corresponding WebSphere MQ connection factory.

First, we decided to configure the WebSphere MQ connection factory. Configuration can be done using the administrative console. Click **Resources** → **JMS providers** → **WebSphere MQ** → **WebSphere MQ connection factories** → **New**, in order to create a new JMS 1.1 unified connection factory. For our migrated TraderMQ application the properties had to be configured as indicated by Figure 4-39 on page 144, Figure 4-40 on page 145, Figure 4-41 on page 146, and Figure 4-42 on page 147, which are scroll-downs of the same panel.

WebSphere MQ messaging provider

WebSphere MQ messaging provider > WebSphere MQ connection factories > TraderCF

A unified JMS connection factory can be used to create JMS connections to both queue and topic destinations.

Configuration

General Properties

*

Scope

cells:d6481:nodes:nd6481

*

Name

TraderCF

*

JNDI name

jms/TraderCF

Description

TraderMQ Connection Factory

Category

Component-managed authentication alias

(none)

Container-managed authentication alias

(none)

Mapping-configuration alias

DefaultPrincipalMapping

Additional Properties

Custom properties

Connection pool

Session pools

Related Items

J2EE Connector Architecture (J2C) authentication data entries

Figure 4-39 WebSphere MQ connection factory properties - 1

Queue manager
MQ4B

Host
wtsc48.itso.ibm.com

Port
1414

Channel
TRADER.CHANNEL

Transport type
CLIENT

Model queue definition

Client ID

CCSID

☒ Enable message retention

☐ XA enabled

☒ Enable return methods during shutdown

Figure 4-40 WebSphere MQ connection factory properties - 2

Local server address

Polling interval

milliseconds

Rescan interval

milliseconds

SSL cipher suite

SSL CRL

SSL peer name

Temporary queue prefix

☒ Enable MQ connection pooling

Broker control queue

Broker queue manager

Broker publication queue

Broker subscription queue

Broker CC subscription queue

Figure 4-41 WebSphere MQ connection factory properties - 3

Broker version
Advanced ▾

Publish and subscribe cleanup level
SAFE ▾

Publish and subscribe cleanup interval
3600000
milliseconds

Broker message selection
BROKER ▾

Publish and acknowledgement interval
25

☐ Enable sparse broker subscriptions

Publish and subscribe status interval
60000
milliseconds

Persistent subscriptions store
MIGRATE ▾

Enable multicast transport
DISABLED ▾

☐ Enable clone support

Direct Broker authorization type
BASIC ▾

Proxy host name

Proxy port

Apply OK Reset Cancel

Figure 4-42 WebSphere MQ connection factory properties - 4

Notice in particular:

- ▶ The convention of the JNDI name being the name prefixed by jms/
- ▶ The absence of component or container managed authentication aliases
- ▶ The specification of the MQ queue manager name
- ▶ The fact that queue and topic related properties are on this same panel

For the client connection channel an equally named server connection channel was defined on the queue manager.

Restriction: At the moment of writing this book we were not able to correctly configure the MQ connection factory with a transport type specified as bindings. We received errors during start-up of the application server indicating “Error encountered binding the J2EE resource, invalid configuration passed to resource binding logic,” but were not able to solve this and decided to submit a PMR.

Note: The JMS 1.1 prepared application can use the same destinations as the previous JMS 1.0 application, but requires a different type of connection factory. When using a different JNDI name for this connection factory, remember to change the listener ports accordingly.

4.7 Using the Default messaging JMS provider

After being able to successfully migrate the TraderMQ application to use the WebSphere MQ (unified) connection factory, we intended to prepare the application to make use of the Default messaging system.

In 4.7.1, “Application changes for using Default messaging” on page 148, we show how the TRADERMQ application was changed in order to use the activation specification and in 4.7.2, “Configuring the Default messaging JMS provider” on page 149, we explain how we set up our infrastructure to use the Default messaging.

Attention: Remember that the application does not need to be migrated to make use of Default messaging. In this case, it may be convenient though to use different JNDI names though.

4.7.1 Application changes for using Default messaging

If you wish to perform the application changes as described in the following sections yourself, you can start out with TRADERMQ application using the JMS 1.1 APIs. You can find this application in the additional material for this book.

For WebSphere Default Messaging JMS provider specify the Activation Specification on the MDB as follows.

Switch to the J2EE perspective. In the Project Explorer, double-click the required EJB. On the Bean tab, select the required MDB. In the WebSphere Bindings section, select **JCA Adapter** and fill in the ActivationSpec JNDI name and the Destination JNDI name. This is shown in Figure 4-43 on page 149.

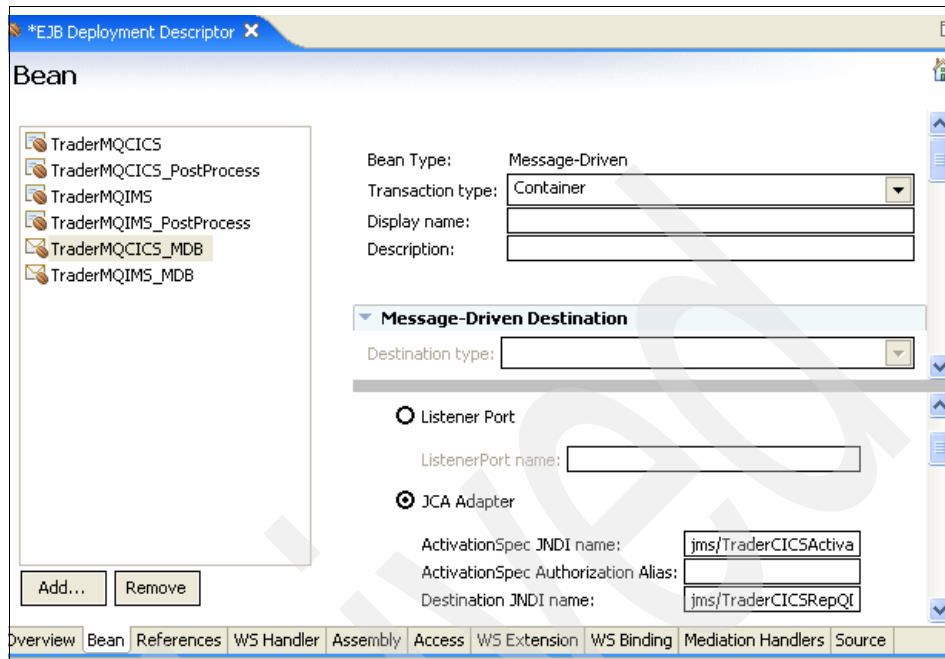


Figure 4-43 Specifying the activation specification for an MDB

4.7.2 Configuring the Default messaging JMS provider

To prevent any changes to the functionality of the application, it is necessary to invoke the CICS program over the MQ-CICS adapter and bridge. To achieve this we set up a local bus and also a foreign bus representing the queue manager connected via an MQ link. Remote queue definitions on the queue manager and activation specifications on the application server are used to trigger the MDB on the response message. The design of the infrastructure used for the TraderMQ application in this case is illustrated by Figure 4-44 on page 150.

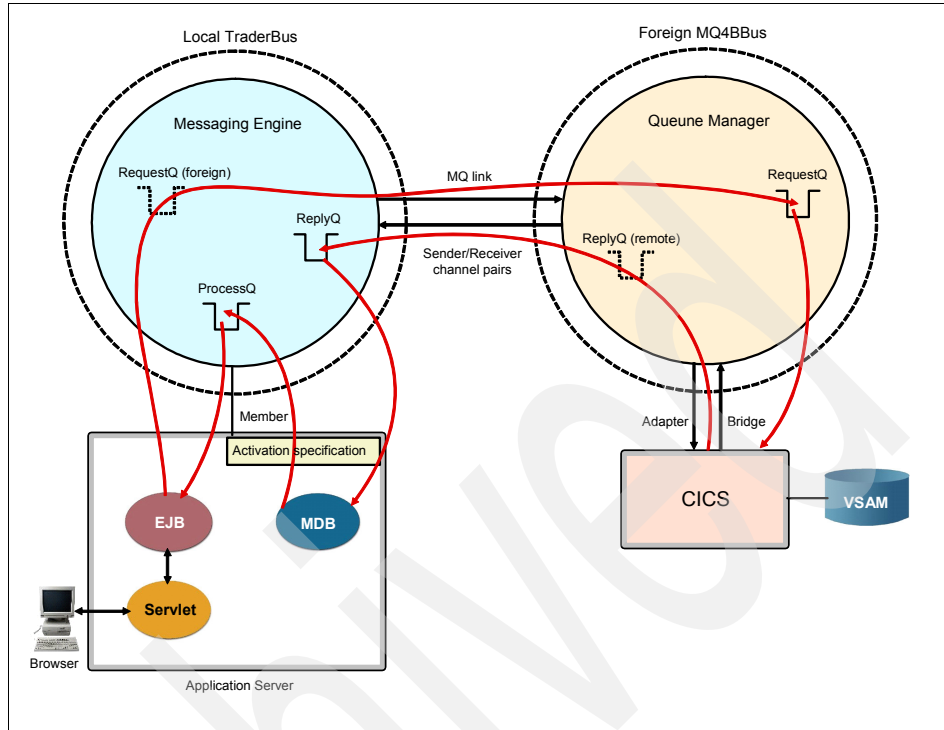


Figure 4-44 Bus infrastructure design of TraderMQ

Bus infrastructure

First, we decided to set up the WebSphere Application Server for z/OS Version 6.01, Default messaging infrastructure. Configuration can be done using the WebSphere Administrative console.

Before a Default messaging JMS connection factory can effectively be defined, a local Service Integration Bus must be created. Click **Service integration** → **Buses** → **New**, in order to create a new local bus. For our TraderMQ application the bus properties were configured as indicated in Figure 4-45 on page 151.

Buses > **TraderBus**

A service integration bus supports applications using message-based and service-oriented architectures. A bus is a group of interconnected servers and clusters that have been added as members of the bus. Applications connect to a bus at one of the messaging engines associated with its bus members.

Configuration

General Properties

Name
TraderBus

UUID
A4CC32397103DB6B

Description

Additional Properties

- Bus members
- Messaging engines
- Destinations
- Mediations
- Foreign buses
- Custom properties
- Inbound Services
- Outbound Services

Security

☐ Secure

Inter-engine authentication alias
(none)

Mediations authentication alias
(none)

Inter-engine transport chain

☐ Discard messages

☒ Configuration reload enabled

Related Items

- J2EE Connector Architecture (J2C) authentication data entries

Figure 4-45 Service integration bus properties

Notice in particular:

- ▶ The specification of the local service integration bus name
- ▶ The fact that we disabled bus security in our scenario
- ▶ The absence of inter-engine or mediations authentication aliases

Then the application server needs to be added as a member of the newly created bus. To achieve this, click **Service integration** → **Buses** → **<bus name>** → **Bus members** → **Add**, and select the server, as illustrated by Figure 4-46 on page 152.

Add a new bus member

Add a server or server cluster as a new member of the bus.

→ **Step 1: Select server or cluster**
Step 2: Confirm the addition of a new bus member

Select server or cluster
Choose the server or cluster to add to the bus

☒ **Server**
Server
nd6481:ws6481
Data store
☒ Default
Data source JNDI name

☐ **Cluster**
Cluster
(none)
Data store
* Data source JNDI name

Next Cancel

Figure 4-46 Adding the application server to be a bus member

Note: We use the default data store, meaning that the embedded Cloudscape database is used for the Messaging Engine.

As explained in “Messaging Engine” on page 92, a Messaging Engine is automatically created as soon as an application server is added to the bus. To display it, click **Service integration** → **Buses** → <bus name> → **Messaging Engines**, as illustrated in Figure 4-47 on page 153.



Figure 4-47 Displaying the automatically created Messaging Engine

Next, since the TraderMQ application in the end accesses a CICS (and IMS) application via point-to-point JMS messages through MQ, a foreign bus needs to be created that represents the MQ queue manager and to which the local TraderBus will be connected using a WebSphere MQ link.

Click **Service integration** → **Buses** → **<bus name>** → **Foreign buses** → **New** in order to create a foreign bus over a WebSphere MQ link, as indicated by Figure 4-48 on page 154.

Buses

[Buses](#) > [TraderBus](#) > [Foreign buses](#) > **MQ4B**

A foreign bus is another bus with which this bus can exchange messages.

Configuration

| General Properties | Additional Properties |
|---|--|
| <p>Name</p> <p>MQ4B</p> | <ul style="list-style-type: none"> Destination defaults WebSphere MQ link routing properties |
| <p>UUID</p> <p>45793E8B5CDE2547B4ACBDAF</p> | |
| <p>Description</p> <p></p> | |
| <p><input checked="" type="checkbox"/> Send allowed</p> | |
| <p>Apply OK Reset Cancel</p> | |

Figure 4-48 Foreign bus properties

Note: Usually, when creating a foreign bus over an MQ link, we recommend specifying the MQ queue manager name as the bus name.

Now, the WebSphere MQ link can be configured on the Messaging Engine. Click **Service integration** → **Buses** → <bus name> → **Messaging Engines** → <engine name> → **WebSphere MQ links (under Additional Properties)** → **New**, as indicated by Figure 4-49 on page 155.

Note: Sometimes more than one path exists to accomplish a certain task. In this case, for example, it is also possible to click **Servers** → **Application servers** → <server name> → **Messaging Engines** → <engine name> → **WebSphere MQ links** → **New** instead.

Buses

[Buses](#) > [TraderBus](#) > [TraderMQlink](#)

A link between the messaging engine and a WebSphere MQ network. The WebSphere MQ link connects engine as a queue manager to WebSphere MQ, thereby providing a bridge between the bus and a WebSphere MQ network.

Runtime **Configuration**

General Properties

Administration

* Name
TraderMQlink

UUID
25EB51913D584092

Description

Connection

* Foreign bus name
MQ4B

* Queue manager name
TRADER

Additional Properties

- [Publish/subscribe broker profile](#)
- [Receiver channel](#)
- [Sender channel](#)

Figure 4-49 WebSphere MQ link properties

Note: On this panel the queue manager name refers to the virtual queue manager name that the local bus is to look like from the external queue manager's perspective.

For the WebSphere MQ link to be really established, a pair of MQ style receiver and sender channels are required when the link is created. Afterwards, click **WebSphere MQ links** → **<link name>** → **Receiver channel (under Additional Properties)** to operate the receiver channel or change its configuration, as indicated by Figure 4-50 on page 156.



Figure 4-50 MQ link receiver channel status

Click **WebSphere MQ links** → **<link name>** → **Sender channel (under Additional Properties)** to operate the sender channel or change its configuration, as indicated by Figure 4-51.



Figure 4-51 MQ link sender channel status

Note: The default InboundBasicMQLink port is 5558; the default InboundSecureMQLink port is 5578. Click **Servers** → **Application servers** → **<server name>** → **Ports (under Communications)** → **SIB_MQ_ENDPOINT_ADDRESS** to verify or change it.

To complement the WebSphere MQ link channels, a corresponding pair of channels has to be defined on the MQ queue manager, as shown in Example 4-11.

Example 4-11 Corresponding queue manager channel definitions

```
DEFINE QLOCAL('Trader') USAGE(XMITQ)

DEFINE CHANNEL('Trader2MQ4B') CHLTYPE(RCVR) TRPTYPE(TCP)

DEFINE CHANNEL('MQ4B2Trader') CHLTYPE(SDR) TRPTYPE(TCP) +
      CONNAME('9.12.4.38(5558)') XMITQ('Trader')
```

Note: The name of the link sender channel must match the name of the queue manager receiver channel, and the name of the link receiver channel must match the name of the queue manager sender channel.

Attention: When using mixed case queue manager resource names, put them inside quotes.

After completing the definitions, start the sender channels on both sides. Check if all channels have the correct status.

JMS connection factory

Now we have arrived at the point where we can create a new Default messaging (unified) connection factory. Click **Resources** → **JMS providers** → **Default messaging** → **JMS connection factories** → **New**. For our migrated TraderMQ application the properties had to be configured as indicated by Figure 4-52 on page 158, Figure 4-53 on page 159, and Figure 4-54 on page 160, which are scroll-downs of the same panel.

Default messaging provider

Default messaging provider > JMS connection factory > TraderDCF

A JMS connection factory is used to create connections to the associated JMS provider of JMS destinations, for both point-to-point and publish/subscribe messaging. Use connection factory administrative objects to manage JMS connection factories for the default messaging provider.

Configuration

General Properties

Administration

* Scope
cells:cd6481:nodes:nd6481

* Name
TraderDCF

* JNDI name
jms/TraderDCF

Description

Category

Connection

* Bus name
TraderBus

Additional Properties

- Connection pool properties

Related Items

- J2EE Connector Architecture (J2C) authentication data entries
- Buses

Figure 4-52 Default messaging JMS connection factory properties - 1

The image shows a screenshot of a JMS connection factory properties dialog box. The dialog is divided into three main sections: Target, Durable Subscription, and Quality of Service. The Target section contains fields for Target, Target type (set to 'Bus member name'), Target significance (set to 'Preferred'), Target inbound transport chain, and Provider endpoints (a list box). The Durable Subscription section contains fields for Client identifier and Durable subscription home. The Quality of Service section contains two sections: Nonpersistent message reliability (set to 'Express nonpersistent') and Persistent message reliability (set to 'Reliable persistent').

Target

Target type
Bus member name

Target significance
Preferred

Target inbound transport chain

Provider endpoints

Connection proximity
Bus

Durable Subscription

Client identifier

Durable subscription home

Quality of Service

Nonpersistent message reliability
Express nonpersistent

Persistent message reliability
Reliable persistent

Figure 4-53 Default messaging JMS connection factory properties - 2

Advanced Messaging

Read ahead
Default

Temporary queue name prefix

Temporary topic name prefix

Share durable subscriptions
In cluster

Advanced Administrative

Component-managed authentication alias
(none)

☐ Log missing transaction contexts

☐ Manage cached handles

☐ Share data source with CMP

XA recovery authentication alias
(none)

Apply OK Reset Cancel

Figure 4-54 Default messaging JMS connection factory properties - 3

Notice in particular:

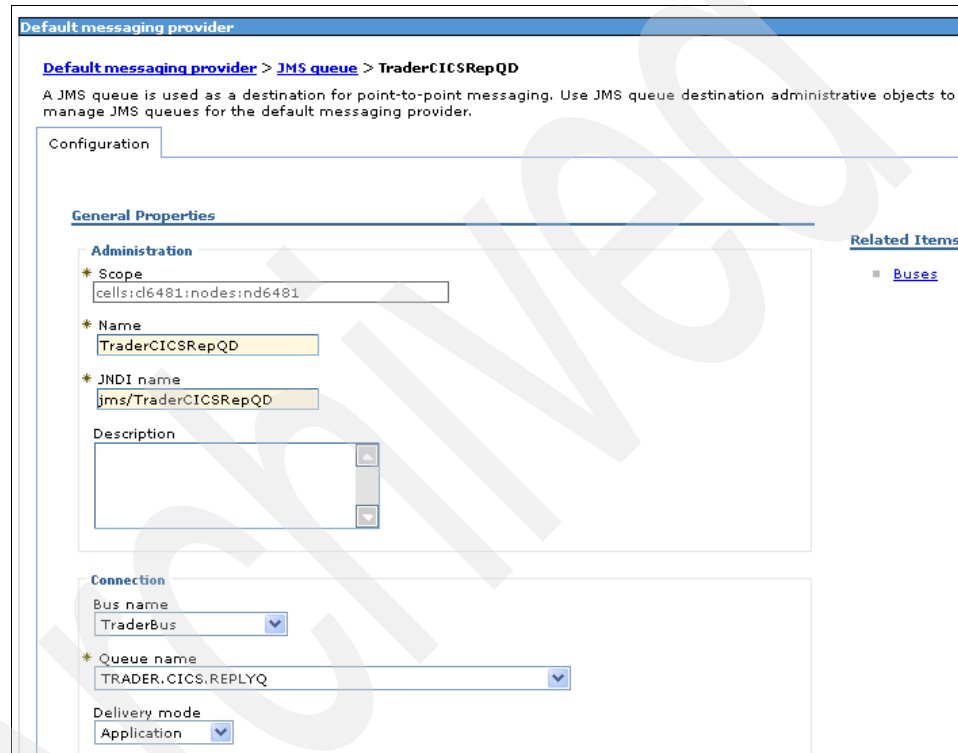
- ▶ The convention of the JNDI name being the name prefixed by jms/
- ▶ The absence of a component managed authentication alias
- ▶ The specification of the local service integration bus name
- ▶ The fact that queue and topic related properties are on this same panel

Tip: When you wish to make use of the Default messaging system but do not like to migrate your applications, as explained in “Application changes for using Default messaging” on page 148, you can still make use of a JMS queue connection factory.

Queues and bus destinations

Next, the Default messaging JMS queues and bus destinations have to be defined, as well as some remote queue definitions on the queue manager. For a review of the infrastructure we are planning to set up, refer to Figure 4-44 on page 150.

For the JMS queue definitions, configuration can be done using the WebSphere Administrative Console. Although more queue destinations are required for TraderMQ, two definitions are presented as an example. For a complete configuration survey, refer to 4.8, “The TraderMQ application environment” on page 166. Click **Resources** → **JMS providers** → **Default messaging** → **JMS queue** → **New** in order to create a new queue. For our TraderMQ application, local and foreign JMS queues had to be configured, as indicated by Figure 4-55 and Figure 4-56 on page 162, respectively.



Default messaging provider

[Default messaging provider](#) > [JMS queue](#) > **TraderCICSRepQD**

A JMS queue is used as a destination for point-to-point messaging. Use JMS queue destination administrative objects to manage JMS queues for the default messaging provider.

Configuration

General Properties

Administration

* Scope
cells:cl6481:nodes:nd6481

* Name
TraderCICSRepQD

* JNDI name
jms/TraderCICSRepQD

Description

Connection

Bus name
TraderBus

* Queue name
TRADER.CICS.REPLYQ

Delivery mode
Application

Related Items

- Buses

Figure 4-55 Default messaging local JMS queue properties

Default messaging provider

[Default messaging provider](#) > [JMS queue](#) > **TraderCICSReqQD**

A JMS queue is used as a destination for point-to-point messaging. Use JMS queue destination administrative objects to manage JMS queues for the default messaging provider.

Configuration

General Properties

Administration

* Scope
cells:cl6481:nodes:nd6481

* Name
TraderCICSReqQD

* JNDI name
jms/TraderCICSReqQD

Description

Connection

Bus name
MQ4B

* Queue name
TRADER.CICS.BRIDGEQ

Delivery mode
Application

Related Items

- Buses

Figure 4-56 Default messaging foreign JMS queue properties

Notice in particular:

- ▶ The convention of the JNDI name being the name prefixed by jms/
- ▶ The specification of the service integration bus name for the local queue
- ▶ The specification of the queue manager name for the foreign queue

For the JMS queues, accompanying bus destinations have to be configured using the administrative console. Click **Service integration** → **Buses** → **<bus name>** → **Destinations** → **New** to create a new destination. For our TraderMQ application, local and foreign bus destinations had to be configured, as indicated by Figure 4-57 on page 163 and Figure 4-58 on page 164, respectively.

Buses

[Buses](#) > [TraderBus](#) > [Destinations](#) > **TRADER.CICS.REPLYQ**

A queue for point-to-point messaging.

Configuration

General Properties

Identifier
TRADER.CICS.REPLYQ

UUID
5FDB57C3444BA46A60E59176

Type
Queue

Description

Mediation

Default reliability
Assured persistent

Maximum reliability
Assured persistent

☒ Enable producers to override default reliability

Default priority
0

Maximum failed deliveries
5

Message points

- Queue points
- Mediation points

Additional Properties

- Context properties

Figure 4-57 Default messaging local bus destination properties

Buses

[Buses](#) > [TraderBus](#) > [Destinations](#) > **TRADER.CICS.BRIDGEQ**

The name by which this foreign destination is known for administrative purposes.

Configuration

General Properties

Identifier
TRADER.CICS.BRIDGEQ

UUID
19EB5D909AA69394D8940844

Type
Foreign

Description

Bus
MQ4B

Default reliability
Assured persistent

Maximum reliability
Assured persistent

☒ Enable producers to override default reliability

Default priority
0

☒ Send allowed

Additional Properties

[Context properties](#)

Figure 4-58 Default messaging foreign bus destination properties

Notice in particular:

- ▶ The specification of the physical queue name as the identifier
- ▶ The specification of the destination type
- ▶ The specification of the bus name as the queue manager for the foreign queue

For the queues that are local to the service integration bus, it is required to have accompanying remote queue definitions on the queue manager side.

Example 4-12 Corresponding remote queue definitions

```
DEFINE QREMOTE(TRADER.CICS.REPLYQ) RNAME(TRADER.CICS.REPLYQ) +
```

```
XMITQ('Trader') RQMNAME('TraderBus')  
  
DEFINE QREMOTE(TRADER.IMS.REPLYQ) RNAME(TRADER.IMS.REPLYQ) +  
XMITQ('Trader') RQMNAME('TraderBus')
```

The reason to configure things this way is that an MDB can only be triggered from a local bus destination, as mentioned in “Activation specification” on page 103.

JMS activation specifications

Finally, this section describes how to define the activation specifications to be used by the TraderMQ application. Configuration can be done using the administrative console. Although more activation specifications are required, one definition is presented as an example. Click **Resources** → **JMS providers** → **Default messaging** → **JMS activations specification** → **New** to create a new activation specification. For our TraderMQ application the properties had to be configured as indicated by Figure 4-59 on page 166.

Default messaging provider

Default messaging provider > JMS activation specification > TraderCICSActivation

A JMS activation specification is associated with one or more message-driven beans and provides the configuration necessary for them to receive messages.

Configuration

General Properties

Administration

* Scope
cells:cl6481:nodes:nd6481

* Name
TraderCICSActivation

* JNDI name
jms/TraderCICSActivation

Destination

* Destination type
Queue

* Destination JNDI name
jms/TraderCICSRepQ

Message selector

Bus name
TraderBus

Acknowledge mode
Auto-acknowledge

Additional

Authentication alias
(none)

Related Items

J2EE Connector Architecture (J2C) authentication data entries

Buses

Figure 4-59 Default messaging activation specification properties

Notice in particular:

- ▶ The convention of the JNDI name being the name prefixed by jms/
- ▶ The specification of the destination type and JNDI name

4.8 The TraderMQ application environment

This is a small section containing a few tables to help set up the WebSphere Application Server for z/OS Version 6.01 and WebSphere MQ infrastructure for the sample TraderMQ application.

Note: In addition to these resources it is required to have the CICS (and IMS) adapter and bridge installed and configured.

4.8.1 JMS 1.0 TraderMQ using WebSphere MQ

This section specifies our environment for the JMS 1.0 TraderMQ application (both the V5 and V6 applications) as illustrated in Figure 4-60.

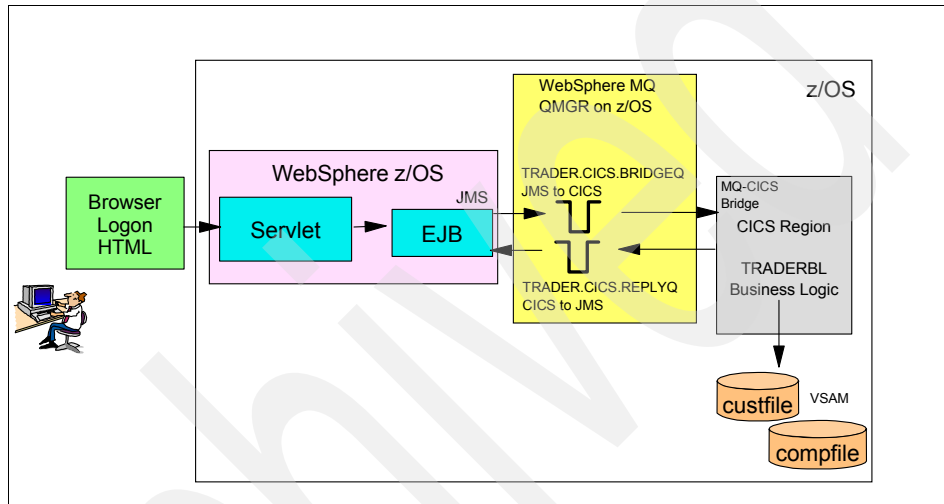


Figure 4-60 TraderMQ environment

Application source

The name of the ear file in the additional material is TraderMQ2004.ear for the Version 5 source and TraderMQ2005.ear for the Version 6 source (converted from Version 5 using the RAD migration wizard). They can be found in the “Messaging - TraderMQ” subdirectory of the additional material.

MQ queue connection factory in WebSphere

Table 4-7 MQ queue connection factory

| Name | JNDI name | Queue manager | Transport |
|-----------|---------------|---------------|-----------|
| TraderQCF | jms/TraderQCF | MQ4B | Bindings |

MQ queue destinations in WebSphere

Table 4-8 MQ queue destinations

| Name | JNDI name | Queue name | Target |
|----------------|--------------------|---------------------|--------|
| TraderCICSReqQ | jms/TraderCICSReqQ | TRADER.CICS.BRIDGEQ | MQ |
| TraderCICSRepQ | jms/TraderCICSRepQ | TRADER.CICS.REPLYQ | JMS |
| TraderIMSReqQ | jms/TraderIMSReqQ | TRADER.IMS.BRIDGEQ | MQ |
| TraderIMSRepQ | jms/TraderIMSRepQ | TRADER.IMS.REPLYQ | JMS |
| TraderProcessQ | jms/TraderProcessQ | TRADER.PROCESSQ | JMS |

Listener ports in WebSphere

Table 4-9 Listener ports

| Name | JNDI name | Destination JNDI |
|----------------------|--------------------------|--------------------|
| TraderMQCICSListener | jms/TraderMQCICSListener | jms/TraderCICSRepQ |
| TraderMQIMSListener | jms/TraderMQIMSListener | jms/TraderIMSRepQ |

Queue definitions in WebSphere MQ

The following queues have been defined in WebSphere MQ for the TraderMQ application.

TRADER.CICS.BRIDGEQ and TRADER.IMS.BRIDGEQ

| | |
|---------------------------|---|
| Message delivery sequence | F |
| Permit shared access | Y |
| Default share option | S |
| Trigger type | F |
| Trigger set | Y |
| Trigger depth | 1 |

TRADER.CICS.REPLYQ and TRADER.IMSREPLYQ

| | |
|---------------------------|---|
| Message delivery sequence | F |
| Permit shared access | Y |
| Default share option | S |

| | |
|---------------|---|
| Trigger type | N |
| Trigger set | N |
| Trigger depth | 1 |

TRADER.PROCESSQ

| | |
|---------------------------|---|
| Message delivery sequence | F |
| Permit shared access | Y |
| Default share option | S |
| Trigger type | N |
| Trigger set | N |
| Trigger depth | 1 |

4.8.2 JMS 1.1 TraderMQ using WebSphere MQ

This section specifies our environment for the JMS 1.1 TraderMQ application. The topology is the same as illustrated in Figure 4-60 on page 167. The only differences are in the coding and the usage of the connection factory.

Application source

The name of the ear file in the additional material is TraderMQ2005_JMS_1_1.ear. This application is at Version 6 level. It can be found in the “Messaging - TraderMQ” subdirectory of the additional material.

MQ connection factory in WebSphere

Table 4-10 MQ connection factory

| Name | JNDI name | Queue manager | Transport |
|----------|--------------|---------------|-----------|
| TraderCF | jms/TraderCF | MQ4B | Client |

Other settings

The configuration of the queue destinations and listener ports in WebSphere and the queue definitions in MQ are the same as specified for the JMS 1.0 version of TraderMQ and will not be repeated here.

4.8.3 JMS 1.1 TraderMQ using Default messaging

This section specifies our environment for the JMS 1.1 TraderMQ application using Default messaging. The topology is the same as illustrated in Figure 4-60 on page 167.

Application source

The name of the ear file in the additional material is TraderMQ2005_DM_JMS1_1.ear. This application is at Version 6 level. It can be found in the “Messaging - TraderMQ” subdirectory of the additional material.

JMS connection factory in WebSphere

Table 4-11 JMS connection factory

| Name | JNDI name | Bus name |
|-----------|---------------|-----------|
| TraderDCF | jms/TraderDCF | TraderBus |

JMS queue destinations in WebSphere

Table 4-12 JMS queue destinations

| Name | JNDI name | Queue name | Bus name |
|-----------------|---------------------|---------------------|-----------|
| TraderCICSReqQD | jms/TraderCICSReqQD | TRADER.CICS.BRIDGEQ | MQ4B |
| TraderCICSRepQD | jms/TraderCICSRepQD | TRADER.CICS.REPLYQ | TraderBus |
| TraderIMSReqQD | jms/TraderIMSReqQD | TRADER.IMS.BRIDGEQ | MQ4B |
| TraderIMSRepQD | jms/TraderIMSRepQD | TRADER.IMS.REPLYQ | TraderBus |
| TraderProcessQD | jms/TraderProcessQD | TRADER.PROCESSQ | TraderBus |

JMS bus destinations

Table 4-13 Bus destinations

| Name | Type | Bus name |
|---------------------|---------|-----------|
| TRADER.CICS.BRIDGEQ | Foreign | MQ4B |
| TRADER.CICS.REPLYQ | Queue | TraderBus |
| TRADER.IMS.BRIDGEQ | Foreign | MQ4B |
| TRADER.IMS.REPLYQ | Queue | TraderBus |
| TRADER.PROCESSQ | Queue | TraderBus |

Activation specifications

Table 4-14 Activation specifications

| Name | JNDI name | Destination JNDI |
|----------------------|--------------------------|--------------------|
| TraderCICSActivation | jms/TraderCICSActivation | jms/TraderCICSRepQ |
| TraderIMSActivation | jms/TraderIMSActivation | jms/TraderIMSRepQ |

Additional definitions in WebSphere MQ

In WebSphere MQ channels have to be defined (one receiver and one sender channel, as defined in Table 4-15).

Table 4-15 WebSphere MQ channels required for SIB

| Name | Type |
|-------------|----------|
| Trader2MQ4B | Receiver |
| MQ4B2Trader | Sender |

Introduction to J2EE Connector Architecture

This chapter provides an overview of the J2EE Connector Architecture.

In J2CA the Java classes needed by an application to access an Enterprise Information System (EIS), such as CICS or IMS, through connectors have been standardized. This section provides an overview of the J2CA and describes the main concepts and principles of this architecture.

For detailed information about the J2CA, refer to:

<http://java.sun.com/j2ee/connector/>

Note: The acronyms J2CA and J2EE Connector (J2C) sometimes appear to be used interchangeably in this document and in the product documentation and tools. This book uses J2CA to refer to the architecture and J2C to refer to an implementation of the architecture. The panels on the WebSphere Administrative Console use J2C. We intentionally did not use JCA, as that abbreviation is reserved for the Java Cryptographic Architecture.

Later on in this book we describe how you can develop applications accessing CICS and IMS using J2C in Chapter 6, “Developing J2C applications accessing

CICS” on page 183 and Chapter 7, “Developing J2C applications accessing IMS” on page 211, respectively.

In Chapter 8, “Configuring J2C for CTG” on page 265 and Chapter 9, “Configuring J2C for IMS Connect” on page 295 we describe how to set up the infrastructure on z/OS to use J2C.

5.1 Connector components

Figure 5-1 illustrates the main concept behind the J2CA.

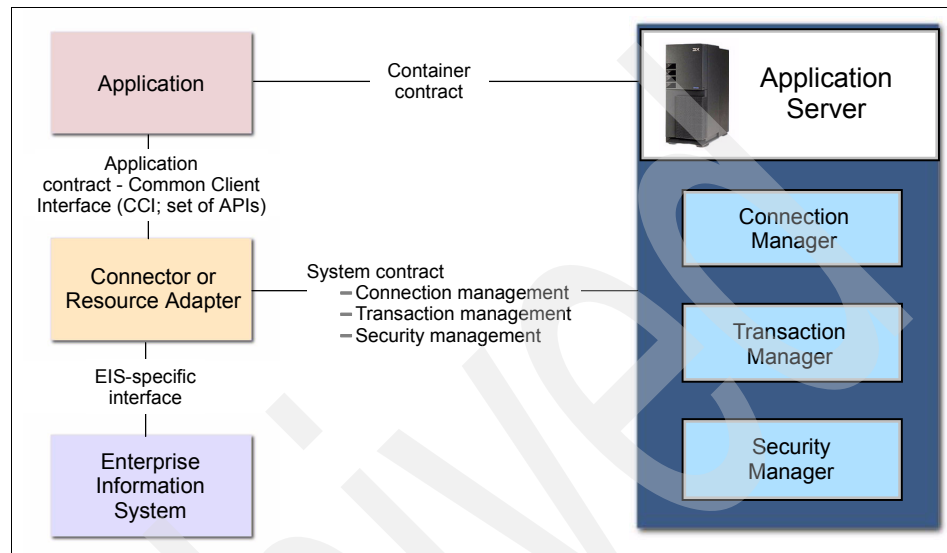


Figure 5-1 J2EE Connector architecture

The *application server* is the Enterprise JavaBean (EJB) runtime engine. It provides such facilities as transaction support, security support, and persistence capabilities.

The *connector* or *resource adapter* concept (and corresponding objects) has been used within Java since the inception of the Java language. Objects that implement the adapter pattern have been used to interface Java with the native graphical user interface (GUI), for instance.

The *application* represents the client application code that uses the client Application Programming Interface (API) provided by the J2CA framework. This API is called the *Common Client Interface (CCI)*. It defines a common API, which client programs can use to access the EIS. The CCI is the connection between the client program and the resource adapter.

The *Enterprise Information System* is the back-end system (for example, CICS or IMS) where the most important business logic processing of a business occurs.

J2CA defines the *contracts* between the application, the connector, and the application server where the application will be deployed:

Container contract This defines what the application server expects to find in a deployed application. This is the standard contract between an EJB and its container. It consists of the Bean *callback methods* such as `ejbCreate()`, `ejbLoad()`, and `ejbActivate()`.

Application contract This defines what the connector expects to receive from the application. It is defined by the CCI.

System contract This specifies the behavior that every resource adapter must support. The contract components are:

Connection management contract

Allows the application server, with the assistance of the adapter, to pool connections to the EIS.

Transaction management contract

Transactions are a key concept needed to support distributed computing.

Security management contract

Details the sign-on procedures that are carried out when the client in WebSphere establishes a connection to the resource adapter or EIS.

These contracts imply that all participating components are J2EE Connector Architecture-compliant. The application, connector, and application server must all be compliant with the J2EE architecture.

5.2 The Common Client Interface

The Common Client Interface (CCI) defines a unified remote function call interface, which focuses on executing functions in the EIS and retrieving the results. From a programming perspective, this means that programmers only have to use a single unified interface, with which they can get data from the EIS (for example, from CICS or IMS). The EIS-supplied resource adapter handles abstracting the difference and provides a unified programming model to the programmer. This model is independent of actual EIS behavior and communication requirements.

5.2.1 Establishing a connection to a resource

The J2EE Connector Framework supports access to EISs from:

- ▶ Managed connections

With managed connections, the application server handles all aspects of the connection. The application server handles the Quality of Service (QoS). This includes, for example, looking up a connection factory instance, getting an EIS connection, and finally closing the connection.

- ▶ Non-managed connections

In a non-managed application scenario, the application developer follows a similar programming model to the managed application scenario, but must handle all aspects of the connection within the application code.

Figure 5-2 on page 178 illustrates the process of establishing a connection to a resource. In this case, it is a managed connection, since the application server provides the QoS. The application starts the process with a request to the Java Naming and Directory Interface (JNDI) for a connection. This is seen as a map that links applications with services.

JNDI returns a *ConnectionFactory* object. A factory object can create other objects, in this case *connections*.

To create a connection with QoS, the *ConnectionFactory* object requests a *Connection* object from the *ConnectionManager* object at the application server. A *Connection* object is returned to the application with the QoS, as defined by the application server (with the QoS that you specified when you set up your application on the application server). The *Connection* object interacts with the connector to provide data to the application. The application can influence the behavior of the connection by specifying a *ConnectionSpec* object when obtaining the *Connection* object.

A non-managed connection is identical except that you (not the application server) provide the QoS using the *DefaultConnectionManager* rather than the *ConnectionManager*.

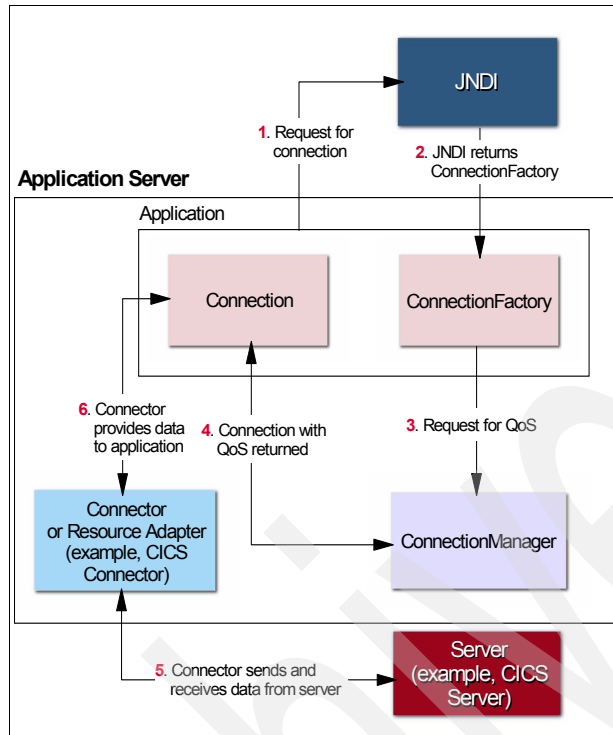


Figure 5-2 Establishing a connection to a resource using J2CA

5.2.2 Interacting with the resource

Once a connection has been established, the application interacts with the EIS to perform a specified action appropriate for the EIS (for example, do a Dynamic Program Link (DPL) to a CICS program or place an IMS transaction in the input message queue and wait for the response). This interaction with the EIS is governed by the property of an object, which the application obtains from the application server, an *interaction* object. The application can influence behavior or properties of the interaction object by specifying a *InteractionSpec* object when requesting the creation of the *interaction* object.

Example 5-1 is an sample of the CCI code to obtain a connection factory, create a connection to an EIS, obtain an interaction object, and then interact with the EIS.

Example 5-1 Example of code using the CCI to interact with a resource

```
// JNDI lookup of a Connection Factory
InitialConext ic = new InitialContext();
```



```

ConnectionFactory connectionFactory = (ConnectionFactory) ic.lookup("EIS");

// Create a ConnectionSpec object
ConnectionSpec connectionSpec = new ConnectionSpec(user,password);
// Create a Connection object using the above ConnectionSpec object
Connection connection = connectionFactory.getConnection(connectionSpec);

// Create an InteractionSpec object
InteractionSpec interactionSpec = new InteractionSpec();
interactionSpec.setUserName("USERID");
interactionSpec.setPassword("PASSWORD");
// Create an interaction using the above InteractionSpec object
Interaction interaction = conn.createInteraction(interactionSpec);

// Execute the interaction object passing an input record and
outputRecord = interaction.execute(inputRecord);

```

5.3 Enabling Rational Application Developer for J2C

Before being able to develop J2C applications in Rational Application Developer Version 6 you must prepare the environment. There are two activities involved in this:

- ▶ Installing the J2EE Connector Tools feature, as explained in “Installing the J2EE Connector Tools feature” on page 179
- ▶ Enabling the J2C feature from within Rational Application Developer Version 6, as explained in Figure 5-4 on page 181

5.3.1 Installing the J2EE Connector Tools feature

You can install this feature by starting the IBM Rational Product Updater, then click **Optional Features** and select **J2C Connector Tools** to be installed, as shown in Figure 5-3 on page 180.

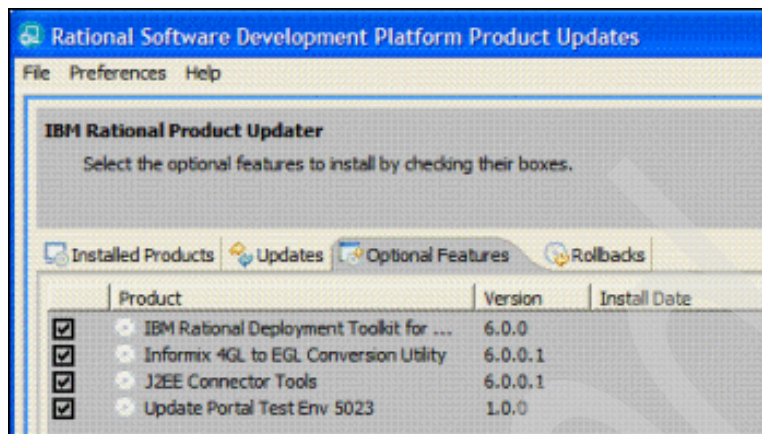


Figure 5-3 Using the updated tool to install J2C Connector tools

5.3.2 Enabling the J2C feature in RAD

The second step is to ensure the J2C feature has been enabled. Open the Capabilities view under the Workbench in the Preference view and ensure that Enterprise Java has been checked (see Figure 5-4 on page 181).

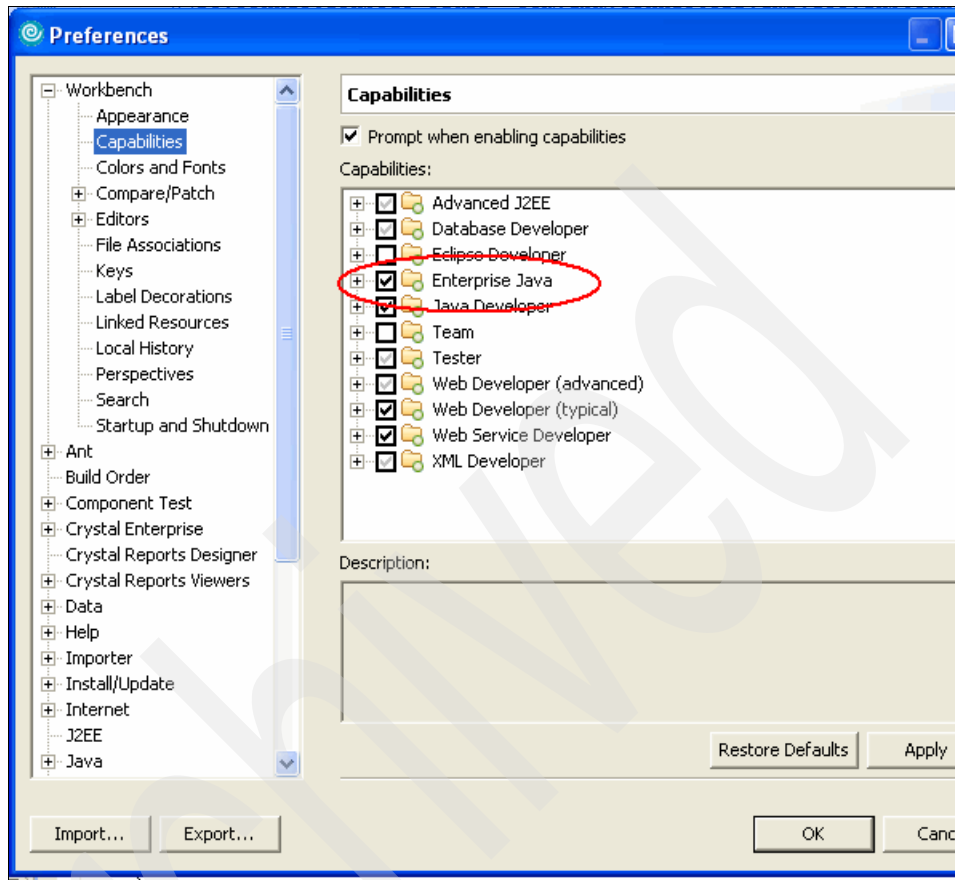


Figure 5-4 Enabling the J2C feature in Rational Application Developer

Developing J2C applications accessing CICS

This chapter explains how to develop or migrate a simple J2C application connecting through CICS Transaction Gateway V6 to CICS Transaction Server Version 3.1. The application being used in this chapter is the Trader application, which has been used in previous Redbooks as well.

Later on, in Chapter 8, “Configuring J2C for CTG” on page 265, we describe how to configure the resource adapter for CICS Transaction Gateway V6 in WebSphere Application Server for z/OS Version 6.01 and how we deployed the sample application.

The topics covered are:

- ▶ “CICS Transaction Gateway topologies” on page 184
- ▶ “CICS J2C application development” on page 185

6.1 CICS Transaction Gateway topologies

In a *local* configuration (Figure 6-1), you access a CICS region directly from WebSphere Application Server for z/OS Version 6.01. An active CICS Transaction Gateway V6 task is not required. CICS Transaction Gateway code must still be installed on the same logical partition (LPAR) as WebSphere Application Server for z/OS Version 6.01.

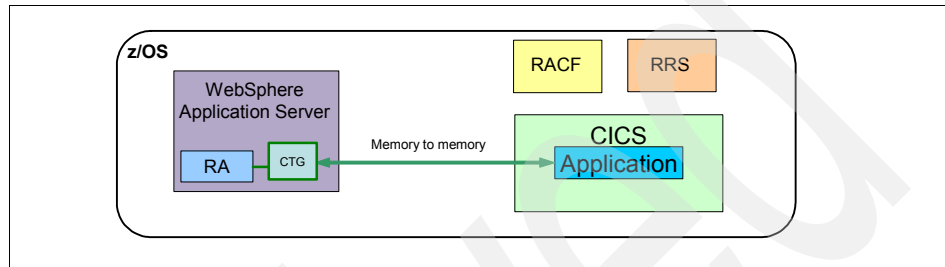


Figure 6-1 CICS Transaction Gateway local configuration

In a *remote* configuration (Figure 6-2), a CICS Transaction Gateway task must be active and listening for a request on the same LPAR as the CICS region. CICS Transaction Gateway code must still be installed on the same LPAR that WebSphere is running on, but the active CICS Transaction Gateway task does not have to be on the same LPAR.

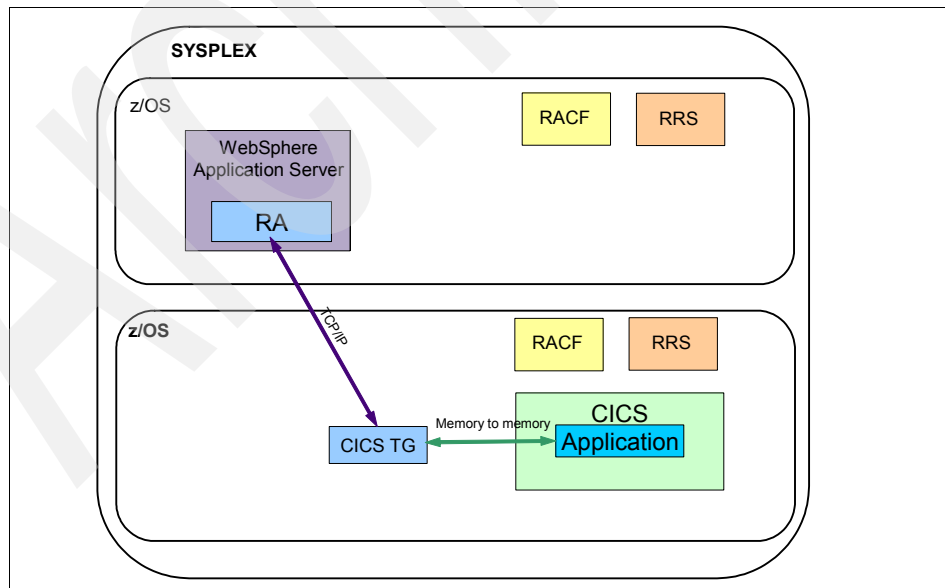


Figure 6-2 CICS Transaction Gateway remote configuration

Note: If a WebSphere-accessed CICS were on the same LPAR, using a remote connection, Resource Recovery Services (RRS) manages their protected resources independently of each other. No two-phase commit (2PC) is supported between WebSphere and CICS in that case.

6.2 CICS J2C application development

In Chapter 1 of *WebSphere for z/OS V5 Connectivity Handbook*, SG24-7064-01, we described how to develop an application to use the J2C architecture with the Version 5 level of WebSphere Application Server for z/OS and WebSphere Studio Application Developer Integration Edition. The following sections describe how to develop a similar application using the Version 6 levels of the products¹.

6.2.1 Backward compatibility

WebSphere Application Server for z/OS Version 6.01 provides backward compatibility support to allow applications that use the Version 5 development approach for J2C connectors to run unchanged. We verified this by installing the TraderCICS ear file available with the *WebSphere for z/OS V5 Connectivity Handbook*, SG24-7064-01, into a WebSphere Application Server for z/OS Version 6.01 server and ran it successfully.

6.2.2 Version 6 J2C CICS support

Rational Application Developer Version 6 provides support to quickly develop code to allow an application to call a program in CICS via the new J2C architecture supported in WebSphere Application Server for z/OS Version 6.01.

Sample tutorials on how to run the J2C wizards are available in Rational Application Developer Version 6, by selecting **Help** → **Tutorials** to reach the display shown in Figure 6-3 on page 186.

¹ Instead of WebSphere Studio Application Developer Integration Edition we are using now Rational Application Developer Version 6.

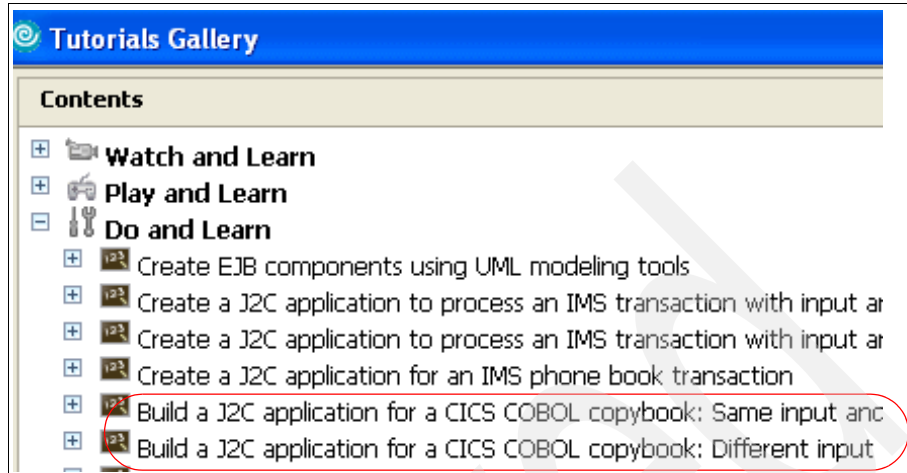


Figure 6-3 Sample tutorials in Rational Application Developer Version 6

To demonstrate this process we show how we modified the TraderCICS sample application from the *WebSphere for z/OS V5 Connectivity Handbook*, SG24-7064-01.

If you were developing a new application from scratch or changing an existing application to call a CICS program, then the process described here is the same.

6.2.3 Import TraderCICS ear

From the RAD workbench select **File** → **import**, select **EAR** as the type of file to import, and in the window displayed locate the TraderCICS ear file shipped with *WebSphere for z/OS V5 Connectivity Handbook*, SG24-7064-01. When located, this sets the name of the project. Change the name as shown in Figure 6-4 on page 187.

Import

Enterprise Application Import

Import an Enterprise Application project based on selected resources.

EAR file: C:\zResidency\V5-ear\TraderCICS2004.ear

EAR project: TraderCICS2005

☒ Import EAR Project

☐ Overwrite existing resources without warning.

☐ Delete project on overwrite

Target server: WebSphere Application Server v6.0

Figure 6-4 Locating the TradeCICS sample EAR file

Click **Next** and select to import the utility Jars as projects, as shown in Figure 6-5 on page 188.

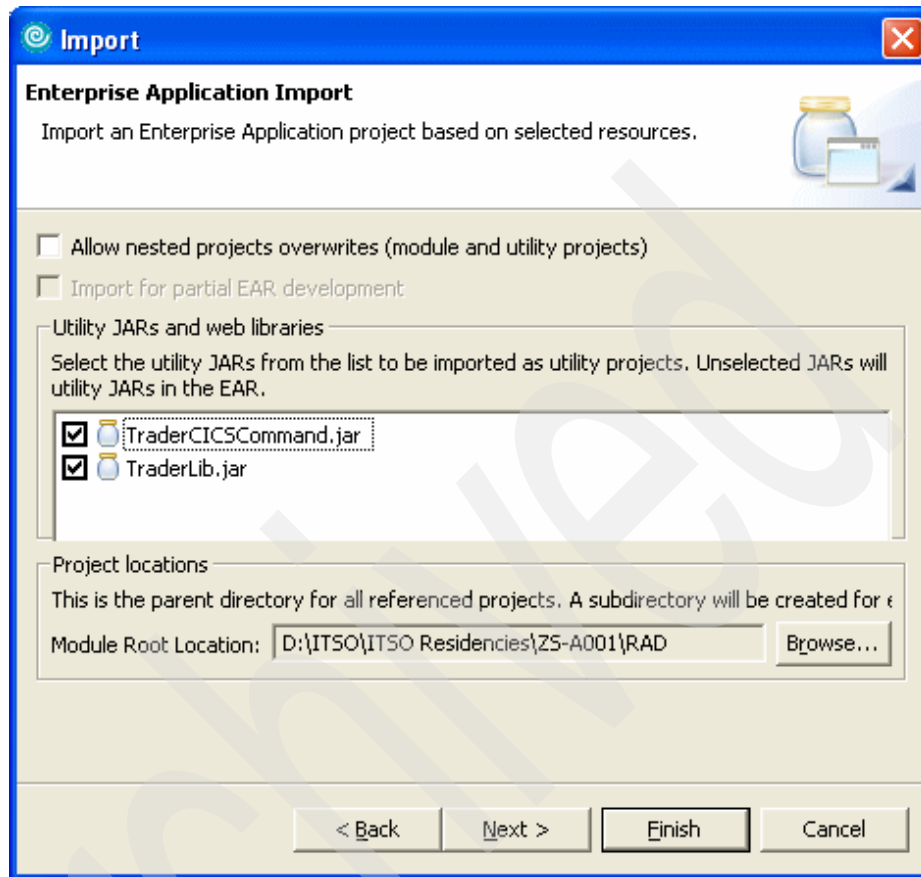


Figure 6-5 Importing utility Jars as projects

Click **Finish** and the application is imported into Rational Application Developer Version 6.

6.2.4 Delete TraderCICSEJB2

In the TraderCICS sample used in *WebSphere for z/OS V5 Connectivity Handbook*, SG24-7064-01, the TraderCICSEJB2 part of the application handled the part of interfacing with the J2C CICS connector. As we are going to be generating new code to handle this task, the TraderCICSEJB2 EJB should be deleted.

Expand **EJB Projects**, right-click **TraderCICSEJB2**, then set the options in the Delete window as shown in Figure 6-6 on page 189.

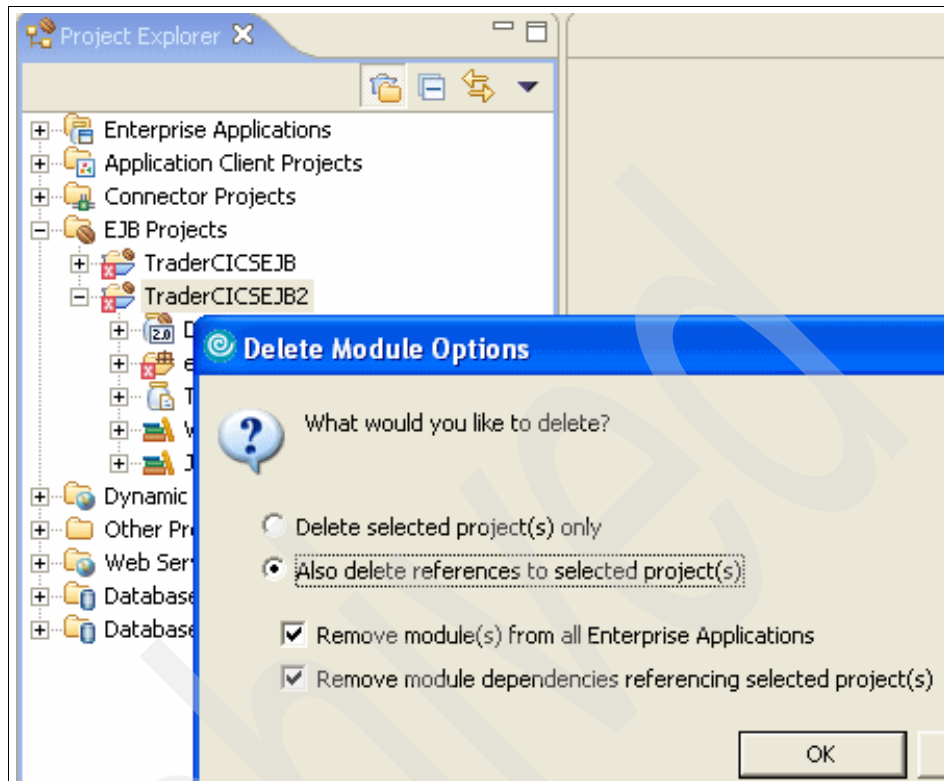


Figure 6-6 Deleting the TraderCICSEJB2 EJB

Click **OK** and another confirmation window is displayed. Set the option in this as shown in Chapter 6-7, “Select option to delete all content of the EJB” on page 190.

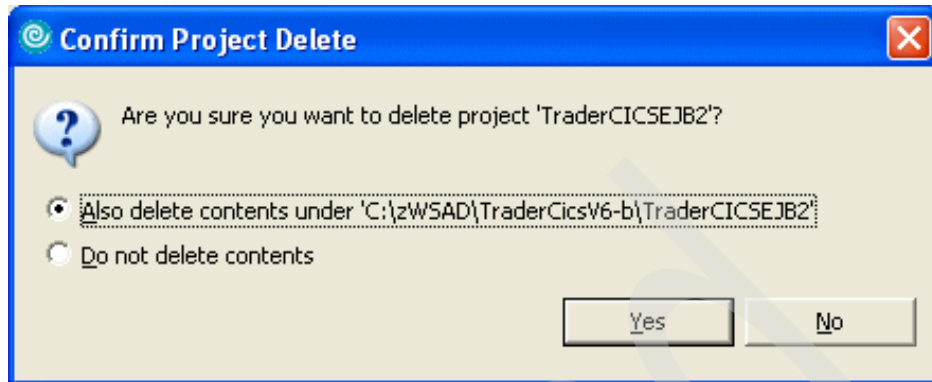


Figure 6-7 Select option to delete all content of the EJB

Click **OK** and the EJB is deleted from the project. The TraderCICSEJB will have a red cross on it to indicate it has errors. This is due to the TraderCICSEJB2 EJB having just been deleted. These errors will be resolved once we have generated the new code to handle connecting to CICS.

6.2.5 Generating J2C CICS Connector Code

Rational Application Developer Version 6 provides a wizard to guide you through the process of generating the CICS J2C connector code.

Note: To be able to perform this task, ensure that the J2C connectivity feature is installed in Rational Application Developer Version 6. This is explained in Figure 5-4 on page 181.

Invoke the J2C Java Bean Wizard

To start the process, click **File** → **New** → **Other**, then select the J2C Java Bean option as shown in Figure 6-8 on page 191.

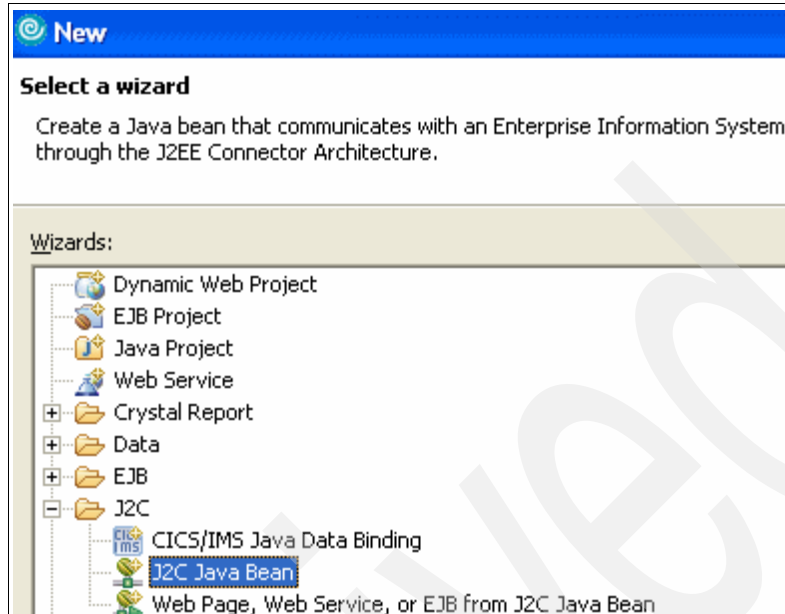


Figure 6-8 Wizard to create J2C Java Bean

If the J2C option is not displayed this means you do not have the J2C Connector Tools installed into Rational Application Developer Version 6 and/or you do not have the J2C feature enabled. Refer to “Enabling Rational Application Developer for J2C” on page 179 to make sure your development environment is complete. If you have the J2C Connector Tools installed, then in the window displayed, select the Version 6 CICS resource adapter as shown in Figure 6-9 on page 192.

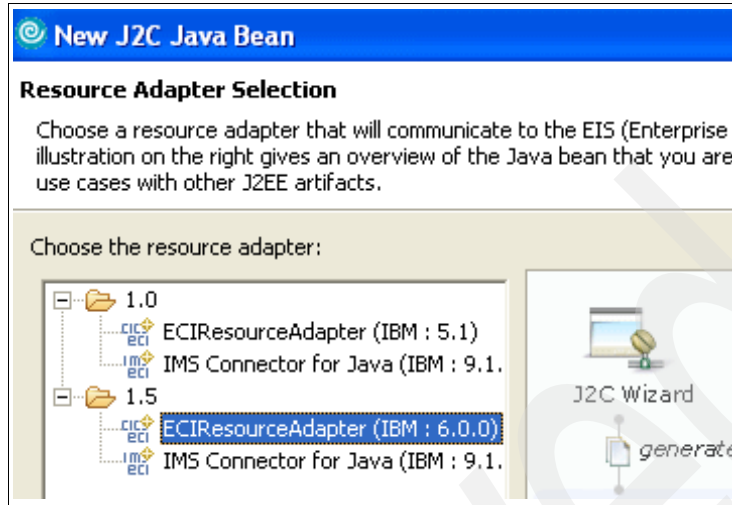


Figure 6-9 Selecting the V6 CICS Resource Adapter

Click **Next**.

Set JNDI Name

The next window displayed is where you set the JNDI name that the generated code will use to locate the CICS resource adapter. Take the default of Managed Connection and set the JNDI lookup name to eis/CICSLocal as shown in Figure 6-10.

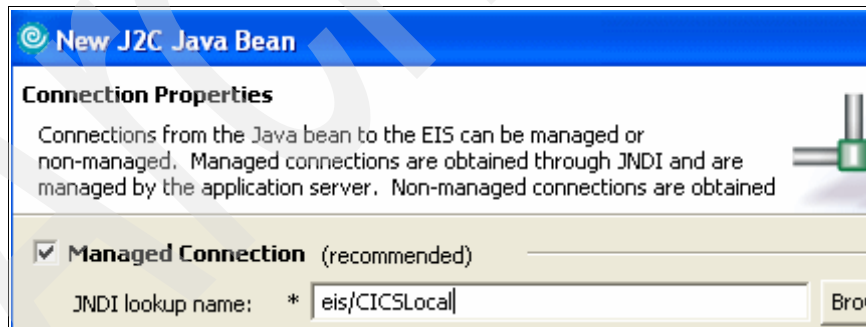


Figure 6-10 Setting JNDI Name

The value of eis/CICSLocal will become the logical name of the J2C CICS Connector that the application program uses to locate it by, as explained later in “Update the ejb-jar.xml” on page 206.

The window shown in Figure 6-11 on page 193 is displayed; click **Yes**.

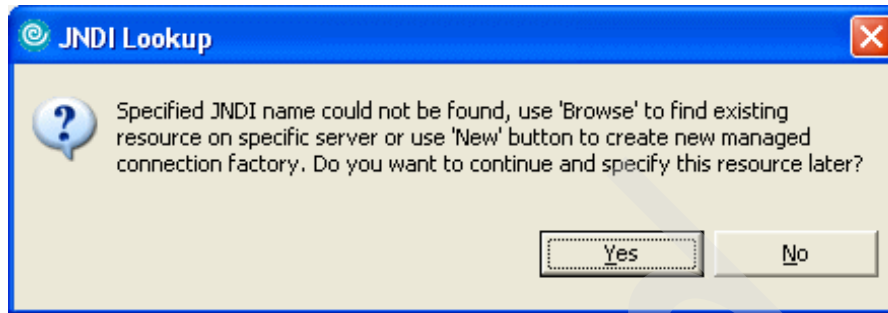


Figure 6-11 Warning message

Click **Yes**.

Name and location of J2C Java Bean

The next window displayed is where you set properties to name the J2C Java Bean that will be generated and where it will be saved in the project. Set these as shown in Figure 6-12 on page 194, then click **Next**.

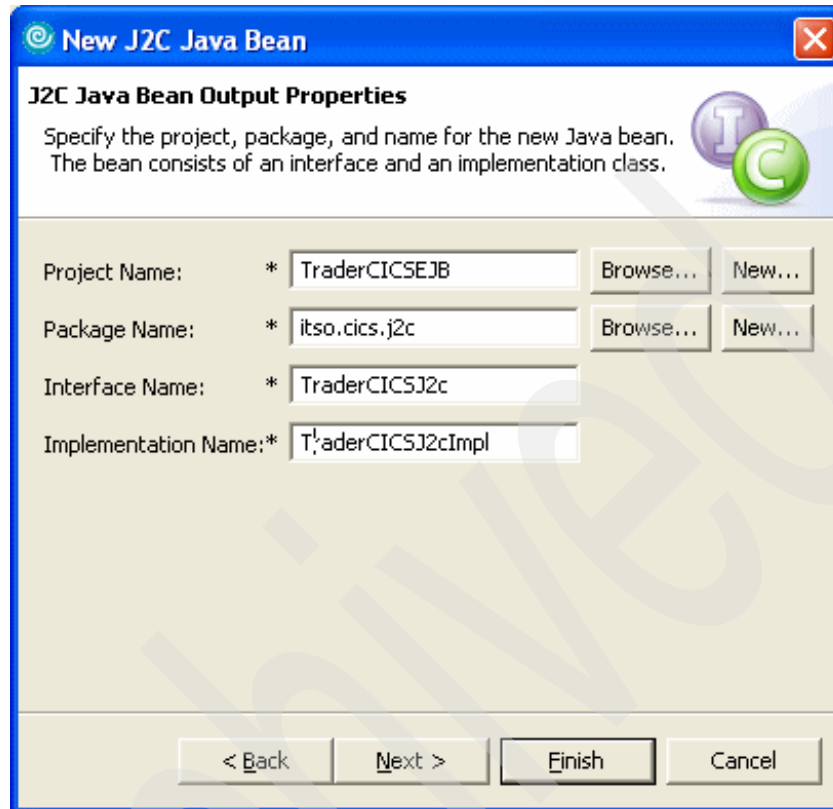


Figure 6-12 Setting the name of the J2C Java Bean

Click **Next**.

Add method to J2C Java Bean

The next window displayed (shown in Figure 6-13 on page 195) is where you manage the creation of methods to be in the J2C Java Bean.

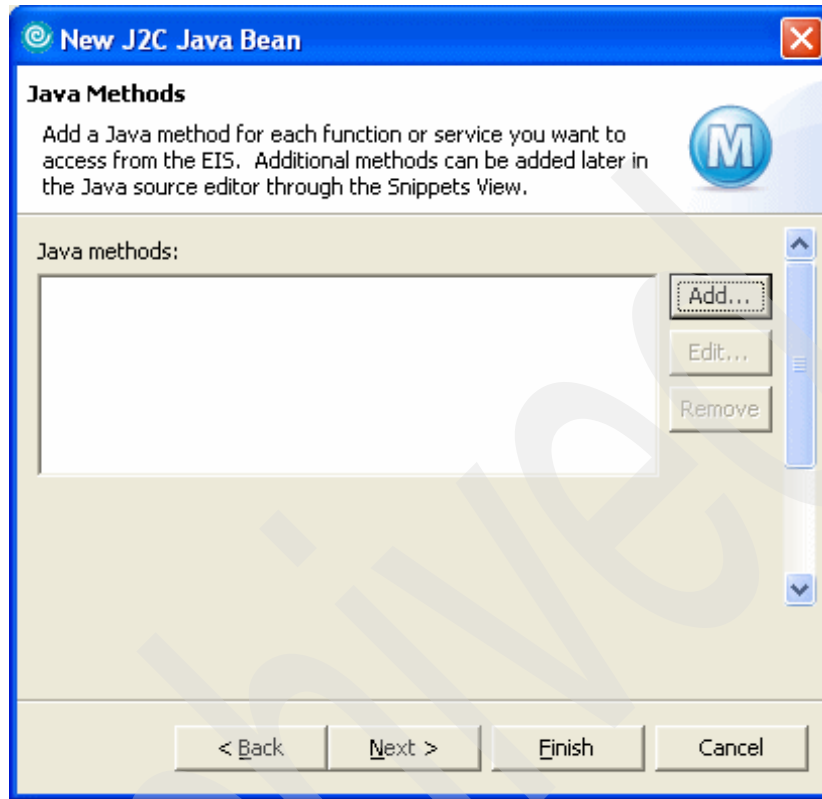


Figure 6-13 Start of process to define Java method in the bean

In our case, we need to add a method called `runTrader`, as this is the method called by the existing `TraderCICSECIBean`. This method used to exist in the `TraderCICSEJB2` code that we deleted. Click the **Add** button to begin the process of creating the method.

A new window displays; enter the method name of `runTrader`, as shown in Figure 6-14 on page 196.

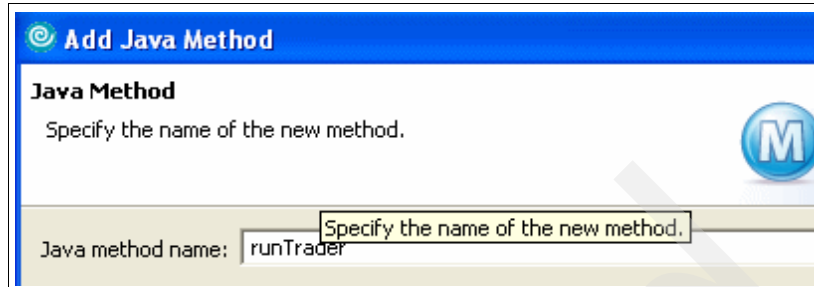


Figure 6-14 Setting name of method to add

Click **Next**.

Define input and output types for the method

A new window is displayed, as shown in Figure 6-13 on page 195, which allows you to specify the input and output data types for the method.

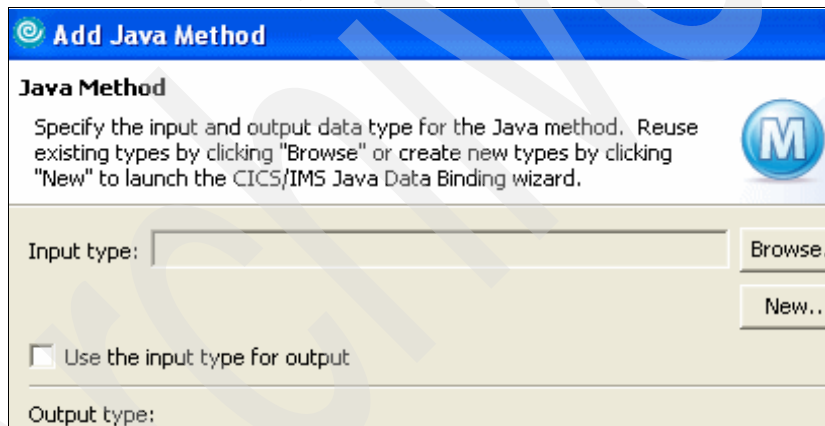


Figure 6-15 Start of process to define input and output data types for the method

As this method is calling a CICS program, only one object will be passed on the call to the method. This object will in effect be passed to CICS as a standard commarea. When the call from CICS completes, the commarea is passed back to this method. For our sample application the commarea structure is the same for both input and output, which means this method will return the same sort of object. This does not have to be the case for an application you may be writing.

Create Java class for the input and output data type

What we need to do now is to create a Java class that will handle the mapping of the commarea used in the CICS call. To do this click **New** and a window is

displayed. Use this window to locate the COBOL program source code that is going to be called, as shown in Figure 6-16.

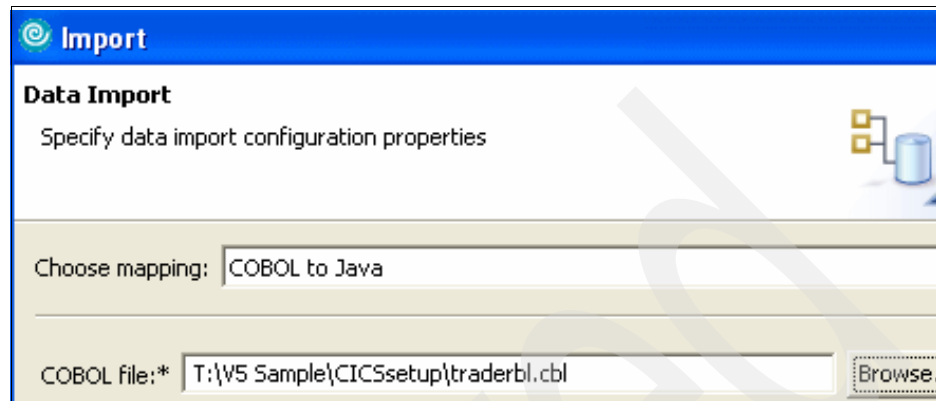


Figure 6-16 Locating the COBOL program

Click **Next** and the window shown in Figure 6-17 is displayed. There are two key fields to change in this display.

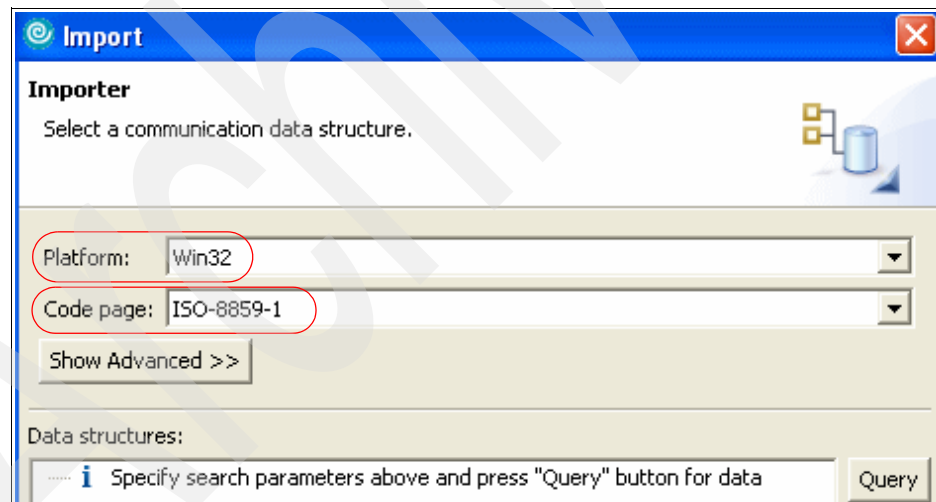
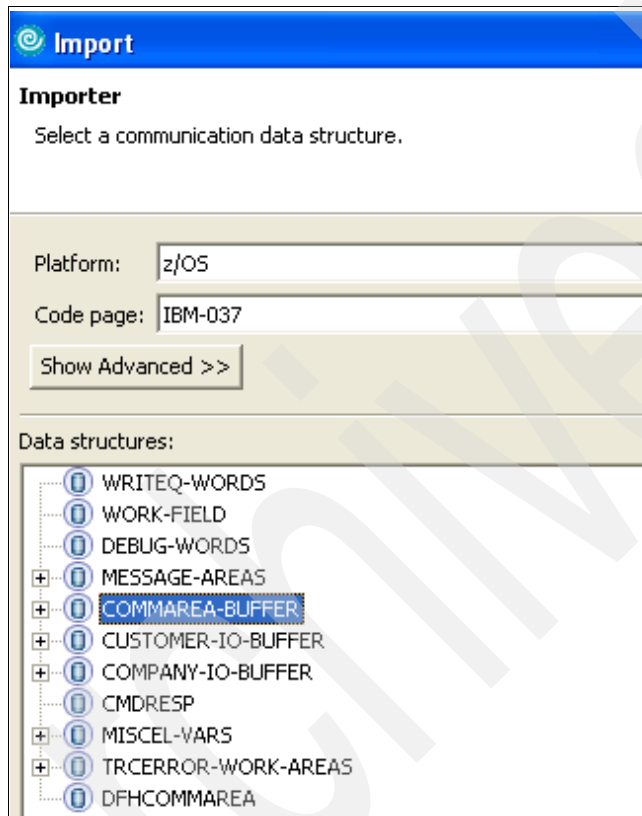


Figure 6-17 Importing the data structure from the COBOL source code

Change the platform to z/OS, which will also automatically change the code page to IBM-037. Then click the **Query** button; this results in the data structures found in the COBOL source code being displayed in the window. The window should now look as shown in Figure 6-18 on page 198.

Note: If you forget to set the platform and code page correctly, there is still a way to correct the settings by hand later on. This is described in “Rational Application Developer Version 6 and meta-data” on page 204.



Import

Importer

Select a communication data structure.

Platform: z/OS

Code page: IBM-037

Show Advanced >>

Data structures:

- WRITEQ-WORDS
- WORK-FIELD
- DEBUG-WORDS
- MESSAGE-AREAS
- COMMAREA-BUFFER**
- CUSTOMER-IO-BUFFER
- COMPANY-IO-BUFFER
- CMDRESP
- MISCEL-VARS
- TRCERROR-WORK-AREAS
- DFHCOMMAREA

Figure 6-18 Display after setting z/OS as the platform and querying the data structure

Select **COMMAREA-BUFFER** and click **Next**. The next window displayed is where you specify the name of the Java class that will handle manipulation of the object passed to and from CICS and where it is saved in the project. Set the fields as shown in Figure 6-19 on page 199.

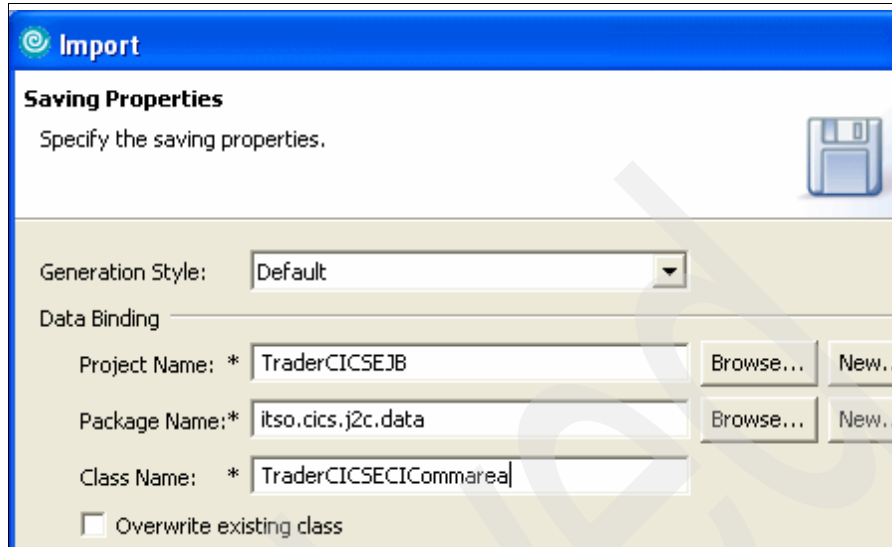


Figure 6-19 Setting properties for the Java Class that will handle commarea

Click **Finish** and an updated version of the window shown in Figure 6-15 on page 196 is displayed. Click the option **Use the input type for output** and the window will now look as shown in Figure 6-20.

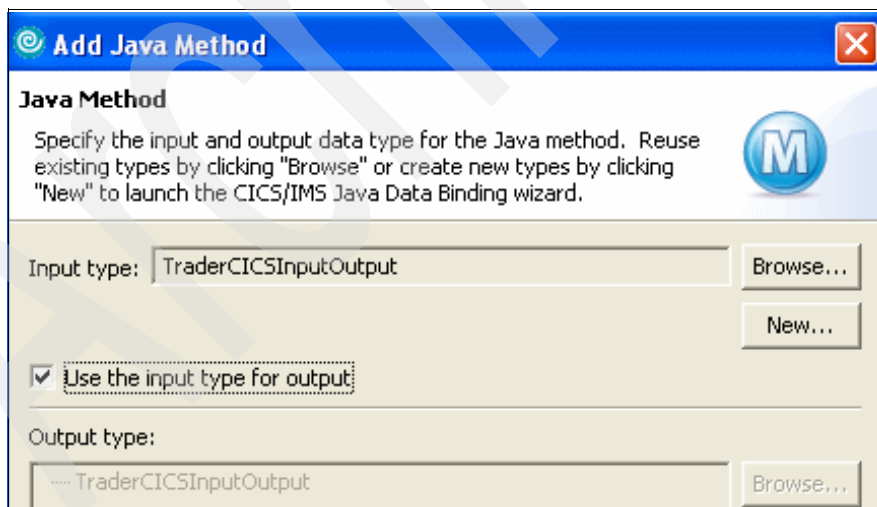


Figure 6-20 Updated add Java method window

Click **Finish**.

Set J2C Connector properties

A new window is displayed. This new window is where you define properties dealing with the J2C resource adapter. Click **Show Advanced** and set the properties as shown in Table 6-1.

Table 6-1 Properties to set for the CICS J2C connection factory

| Property | Setting |
|------------------|-----------------|
| Function Name | TRADERBL |
| Commarea Length | 500 |
| Reply Length | 500 |
| TPN name | blank |
| Tran name | blank |
| Execute timeout | 0 |
| Interaction verb | SYNC_RECEIVE(1) |

The window will look as shown in Figure 6-21 on page 201.

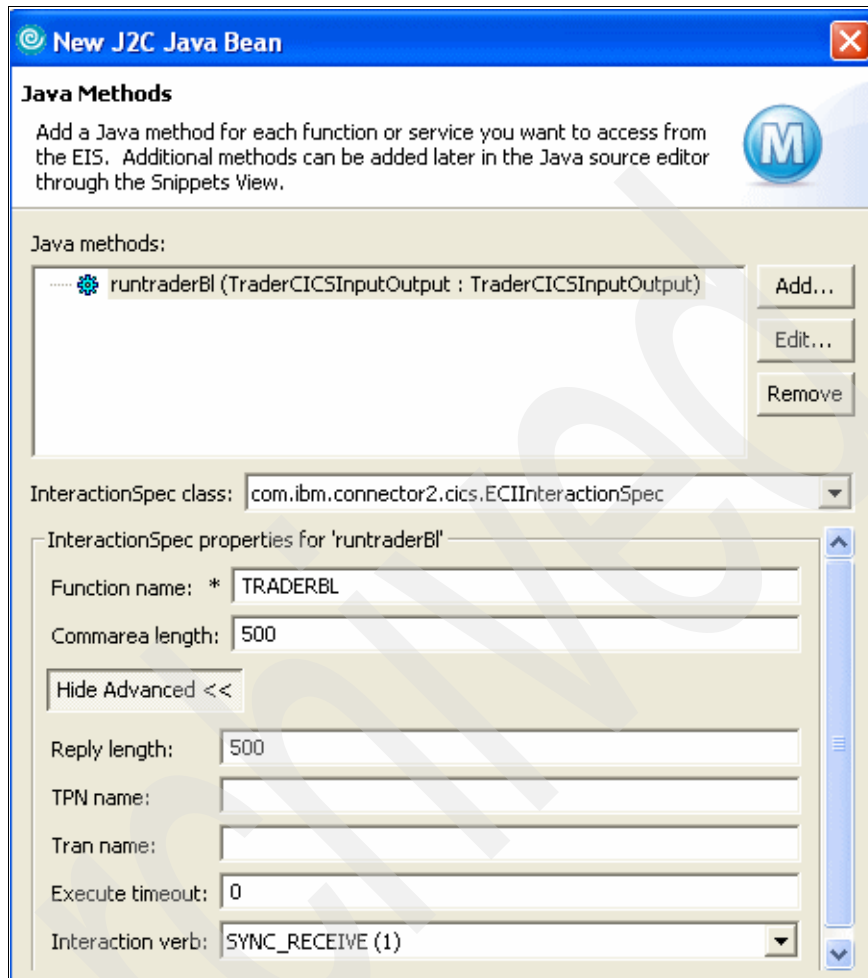


Figure 6-21 Completed Java Bean method display

Click **Finish** and the new code is generated. Once complete your workbench will look as shown in Figure 6-22 on page 202.

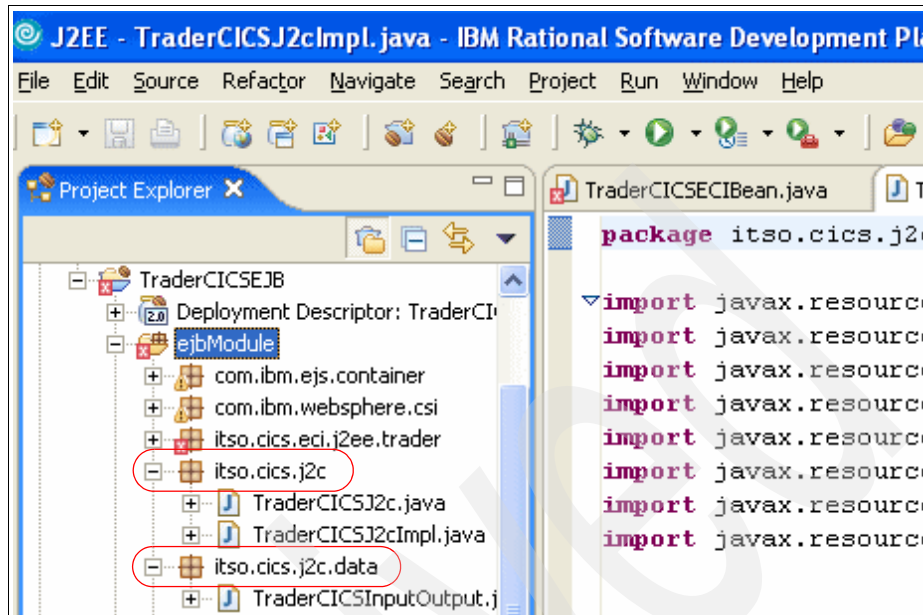


Figure 6-22 Display showing the new Java Beans

The two Java programs generated in the `itso.cics.j2c` package are the ones that will handle interacting with the CICS J2C Connection Factory. The Java program in the `itso.cics.j2c.data` package handles the conversion of Java objects to and from the commarea structure that is passed to the CICS COBOL program.

Correct outstanding Java errors

To clear the errors in the `TraderCICSECIBean` class, right-click **TraderCICSEJB** under EJB Projects, and select **Properties**; then click **Java Build Path**, then the **Projects** tab. Unselect the `TraderCICSCommand` project from the build path, as shown in Figure 6-23 on page 203. This is done so that the `TraderCICSEJB` code uses the new `TraderCICSECICommarea` Java class we have just created rather than the old one.

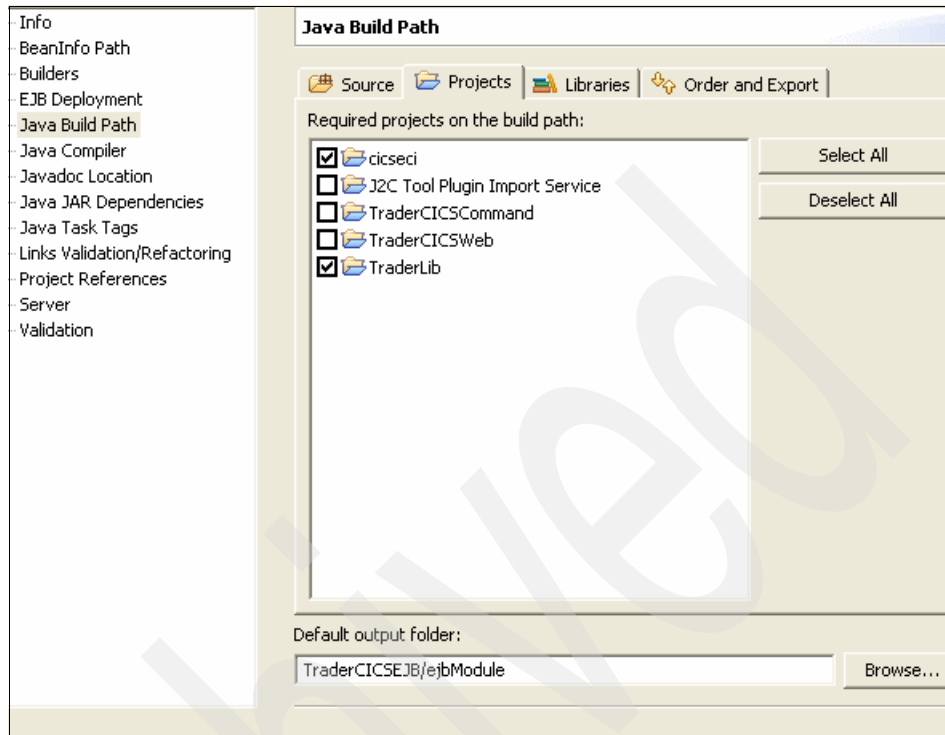


Figure 6-23 Changing the Java Build Path

Then double-click the `TraderCICSECIBean` java source program in `EJB Projects/TraderCICSEJB/ejbModule/itso.cics.eci.j2ee.trader` so you can make the following changes.

1. Add the following import statement:

```
import itso.cics.j2c.*;
import itso.cics.j2c.data.*;
```

2. Locate this line:

```
private TraderCICSECICommandEJBProxy traderCICSEjb;
```

3. Change it to:

```
private TraderCICSJ2cImpl traderCICSEjb;
```

4. Save this change.

5. Locate this line:

```
traderCICSEjb = new TraderCICSECICommandEJBProxy();
```

6. Change it to:

```
traderCICSEjb = new TraderCICSJ2cImpl();
```

Then save this change. The Java code should now have no compilation error messages.

Note the TestTraderProxy Java class described in *WebSphere for z/OS V5 Connectivity Handbook*, SG24-7064-01, has not been updated to use the new TraderCICSECICommarea Java class so will not function correctly.

6.2.6 Rational Application Developer Version 6 and meta-data

During the residency, when performing the J2C Java Bean generation process described in “Generating J2C CICS Connector Code” on page 190, during the part described in “Define input and output types for the method” on page 196, we initially did not set the Platform field shown in Figure 6-17 on page 197 to z/OS.

When we tested the Trader application, it ran without failing, but displayed no company names. After further investigation we determined that this was because the data being passed in the commarea to the COBOL program was in ASCII, not EBCDIC, thus causing the COBOL program to return no data.

Having determined the cause of the error, the challenge was then how to correct it. In the generated code for TraderCICSECICommarea.java we could see that several methods used ISO8859-1 as string passed on various Java APIs. ISO8859-1 identifies the ASCII character set. We could have tried just changing this string to IBM-037, the setting for EBCDIC, but there was also the issue of how the code was handling conversion of binary data as well to consider.

We then noticed that at the start of the generated code was the comment block shown in Example 6-1.

Example 6-1 Meta-data in the TraderCICSECICommarea java program

```
/**
 * @generated
 * Generated Class: TraderCICSECICommarea
 * @type-descriptor.aggregate-instance-td accessor="readWrite"
 * contentSize="372" offset="0" size="372"
 * @type-descriptor.platform-compiler-info language="COBOL"
 * defaultBigEndian="false" defaultCodepage="ISO-8859-1"
 * defaultExternalDecimalSign="ascii" defaultFloatType="ieeeNonExtended"
 */
```

We then changed this code, updating three fields in the type-descriptor.platform line to that shown in Example 6-2 and then saved the changes.

Example 6-2 Updated type-descriptor.platform settings

```
/**
```

```

* @generated
* Generated Class: TraderCICSECICommarea
* @type-descriptor.aggregate-instance-td accessor="readWrite"
contentSize="372" offset="0" size="372"
* @type-descriptor.platform-compiler-info language="COBOL"
defaultBigEndian="false" defaultCodepage="IBM-037"
defaultExternalDecimalSign="ebcdic" defaultFloatType="ibm390Hex"
*/

```

Rational Application Developer Version 6 then automatically updated the generated code to reflect the changes we had made in the meta-data.

6.2.7 Deprecated use of direct JNDI lookup

In the generated code for the J2C Java Bean `TraderCICSJ2cImpl.java` is code to perform the lookup of the J2C CICS Resource Adapter as shown in Example 6-3.

Example 6-3 Java code to perform the resource lookup

```

protected void initializeBinding() throws ResourceException {
    ConnectionFactory cf = null;
    String jndiName = "eis/CICSLocal";
    javax.naming.Context ctx = null;
    try {
        ctx = new javax.naming.InitialContext();
        if (ctx != null)
            cf = (ConnectionFactory) ctx.lookup("java:comp/env/" + jndiName);
    }
}

```

If you where to export the application from Rational Application Developer Version 6 and import the ear file into WebSphere Application Server for z/OS Version 6.01 and run it, when the code in Example 6-3 was executed you would see a message similar to that shown in Example 6-4.

Example 6-4 Warning message regarding use of direct JNDI lookups

```

BossLog: { 0084} 2005/06/10 17:53:55.195 01 SYSTEM=SC48 SERVER=WS6483
PID=0X010C05F4 TID=0X22E6C7D0 0X00002A c=A.4
./bborjtr.cpp+901 ... BB000221W: J2CA0294W: Deprecated usage of direct JNDI
lookup of resource eis/CICSLocal. The following default values are used:
[Resource-ref settings]

```

```

res-auth:                1 (APPLICATION)
res-isolation-level:      0 (TRANSACTION_NONE)
res-sharing-scope:        true (SHAREABLE)
loginConfigurationName:   null
loginConfigProperties:     null
[Other attributes]

```

```
res-resolution-control: 999 (undefined)
isCMP1_x:                false (not CMP1.x)
isJMS:                   false (not JMS)
```

This warning message would be generated because the application code is doing a direct JNDI lookup of the resource. You might argue that it appears the application is not doing a direct JNDI lookup since the code has the following:

```
cf = (ConnectionFactory) ctx.lookup("java:comp/env/" + jndiName);
```

which is the correct convention to use to perform a logical resource lookup.

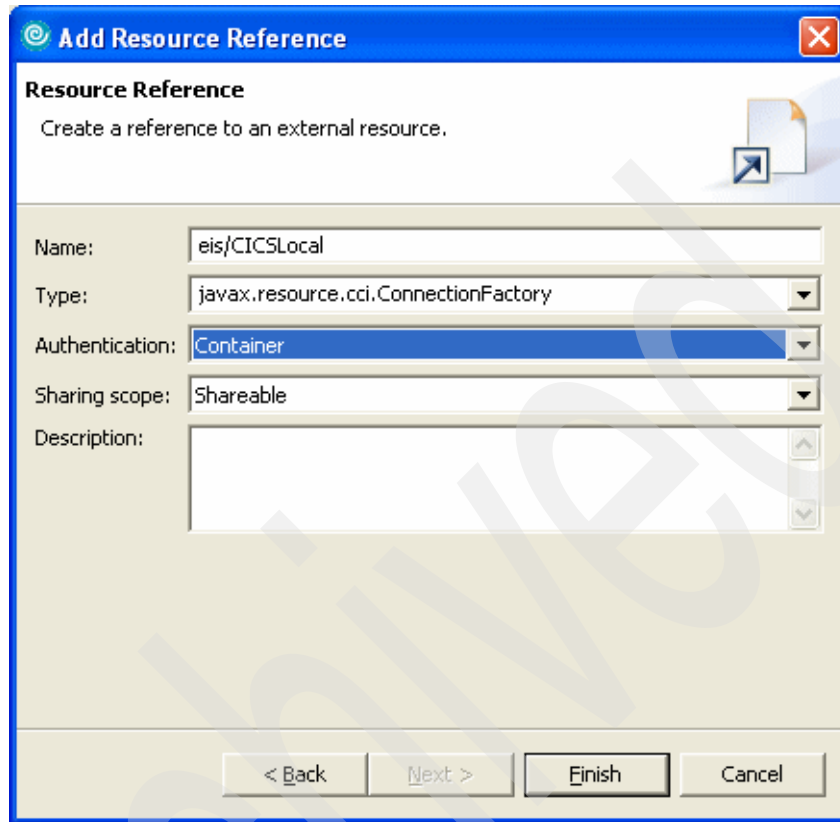
However, even though this would mean that a logical JNDI lookup of `java:comp/env/CICSLocal` was done, since no logical resource has been defined in the `ejb-jar.xml` at this time, the logical lookup fails, and WebSphere then does a direct JNDI lookup using just the value of `eis/CICSLocal`. Assuming that the server has a J2C CICS connection factory defined with a JNDI value of `eis/CICSLocal`, then the direct lookup succeeds and the warning message is issued.

As the warning message says, this approach is deprecated it will in all likelihood fail in a future versions of WebSphere.

6.2.8 Update the `ejb-jar.xml`

To ensure that a logical JNDI lookup is performed, under **EJB Projects** → **TraderCICSEJB** → **ejbModule** → **META-INF**, double-click **ejb-jar.xml** to open it for updating.

Click the **References** tab, then click **TraderCICSECI**, and then click **Add**. On the window displayed select **Resource reference** and click **Next**. In the window displayed set the fields as shown in Figure 6-24 on page 207.



The image shows a Java IDE dialog box titled "Add Resource Reference". It has a blue title bar with a close button. The main area is titled "Resource Reference" and contains the instruction "Create a reference to an external resource." with a document icon. Below this are five fields: "Name:" with the text "eis/CICSLocal", "Type:" with a dropdown menu showing "javax.resource.cci.ConnectionFactory", "Authentication:" with a dropdown menu showing "Container", "Sharing scope:" with a dropdown menu showing "Shareable", and "Description:" with an empty text area. At the bottom are four buttons: "< Back", "Next >", "Finish", and "Cancel".

Add Resource Reference

Resource Reference
Create a reference to an external resource.

Name: eis/CICSLocal

Type: javax.resource.cci.ConnectionFactory

Authentication: Container

Sharing scope: Shareable

Description:

< Back Next > Finish Cancel

Figure 6-24 Defining a logical resource reference for the EJB

Click **Finish** and the logical resource definition will be shown in `ejb-jar.xml`, as shown in Figure 6-25 on page 208.

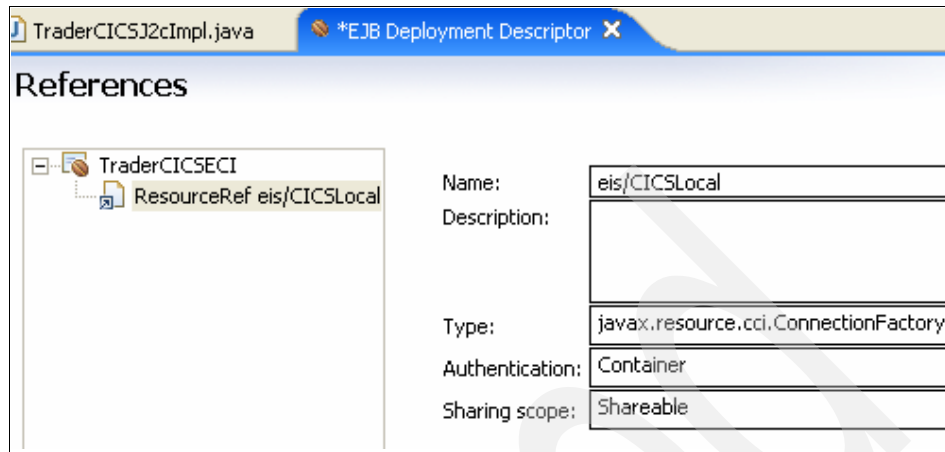


Figure 6-25 Defined logical resource reference

Set the expected JNDI name

In the References tab of the `ejb-jar.xml`, scroll the display down and you will see the display shown in Figure 6-26.

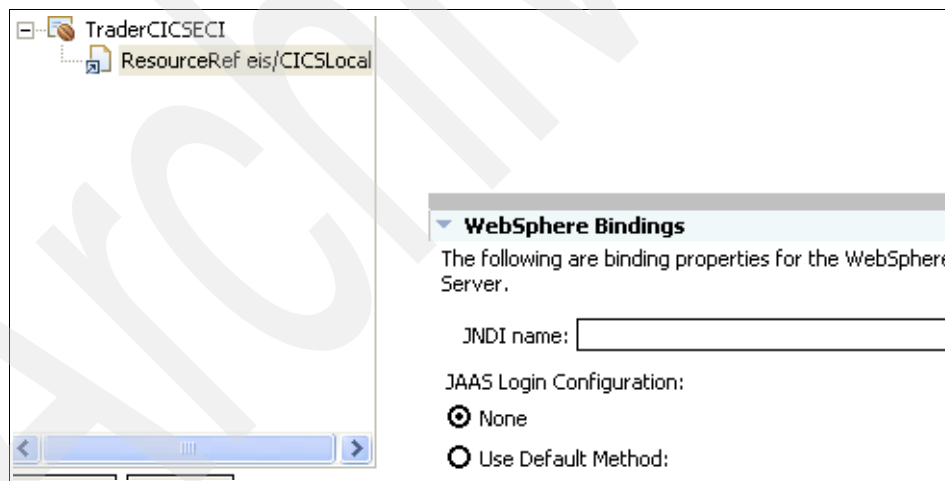
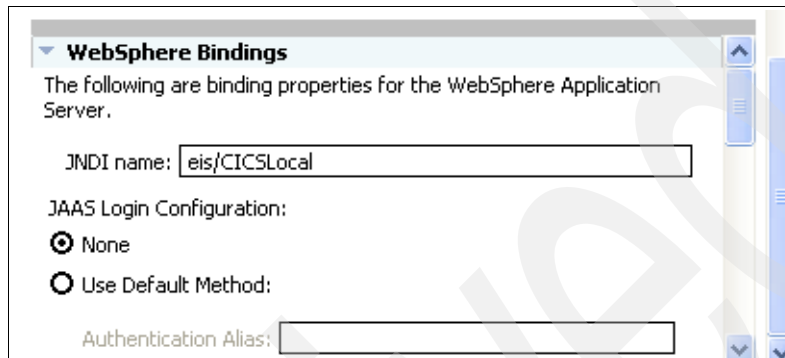


Figure 6-26 Area in `ejb-jar.xml` to set JNDI name to match with in WebSphere server

Under the heading WebSphere Bindings is a field called JNDI name. If you leave this blank, then when you deploy the EAR file later on, you will have to map this logical resource to the JNDI name of a physical resource defined in the WebSphere server. This is a simple process where you just need to select from a list of resources defined to WebSphere.

However, you can save yourself having to do this if you set in this field the JNDI name of an actual resource you expect to be defined in the WebSphere server.

If, for example, the J2C CICS Connection® Factory you will define in the WebSphere server has a JNDI name of eis/CICSLocal, then set this as the value of this field in the ejb-jar.xml, as shown in Figure 6-27.



The screenshot shows a configuration window titled "WebSphere Bindings". It contains the following elements:

- A header section with the text: "The following are binding properties for the WebSphere Application Server."
- A text field labeled "JNDI name:" containing the value "eis/CICSLocal".
- A section labeled "JAAS Login Configuration:" with two radio buttons: "None" (which is selected) and "Use Default Method:".
- A text field labeled "Authentication Alias:" which is currently empty.

Figure 6-27 Setting JNDI name to match with in the WebSphere server

Note that in this case we have the logical name of the resource that the application uses to perform the resource lookup and the JNDI name for the actual resource in the WebSphere server both set to the same value of eis/CICSLocal.

This does not have to be the case. We could have the application use a different (logical) JNDI value for the logical lookup if desired.

Save this change in the project.

Export the application

Export the application as an EAR file from Rational Application Developer Version 6 to a directory on your PC. The next step is to define a J2C CICS Resource Adapter in the WebSphere server, as described in "Installing the CICS resource adapter" on page 266, after which we will install the application.

Developing J2C applications accessing IMS

This chapter provides step-by-step examples of developing and testing applications using Rational Application Developer (RAD), which accesses IMS.

In this chapter the following topics are covered:

- ▶ An example of developing and testing J2C Java Beans accessing IMS, in “Create J2C Java Data Beans” on page 214 through “Creating an EJB J2C Java Bean” on page 227
- ▶ An example of developing and testing a SOA J2C Java Bean accessing IMS, in “Generating SOA J2C Java Beans” on page 236

7.1 Preparing the project

First, we created an Enterprise Application project to manage our EJB application, and followed these steps:

1. We opened the J2EE perspective.
2. We launched the Create an Enterprise Application project wizard by clicking the icon shown in Figure 7-1.

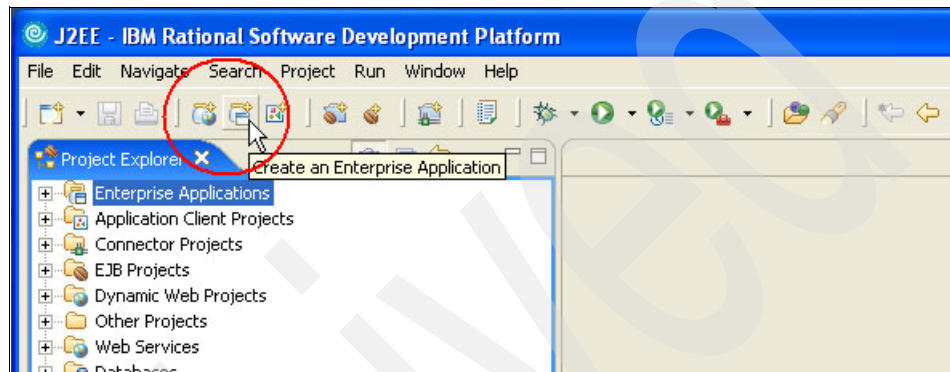


Figure 7-1 Create an Enterprise Application

3. In the New Enterprise Project window that opened, we entered TraderITSOEJB as the project name and clicked **Finish**.
4. Next, we added an EJB project to the Enterprise Application by launching the **Create an EJB project** wizard; see Figure 7-2

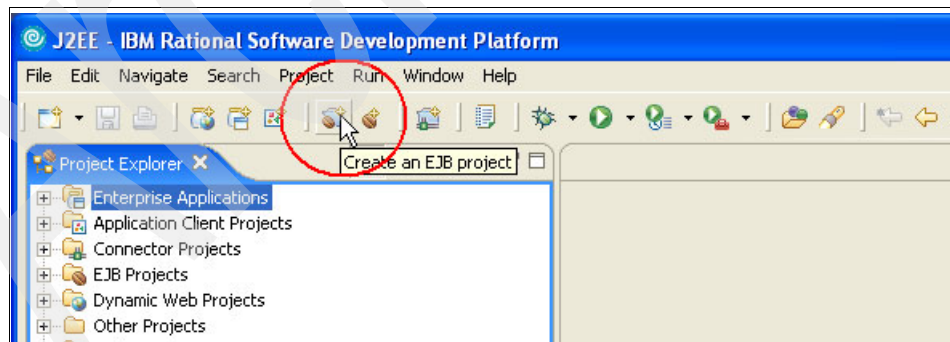


Figure 7-2 Create an EJB Project

5. In the New EJB Project window that opened, we entered TraderIMSEJB as the name of the new project. We also added this project to the Enterprise

project we just created. We clicked **Finish** after we ensured the box to create an EJB client JAR project was checked (see Figure 7-3).

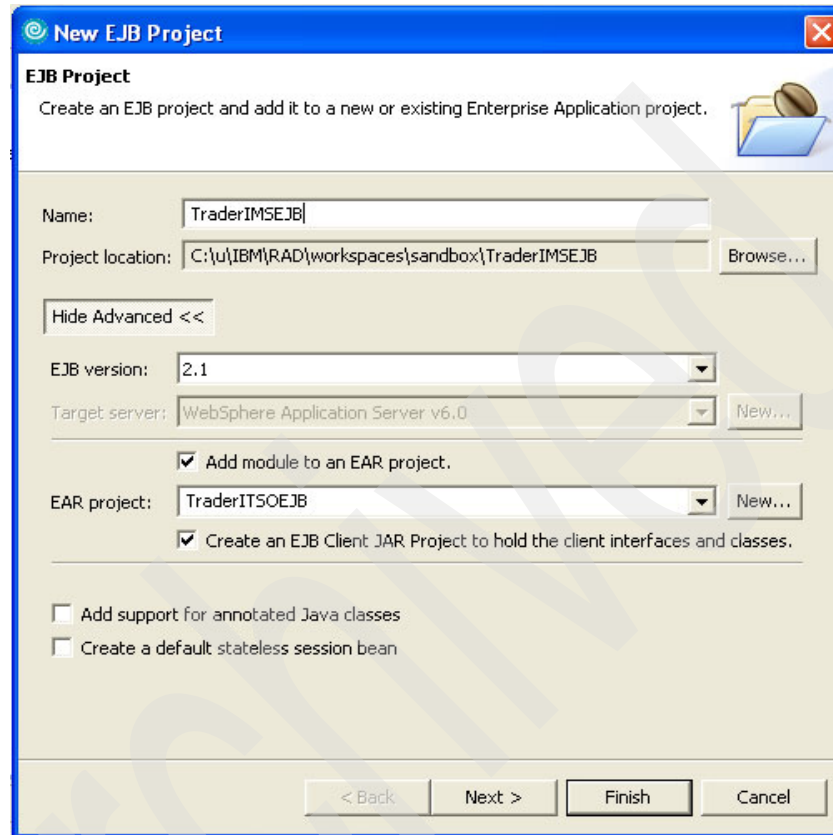


Figure 7-3 Create a new EJB project

6. We then created a Java package for use within the TraderIMSEJBClient project. We selected the TraderIMSEJBClient project under Other Projects in the Project Explorer pane and then clicked the right mouse button. We selected **New - Package** to open the New Java Package window where we entered `itso.ims.trader` as the package name (see Figure 7-4 on page 214). We clicked **Finish** to continue.

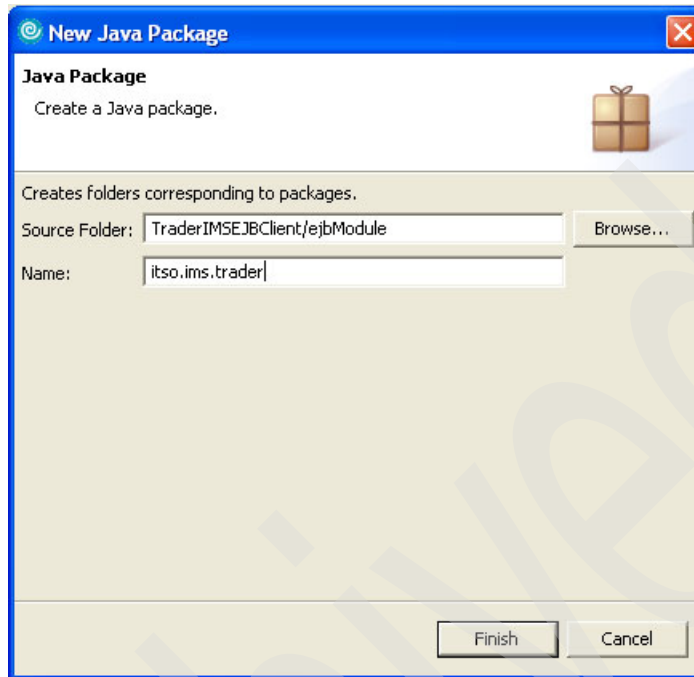


Figure 7-4 New Java Package

7.2 Create J2C Java Data Beans

To do this:

1. Next, we created J2C Java Data binding beans. We selected the **TraderIMSEJBClient** project under Other Projects in the Project Explorer pane and then clicked the right mouse button. We selected **New - Other...** to open the Select a wizard window, where we selected **CICS/IMS Java Data Binding** under the J2C folder (see Figure 7-5 on page 215). We clicked **Next** to continue.

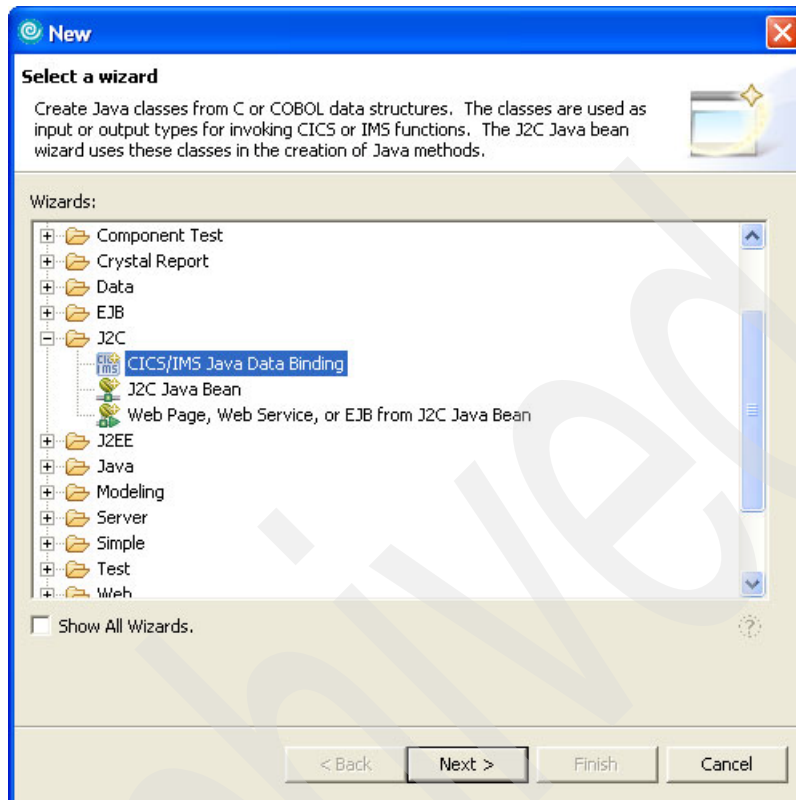


Figure 7-5 Select CICS/IMS Java Data Binding wizard

2. In the Data Import window we selected **COBOL to Java** mapping and then clicked the **Browse** button to traverse to the source of the target COBOL program (see Figure 7-6 on page 216).

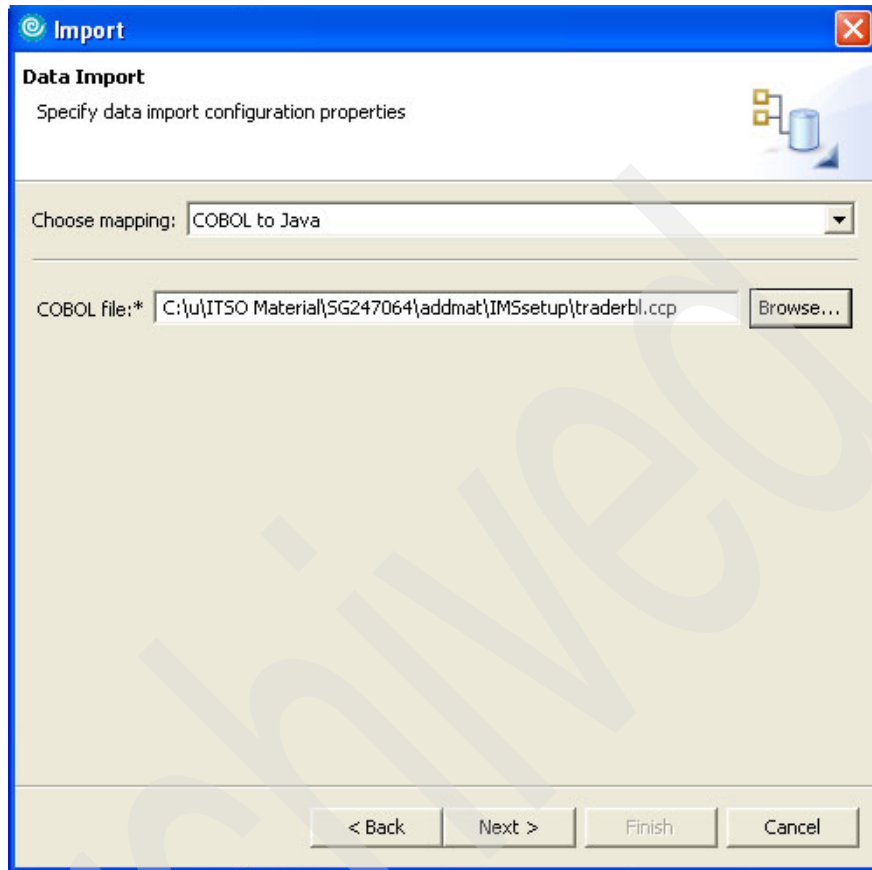


Figure 7-6 Import COBOL source for COBOL to Java mapping

3. We click the **Next** button to go to the Importer window. In this window we selected **z/OS** as the target platform and queried the COBOL source and located the IN-BUFFER data structure (see Figure 7-7 on page 217).

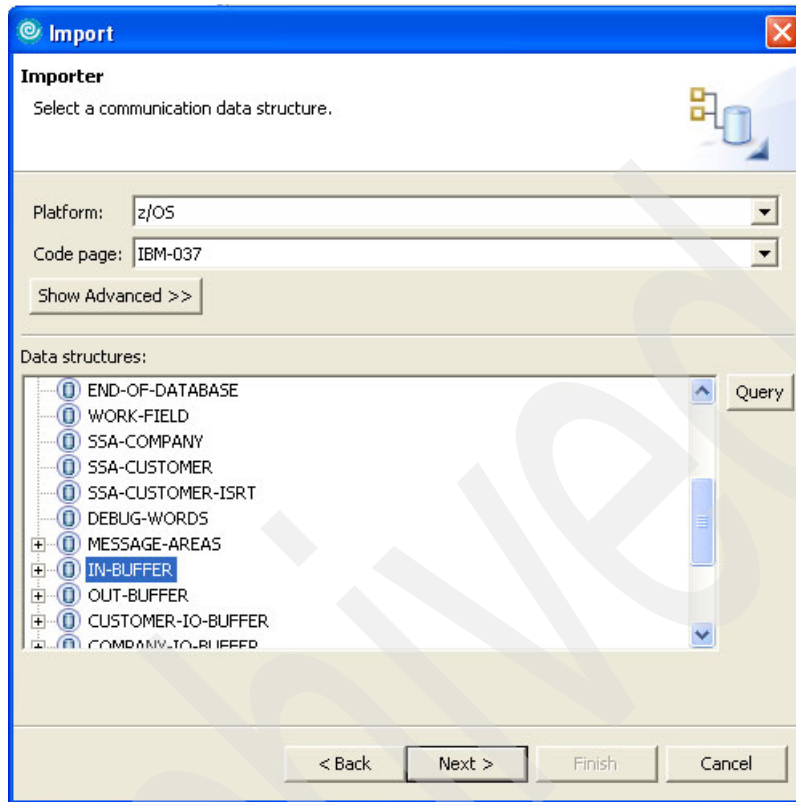


Figure 7-7 Import and parse COBOL source

4. We clicked the **Next** button to go to the Saving Properties window. On this window we selected the `itso.ims.trader` package name in the `TraderIMSEJBClient` project (see Figure 7-8 on page 218). We clicked the **Finish** button to continue.

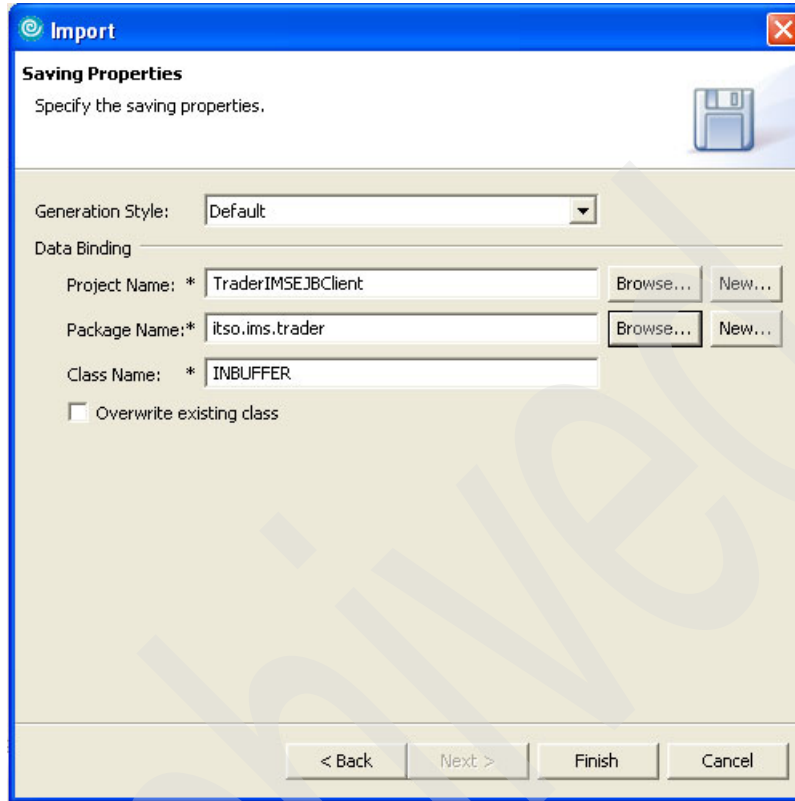


Figure 7-8 Save imported properties

5. We repeated the above steps to import and save the OUT-BUFFER data structure as Java class OUTBUFFER.

7.3 Creating a J2C Java Bean

To do this:

1. We next created a J2C Java Bean. We selected the **TraderIMSEJBClient** project under Other Projects in the Project Explorer pane and then clicked the right mouse button. We selected **New - Other...** to open the Select a wizard window, where we selected **J2C Java Bean** under the J2C folder (see Figure 7-9 on page 219). We clicked **Next** to continue.

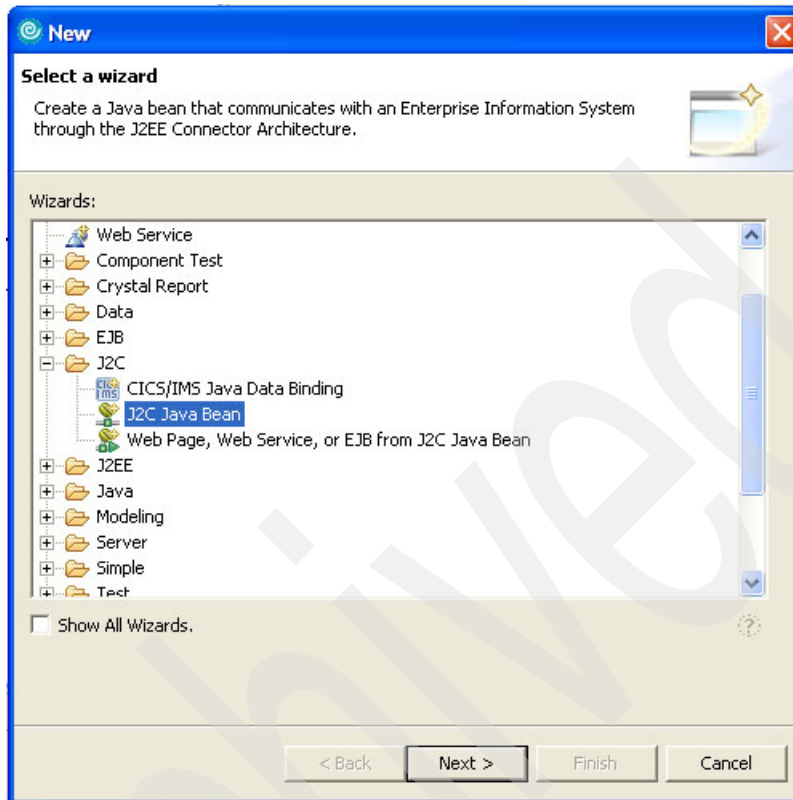


Figure 7-9 Select the J2C Java Bean wizard

2. On the Resource Adapter Selection window we chose the **IMS Connector for Java J2CA 1.5 adapter** (see Figure 7-10 on page 220) and clicked **Next**.

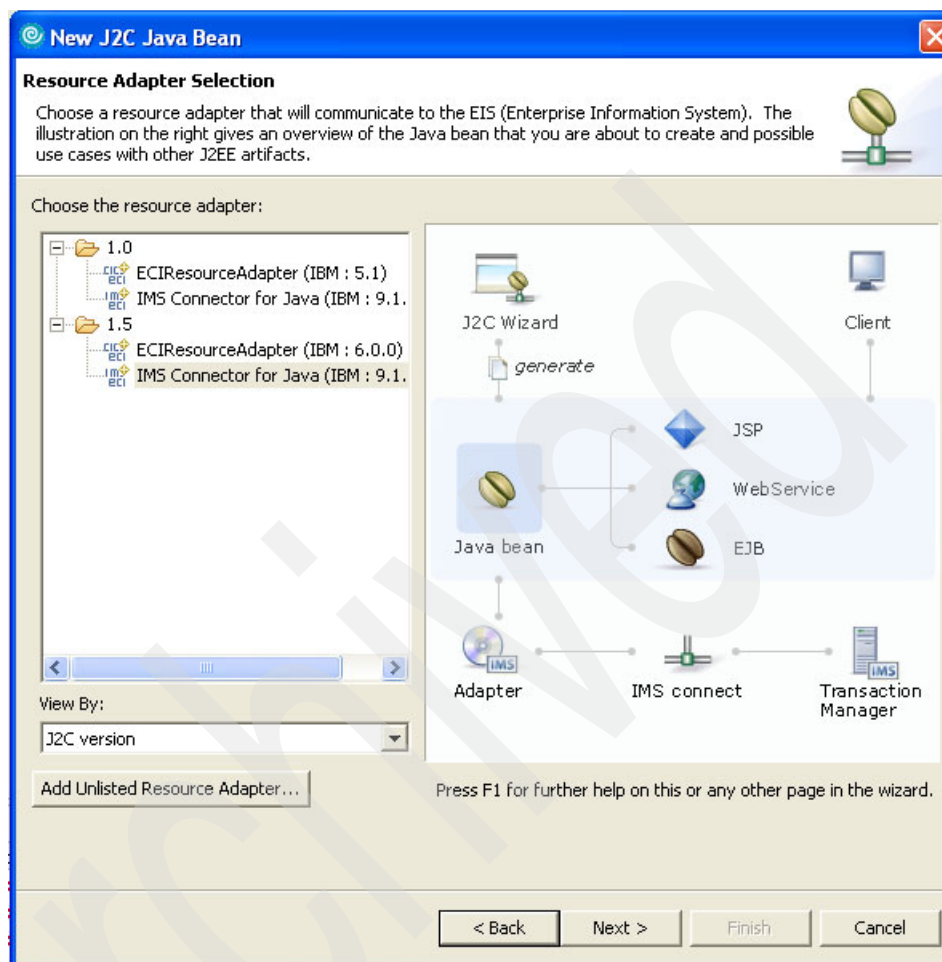


Figure 7-10 Resource Adapter Selection

3. On the Connection Properties (see Figure 7-11 on page 221) we entered `eis/IMSRemote` as the JNDI lookup name and clicked **Next** to continue.

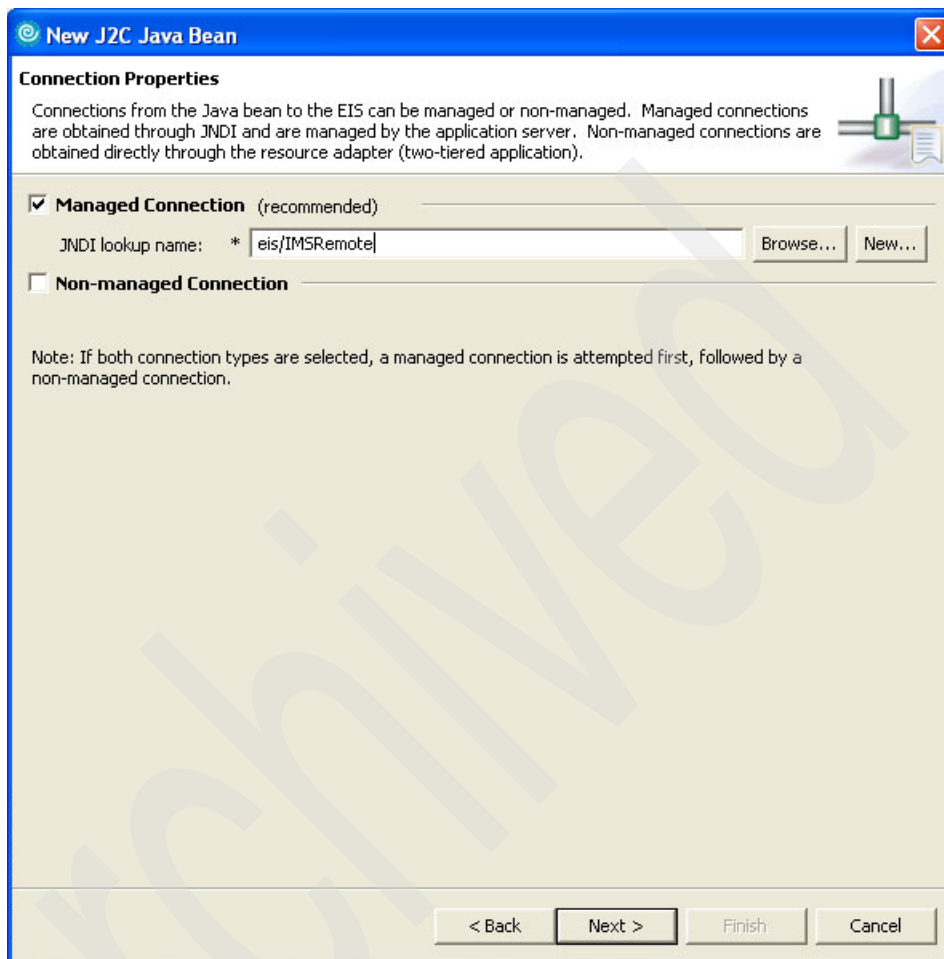


Figure 7-11 J2C Java Bean Connection Properties

Note: The JNDI lookup name is maintained in the J2C Java Bean using the following doclet code:

```
/**
 * @j2c.connectionFactory jndi-name="eis/IMSRemote"
 * @generated
 */
```

Attention: We had previously defined an IMS J2C connection factory in our local WebSphere test server with a name of IMS and an JNDI name of eis/IMSRemote. When we specified eis/IMSRemote as the JNDI lookup name, a direct lookup (versus an indirect lookup using a resource reference in the deployment descriptor) of a J2C connection factory is generated in the J2C Java Bean. A direct lookup of a J2C connection factory is not a best practice and has been deprecated.

We chose to go ahead and generate a direct JNDI lookup in the J2C Java Bean and change to an indirect lookup later when we generated EJB and SOA versions of our J2C Java Beans.

The Browse and New buttons provide an opportunity to either retrieve an existing J2C connection factory from your local test server or create a new J2C connection factory in your test server configuration.

4. We clicked **Next** to go the J2C Java Bean Output Properties window, where we specified itso.ims.trader as the package name and Traderb_J2C_Bean as the interface name (see Figure 7-12 on page 223).

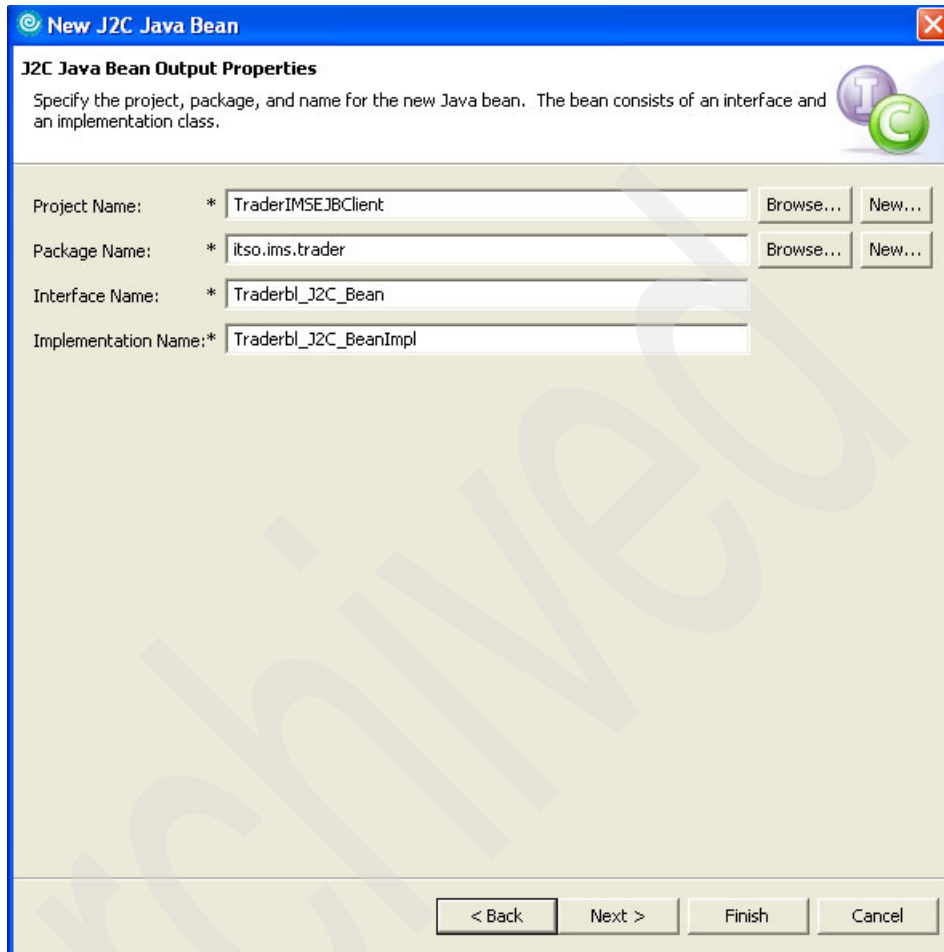


Figure 7-12 J2C Java Bean Output Properties

5. We clicked **Next** to go to the Add Java Method window, where we clicked the **Add** button. We entered `runTraderbl` as the method name (see Figure 7-13 on page 224).

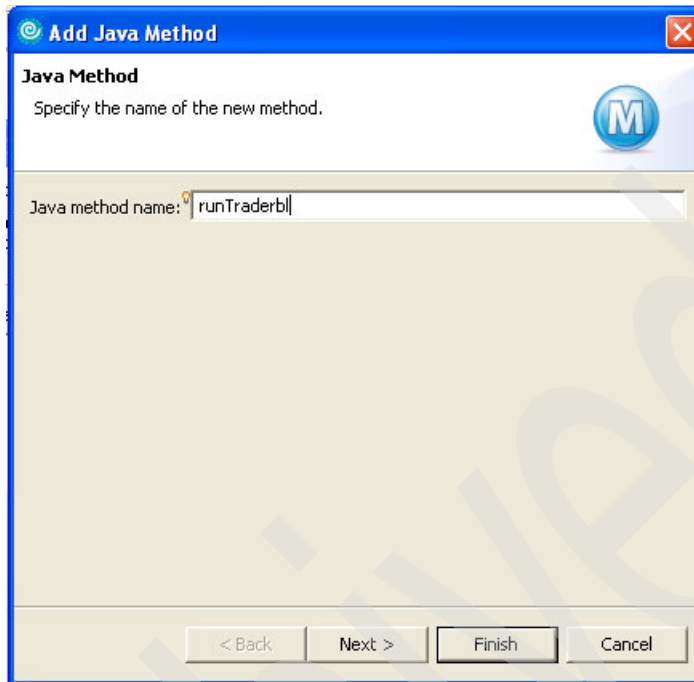


Figure 7-13 Add a method to a J2C Java Bean

6. We clicked **Next** in order to add our INBUFFER and OUTBUFFER data beans to the runTraderbl method signature. In the next window we clicked the **Browse** buttons beside the Input type and Output type and selected **INBUFFER** data bean as the input type and **OUTBUFFER** data bean as the output type; see Figure 7-14 on page 225.

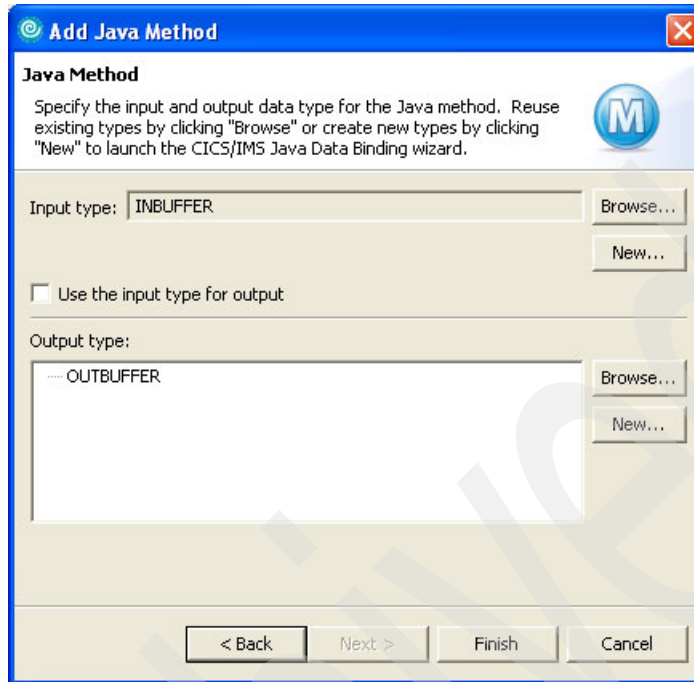


Figure 7-14 Adding Data Beans to a J2C Java Bean

7. We clicked the **Finish** button and reviewed the properties of our J2C Java Bean (see Figure 7-15 on page 226) and clicked **Finish** to complete the building of our J2C Java Bean.

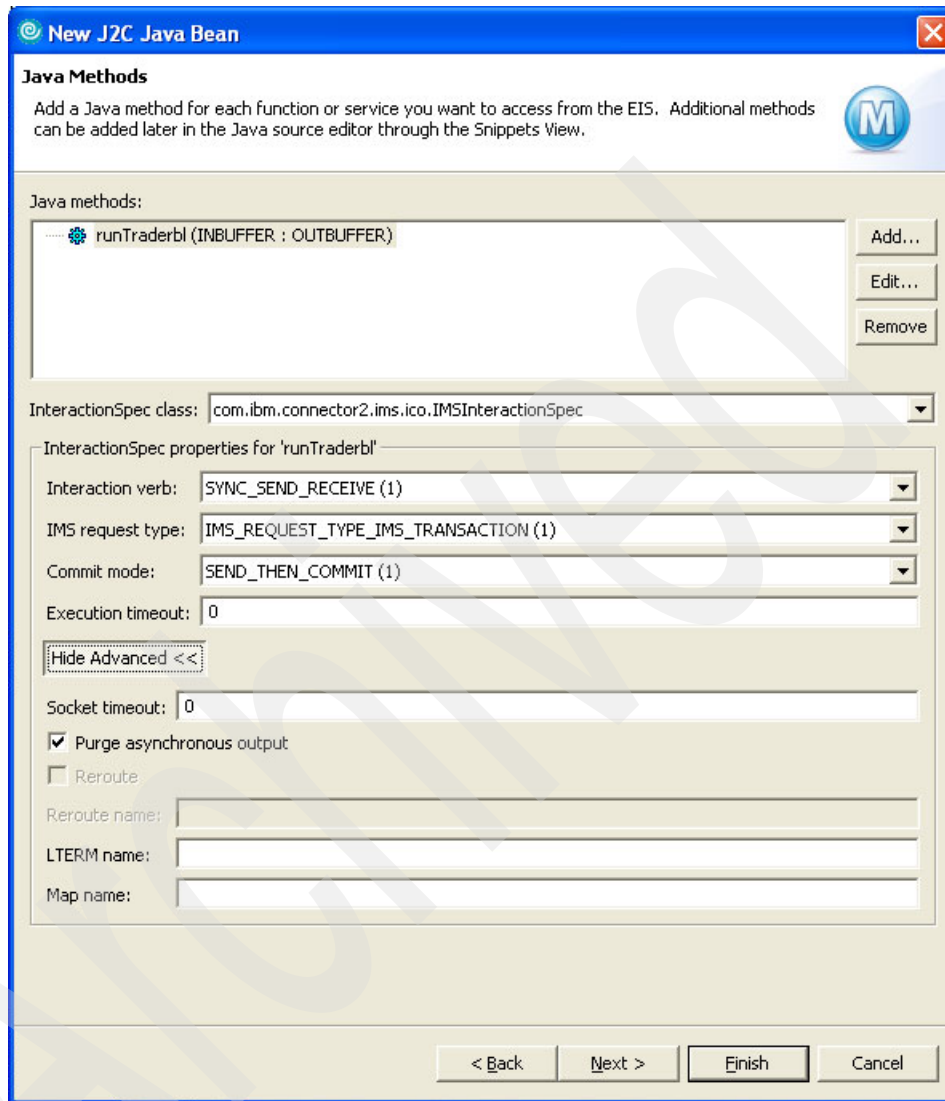


Figure 7-15 J2C Java Bean properties

8. The wizard now generates two Java classes in the TraderIMSEJBClient project, the interface Java class (Traderbl_J2C_Bean) and the implementation Java class (Traderbl_J2C_BeanImpl). It is the implementation class that contains the J2C CCI code.

7.4 Creating an EJB J2C Java Bean

To do this:

1. We selected the **Traderbl_J2C_BeanImpl** java code in the TraderIMSEJBClient project under Other Projects in the Project Explorer pane and then clicked the right mouse button. We selected **New - Other...** to open the Select a wizard window where we selected Web Page, Web services, or EJB from J2C Java Bean wizard under the J2C folder. We clicked **Next** to continue until the Deployment Information window was displayed (see Figure 7-16).

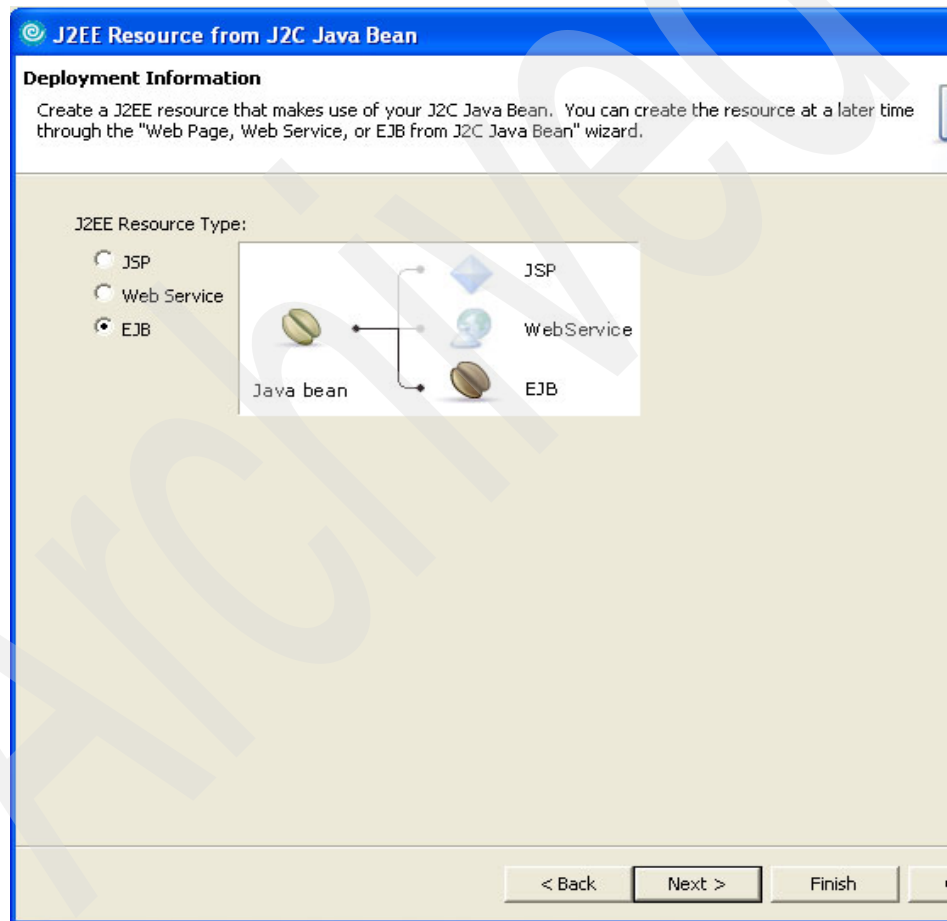


Figure 7-16 J2C Java Bean Deployment Information

2. We ensured EJB was selected and clicked **Next**.

3. On the J2EE Resource from J2C Java Bean window we enter Traderbl as the Bean name and under Advanced options we entered IMS as the resource reference name (see Figure 7-17). We clicked **Finish**.

J2EE Resource from J2C Java Bean

EJB Creation

J2C Java bean has to be generated directly into the EJB project because it is required for EJB doclets support. If the selected EJB project has a separate EJB Client project all the data type binding classes will be generated into the EJB Client project.

EJB project: TraderIMSEJB New...

EAR Project: TraderITSOEJB New...

Session bean name: Traderbl

Session type: Stateless

Transaction type: Container

JNDI Name: ejb/itso/ims/trader/TraderblHome

☒ Remote client view

Remote home interface: itso.ims.trader.TraderblHome

Remote interface: itso.ims.trader.Traderbl

☐ Local client view

Hide Advanced <<

Resource Reference: IMS

JNDI lookup name: eis/IMSRemote Browse... New...

< Back Next > Finish Cancel

Figure 7-17 J2EE Resource from J2C Java Bean

4. Since we were creating an EJB based on a J2C Java Bean that was not currently in an EJB project, we received a message that our J2C Java Bean would be copied to the EJB project (see Figure 7-18 on page 229).

Note: The above step creates a resource in the EJB deployment descriptor that can subsequently be modified to change security management, transactional attributes, and other characteristics of the connection to the resource.

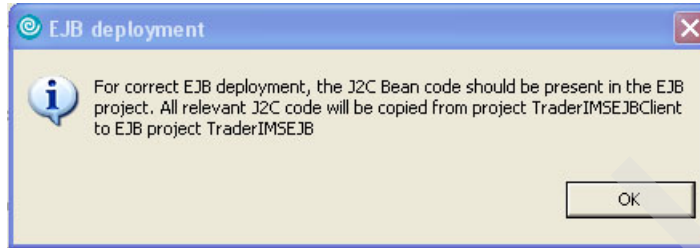


Figure 7-18 EJB deployment message

Note: The J2C doclet tag in the implementation class created in the EJB project differs from the original by the addition of EJB-related information and the changed JNDI name.

```
/**
 * @j2c.connectionFactory jndi-name="IMS"
 * @generated
 * @ejb.session name="Traderbl" type="Stateless"
 "jndi-name="ejb/itso/ims/trader/TraderblHome"view-type="remote"
 transaction-type="Container"
 * @ejb.home remote-class="itso.ims.trader.TraderblHome"
 * @ejb.interface remote-class="itso.ims.trader.Traderbl"
 * @ejb.resource-ref res-ref-name=IMS
 res-type=javax.resource.cci.ConnectionFactory res-auth=Application
 jndi-name=eis/IMSRemote
 */
```

Testing the EJB J2C Java Bean

We followed these steps to test the EJB J2C Java Bean:

1. We added the TraderIMSEJB project to our local test server.
2. We selected the **itso.ims.trader** package in the TraderIMSEJBClient project under Other Projects in the Project Explorer pane and then clicked the right mouse button. We selected **New - Class** to open the Create a new Java class window. We entered a class name of TestTraderblBean and ensured a main method was created; see Figure 7-19 on page 230. We clicked **Finish** to continue.

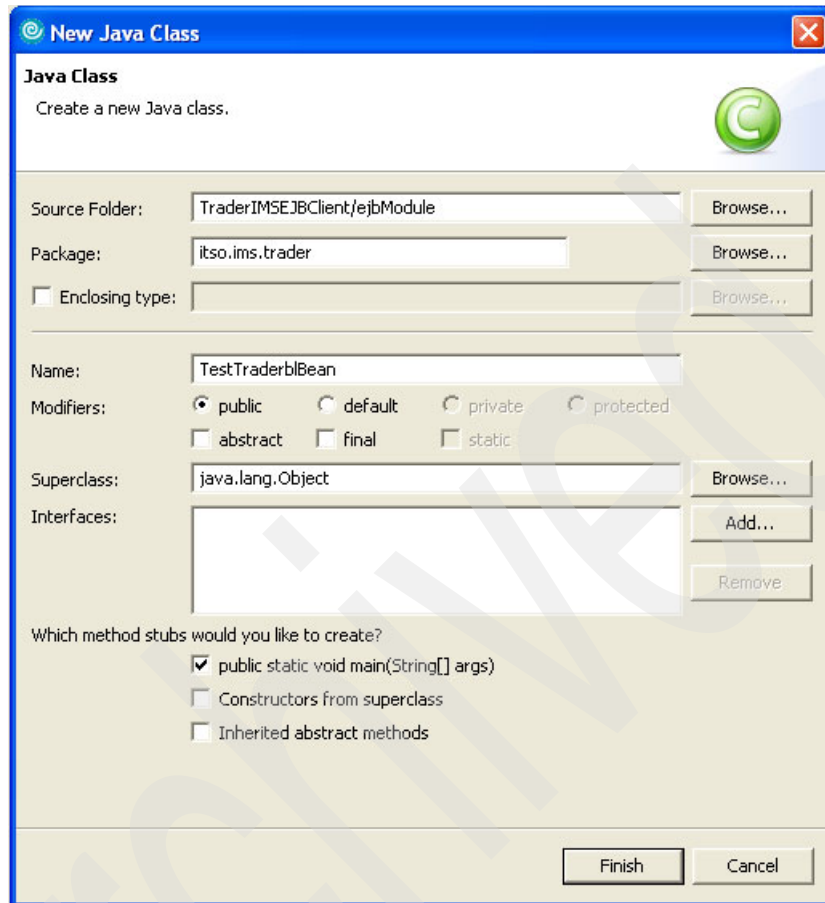


Figure 7-19 New TestTraderblBean Java Class

3. We entered the code shown in Example 7-1 for the TestTraderblBean class and saved the code using a Ctrl+S key sequence.

Example 7-1 TestTraderblBean

```
package itso.ims.trader;

/**
 * @author mitchj
 */
public class TestTraderblBean {

    public static void main(String[] args) {
        try {
            OUTBUFFER outbuffer = new OUTBUFFER();
```

```

INBUFFER inbuffer = new INBUFFER();
inbuffer.setIn__ll((short)386);
inbuffer.setIn__trcd("TRADERBL");
inbuffer.setRequest__type("Get_Company    ");
//    create initial context
javax.naming.InitialContext ctx = new javax.naming.InitialContext();
//    lookup TraderHome
Object obj = ctx.lookup("ejb/itso/ims/trader/TraderblHome");
//    narrow to TraderHome and create Trader
TraderblHome traderHome =
(TraderblHome) javax.rmi.PortableRemoteObject
.narrow((org.omg.CORBA.Object) obj, TraderblHome.class);
Traderbl trader = traderHome.create();
outbuffer = trader.runTraderbl(inbuffer);
for (int i = 0; i <4; i++) {
    System.out.println(outbuffer.getCompany__name__tab(i));
}
} catch (Exception e) {
    e.printStackTrace();
}
}

```

4. Next we selected the down arrow next to the Run wizard on the toolbar and selected **Run** on the drop-down list; see Figure 7-20.

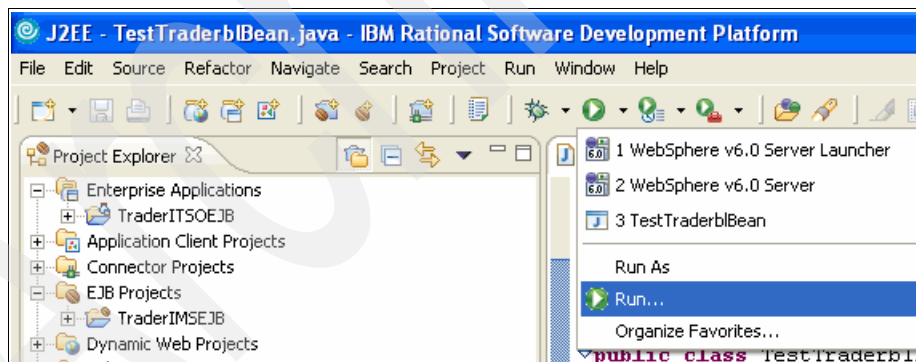


Figure 7-20 Running an application in RAD

5. On the Launch Configuration window, we selected **Java Application** and click **New**.
6. We clicked the **Classpath** tab and selected **User Entries**.

The following steps, which add support for name resolution, are required to execute this code outside of the container environment (Figure 7-21):

1. We clicked **Add External JARs**.
2. We traversed to the RAD runtimes directory ...\\runtimes\\base_v6\\lib and selected **namingclient.jar**.
3. We clicked **Open** to add this jar to our client runtime environment.
4. We clicked **Advanced**, selected **Add External Folder**, and clicked **OK**.
5. We traversed to the ...\\runtimes\\base_v6\\property directory and clicked **OK**.
6. We clicked **Apply** to save these changes.

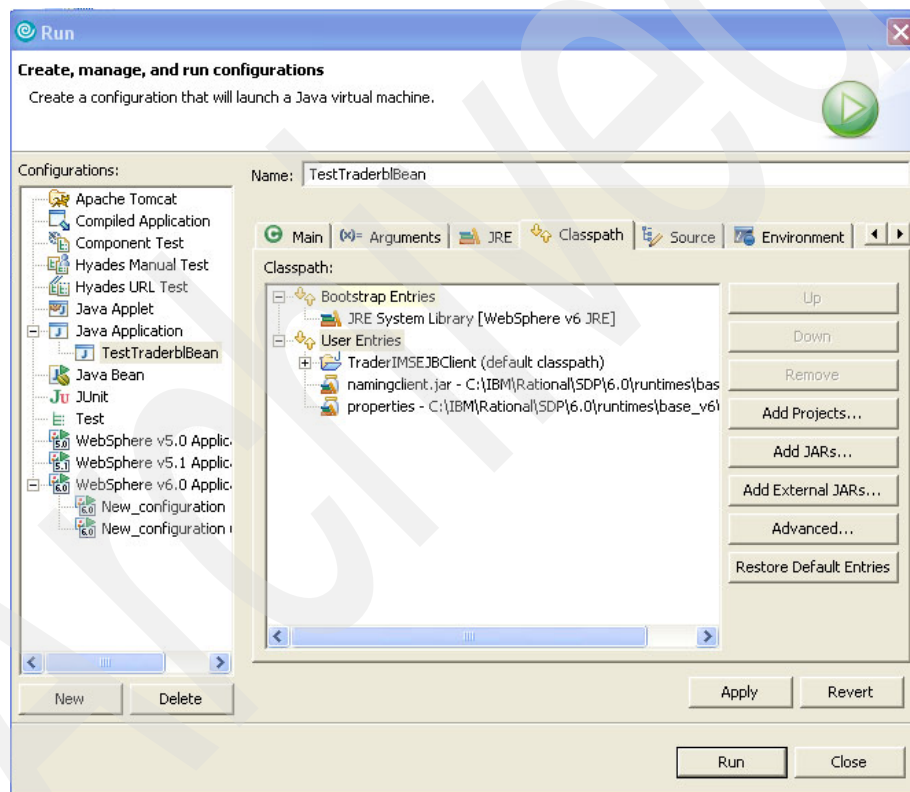


Figure 7-21 Run configuration in RAD for EJB J2C Bean

7. Finally we clicked **Run** (see Figure 7-21) and executed the client code.

The Console window (see Figure 7-22 on page 233) shows the results of our Get_Company request.

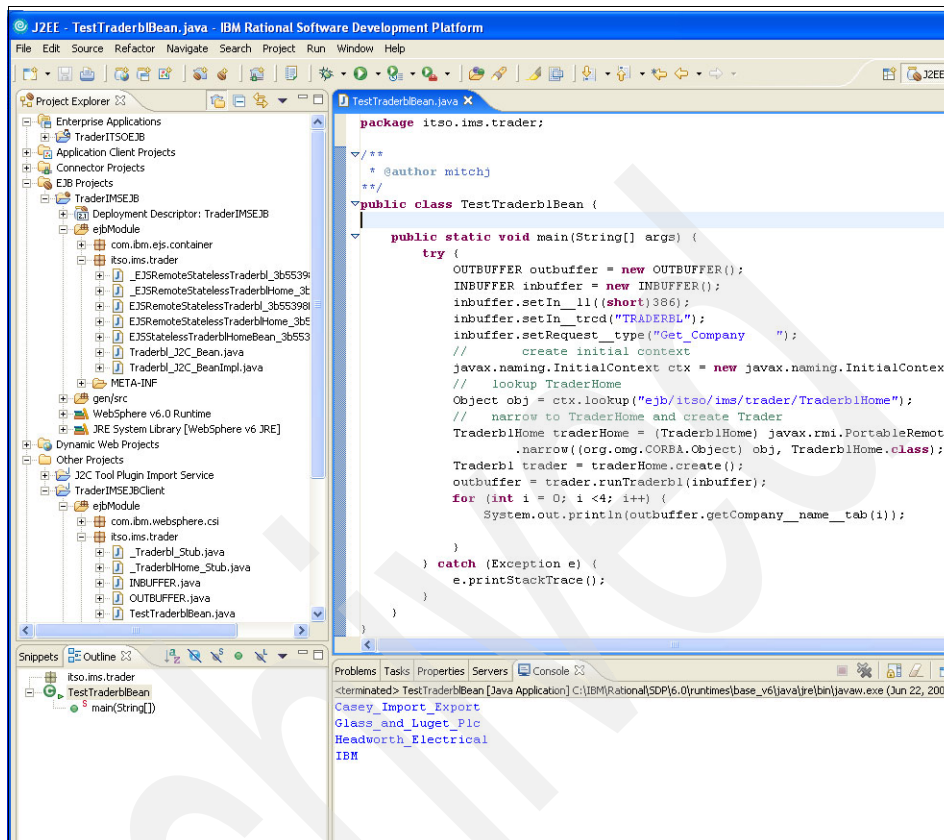


Figure 7-22 Results of testing the EJB J2C bean in RAD

7.4.1 Exposing connection and interaction properties

Next we wanted to provide a user ID and password on our requests to IMS. User IDs and passwords are properties of the Connection object and are specified by an application in a ConnectionSpec object when the connection is established or requested. The J2C Java Bean by default does not expose these properties. We had to modify the J2C Java Bean by adding J2C doclet tags to the source of the J2C Java Bean. It is the saving of the source with our new doclet tags that will generate the code that allows the application to set these properties.

See 9.7.1, “Connection properties” on page 318, and 9.7.2, “Interaction properties” on page 318, for a complete list of IMS ConnectionSpec and InteractionSpec properties, respectively.

1. We located the J2C doclet tags prior to the runTraderbl method in the J2C Java Bean (Traderbl_J2C_Bean). By default, the runTraderbl method creates

a generic ConnectionSpec object and an IMSInteractionSpec object (see Example 7-2).

Example 7-2 runTraderbl method

```
/**
 * @j2c.interactionSpec class="com.ibm.connector2.ims.ico.IMSInteractionSpec"
 * @generated
 * @ejb.interface-method view-type="remote"
 */
public itso.ims.trader.OUTBUFFER runTraderbl(itso.ims.trader.INBUFFER arg)
    throws javax.resource.ResourceException {
    ConnectionSpec cs = getConnectionSpec();
    InteractionSpec is = interactionSpec;
    if (is == null) {
        is = new com.ibm.connector2.ims.ico.IMSInteractionSpec();
    }
    itso.ims.trader.OUTBUFFER output = new itso.ims.trader.OUTBUFFER();
    invoke(cs, is, arg, output);
    return output;
}
```

2. We changed the J2C meta data by adding j2c.connectionSpec and j2c.connectionSpec-property tags (see the doclet tags in Example 7-3). When we saved the code by pressing Ctrl+S the runTraderbl method was regenerated and now creates an IMSConnectionSpec and the setter methods for the UserName and Password connection properties (see Example 7-3).

Example 7-3 runTraderbl method with IMSConnectionSpec properties

```
/**
 * @j2c.interactionSpec class="com.ibm.connector2.ims.ico.IMSInteractionSpec"
 * @j2c.connectionSpec class="com.ibm.connector2.ims.ico.IMSConnectionSpec"
 * @j2c.connectionSpec-property name="UserName" argumentBinding="ivUserID"
 * @j2c.connectionSpec-property name="Password" argumentBinding="ivPassword"
 * @generated
 * @ejb.interface-method view-type="remote"
 */
public itso.ims.trader.OUTBUFFER runTraderbl(itso.ims.trader.INBUFFER arg, String ivUserID,
String ivPassword) throws javax.resource.ResourceException {
    ConnectionSpec cs = getConnectionSpec();
    if (cs == null) {
        cs = new com.ibm.connector2.ims.ico.IMSConnectionSpec();
        ((com.ibm.connector2.ims.ico.IMSConnectionSpec) cs).setUserName(ivUserID);
        ((com.ibm.connector2.ims.ico.IMSConnectionSpec) cs).setPassword(ivPassword);
    }
    InteractionSpec is = interactionSpec;
    if (is == null) {
        is = new com.ibm.connector2.ims.ico.IMSInteractionSpec();
    }
}
```



```

itso.ims.trader.OUTPUT output = new itso.ims.trader.OUTPUT();
invoke(cs, is, arg, output);
return output;
}

```

3. These changes introduced problems in our code since the variables `ivUserID` and `ivPassword` were undefined. We located the `runTraderbl` method in the Outline pane (see Figure 7-23).

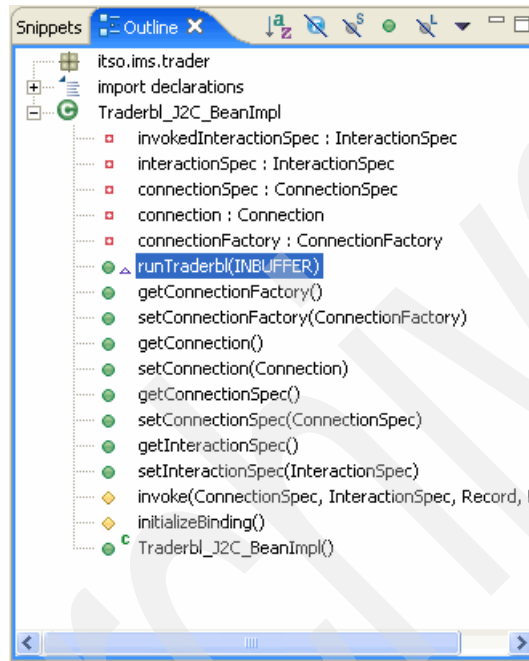


Figure 7-23 Traderbl_J2C_BeanImpl outline

4. We selected the `runTraderbl` method and right-clicked. We select **Refactor** → **Change Method Signature**. We replied Yes to the message about continuing and added two String parameters (`ivUserID` and `ivPassword`) to the method so it would now accept the user ID and password as parameters (see Figure 7-24 on page 236).

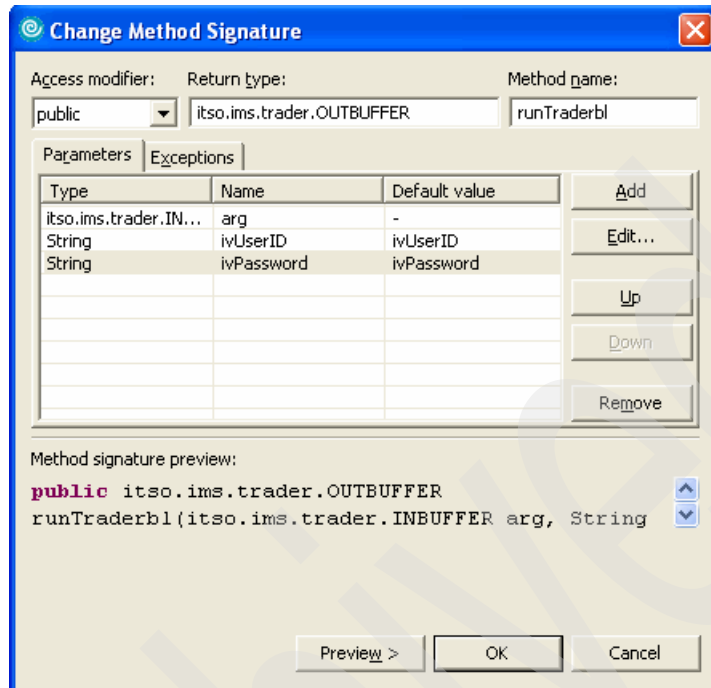


Figure 7-24 Change Method Signature

5. This changed the method signature for the runTraderbl method in the J2C Java Bean and it also changed the method signature of any invocation of this method in any other EJB in the same project. We found this latter behavior very useful.
6. Finally, we had to ensure the EJB deployment descriptor for the TraderIMSEJB project specified application managed authentication for our IMS resource referenced.
7. We republished our TraderIMSEJB project, added the Username and Password parameters to our TestTraderblBean Java class, and successfully retested with our application-provided user ID and password.

7.5 Generating SOA J2C Java Beans

First, we created a Enterprise Application project to manage out SOA application. We followed these steps:

1. We opened the J2EE perspective.

2. We launched the Create an Enterprise Application project wizard by clicking the icon shown in Figure 7-25.

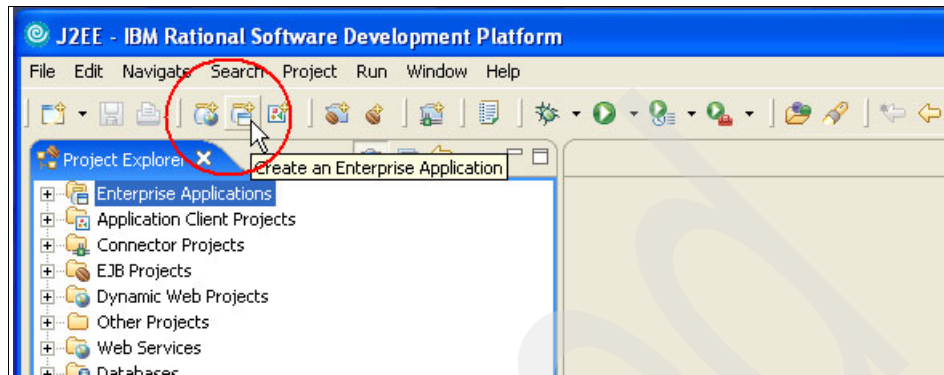


Figure 7-25 Create an Enterprise Application Project wizard

3. On the New Enterprise Project window that opened, we entered TraderITS0S0A as the project name and clicked **Finish**.

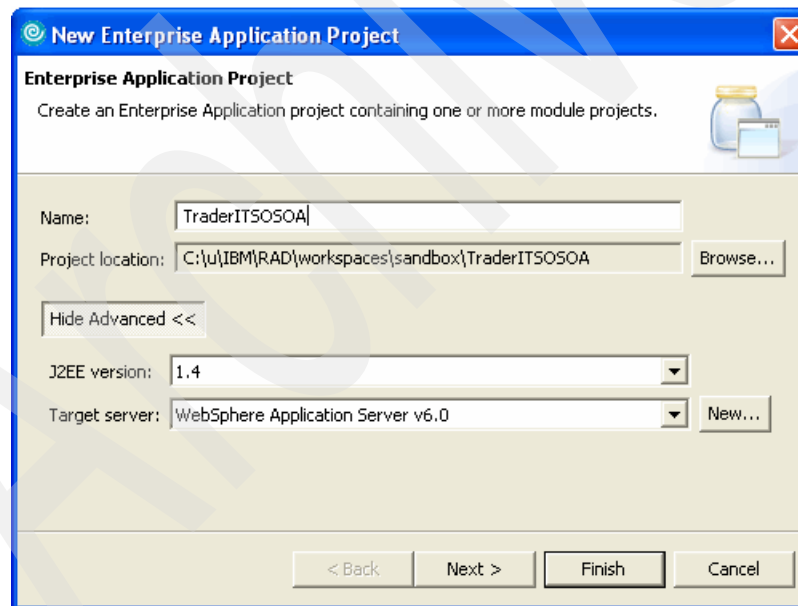


Figure 7-26 Create an Enterprise Application Project

4. Next we added a dynamic Web project to our enterprise application by launching the Create an WEB project wizard icon; see Figure 7-27 on page 238.

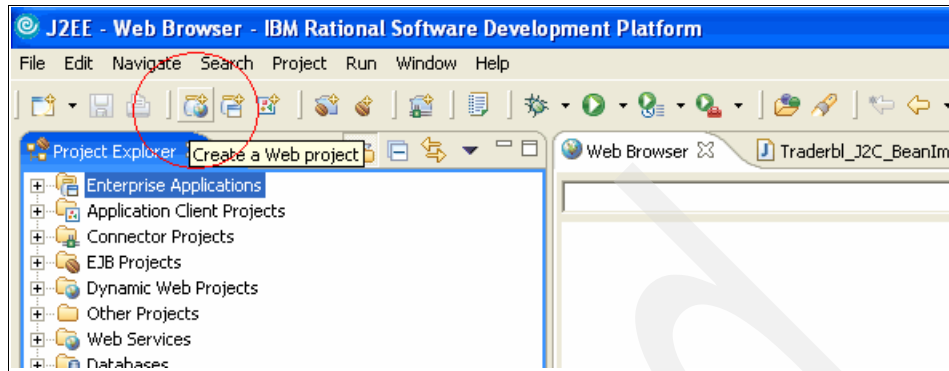


Figure 7-27 Create a Web project wizard

5. On the New Dynamic Web Project window that opened, we entered TraderIMSS0A as the name of the new project. We added this project to the Enterprise project we just created. We clicked **Finish** to continue (see Figure 7-28 on page 239).

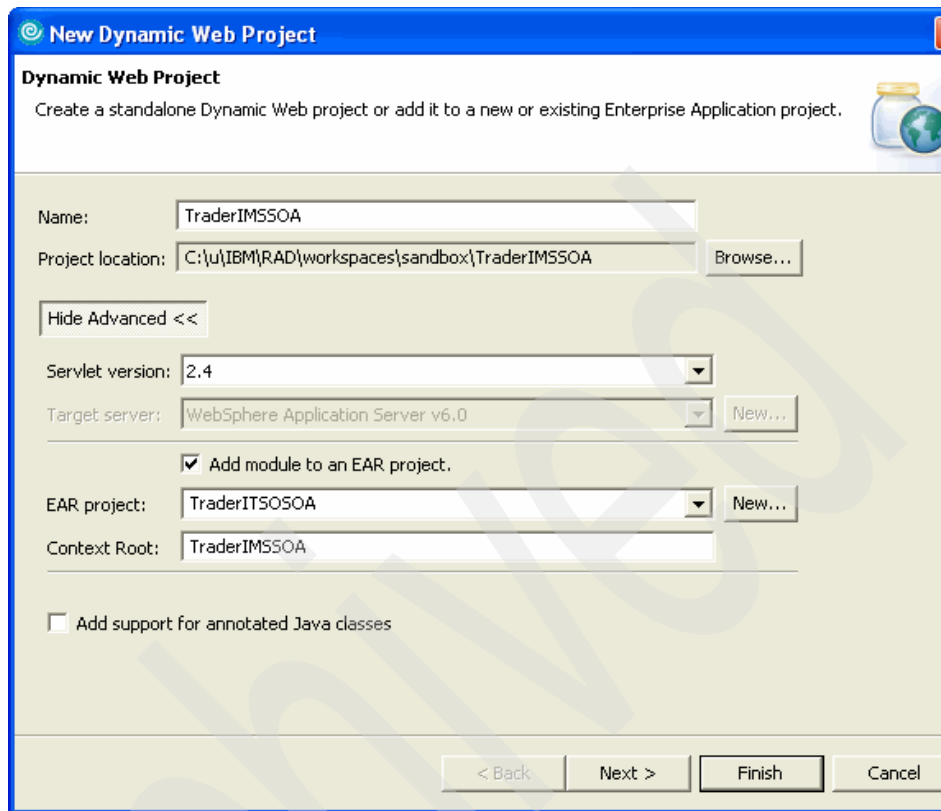


Figure 7-28 New Dynamic Web Project

6. We repeated this process to create a *client* dynamic Web project with a name of TraderIMSSOAClient and also added it to the TraderITSOSOA enterprise application project (see Figure 7-29 on page 240).

Note: Normally it is not considered a best practice to include the client project in the same EAR file with the Web service. We did so in this example only for simplification purposes.

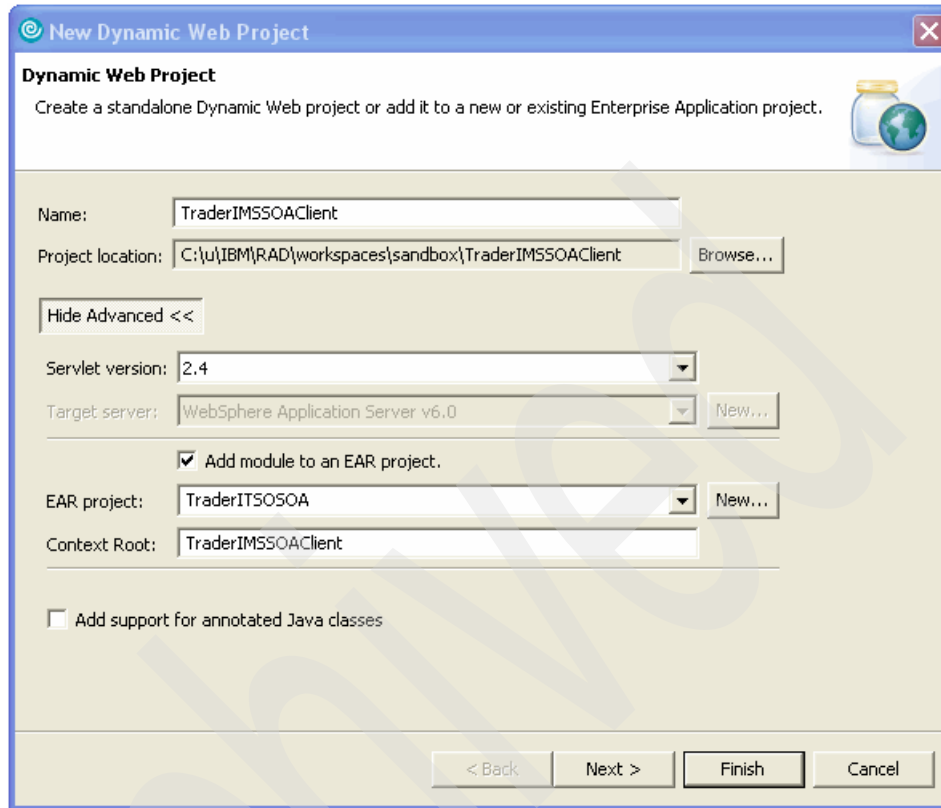


Figure 7-29 New dynamic Web project for client use

7. We next created a Java package for use within the TraderIMSSOA project. We selected the TraderIMSSOA project under Dynamic Web Projects in the Project Explorer pane and then clicked the right mouse button. We selected **New - Package** to open the New Java Package window, where we entered `itso.ims.trader` as the package name (see Figure 7-30 on page 241). We clicked **Finish** to continue.

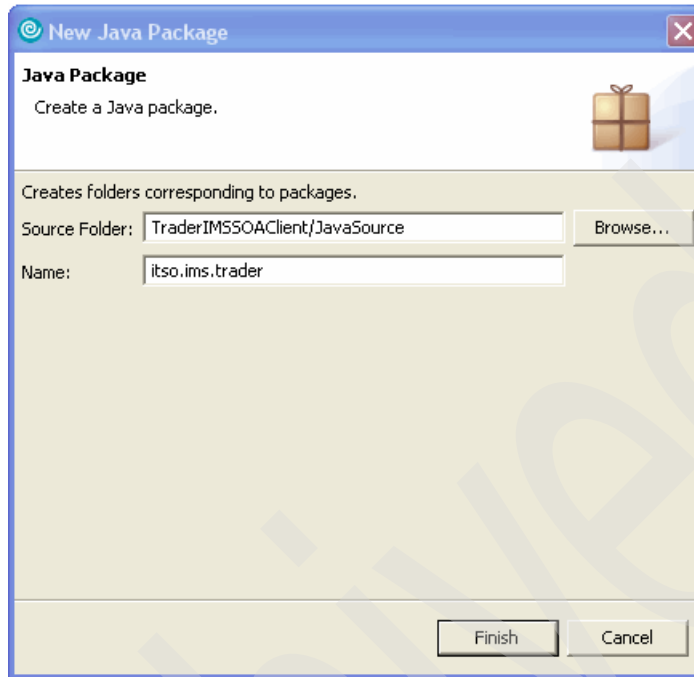


Figure 7-30 New Java Package

7.5.1 Create J2C Java Data Beans

To do this:

1. Next, we created J2C Java Data binding beans. We selected the **TraderIMSSOA** project under Dynamic Web Projects in the Project Explorer pane and then clicked the right mouse button. We selected **New - Other...** to open the Select a wizard window, where we selected **CICS/IMS Java Data Binding** under the J2C folder (see Figure 7-31 on page 242). We clicked **Next** to continue.

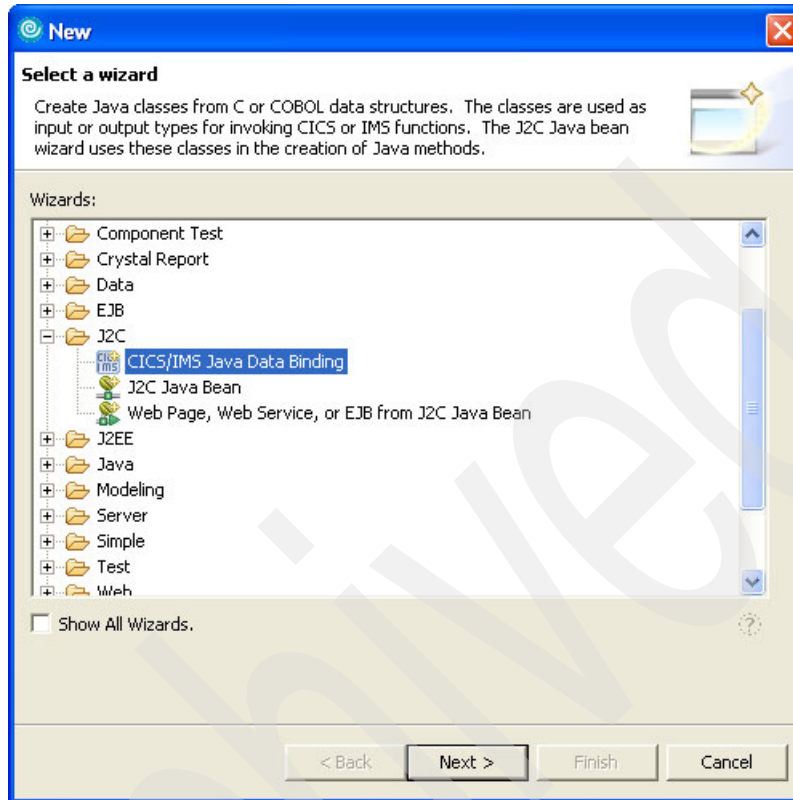


Figure 7-31 CICS/IMS Java Data Binding wizard

2. On the Data Import window we selected **COBOL to Java** mapping and clicked the **Browse** button to traverse to the source of the target COBOL program (see Figure 7-32 on page 243).

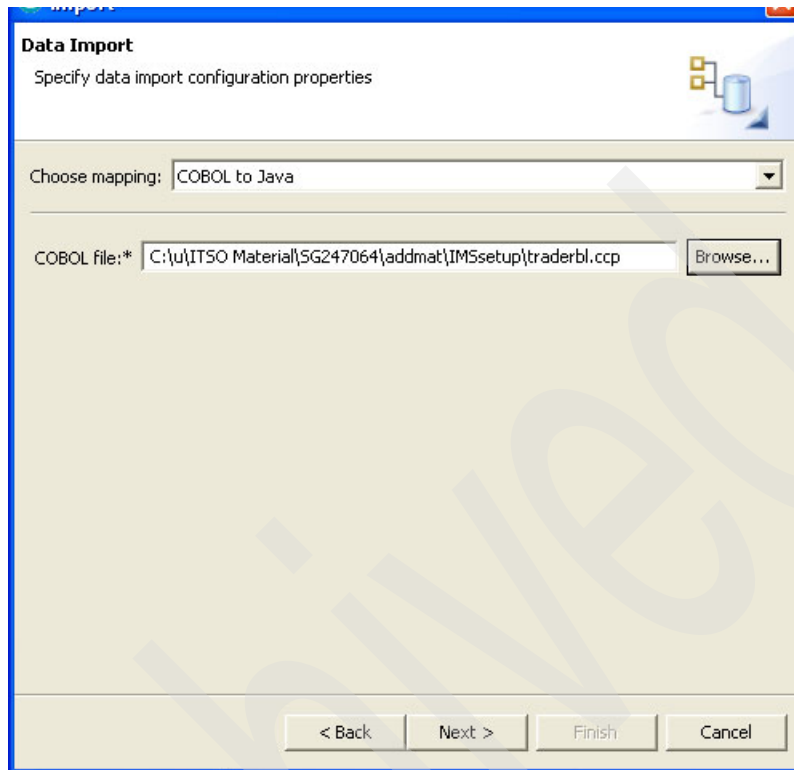


Figure 7-32 Import COBOL source for COBOL to Java mapping

3. We clicked the **Next** button to go to the Importer window. On this window we selected **z/OS** as the target platform and queried the COBOL source and located the IN-BUFFER data structure (see Figure 7-33 on page 244).

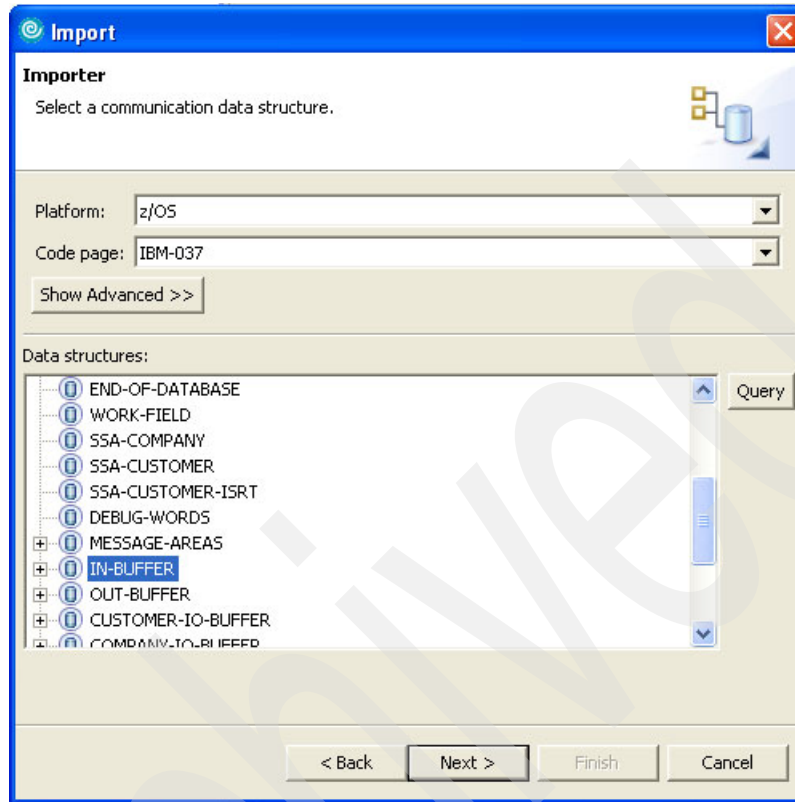


Figure 7-33 Import and parse COBOL source

4. We clicked the **Next** button to go to the Saving Properties window. On this window we selected the **itso.ims.trader** package in the TraderIMSSOA project (see Figure 7-34 on page 245). We clicked the **Finish** button to continue.

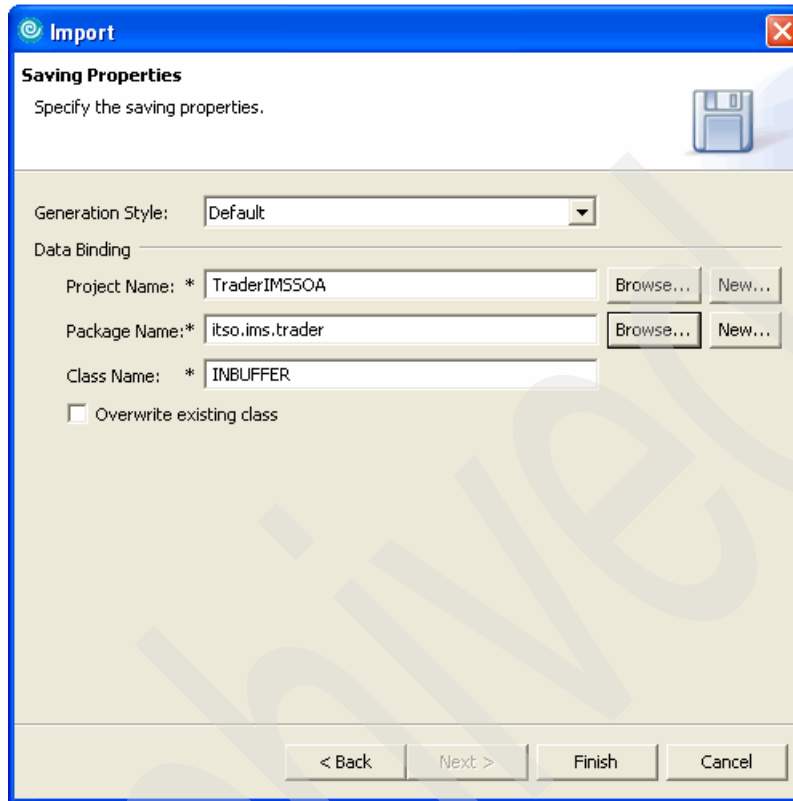


Figure 7-34 Save imported properties

5. We repeated the above steps to import and save the OUT-BUFFER data structure as Java class OUTBUFFER.

7.5.2 Create a J2C Java Bean

To do this:

1. We next created a J2C Java Bean. We selected the **TraderIMSSOA** project under Other Projects in the Project Explorer pane and then clicked the right mouse button. We selected **New - Other...** to open the Select a wizard window, where we selected J2C Java Bean under the J2C folder (see Figure 7-35 on page 246). We clicked **Next** to continue.

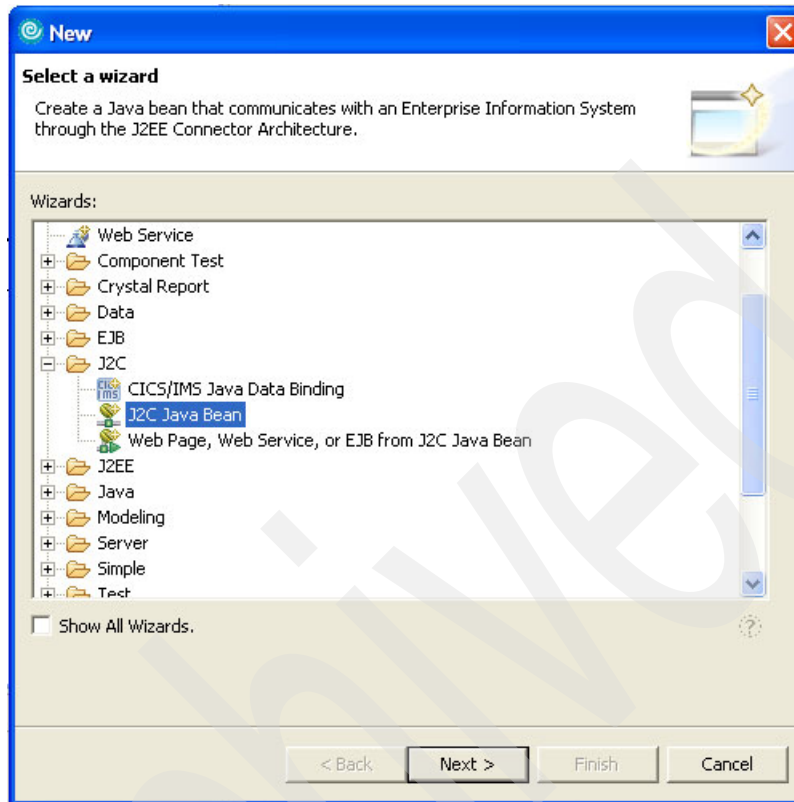


Figure 7-35 J2C Java Bean wizard

2. On the Resource Adapter Selection window we chose the IMS Connector for Java J2CA 1.5 adapter (see Figure 7-36 on page 247) and clicked **Next**.

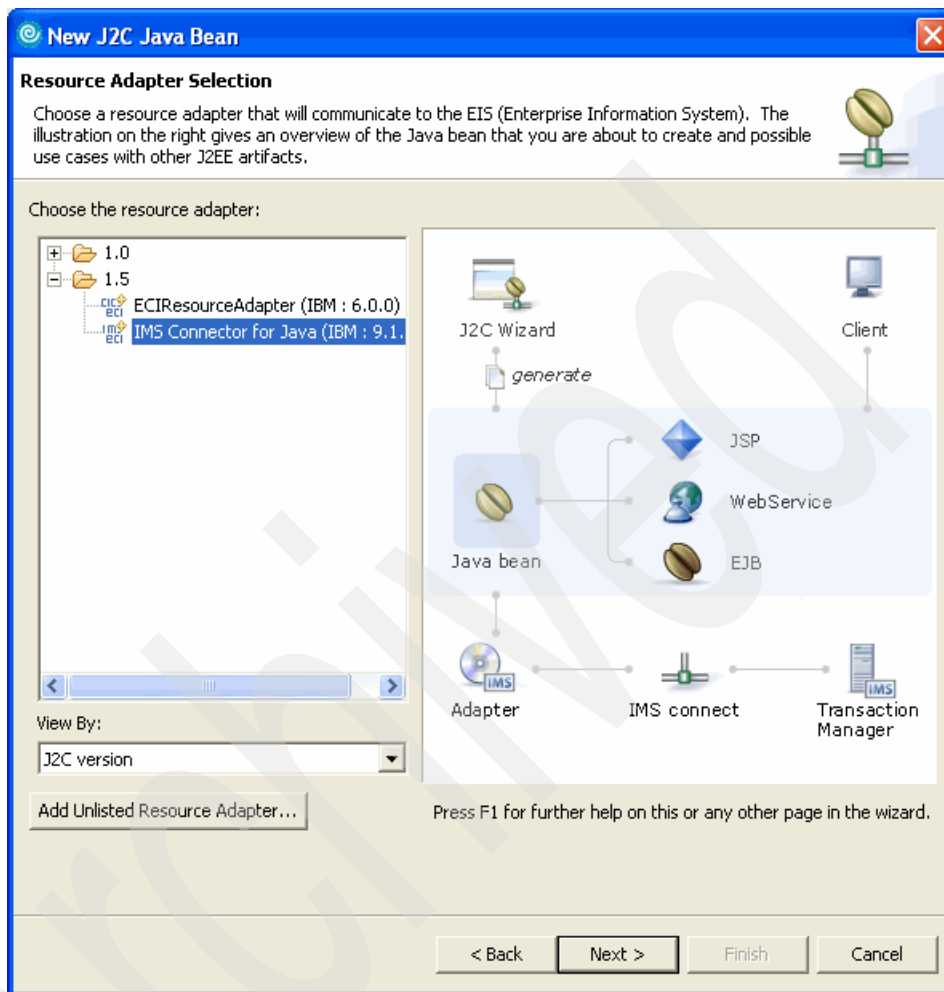


Figure 7-36 Resource Adapter Selection

3. On the Connection Properties (see Figure 7-37 on page 248) we entered `eis/IMSRemote` as the JNDI lookup name and clicked **Next** to continue.

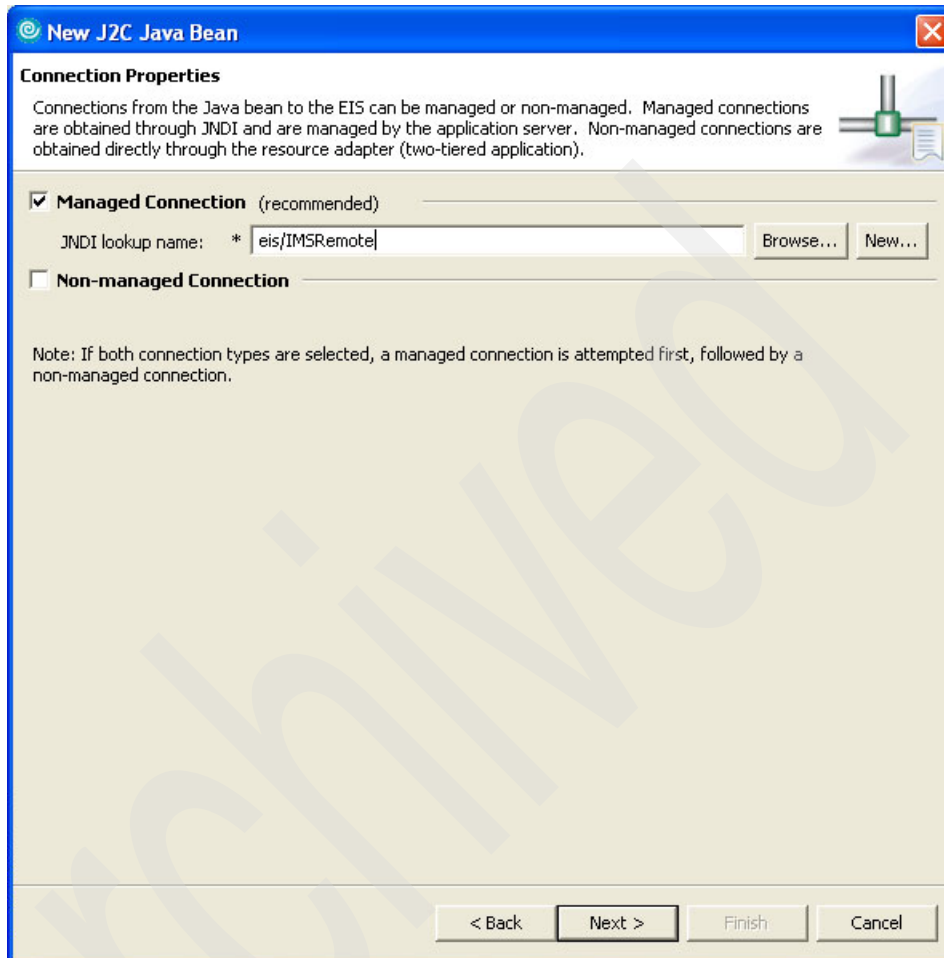


Figure 7-37 J2C Java Bean Connection Properties

Note: The JNDI lookup name is maintained in the J2C Java Bean using the following doclet code.

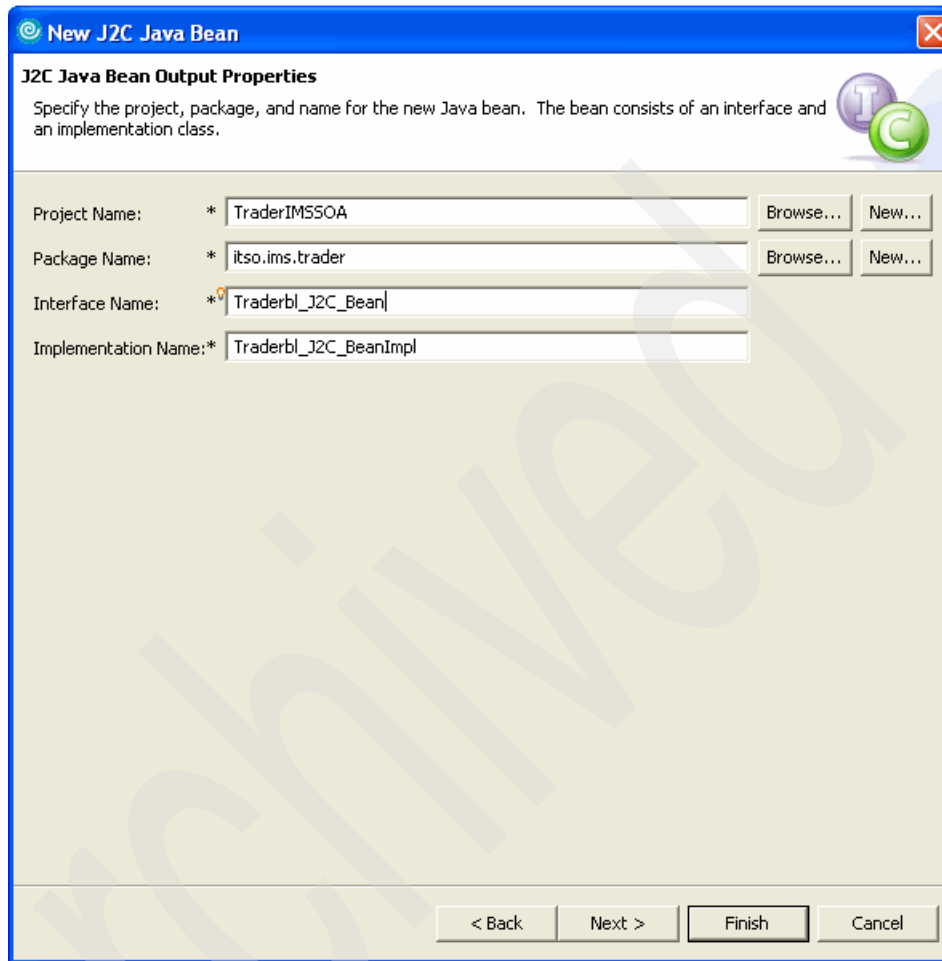
```
/**
 * @j2c.connectionFactory jndi-name="eis/IMSRemote"
 * @generated
 */
```

Attention: We had previously defined an IMS J2C connection factory in our local WebSphere test server with a name of IMS and an JNDI name of eis/IMSRemote. When we specified eis/IMSRemote as the JNDI lookup name, a direct lookup (versus an indirect lookup using a resource reference in the deployment descriptor) of a J2C connection factory is generated in the J2C Java Bean. A direct lookup of a J2C connection factory is not a best practice and has been deprecated.

We chose to go ahead and generate a direct JNDI lookup in the J2C Java Bean and change to an indirect lookup later when we generated an SOA versions of our J2C Java Bean.

The Browse and New buttons provide an opportunity to either retrieve an existing J2C connection factory from your local test server or create a new J2C connection factory in your test server configuration.

4. We clicked **Next** to go the J2C Java Bean Output Properties window where we specified `itso.ims.trader` as the package name and `Traderb_J2C_Bean` as the interface name (see Figure 7-38 on page 250).



The image shows a 'New J2C Java Bean' dialog box with a blue title bar. The main area is titled 'J2C Java Bean Output Properties' and contains instructions: 'Specify the project, package, and name for the new Java bean. The bean consists of an interface and an implementation class.' There are four input fields: 'Project Name' with value 'TraderIMSSOA', 'Package Name' with value 'itso.ims.trader', 'Interface Name' with value 'Traderbl_J2C_Bean', and 'Implementation Name' with value 'Traderbl_J2C_BeanImpl'. Each of the first three fields has 'Browse...' and 'New...' buttons to its right. At the bottom are '< Back', 'Next >', 'Finish', and 'Cancel' buttons.

| Field | Value | Buttons |
|----------------------|-------------------------|------------------|
| Project Name: | * TraderIMSSOA | Browse... New... |
| Package Name: | * itso.ims.trader | Browse... New... |
| Interface Name: | * Traderbl_J2C_Bean | |
| Implementation Name: | * Traderbl_J2C_BeanImpl | |

Figure 7-38 J2C Java Bean Output Properties

5. We clicked **Next** to go to the Add Java Method window where we clicked the **Add** button. We entered runTraderbl as the method name (see Figure 7-39 on page 251).

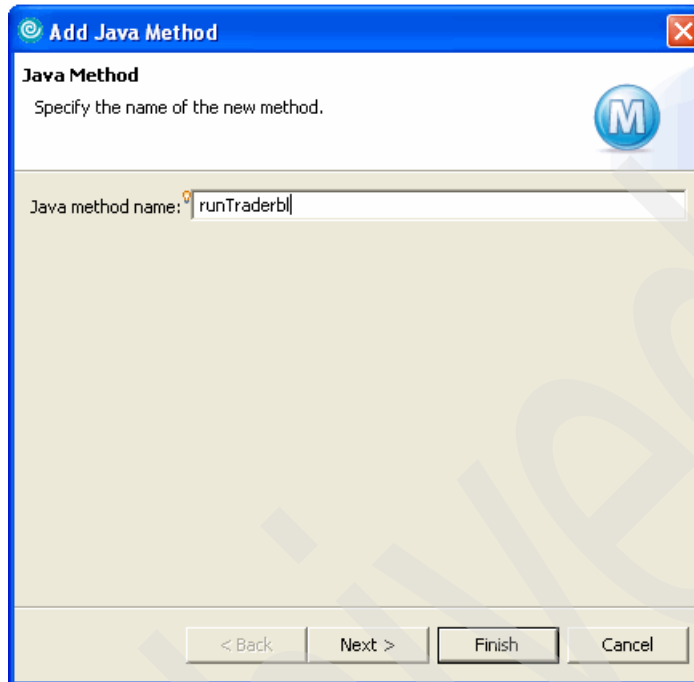


Figure 7-39 Add a Java method to a J2C Java Bean

6. We clicked **Next** in order to add our INBUFFER and OUTBUFFER data beans to the runTraderbl method signature. On the next window we clicked the **Browse** buttons beside the Input type and Output type and selected **INBUFFER** data bean as the input type and **OUTBUFFER** data bean as the output type; see Figure 7-40 on page 252.

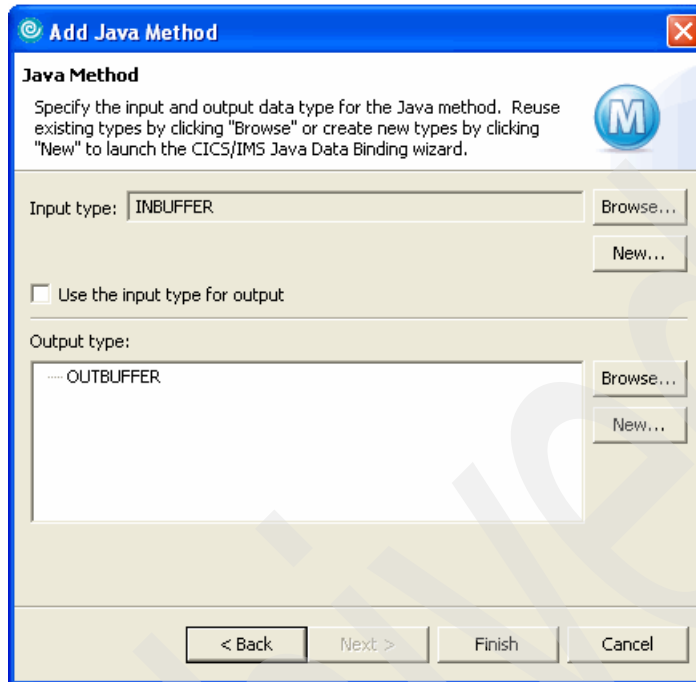


Figure 7-40 Adding data Beans to a J2C Java Bean

7. We clicked the **Finish** button and reviewed the properties of our J2C Java Bean (see Figure 7-41 on page 253), and clicked **Finish** to complete the building of our J2C Java Bean.

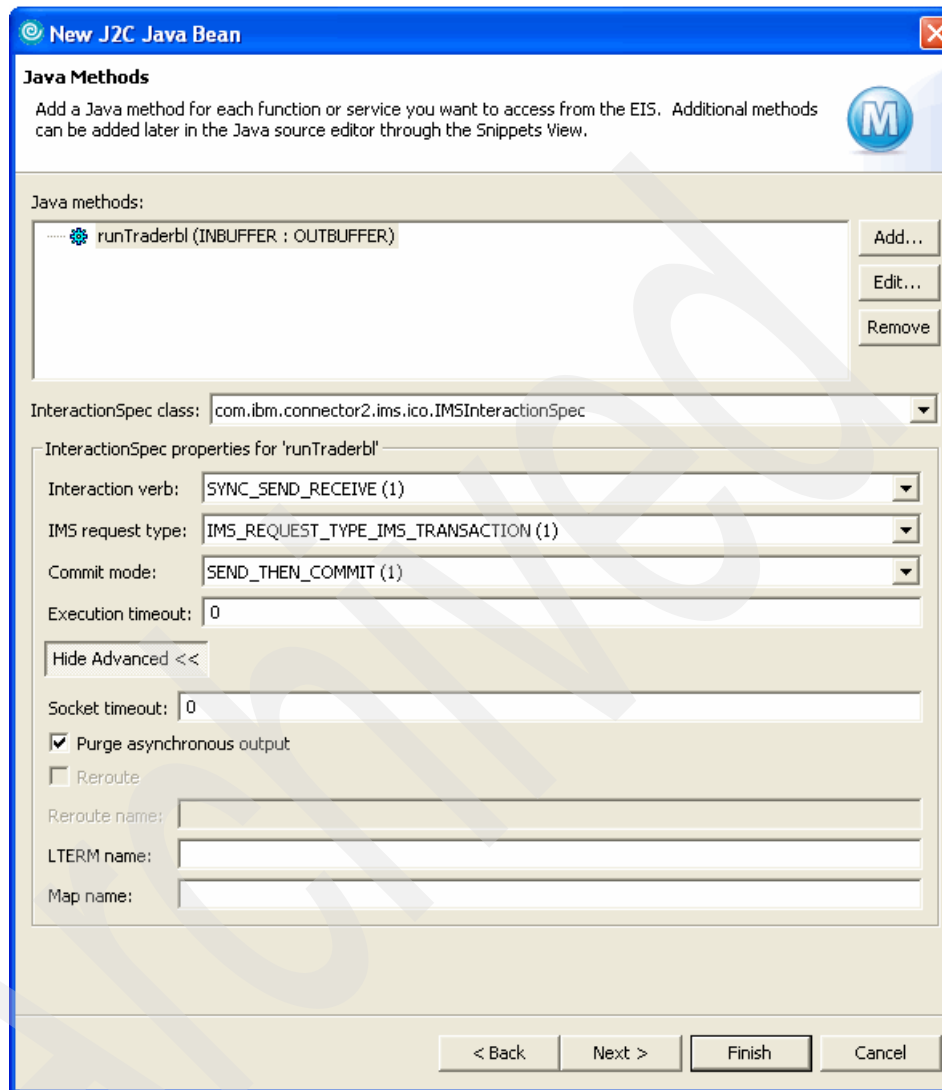


Figure 7-41 J2C Java Bean properties

8. The wizard now generates two Java classes in the TraderIMSEJBClient project, the interface Java class (Traderbl_J2C_Bean) and the implementation Java class (Traderbl_J2C_BeanImpl). It is the implementation class that contains the J2C CCI code.

7.5.3 Creating a SOA J2C Java Bean

To do this:

1. We selected the **Traderbl_J2C_BeanImpl** Java code in the TraderIMSSOA project under Dynamic Web Projects in the Project Explorer pane and then clicked the right mouse button. We selected **New - Other...** to open the Select a wizard window, where we selected Web Page, Web services, or EJB form J2C Java Bean wizard under the J2C folder. We clicked **Next** to continue until the Deployment Information window was displayed (see Figure 7-42).

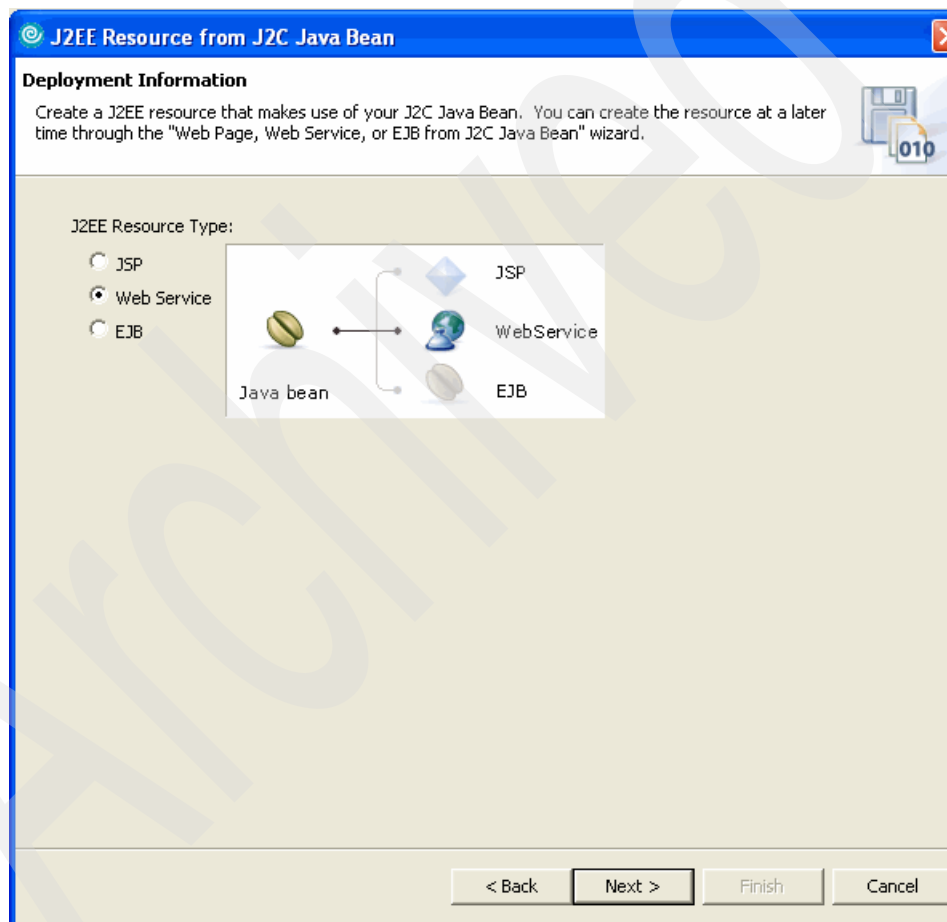


Figure 7-42 J2C Java Bean Deployment Information

2. We ensured Web services was selected and clicked **Next**.

3. On the J2EE Resource from J2C Java Bean window we entered IMS as the resource reference name under Advanced options (see Figure 7-43). We clicked **Finish**.

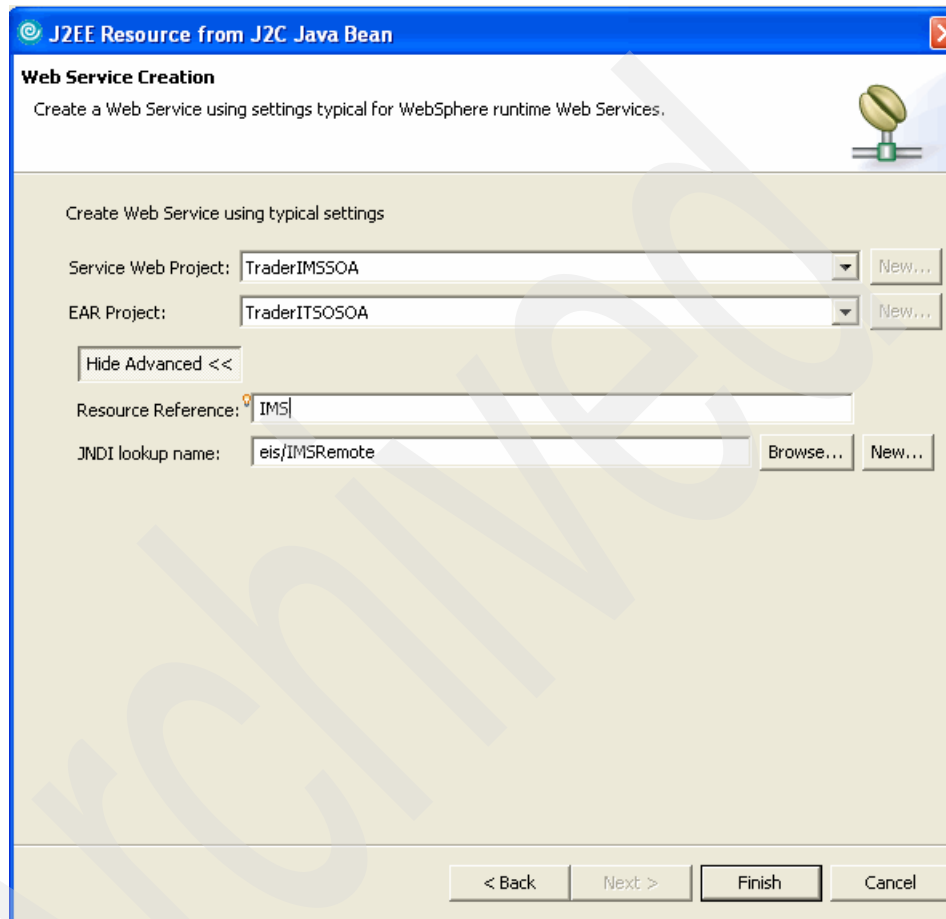


Figure 7-43 SOA J2EE Resource from J2C Java Bean

Note: The above step creates a resource in the Web deployment descriptor, which can subsequently be modified to change security management, transactional attributes, and other characteristics of the connection to the resource.

Note: The J2C doclet tag in the implementation class created in the SOA project differs from the original by containing the changed JNDI name.

```
/**  
 * @j2c.connectionFactory jndi-name="IMS"  
 * @generated
```

7.5.4 Testing the SOA J2C Java Bean

We followed these steps to test the SOA J2C Java Bean:

1. We added the TraderITSOSOA project to our local test server to ensure the project was started.
2. We next expanded the **WebContent** folder in project TraderITSOSOA and the **Services** folder under Web Server (see Figure 7-44 on page 257) to expose the WSDL files created for our SOA J2C Java Bean.

WSDL files now appeared in three places:

- ▶ WEB-INF folder

This copy is used by the server for deployment purposes, but is not accessible to external clients through HTTP (the Servlet specification states that resources contained within the WEB-INF folder are not visible externally).

- ▶ wsdl folder

This copy is accessible to external clients and can therefore be used by a client to obtain all the necessary information about the Web service.

- ▶ An externally visible version of the same WSDL in the WEB-INF folder

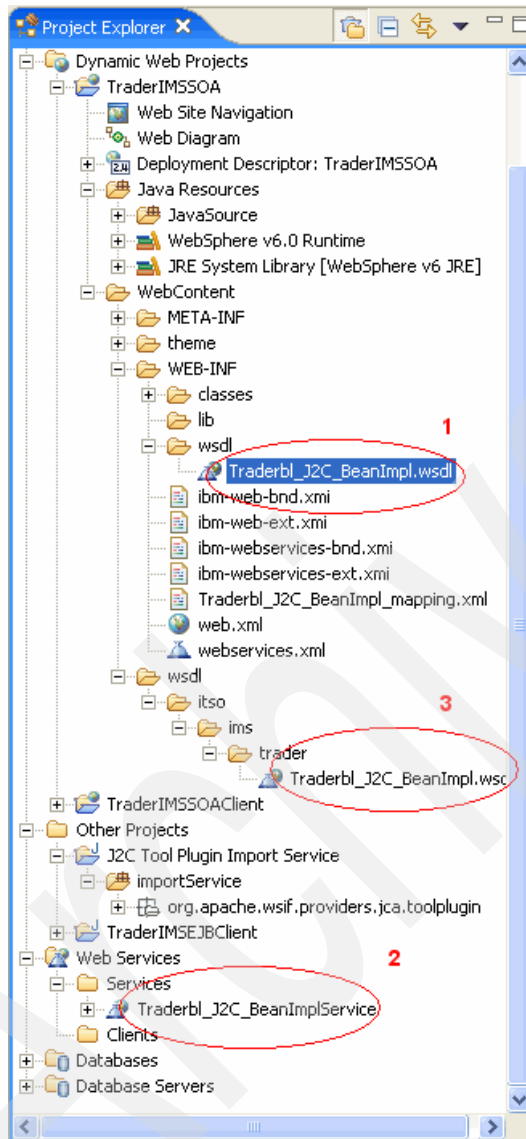


Figure 7-44 Expanded folders

3. We selected the Traderbl_J2C_BeanImplService in the Web Services - Services folder and right-clicked and selected **Generate Client**. This displayed the Web Services Client window, shown in Figure 7-45 on page 258.

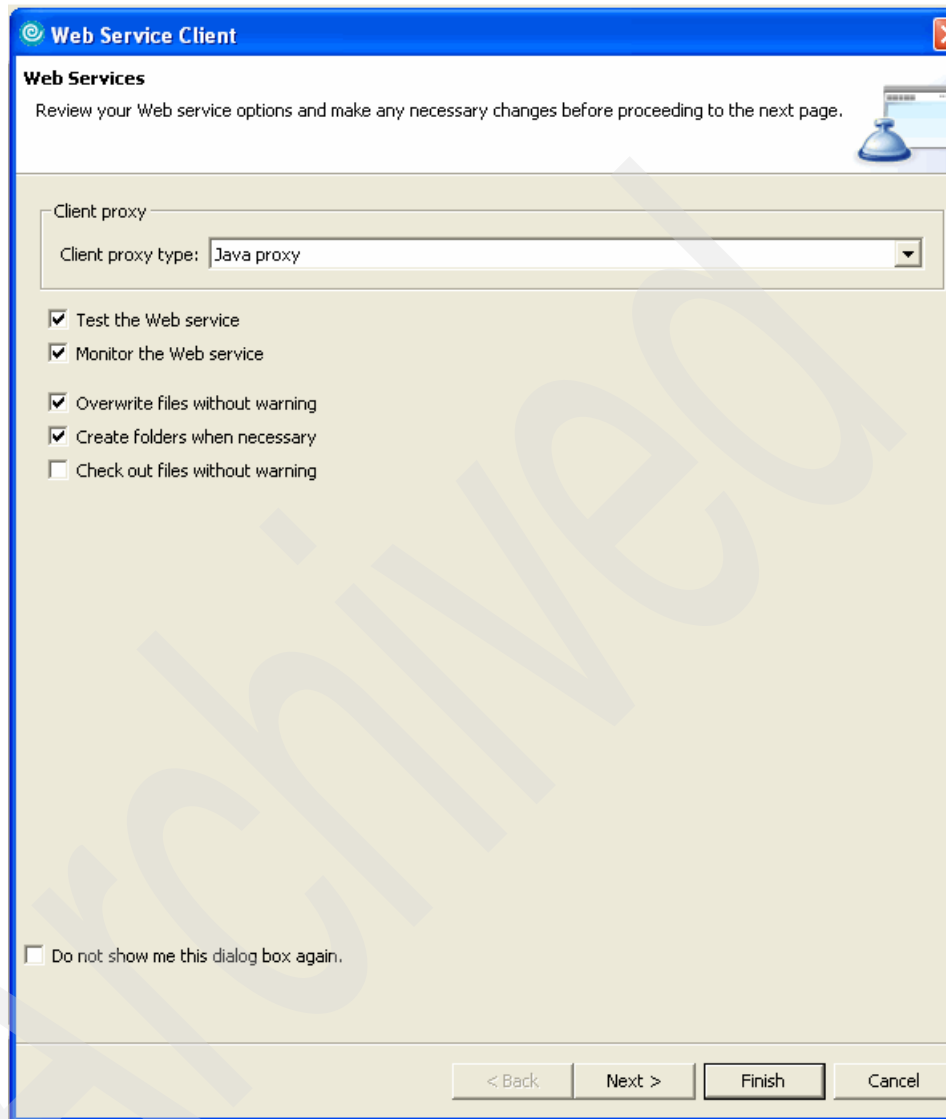


Figure 7-45 Web services generate client

4. In this window we check Test the Web Service and Monitor the Web Service and clicked **Next** until the Client Environment Configuration window was displayed (see Figure 7-46 on page 259).

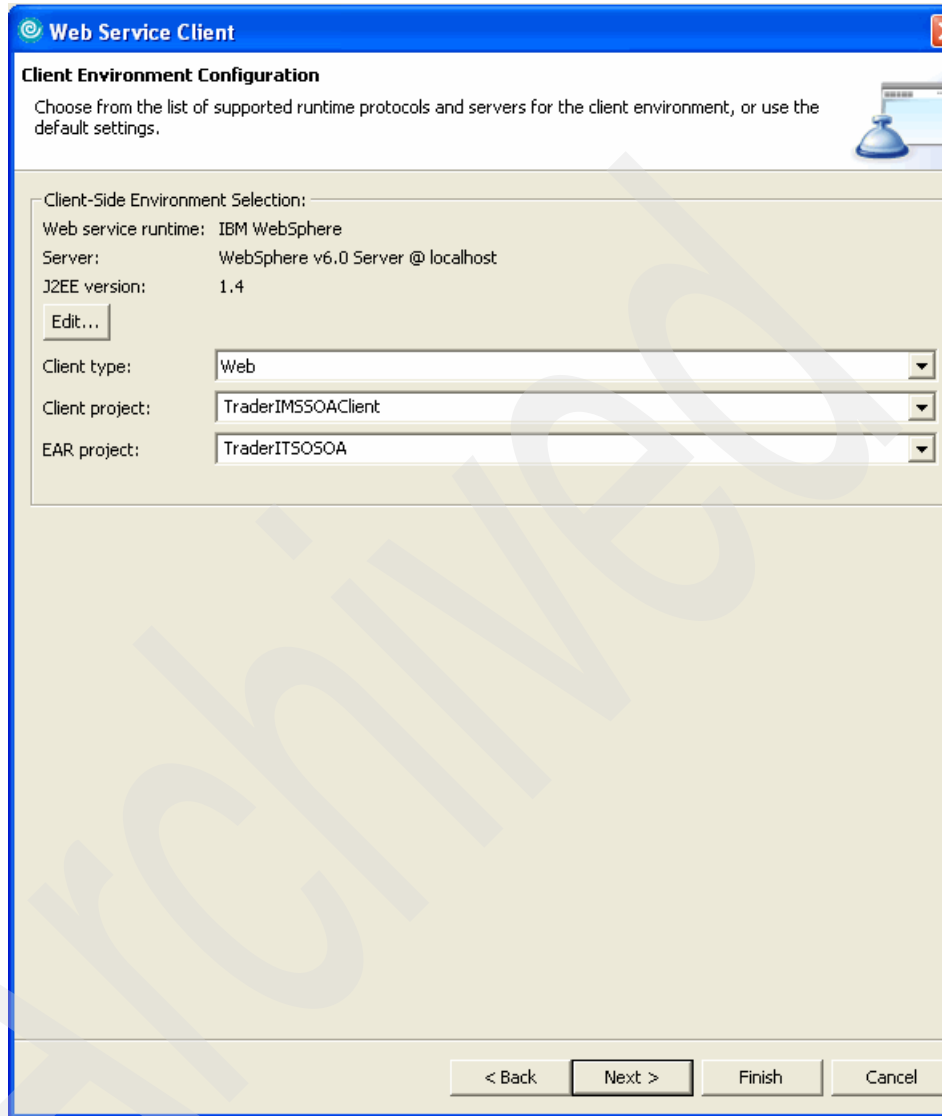


Figure 7-46 Client Environment Configuration

5. We ensured the client project was set to TraderIMSSOAClient and clicked **Finish** to generate a test application consisting of JSPs and proxy code for the Web service. When the generation process was complete the window in Figure 7-47 on page 260 was displayed.

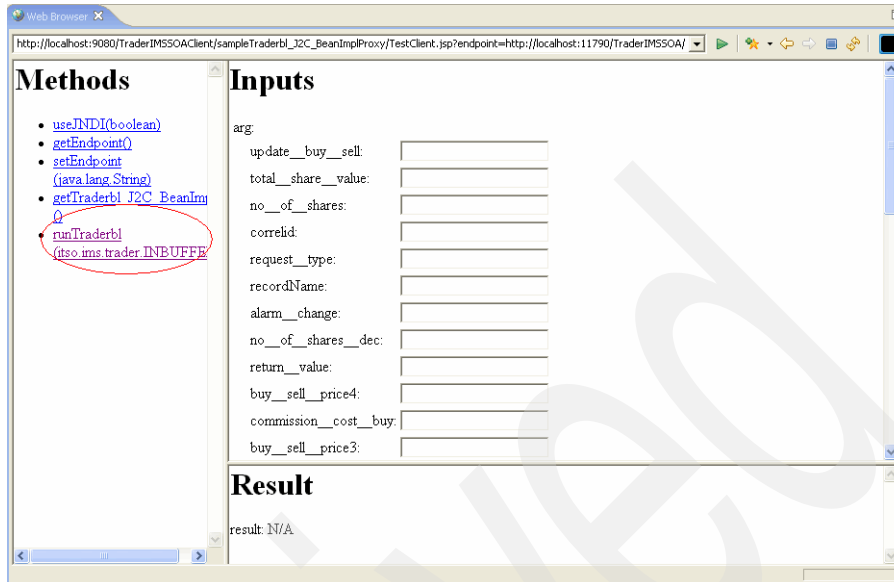


Figure 7-47 Web service test application

6. We clicked the **runTraderbl** method and filled in the following fields in the input record:
 - Get_Company as the value for request_type
 - 0 as the value for no_of_shares_dec
 - TRADERL as the value for in_trcd
 - 386 as the value for in_ll
 - 0 as the value for in-zz
7. We clicked the **Invoke** button (see Figure 7-48 on page 261) and saw the company names returned in the company__name__tab field in the Result pane.

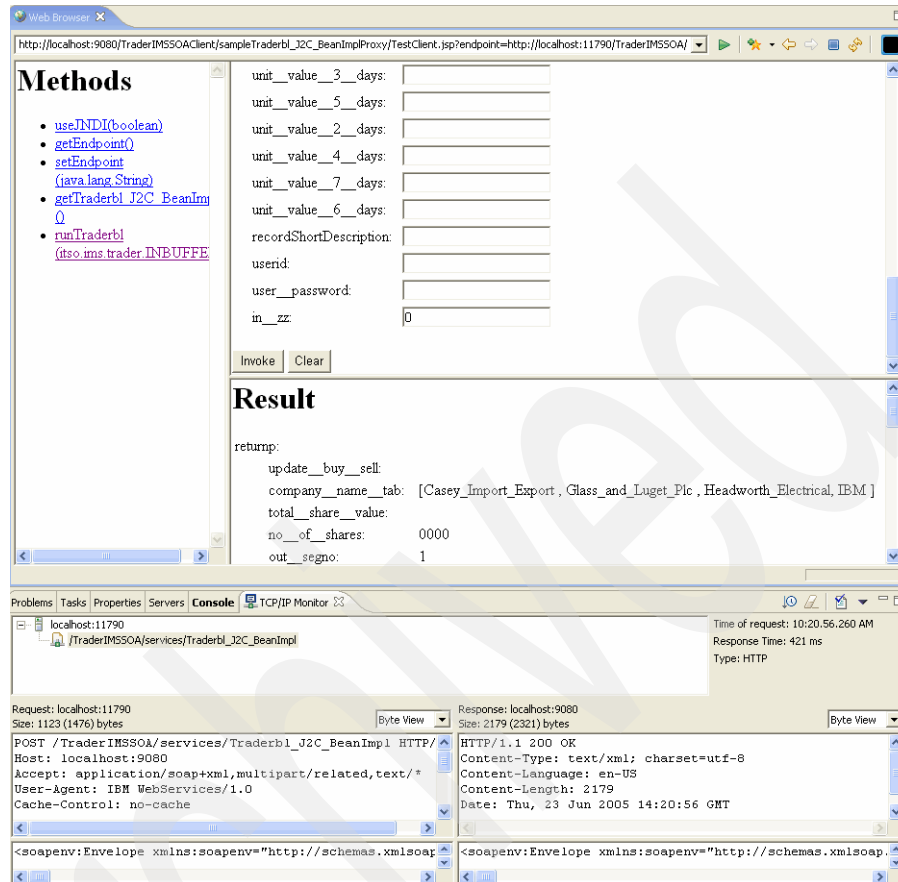


Figure 7-48 Test results from J2C Web service

8. Next, we selected the **itso.ims.trader** package in the TraderIMSSOAClient project under Other Projects in the Project Explorer pane and then clicked the right mouse button. We selected **New - Class** to open the Create a new Java class window. We entered a class name of TestTraderblBeanImplProxy and ensured a main methods was created. We clicked **Finish** to continue.
9. We entered the code shown in Figure 7-4 for the TestTraderblBeanImplProxy class and saved the code by pressing Ctrl+S.

Example 7-4 TestTraderblBeanImplProxy source

```
package itso.ims.trader;
public class TestTraderblBeanImplProxy {

    public static void main(String[] args) {
        Traderbl_J2C_BeanImplProxy proxy = new Traderbl_J2C_BeanImplProxy();
```

```

OUTBUFFER outbuffer = new OUTBUFFER();
INBUFFER inbuffer = new INBUFFER();
inbuffer.setRequest_type("Get_Company  ");
inbuffer.setIn_trcd("TRADERBL");
inbuffer.setIn_ll((short) 386);
try {
    outbuffer = proxy.runTraderbl(inbuffer);
    for (int i = 0; i <4; i++) {
        System.out.println(outbuffer.getCompany_name_tab()[i]);
    }
} catch (Exception e) {
}
}
}

```

10. Next, we selected the down arrow next to the Run wizard on the toolbar and selected **Run** on the drop-down list; see Figure 7-49.

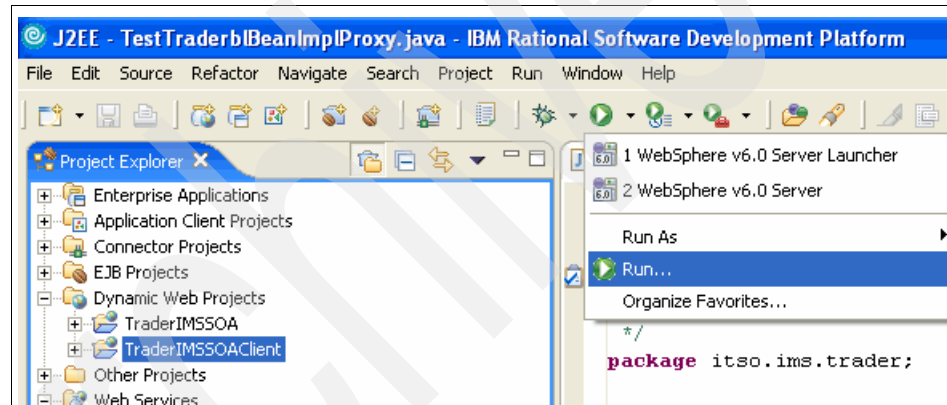


Figure 7-49 Run wizard

11. On the Launch Configuration window, we selected **Java Application** and clicked **New**. Then we clicked **Run** (see Figure 7-21 on page 232) and executed the client code.

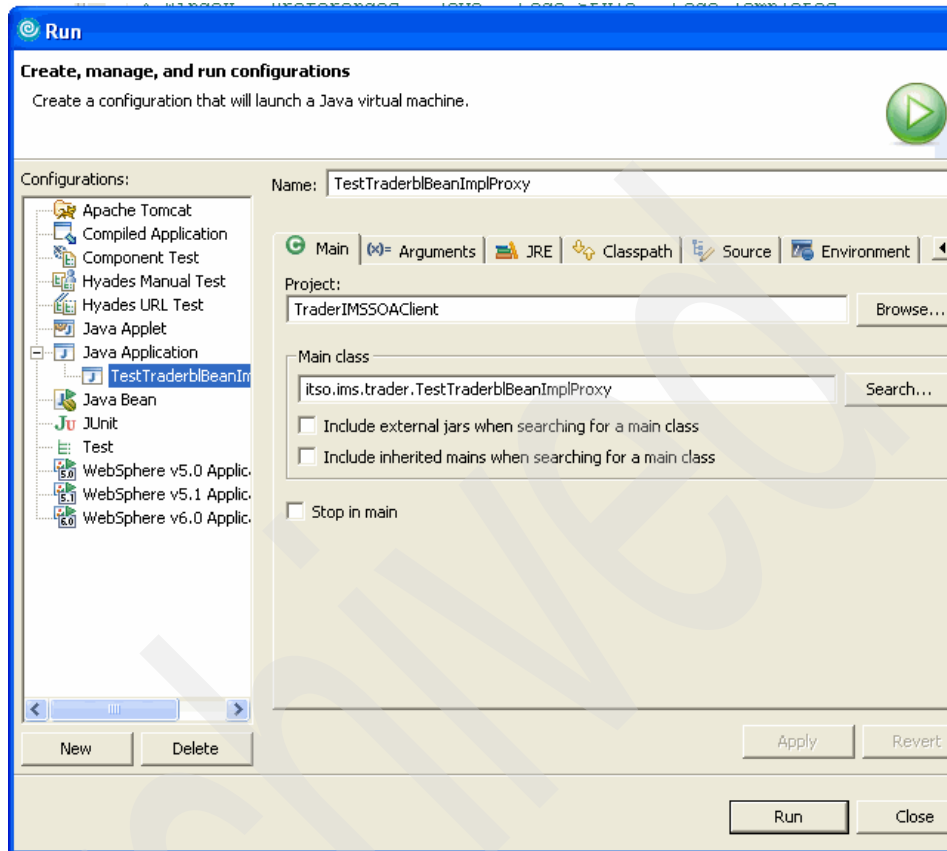


Figure 7-50 TestTraderblBeanImplProxy configuration

12. The Console window (see Figure 7-51 on page 264) shows the results of our Get_Company request.

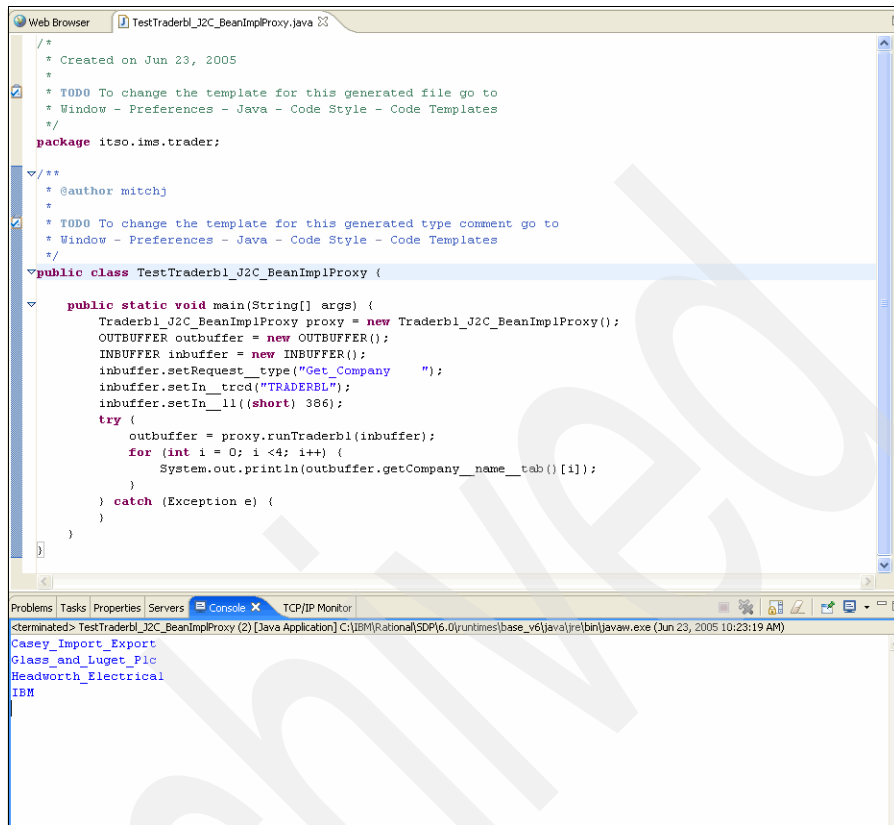


Figure 7-51 Results of testing the Trader SOA J2C Bean from a Java client

Configuring J2C for CTG

In Chapter 6, “Developing J2C applications accessing CICS” on page 183 we described how to develop or migrate an application using J2C to connect to CICS Transaction Server Version 3.1 using CICS Transaction Gateway V6. This chapter describes how to configure the infrastructure to be able to use the application we developed. We also show how we deployed the sample application. The sample application has been used in previous versions of this book and is called TRADER.

The topics covered are:

- ▶ “Installing the CICS resource adapter” on page 266
- ▶ “Configuring connection factories” on page 269
- ▶ “Command-line CICS J2C definition” on page 276
- ▶ “Application deployment” on page 278
- ▶ “TranName and TPNNName” on page 279
- ▶ “Security and CICS J2C Connector” on page 280
- ▶ “Problem determination” on page 290
- ▶ “Transaction management” on page 292

8.1 Installing the CICS resource adapter

For an application to use the CICS resource adapter, it must first be installed. The adapter file is called `cicseci.rar`. The process to configure this adapter is explained here:

1. On the WebSphere Administrative Console (Figure 8-1), expand **Resources** and click **Resource Adapters**.
2. In the Resource Adapters panel, select the node on which to install the resource adapter. Click **Install RAR** to start the installation process (Figure 8-1).

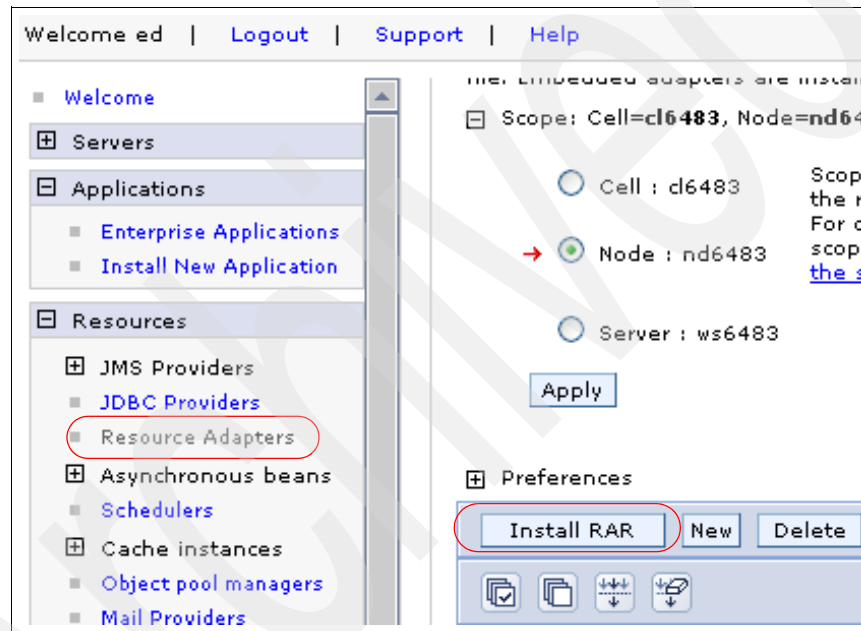


Figure 8-1 Install CICS Resource Adapter

3. On the next pane (Figure 8-2 on page 267), identify the location of the CICS Resource Adapter Repository (RAR) file and make sure the that desired node for this resource is selected and click **Next**. If your WebSphere cell is a Network Deployment configuration then the target node *cannot* be the deployment manager node.

Tip: You can install RAR files from either of two locations—you can load the file from a path on the local workstation or from a path on the server.

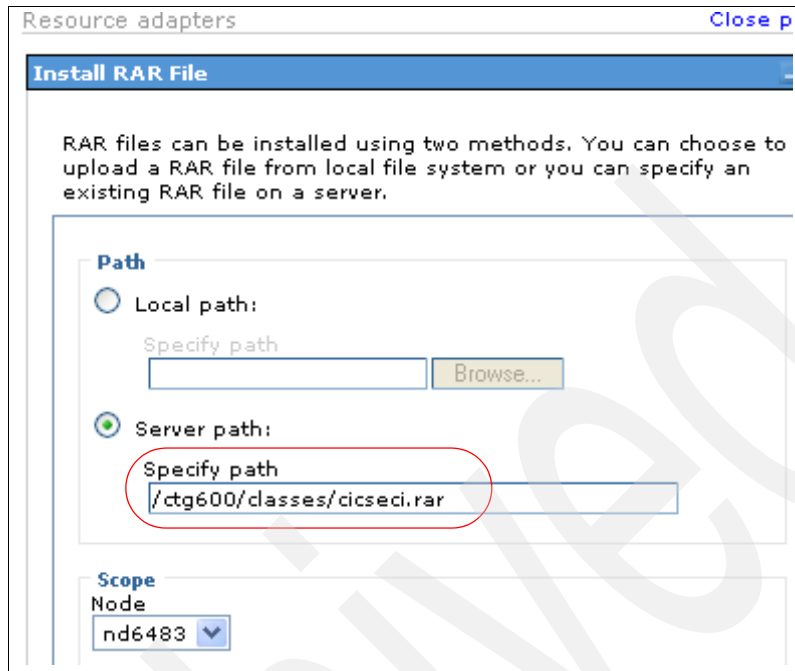


Figure 8-2 Locating the CICS Resource Adapter file

Important: If you deploy an application developed with WebSphere Studio-Integration Edition, which uses WSIF, then WebSphere for z/OS requires access to Java Archive (JAR) file cicsecitools.jar. Since the RAR shipped with CICS Transaction Gateway for z/OS does not include this JAR file, you must manually place it in the installed Connectors directory, which is /WebSphere5/bs0a/AppServer/installedConnectors/cicseci.rar in our case. You can obtain this JAR file from the CICS RAR (cicseci.rar) provided with WebSphere Studio Application Developer Integration Edition. Be sure the attributes of this file are set to 660.

4. A resource adapter configuration pane (Figure 8-3 on page 268) opens. Enter a name for the resource adapter. We enter CICS ECI. Because we intend to use local access to CICS, for the Native Path, we specify /ctg600/bin. Click **OK**.

Configuration

General Properties

* Scope
cells:cl6483:nodes:nd6483

* Name
CICS ECI

Description

* Archive path
\${CONNECTOR_INSTALL_ROOT}/cicseci.rar

Class path
\${CONNECTOR_INSTALL_ROOT}/cicseci.rar

Native path
/ctg600/bin

Thread pool alias
Default

Figure 8-3 Setting properties for the CICS Resource Adapter

This adapter now displays as being installed on this node (Figure 8-4 on page 269).

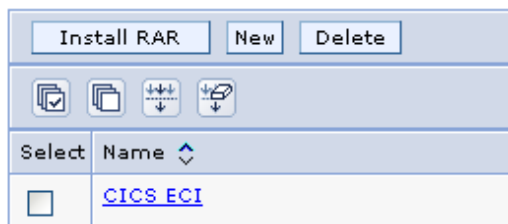


Figure 8-4 Display showing CICS Resource Adapter defined

You can save the configuration changes at this time, but we configure a J2EE Connector (J2C) connection factory for the CICS ECI adapter next.

8.2 Configuring connection factories

To configure the connection factories, follow these steps:

1. Click the CICS ECI resource adapter that you just installed.
2. On the Configuration pane (Figure 8-5) that opens, on the right-hand side under Additional Properties, click **J2C Connection Factories**.

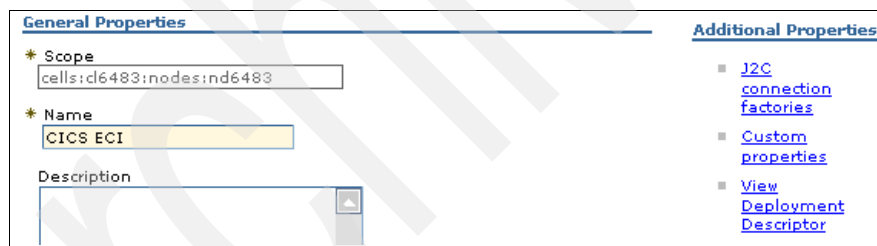


Figure 8-5 Start of process to define Connection Factory

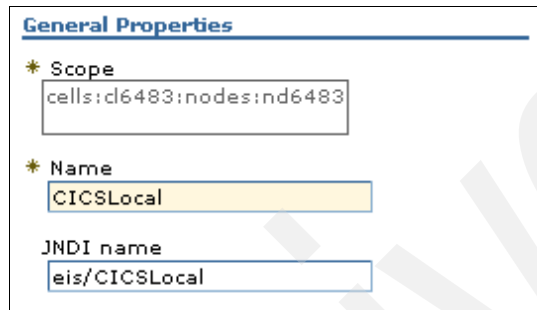
3. The Resource Adapter → CICS ECI → J2C Connection Factories pane (not shown) opens. Click **New**.

There are two types of CICS ECI J2C connection factories. One directly accesses a CICS region using Java Native Interface (JNI) calls to CICS Transaction Gateway code (LOCAL). The other one accesses a CICS Transaction Gateway task using TCP/IP (REMOTE). The following sections explain both configurations.

8.2.1 Configuring a local J2C connection factory resource

To configure a local J2C connection factory resource, follow these steps:

1. On the Resource Adapter → CICS ECI → J2C Connection Factories - New pane (shown in part in Figure 8-6), for Name (factory), enter CICSLocal. For its Java Naming and Directory Interface (JNDI) name, enter eis/CICSLocal. On this pane (but not shown in Figure 8-6) are fields to do with container and component authentication. These are discussed in “Security and CICS J2C Connector” on page 280. Click **Apply**.



General Properties

* Scope
cells:cl6483:nodes:nd6483

* Name
CICSLocal

JNDI name
eis/CICSLocal

Figure 8-6 Main part of the display to define a new Connection Factory

2. After clicking **Apply**, the previously greyed out options under Additional Properties, to the right of the display, will now be available.

Click **Custom Properties** to access the pane where you can set the details of the CICS region to be accessed.

3. The Custom Properties pane (Figure 8-7 on page 271) for the connection factory opens. Enter a ConnectionURL (local :) and the ServerName (SCSCERW1) provided by the CICS system programmer.

Preferences

| Name | Value | Description | Required |
|-----------------|----------|-------------|----------|
| TPNName | | | false |
| ClientSecurity | | | false |
| ConnectionURL | local: | | false |
| KeyRingClass | | | false |
| KeyRingPassword | | | false |
| Password | | | false |
| PortNumber | 2006 | PortNumber | false |
| ServerName | SCSCERW1 | | false |
| ServerSecurity | | | false |
| TraceLevel | 1 | TraceLevel | false |
| TranName | | | false |
| UserName | | | false |
| Total 12 | | | |

Figure 8-7 Setting properties for the CICS Connection Factory

Note the following explanations of the names shown in Figure 8-7:

- ConnectionURL: Must be set to *local* for a local connection factory.
- ServerName: The name of the CICS server to which CICS Transaction Gateway connects for all interactions for this connection factory. This is the value of APPLID specified for the target CICS region.
- PortNumber: Has no meaning for a local connection factory.
- TPNName: The name of the CICS transaction under which the initial CICS mirror program and target application program run. If TPNName is specified, it overrides TranName.
- TraceLevel
 - 0: No tracing or logging occurs.
 - 1: Only errors and exceptions are logged.
 - 2: Errors and exceptions, plus the entry and exit of important methods, are logged.

- 3: Errors and exceptions, the entry and exit of important methods, and the contents of buffers sent to and received are logged.
- TranName: The name of the CICS transaction under which the target application program runs. This does not affect the name of the transaction under which the mirror program is initially started.
- Userid: Default user ID to use for the connection if another is not provided.
- Password: A password to be used with the default user ID.

Tip: To add additional J2C connection factories, click **>J2C Connection Factories>** at the top of the pane to redisplay the list of J2C connection factories (Figure 8-8 on page 272) configured for this adapter. Then repeat this process. Otherwise save the changes.



Figure 8-8 Display showing defined J2C connection factories

8.2.2 Configuring a remote CICS J2C factory resource

To configure a remote CICS J2C factory resource, follow these steps:

1. On the Configuration pane of the resource adapter (Figure 8-5 on page 269), click **J2C Connection Factories**.
2. On the J2C connection factory pane, click **New**.
3. On the New J2C Connection Factory pane (Figure 8-9 on page 273), for Name (factory), enter CICSRemote. For its JNDI name, enter eis/CICSRemote. Click **Apply**.

| General Properties | |
|--------------------|---------------------------|
| * Scope | cells:cl6483:nodes:nd6483 |
| * Name | CICSRemote |
| JNDI name | eis/CICSRemote |

The additional properties will not be available until the general properties for this item are saved.

Additional Properties

- Connection pool properties
- Advanced

Figure 8-9 Defining remote CICS Connection factory

4. After clicking **Apply**, the previously greyed out options under Additional Properties, to the right of the display, will now be available.

Click **Custom Properties** to access the pane where you can set the details of the remote CICS region to be accessed.

5. The Custom Properties pane (Figure 8-10 on page 274) opens. Enter the ConnectionURL (tcp://wtsc48.itso.ibm.com), the ServerName (SCSCERW1), and the PortNumber (2006) that are provided by the CICS system programmer.



| + Preferences | | | |
|---|---|----------------------------|----------|
|   | | | |
| Name ▾ | Value ▾ | Description ▾ | Required |
| TPNName | | | false |
| ClientSecurity | | | false |
| ConnectionURL | tcp://wtsc48.itso.ibm.com | | false |
| KeyRingClass | | | false |
| KeyRingPassword | | | false |
| Password | | | false |
| PortNumber | 2006 | PortNumber | false |
| ServerName | | | false |
| ServerSecurity | | | false |
| TraceLevel | 1 | TraceLevel | false |
| TranName | | | false |
| UserName | | | false |
| Total 12 | | | |

Figure 8-10 Setting properties to identify remote CICS Region

Note the following explanation of the items shown in Figure 8-10:

- **ConnectionURL:** The URL of the CICS Transaction Gateway with which the connection factory communicates. The format is tcp:// followed by the host name of the system on which the CICS Transaction Gateway is running.
- **PortNumber:** The port on which the CICS Transaction Gateway is listening.
- **ServerName:** The name of the CICS server to connect to for all interactions for this connection factory. This is the value of APPLID specified for the target CICS region.
- **TPNName:** The name of the CICS transaction under which the initial CICS mirror program and target application run. If TPNName is specified, it overrides TranName.
- **TraceLevel**
 - 0: No tracing or logging occurs.

- 1: Only errors and exceptions are logged.
- 2: Errors and exceptions, plus the entry and exit of important methods, are logged.
- 3: Errors and exceptions, the entry and exit of important methods, and the contents of buffers sent to and received are logged.
- TranName: The name of the CICS transaction under which the target application program runs. This does not affect the name of the transaction under which the mirror program is initially started.
- Userid: A default user ID to use for this connection if another one is not provided.
- Password: A password to use with the previous default user ID

Tip: If you need to add additional J2C connection factories, click **>J2C Connection Factories>** at the top of the pane to redisplay the list of J2C connection factories configured for this adapter and repeat this process. Otherwise save the changes.

6. Since there are no other J2C connection factories to enter, save the configuration changes.

8.3 Cell scope J2C connection factories

To configure J2C connection factories at the cell scope, you must install the CICS ECI resource adapter on each node in the cell. This ensures the implementation code (JAR files and binaries) of the resource adapter is available to the Java Virtual Machines (JVMs) running on that node.

1. On the WebSphere Administration Console, expand **Resource** and click **Resource Adapters**. Set the scope to the Cell level.
2. On the Resource Adapters panel, click **New** to start the configuration process.

3. On the Resource Adapter pane, under General Properties, enter a meaningful name for the configuration. For Archive Path, select the **Choose an archive path from the list of installed RARs** option to display a list of previously installed resource adapters. Select the **cicsecl.rar** file and click **OK**.

J2C connection factories installed at the cell scope will be available to every node in the cell. The adapter is now displayed as being available at the Cell level.

8.4 Command-line CICS J2C definition

An alternative to using the WebSphere Administration GUI to define the Resource Adapter and J2C Connection Factory is to use the **wsadmin** command interface.

As part of the downloadable material available with this book is a JACL script, called Define-CICS-j2c.jacl, that can be used to perform the following:

- ▶ Optionally define an authentication alias.
- ▶ Define a CICS Resource Adapter.
- ▶ Define a CICS J2C Connection Factory to connect to a local CICS region
- ▶ Optionally set the Component-managed authentication alias.

The supplied JACL script needs to be updated with settings to reflect your installation before using it. All the settings to be adjusted are located in a parameter block at the start of the script.

To run the JACL script via the WebSphere wsadmin tool, issue this command:

```
wsadmin.sh -f /tmp/Define-CICS-j2c.jacl
```

Or if your WebSphere cell has global security enabled then use this command:

```
wsadmin.sh -user uid -password pwd -f /tmp/Define-CICS-j2c.jacl
```

The JACL script would need further modification to define a J2C Connection Factory that was going to connect to a remote CICS region.

The advantages of using a JACL script similar to this sample are that you have an easily repeatable process.

Sample output from a run of the JACL script on our system is shown in Example 8-1.

Example 8-1 Sample output from run of JACL script to define J2C definitions

```
EDMCAR @ SC48:/WebSphereAL3/V6R0/BS03/AppServer/bin>wsadmin.sh -f /tmp/Define->
```

```
WASX7209I: Connected to process "ws6483" on node nd6483 using SOAP connector;
The type of process is: UnManagedProcess
Locating cell and node name
Cell Name: c16483(cells/c16483|cell.xml#Cell_1)
Node Id: nd6483(cells/c16483/nodes/nd6483|node.xml#Node_1)

Issuing cmd: AdminConfig create JAASAuthData
(cells/c16483|security.xml#Security_1) {alias cics_ctg} {password xyz} {userId
sample}

Defining new CICS Resource Adapter
Issuing cmd: AdminConfig installResourceAdapter /ctg600/classes/cicseci.rar
nd6483 -rar.name CICS1
CICS Resource Adaptor defined:
CICS1(cells/c16483/nodes/nd6483|resources.xml#J2CResourceAdapter_1118605725008)

Updating attributes of the CICS resource adapter
Issuing cmd: AdminConfig AdminConfig modify
CICS1(cells/c16483/nodes/nd6483|resources.xml#J2CResourceAdapter_1118605725008)
{nativepath /ctg600/bin}

Defining CICS J2C Connection Factory
Issuing cmd: AdminConfig create J2CConnectionFactory
CICS1(cells/c16483/nodes/nd6483|resources.xml#J2CResourceAdapter_1118605725008)
{name CICS-CTG} {jndiName eis/CICSLocal} {authDataAlias cics_ctg}
J2CC Connection Factory:
CICS-CTG(cells/c16483/nodes/nd6483|resources.xml#J2CConnectionFactory_111860572
7048)

Setting method to use to connect to CICS region
Issuing cmd: AdminConfig modify
ConnectionURL(cells/c16483/nodes/nd6483|resources.xml#J2EEResourceProperty_1118
605727050) {value local:}

Setting name of CICS region to connect to
Issuing cmd: AdminConfig modify
ServerName(cells/c16483/nodes/nd6483|resources.xml#J2EEResourceProperty_1118605
727055) {value SCSCERW1}

Setting Component Managed Authentication Alias
Issuing cmd: AdminConfig modify
CICSLocal(cells/c16483/nodes/nd6483|resources.xml#J2CConnectionFactory_11186690
32002) {authDataAlias cics-ctg}

Saving changes to WebSphere Master Configuration :-
Issuing cmd: AdminConfig save
EDMCAR @ SC48:/WebSphereAL3/V6R0/BS03/AppServer/bin>
```

8.5 Application deployment

During deployment of the ear file into WebSphere, we did not need to perform any actions apart from clicking **Next** several times. After locating the ear file and clicking **Next**, the display shown in Figure 8-11 was shown.

Install New Application

Specify options for installing enterprise applications and modules.

→ Step 1: Select installation options

[Step 2](#) Map modules to servers

[Step 3](#) Provide JNDI Names for Beans

[Step 4](#) Map EJB references to beans

[Step 5](#) Map resource references to resources

[Step 6](#) Map virtual hosts for Web modules

[Step 7](#) Ensure all unprotected 2.x methods have the correct level of protection

Select installation options

Specify the various options that are available to install your application.

☐ Pre-compile JSP

Directory to install application

☒ Distribute application

☐ Use Binary Configuration

☐ Deploy enterprise beans

Application name

☒ Create MBeans for resources

☐ Enable class reloading

Reload interval in seconds

☐ Deploy Web services

Validate Input off/warn/fail

Figure 8-11 Main deploy window

None of the steps are highlighted, indicating that no actions are required. You can step through each step to review settings if desired or just click **Step 8** and then **Finish** to complete deployment of the ear file. Save the changes to the WebSphere master configuration.

Tip: You can modify the J2C connection factory later without re-installing the application.

8.6 TranName and TPNNName

Two other custom properties that can be set on a CICS J2C connection factory that are of interest are TranName and TPNNName.

When CICS receives the request via the CICS Transaction Gateway V6, the default behavior is to start a CSMI transaction to process the request, running the specified program.

Typically, this is unsatisfactory for a production system, as most companies would require greater security controls around the processing of these requests. If a company had several different WebSphere application calling several different applications in CICS then it would be unsatisfactory if they all ran under a CICS transaction ID of CSMI.

8.6.1 TranName

Setting TranName results in the EIBTRNID field of the CICS transaction being set to the value specified. This allows the program being run to use a CICS API to obtain a transaction name that is not CSMI. This may be of use for the purpose of the logic in the application. Note that if only this field is set then the transaction run by CICS is still CSMI.

8.6.2 TPNNName

Setting TPNNName to a value results in CICS processing the received request using a transaction of the name specified in the TPNNName property. This allows you to use RACF, for example, to control access to the business purpose represented by this transaction.

Important: You must define to the CICS region a transaction to match the name set in TPNNName. Use the CSMI transaction definition as a template to do this.

Note that setting this field overrides any value set in the TranName field.

8.6.3 Setting TranName and TPName

The TranName and TPName custom properties can be set in the custom properties of the J2C Connection factory, as seen in Figure 8-10 on page 274.

Alternatively, they can be set in the J2C Java Bean. This can be done by updating the meta data of the J2C Java Bean class. If we used this approach for our TraderCICS sample application then we would have modified the TraderCICSJ2cImpl.java program as shown in Example 8-2.

Example 8-2 Setting the TPName custom property

```
* @j2c.interactionSpec-property name="TranName" value="TRDT"
```

8.7 Security and CICS J2C Connector

Each application that accesses a JCA resource has a resource deployment descriptor (res-auth) that determines the authentication behavior when accessing the resource. There are two options that can be set in the field labelled Authentication, as shown in Figure 6-25 on page 208.

- ▶ Container authentication

When container authentication is specified, the user ID and password used on the connection are provided by the container.

- ▶ Application authentication

When application authentication is specified, the user ID and password used on the connection are provided explicitly by the application or the J2C connection factory in the Component-managed authentication alias entry in the Configuration - General Properties pane of the J2C connection factory.

8.7.1 Defining an authentication alias

An authentication alias is a way to store a user ID and password in the WebSphere configuration. It can be defined via the WebSphere Administrative Console, as shown in Figure 8-12 on page 281.

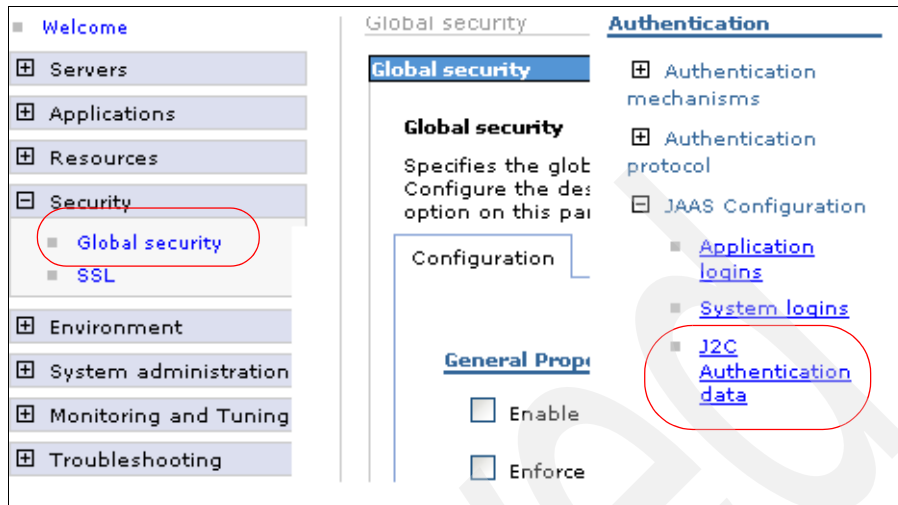


Figure 8-12 Area in Admin GUI to start defining authentication alias

Click **J2C Authentication data** and, in the window displayed, set a name for the alias you are creating and set a user ID and password, as shown in Figure 8-13 on page 282.

Global security > J2EE Connector Architecture (J2C) authentication data entries > New

Specifies a list of user IDs and passwords for Java 2 connector security to use.

Configuration

General Properties

* Alias

* User ID

* Password

Description

Figure 8-13 Defining an authentication alias

WebSphere stores the password in encrypted form within the XML files that manage the configuration.

8.7.2 Container-managed security

Associating an authentication alias with a J2C Connector for container-managed security is deprecated in WebSphere Application Server for z/OS Version 6.01. A message to this effect, along with two other deprecated settings, is displayed in the WebSphere Administrative Console, as shown in Figure 8-14 on page 283.

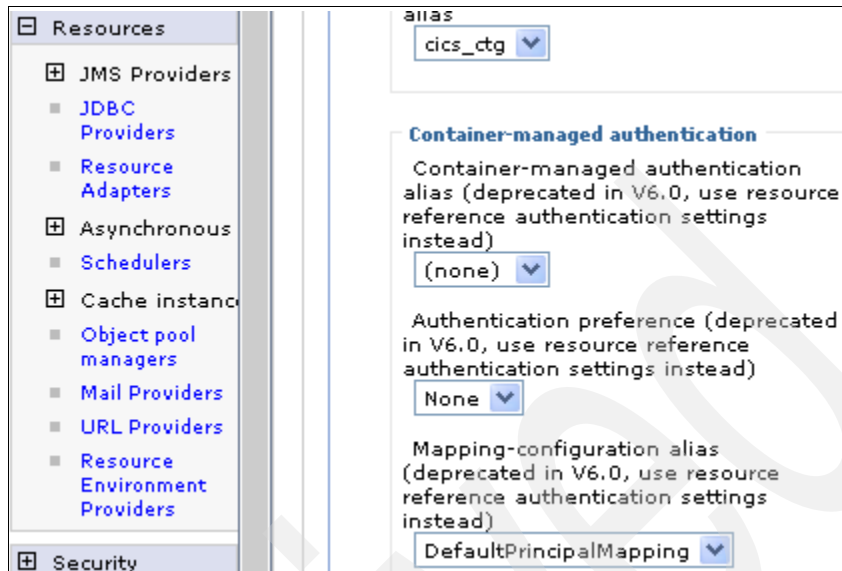


Figure 8-14 *Deprecated container-managed settings*

The way to associate an authentication alias with a resource for container-managed security is to either perform that association during deployment of the ear file, or you can also pre-configure it. In Figure 6-26 on page 208 where we showed how to pre-set the JNDI name of the actual resource, is displayed another check box labelled JAAS Logon Configuration.

Click the radio button labelled **Use Default Method** and enter in the field Authentication Alias the name of an authentication alias that will be defined in the WebSphere server. The display in Figure 8-15 on page 284 shows an example of setting the authentication alias to cics-ctg.

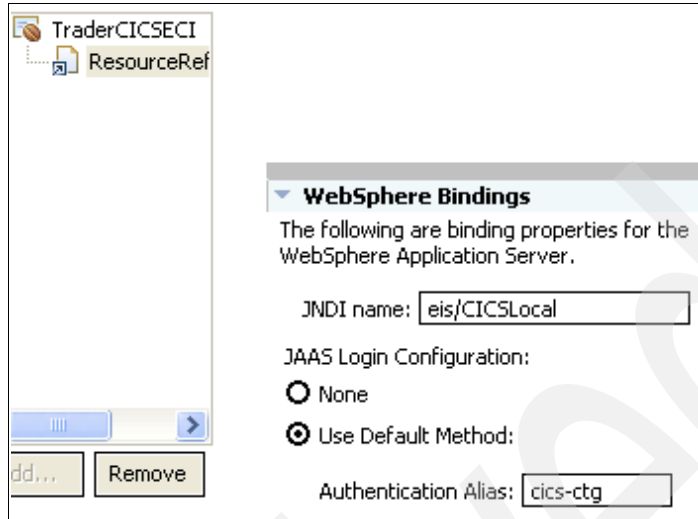


Figure 8-15 Setting an authentication alias for container managed security

Rational Application Developer Version 6 saves this information in a file called `ibm-ejb-jar-bnd.xml`. This is not part of the J2EE specification but is a feature of WebSphere.

If, after deploying the application, you want to change the container-managed security setting, then this can be done via the WebSphere Administrative Console. Expand **Applications**, click **Enterprise Applications**, click the application, and click **Map Resource Reference to Resources**. Then in the window displayed you can modify the container-managed security settings for the application resource.

8.7.3 Application-managed security

Application-managed security is also referred to as component-managed security.

To use application authentication, set the Authentication field shown in Figure 6-25 on page 208 to **Application**. When you do this the JAAS Login Configuration area of the display, shown previously in Figure 6-26 on page 208, will become greyed out, meaning they are no longer applicable.

When used, there are three ways that a user ID and password can be supplied for the connection to CICS; they are:

- ▶ “User ID and password supplied via the application” on page 285
- ▶ “User ID and password supplied via authentication alias” on page 286

- “User ID and password supplied via custom properties” on page 286

User ID and password supplied via the application

The application can supply a user ID and password when the connection is acquired.

To use this approach, you update the meta data in the J2C Java Bean generated code adding properties for the user ID and password. If we did this for the TraderCICS sample application then the change would be to TraderCICSJ2cImpl.java, as shown in Example 8-3.

Example 8-3 Updating the meta data to add user ID and password properties

```
/**
 * @j2c.interactionSpec class="com.ibm.connector2.cics.ECIInteractionSpec"
 * @j2c.interactionSpec-property name="functionName" value="TRADERBL"
 * @j2c.interactionSpec-property name="commareaLength" value="500"
 * @j2c.interactionSpec-property name="replyLength" value="500"
 * @j2c.connectionSpec class="com.ibm.connector2.cics.ECIConnectionSpec"
 * @j2c.connectionSpec-property name="userName" argumentBinding="ivUserid"
 * @j2c.connectionSpec-property name="password"
argumentBinding="ivPassword"
 * @generated
 */
```

Save these changes. Rational Application Developer Version 6 automatically adds additional code to cater for the use of a user ID and password on the connection call, but you will still have to make some other changes.

The runTrader method in this Java program then needs to be updated as well, as shown in Example 8-4.

Example 8-4 Updated runTrader method

```
public itso.cics.j2c.data.TraderCICSECICommarea runTrader
    (itso.cics.j2c.data.TraderCICSECICommarea arg,
     String ivUserid, String ivPassword )
```

Then this in turn requires updating the TraderCICSJ2c.java program, as shown in Example 8-5.

Example 8-5 Updating runTrader method in the interface code

```
public itso.cics.j2c.data.TraderCICSECICommarea
    runTrader(itso.cics.j2c.data.TraderCICSECICommarea arg,
              String ivUserid, String ivPassword)
    throws javax.resource.ResourceException;
```

Any Java program that uses the updated method needs to be modified to now pass in arguments for the user ID and password. It is now up to the application to obtain the user ID and password to pass on the method call.

User ID and password supplied via authentication alias

WebSphere can still be used to supply a user ID and password via an authentication alias.

If you want WebSphere to supply a user ID and password on the connection request for application-managed security, then select a pre-defined authentication alias for the Component-managed authentication alias in the J2C connection definition, as shown in Figure 8-16.

If the application will supply a user ID and password then do not set the Component-managed authentication alias.

The screenshot shows the WebSphere Administration Console interface for defining a J2C connection. On the left is a tree view with the following structure:

- Welcome
- Servers
- Applications
- Resources
 - JMS Providers
 - JDBC Providers
 - Resource Adapters
 - Asynchronous beans
 - Schedulers
 - Cache instances
 - Object pool managers
 - Mail Providers
 - URL Providers
 - Resource Environment Providers

The main panel on the right contains the following fields:

- JNDI name: eis/CICSLocal
- Description: (empty text area)
- Connection factory interface: javax.resource.cci.ConnectionFactory
- Category: (empty text field)
- Component-managed authentication alias: (highlighted with a red circle) cics_ctg (selected from a dropdown menu)

Figure 8-16 Setting authentication alias for application managed security

User ID and password supplied via custom properties

The third approach is to set a user ID and password in the custom properties of the CICS J2C connection factory. Do this by setting the fields userName and Password, which can be seen in Figure 8-7 on page 271.

Important: The password property is treated like all the other properties—the value is not encrypted and is displayed in clear text. We recommend that this approach should not be used for this reason.

8.7.4 CICS J2C security decision tree

CICS J2C connection factories support the assignment of the current thread identifier as the owner of a connection for authentication purposes when global security is enabled. Thread identity assignment is performed when the following conditions are met:

- ▶ Container-managed resource authority (res-auth=Container) is specified in the application deployment descriptors.
- ▶ The J2C connection factory uses a local connection (no TCP/IP), and no Container-managed authentication alias is specified.

The security decision tree shown in Figure 8-17 on page 288 shows how the user ID that flows to the CICS region is determined.

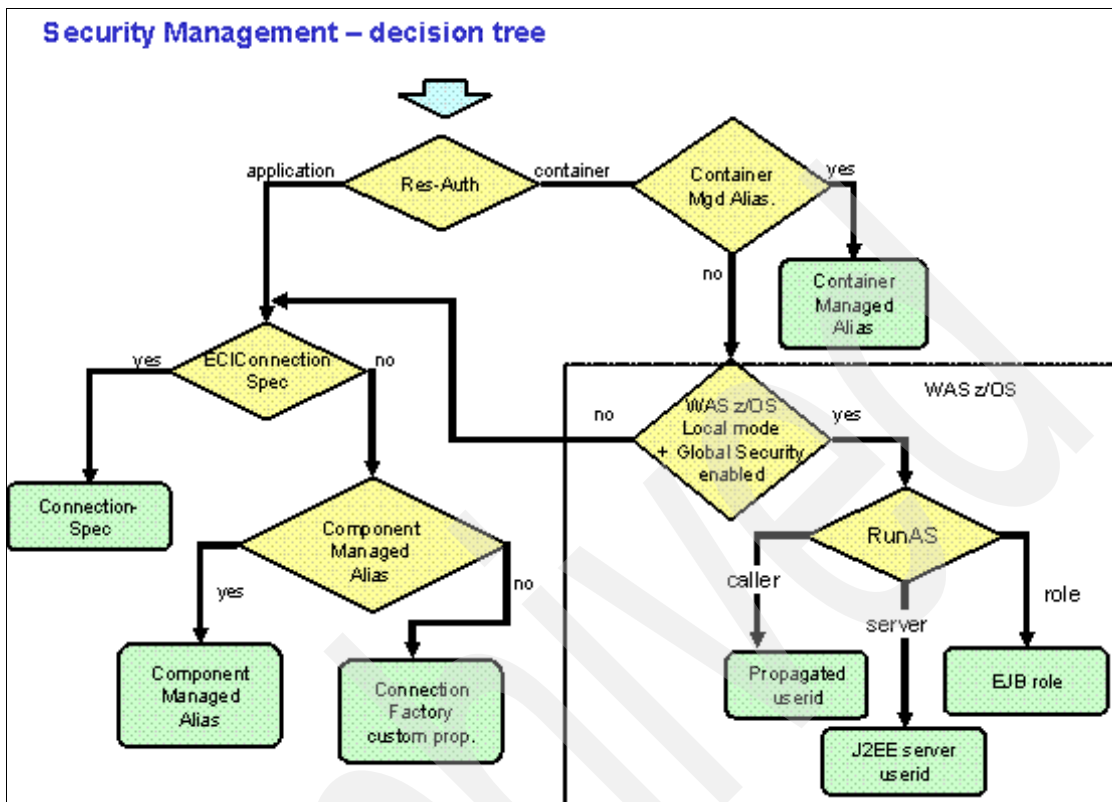


Figure 8-17 CICS J2C Security decision tree

8.8 Performance and availability

A local connection provides better performance than a remote connection since no network-related overhead is involved in processing a request. The implication is that a local connection has a single point of failure if the local CICS region is not available. Use of a remote connection along with sysplex distributor provides a solution for this single point of failure problem (Figure 8-18 on page 289).

Attention: In this case, two-phase commit support is not available, and thread identity is not passed over the TCP/IP connection.

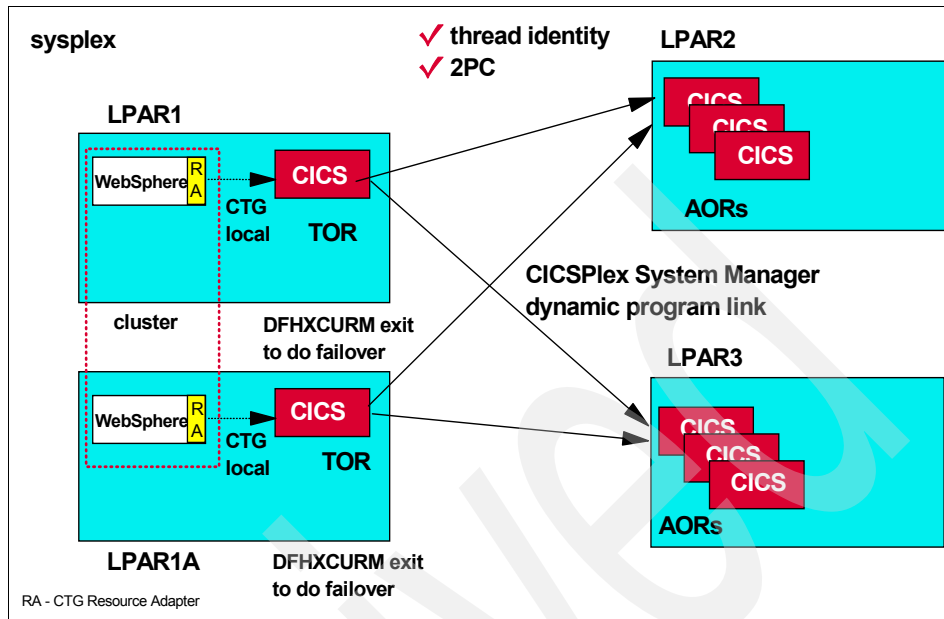


Figure 8-19 High availability solution using CICS Transaction Gateway local option

Although this configuration seems more complicated, it provides more quality of service than the previous solution.

8.9 Problem determination

To perform problem determination, we recommend that you enable the following WebSphere traces, as shown in Figure 8-20 on page 291:

- ▶ `com.ibm.ejs.j2c.*=all=enabled`
- ▶ `com.ibm.connector2.*=all=enabled`

You can also enable the CICS Transaction Gateway JNI trace to provide additional trace data to see what is happening between WebSphere and CICS. This is done by updating the trace level in the J2C connection factory custom property, setting it to 3.

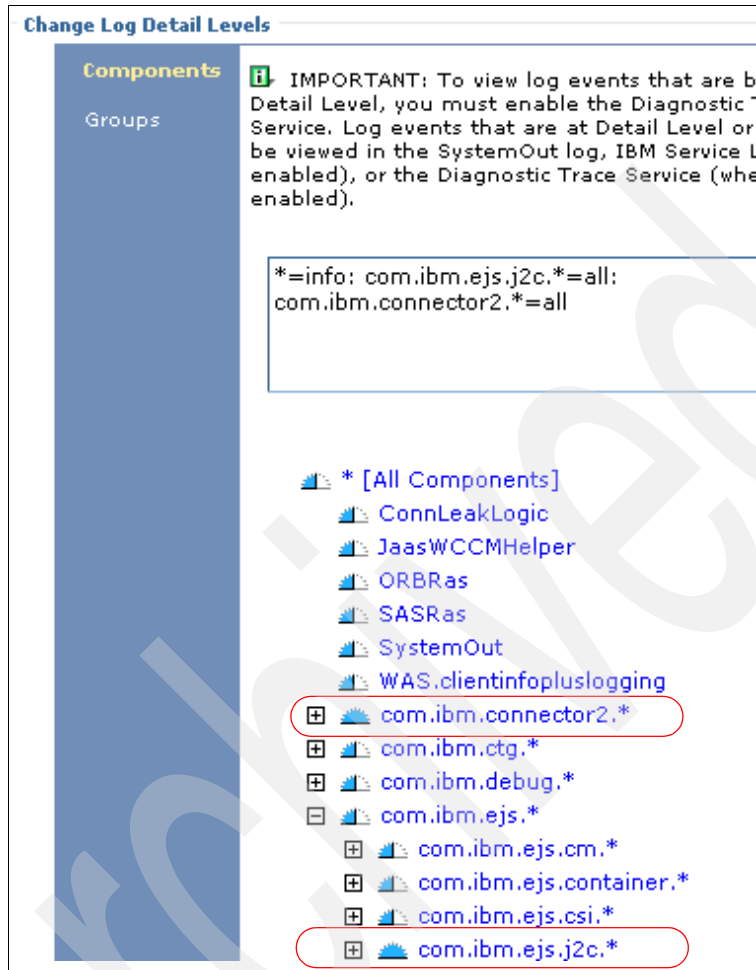


Figure 8-20 Enabling J2C-related trace settings

Tracing of the JNI calls that flow between WebSphere and CICS can be done by setting the TraceLevel to 3 in the custom properties of the CICS J2C Connection factory.

8.9.1 Common errors

Some common errors are:

- ▶ CTG9630E: IOException occurred in communication with CICS
 - Explanation: A configuration error has occurred.

- Usual cause: Review the JNI trace for the root cause. Some possibilities are:
 - RRS register return code 0x300: CICS Transaction Gateway has not been configured properly to use RRS.
 - EXCI reason code 403: CICS Transaction Gateway is unable to contact the target CICS server because of an invalid pipe name.
- ▶ CTG9631E: Error occurred during interaction with CICS. Error Code=ECI_ERR_NO_CICS minor code: 0 completed:
 - Explanation: CICS Transaction Gateway is unable to contact the target CICS server.
 - Usual cause: EXCI reason code 203 - The target CICS server is not active or has not opened IRC communications.
- ▶ CTG9631E: Error occurred during interaction with CICS. Error Code=ECI_ERR_SECURITY_ERROR
 - Explanation: An error occur either validating a user ID or a RACF authorization failure has occurred.
 - Usual cause: Review the JNI trace for the root error. Some possibilities are:
 - EXCI reason code 423: RACF surrogate checking has failed.
 - RACF return code 143: The user ID is unknown or not defined to RACF or does not have an OMVS RACF segment.

8.10 Transaction management

If CICS is accessed via a local connector, it is a participant (subordinate) in the WebSphere client application's global transaction. This does not preclude CICS from being a transaction coordinator for other resource managers (DB2, IMS, WebSphere MQ, or remote CICS regions), which may be accessed by a CICS program.

CICS propagates the *prepare* request to its own subordinates and replies positively to RRS since all resource managers can commit their resources. When CICS receives the *commit* flow from RRS, CICS propagates it to its subordinates. If backout processing is required, CICS conveys that to its subordinates as well. This is a full-function two-phase commit process with RRS acting as the Transaction coordinator. It is available using the local mode connector. Therefore, CICS and WebSphere for z/OS must co-reside on the same z/OS image.

If CICS is accessed via a remote mode connector, WebSphere for z/OS includes CICS as the last participant in the global transaction as long as all other connectors used by the application (for example, Java Database Connection (JDBC) to DB2 for S/390® and Java Message Service (JMS) to WebSphere MQ for z/OS) are local and exploit RRS as their transaction manager. WebSphere *prepares* the other resource manager and then asks CICS to *commit* its work.

If this one-phase commit (1PC) with CICS only succeeds, WebSphere for z/OS directs the other resource managers to commit their changes. If CICS indicates that the commit failed, the other resource managers are directed to roll back their changes.

If the 1-PC to CICS does not return, WebSphere for z/OS detects a communication failure or time out and the current global transaction is in a *heuristic hazard*. Because the type of failure is not known, it may be that CICS has committed or rolled back. However, the outcome of the 1-PC transaction is not known. WebSphere for z/OS issues a rollback on all other resource managers involved in the application global transaction. The global transaction is marked in RRS that a heuristic condition has occurred. Manual intervention may need to occur on the CICS system to restore the affected resources to a consistent state with the 2-PC resource managers that were rolled back.

If CICS is accessed via a remote mode connection and other XA resource managers are involved that do not use RRS as their transaction manager, then WebSphere for z/OS cannot include CICS within the scope of a global transaction. It may create a separate local transaction depending on the requirements of the application. When the local transaction (if any) is committed, the CICS Transaction Gateway uses RRS on the LPAR where it is executing (Figure 8-21 on page 294) to commit the unit of work in CICS. As before, CICS may include subordinate resource managers in its unit of work. Also in this situation, the link from WebSphere to the CICS Transaction Gateway daemon is not a 2PC connection, and any global transaction context is not propagated from WebSphere into CICS.

Note: During two-phase commit processing, RRS may communicate with other resource managers on the same LPAR (DB2, IMS, etc.) as necessary to preserve transactional integrity.

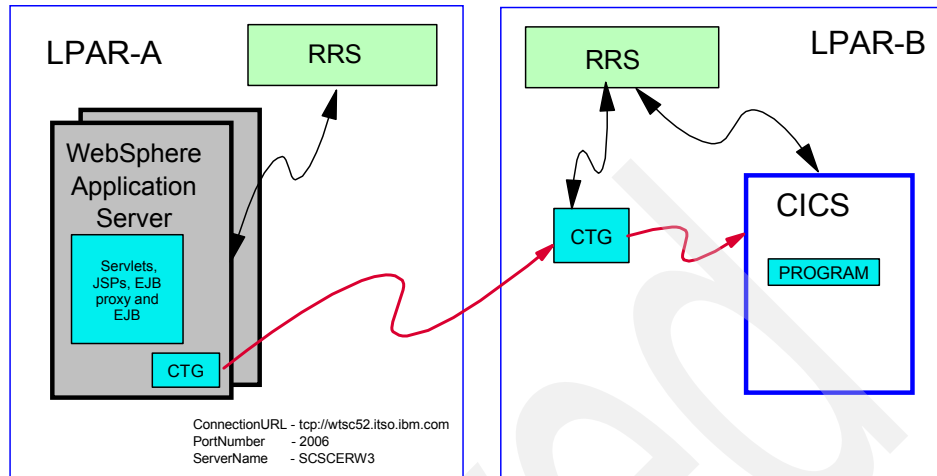


Figure 8-21 Remote CICS with multiple resource managers

Configuring J2C for IMS Connect

In Chapter 5, “Introduction to J2EE Connector Architecture” on page 173, we introduced you to the J2EE J2C Connector Architecture and showed you how to develop a Java component using J2C to connect to IMS. In this chapter we describe in detail the infrastructure side of using J2C with IMS Connect. We describe the steps we took to configure WebSphere Application Server for z/OS Version 6.01 and IMS Connect Version 2.2. We performed these steps in order to install and execute the IMS Trader sample application, which is part of the additional material for this book.

In this chapter the following topics are discussed:

- ▶ A review of IMS Connect topologies, discussed in 9.1, “IMS Connect topologies” on page 297
- ▶ A comprehensive overview of how to configure IMS Connect and IMS, discussed in 9.2, “Configuring IMS Connect and IMS” on page 298
- ▶ Installing the IMS Connect resource adapter, discussed in 9.3, “Installing the IMS Connect Resource Adapter” on page 301
- ▶ Configuring IMS J2C connection factories, discussed in 9.4, “Configuring connection factories” on page 304
- ▶ Cell Scope J2C connection factories, discussed in 9.5, “Cell scope J2C connection factories” on page 314

- ▶ Application deployment, discussed in 9.6, “Application deployment” on page 316
- ▶ Security, discussed in 9.7, “Implementing application security” on page 317 and 9.8, “Implementing infrastructure security” on page 319
- ▶ Configuration automation, discussed in 9.9, “Configuration automation using Jython scripts” on page 326
- ▶ Performance and availability, discussed in 9.10, “Performance and availability” on page 330
- ▶ Problem determination, discussed in 9.11, “Problem determination” on page 331
- ▶ Transaction management, discussed in 9.12, “Transaction management” on page 335

9.1 IMS Connect topologies

In either a local or remote configuration, IMS is accessed from WebSphere for z/OS through an IMS Connect task. In a *local* case, IMS Connect is colocated with WebSphere on the same LPAR, as shown in Figure 9-1.

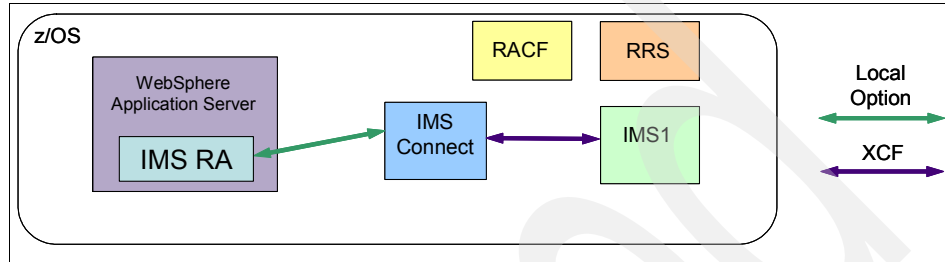


Figure 9-1 IMS Connect local configuration in a single system

In a sysplex, IMS Connect communicates with the IMS on the other system using XCF. See Figure 9-2.

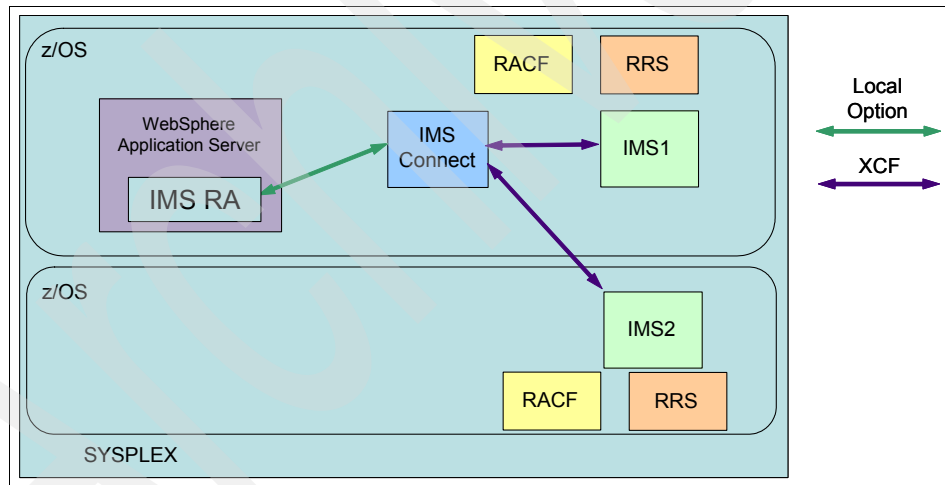


Figure 9-2 IMS Connect local configuration in a sysplex

In a remote case, IMS Connect is colocated with the IMS systems and uses TCP/IP to communicate with the IMS Connector for Java Resource Adapter. See Figure 9-3 on page 298.

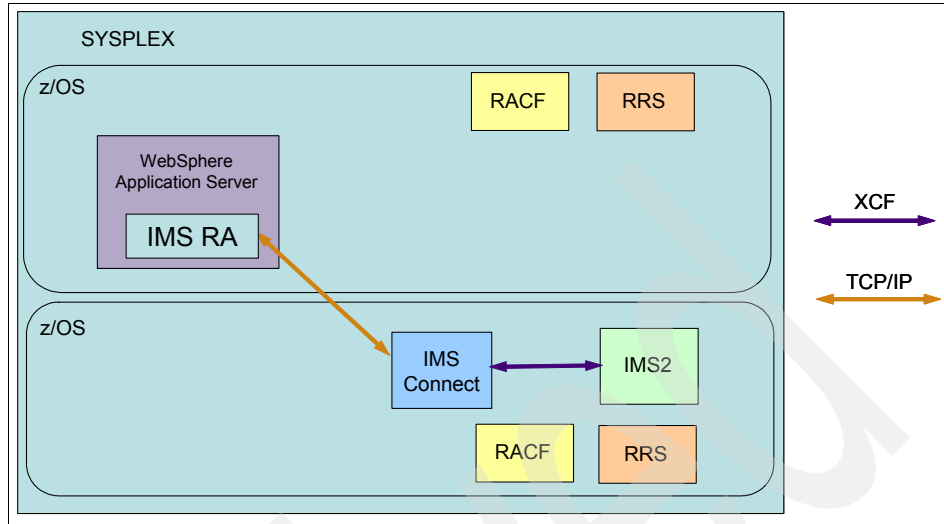


Figure 9-3 IMS Connect remote configuration in a sysplex

9.2 Configuring IMS Connect and IMS

IMS Connect is a TCP/IP server that enables TCP/IP clients to exchange messages with *IMS Open Transaction Manager Access (OTMA)*. This server provides communication links between TCP/IP clients and IMS (data stores). It supports multiple TCP/IP clients accessing multiple datastore resources. To protect information that is transferred through TCP/IP, IMS Connect provides Secure Sockets Layer (SSL) support to protect and secure the information.

IMS Connect runs on a z/OS platform. Our environment consists of:

- ▶ IMS Connect Version 2.2
- ▶ IMS Version 9

We configured IMS Connect as shown in Figure 9-4 on page 299.

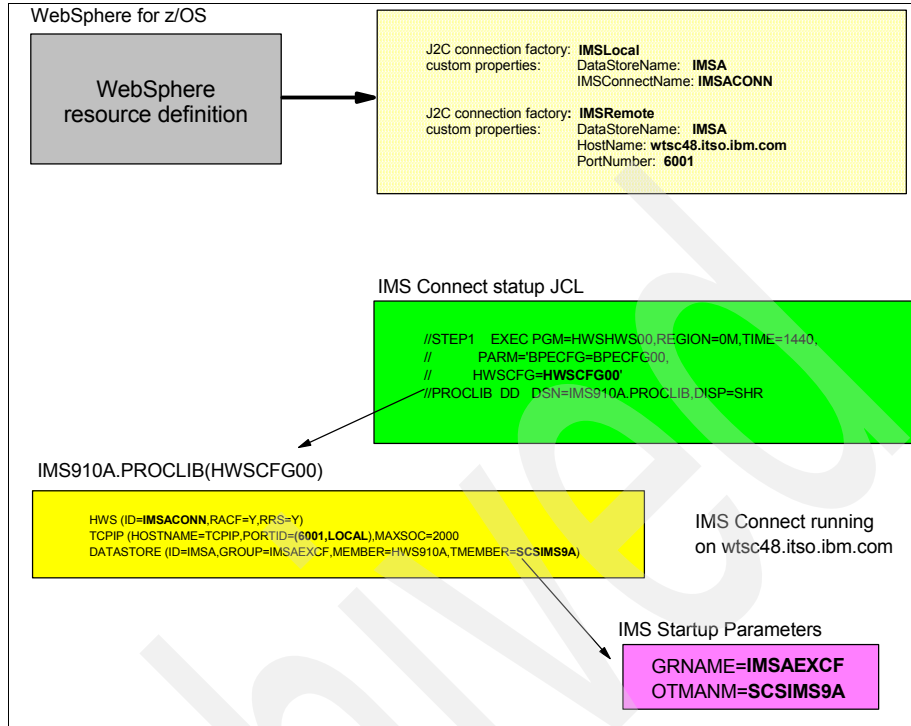


Figure 9-4 IMS Connect definition relationships

9.2.1 IMS Connect setup

We used the JCL procedure as shown in Example 9-1 to start IMS Connect.

Example 9-1 IMS Connect startup procedure

```
//IM4CONN PROC RGN=OM,SOUT=S,SYS1=,
//          BPECFG=BPECFG00,
//          HWSCFG=HWSCFG00,
//          TCPDATA=TCPDATA
//*
//*****
/* BRING UP AN IMS CONNECT SYSTEM *
//*****
//STEP1 EXEC PGM=HWSHWS00,REGION=&RGN,TIME=1440,
//          PARM='BPECFG=&BPECFG,HWSCFG=&HWSCFG'
//STEPLIB DD DSN=IMS910A.SDFSRESL,DISP=SHR
//PROCLIB DD DSN=IMS910A.&SYS1.PROCLIB,DISP=SHR
//SYSTCPD DD DSN=TCP.&SYSNAME..TCPARM(&TCPDATA),DISP=SHR
//SYSPRINT DD SYSOUT=&SOUT
//SYSUDUMP DD SYSOUT=&SOUT
```

```
//HWSRCORD DD DSN=IMS910A.HWSRCORD,DISP=SHR
```

Since we wanted to use local IMS J2C connection factories, we made the IMS Connect program HWSHWS00 non-swappable and added LOCAL as a PORTID (see Example 9-2).

Example 9-2 IMS Connect configuration parameters

```
HWS (ID=IMSACONN,RRS=Y,RACF=Y)
TCPIP (HOSTNAME=TCPIP,PORTID=(6001,LOCAL),MAXSOC=2000)
DATASTORE (ID=IMSA,GROUP=IMSAEXCF,MEMBER=HWS910A,TMEMBER=SCSIMS9A)
```

Notes about the example:

- ▶ The HWS ID provides the value of custom property IMSConnectName for our local J2C connection factory.
- ▶ The DATASTORE GROUP value must match the target IMS region's GRPNAME parameter.
- ▶ The DATASTORE MEMBER value must match the target IMS region's OTMANM parameter.
- ▶ The DATASTORE ID provides the custom property DataStoreName for both of our IMS J2C connection factories.
- ▶ The TCPIP PORTID provides the custom property portNumber for our remote J2C connection factory.

Attention: You must enable Resource Recovery Services (RRS) for WebSphere for z/OS two-phase commit support. Also, PORTID must include LOCAL to support local J2C connection factories.

9.2.2 OTMA setup

There are OTMA-related parameters in the IMS startup procedure. In our setup, IMS910A.PROCLIB member DFSPBIV1 has the OTMA-related parameters (see Example 9-3) we used.

Example 9-3 OTMA parameters in IMS procedure

```
GRNAME=IMSAEXCF,
OTMA=Y,
OTMANM=SCSIMS9A,
OTMASE=N,
RRS=Y
```

9.3 Installing the IMS Connect Resource Adapter

Next, we installed the IMS Connect Resource Adapter (imsico.rar).

1. On the WebSphere Administrative Console (Figure 9-5), we expanded **Resources** and selected **Resource Adapters** (see 1 and 2 in Figure 9-5).
2. In the Resource Adapters pane, we selected the node on which we wanted to install the resource adapter. We click **Install RAR** to start the installation process (see 3 and 4 in Figure 9-5).

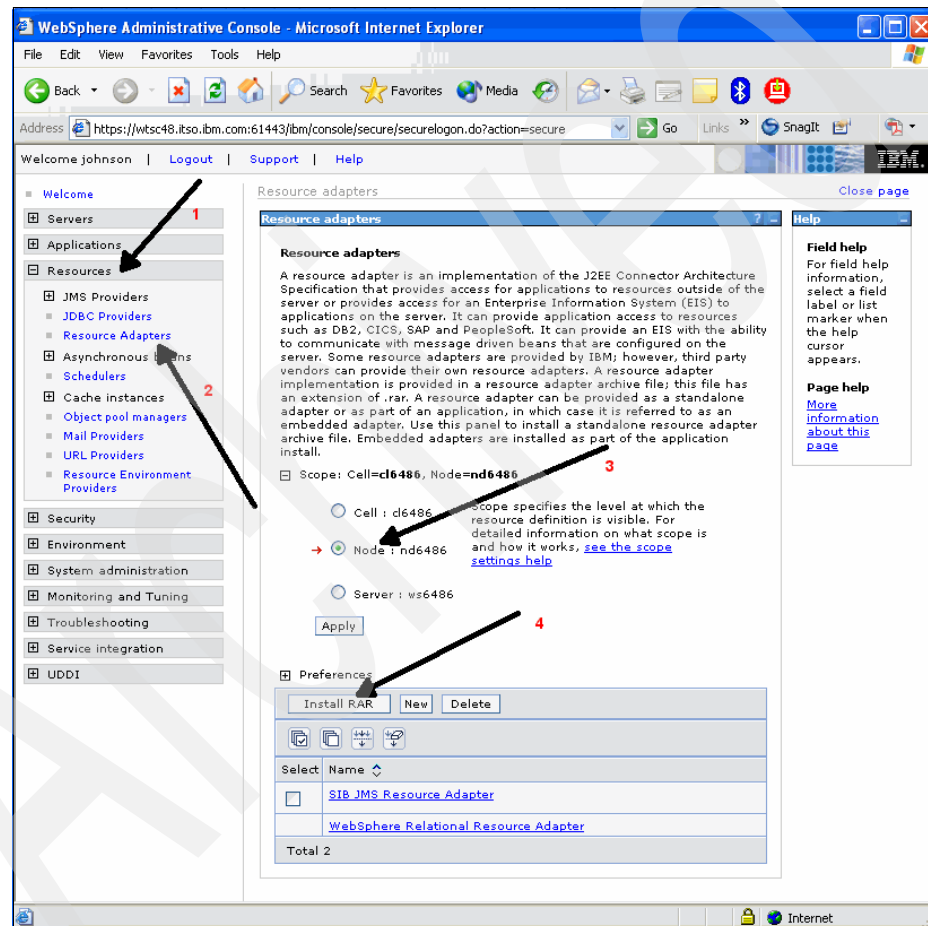


Figure 9-5 WebSphere for z/OS Administrative Console

3. On the next pane (Figure 9-6 on page 302), we traversed to the IMS Connect Resource Adapter Repository (RAR) file, making sure that the desired node

for this resource is selected. The target node *cannot* be the deployment manager node. We clicked **Next**.

Install RAR File

RAR files can be installed using two methods. You can choose to upload a RAR file from local file system or an existing RAR file on a server.

Path

☐ Local path:

Specify path

☒ Server path:

Specify path

Scope

Node

Figure 9-6 Identifying the RAR location

4. We now saw a Resource Adapter configuration pane (Figure 9-7 on page 303). We entered IMS Connect 2.2 as a name for this resource adapter and clicked **OK**.

Configuration

General Properties

* Scope
cells:cl6486:nodes:nd6486

Name
IMS Connect 2.2

Description

Archive path

Class path

Native path

OK Reset Cancel

Figure 9-7 Resource Adapter configuration

This adapter now displays as being installed on this node (see Figure 9-8).

| | |
|---|---------------------------------------|
| Install RAR New Delete | |
| <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> | |
| Select | Name |
| <input type="checkbox"/> | IMS Connect 2.2 |
| <input type="checkbox"/> | SIB JMS Resource Adapter |
| | WebSphere Relational Resource Adapter |
| Total 3 | |

Figure 9-8 List of installed resource adapters

We could have saved the configuration changes at this time, but we wanted to configure J2EE Connector (J2C) connection factories for the IMS Connect adapter.

9.4 Configuring connection factories

To configure the connection factories for IMS Connect, we followed these steps:

1. We clicked the installed IMS Connect Version 2.2 resource adapter; see Figure 9-8 on page 303.

The screenshot shows the 'Configuration' dialog for the 'IMS Connect 2.2' resource adapter. The 'General Properties' tab is selected, displaying the following fields:

- Scope:** cells:cl6486:nodes:nd6486
- Name:** IMS Connect 2.2
- Description:** (empty text area)
- Archive path:** \${CONNECTOR_INSTALL_ROOT}/ims9101.rar
- Class path:** \${CONNECTOR_INSTALL_ROOT}/ims9101.rar
- Native path:** (empty text area)
- Thread pool alias:** Default

At the bottom are buttons for 'Apply', 'OK', 'Reset', and 'Cancel'. The 'Additional Properties' tab is visible on the right, with a red circle highlighting the 'J2C connection factories' link.

Figure 9-9 IMS Connect 2.2 Configuration pane

2. The Resource Adapters → IMS Connect 2.2 - Configuration pane (Figure 9-9) for the resource adapter opened. We clicked **J2C connection factories** under Additional Properties.
3. The Resource Adapters → IMS Connect 2.2 → J2C connection factories pane (not shown) opened. We then clicked **New**.

4. Now the Resource Adapters → IMS Connect 2.2 → J2C connection factories - New Configuration pane opened. There are two types of IMS Connect J2C connection factories we wanted to configure. One directly accesses IMS Connect using XCF (LOCAL). The other accesses IMS Connect using TCP/IP (REMOTE).

9.4.1 Configuring a local J2C connection factory resource

We configured a local J2C connection factory resource by following these steps:

1. On the Resource Adapters → IMS Connect 2.2 → J2C connection factories → New configuration pane (Figure 9-10 on page 306), we entered IMSLOCAL as the name for the factory itself. For its Java Naming and Directory Interface (JNDI) name, we entered eis/IMSLocal and clicked **Apply**.

Connection factory interface
 javax.resource.cdi.ConnectionFactory

Category

Component-managed authentication alias
 Component-managed authentication alias
 (none)

Authentication alias for XA recovery
☒ Use component-managed authentication alias
☐ Specify:

Container-managed authentication
 Container-managed authentication alias
 (deprecated in V6.0, use resource reference authentication settings instead)
 (none)

Related Items

- J2EE Connector Architecture (J2C) authentication data entries

Figure 9-10 IMS local J2C connection factory general properties

Figure 9-10 IMS local J2C connection factory general properties

Note: Component and Container-managed authentication alias are used to administer user ID and password combinations across multiple J2C connection factories:

- ▶ A Component-managed authentication alias is used to provide a user ID and password when the application's res-auth deployment descriptor specifies Application and no explicit user ID and password are provided on the connection request.
- ▶ A Container-managed authentication alias is used to provide a user ID and password when the application's res-auth deployment descriptor specifies Container.

We did not use these aliases so we did not specify them here.

2. We now had access to the Additional Properties (see Figure 9-11).

The screenshot shows a configuration window titled 'Configuration' with a tab labeled 'Configuration'. Inside, there are two main sections: 'General Properties' and 'Additional Properties'. The 'General Properties' section includes fields for 'Scope' (cells:cd6486:nodes:nd6486), 'Name' (IMSLocal), 'JNDI name' (eis/IMSLocal), 'Description' (empty), and 'Connection factory interface' (javax.resource.cci.ConnectionFactory). The 'Additional Properties' section has three links: 'Connection pool properties', 'Advanced connection factory properties', and 'Custom properties'. Below these is a 'Related Items' section with a link to 'J2EE Connector Architecture (J2C) authentication data entries'.

Figure 9-11 J2C connection factory additional properties

3. We next clicked **Custom Properties** under Additional Properties.
4. The J2C connection factory custom properties pane (Figure 9-12 on page 308) opened. We entered the DataStoreName (IMSA) and the IMSConnectName (IMSACONN) provided by the IMS system programmer and clicked **OK**.

| Name | Value | Description | Required |
|---|--------------------------|---|----------|
| IMSConnectName | IMSACONN | Name of the target IMS Connect - for Local Option only | false |
| HostName | myHostNm | TCP/IP host name of the target IMS Connect | false |
| PortNumber | 0 | Target TCP/IP port number of IMS Connect | false |
| DataStoreName | IMSA | Name of the target IMS datastore | false |
| SSLEnabled | FALSE | Indicates if SSL is enabled for this connection factory | false |
| SSLEncryptionType | Weak | The type of cipher suite to be used for encryption | false |
| SSLKeyStoreName | | Name (full path) of SSL keystore for client certificates/private keys | false |
| SSLKeyStorePassword | | Password of SSL keystore for client certificates/private keys | false |
| SSLTrustStoreName | | Name (full path) of SSL keystore for trusted certificates | false |
| SSLTrustStorePassword | | Password of SSL keystore for trusted certificates | false |
| CMODedicated | FALSE | Indicates if sockets are dedicated to specific CM0 clients. | false |
| UserName | | Default name of the user to be authorized | false |
| Password | | Default password of the user | false |
| GroupName | | Default name of the IMS group of the user | false |
| TraceLevel | 1 | Level of information to be traced. | false |
| TransactionResourceRegistration | dynamic | Type of transaction resource registration (enlistment). Valid values are either "static" (immediate) or "dynamic" (deferred). | false |
| MFSXMIRepositoryID | default | Unique identifier of MFS XMI Repository. | false |
| MFSXMIRepositoryURI | | Location of MFS XMI Repository. | false |
| Total 18 | | | |

Figure 9-12 IMS J2C Local custom properties

Note the following explanations for these properties:

- DataStoreName

The name of the target IMS data store. It must match the ID parameter of the DataStore statement specified in the IMS Connect configuration member (Example 9-2 on page 300).

- HostName

Not used for Local J2C connection factories.

- IMSConnectName

The IMS Connect name. It must match the ID parameter of the HWS statement specified in the IMS Connect configuration member.

- PortNumber

Not used for local J2C connection factories.

- TraceLevel

- 0: No tracing or logging occurs.
- 1: Only errors and exceptions are logged.
- 2: Errors and exceptions plus the entry and exit of important methods are logged.
- 3: Errors and exceptions, the entry and exit of important methods, and the contents of buffers sent to and received are logged.

- Userid

A default user ID to be used for this connection if no other is provided.

- Password

A password to be used with the above default user ID.

- GroupName

RACF group to be used with the user ID and password.

Tip: If you need to add additional J2C connection factories, you can click **>J2C connection factories>** at the top of the pane to redisplay the list of J2C connection factories (see Figure 9-13) that are configured for this adapter. Then repeat this process. Otherwise, save the changes.

5. Our local connection factory now shows up in the list of connection factories (Figure 9-13 on page 310).





| New Delete | | | | |
|---|--------------------------|--------------|-------------|-------------------------------|
|     | | | | |
| Select | Name | JNDI name | Description | Connection factory interface |
| <input type="checkbox"/> | IMSLocal | eis/IMSLocal | | javax.resource.cci.Connection |
| Total 1 | | | | |

Figure 9-13 List of installed J2C connection factories

9.4.2 Configuring a remote IMS J2C factory resource

To configure a remote IMS J2C factory resource, we followed these steps:

1. On the Configuration pane of the resource adapter (see Figure 9-9 on page 304), we clicked **J2C connection factories**.
2. The J2C connection factory pane was displayed. On this pane, we clicked **New**.
3. On the New J2C connection factory pane for Name, we entered IMSRemote for the name of the factory. For its JNDI name, we entered eis/IMSRemote and clicked **Apply**.

Note: The Component-managed authentication alias provides a user ID and password when the application's res-auth deployment descriptor specifies Application and no explicit user ID and password is provided on the connection request.

The Container-managed authentication alias provides a user ID and password when the application's res-auth deployment descriptor specifies Container.

4. Additional properties was now available on the pane (see Figure 9-14 on page 311). We clicked **Custom properties**.

Configuration

General Properties

* Scope
cells:cl6486:nodes:nd6486

* Name
IMSRemote

JNDI name
eis/IMSRemote

Description

* Connection factory interface
javax.resource.cdi.ConnectionFactory

Additional Properties

- [Connection pool properties](#)
- [Advanced connection factory properties](#)
- [Custom properties](#)

Related Items

- [J2EE Connector Architecture \(J2C\) authentication data entries](#)

Figure 9-14 IMS Remote J2C connection factory

- The Custom properties pane (see Figure 9-15 on page 312) was displayed. We entered the DataStoreName (IMSA), the HostName (wtsc48.itso.ibm.com), and the PortNumber (6001) provided by the IMS System programmer.

| Name ▾ | Value ▾ | Description ▾ | Required |
|---|-------------------------------------|---|----------|
| IMSConnectName | | Name of the target IMS Connect - for Local Option only | false |
| HostName | wtsc48.itso.ibm.com | TCP/IP host name of the target IMS Connect | false |
| PortNumber | 6001 | Target TCP/IP port number of IMS Connect | false |
| DataStoreName | IMSA | Name of the target IMS datastore | false |
| SSLEnabled | FALSE | Indicates if SSL is enabled for this connection factory | false |
| SSLEncryptionType | Weak | The type of cipher suite to be used for encryption | false |
| SSLKeyStoreName | | Name (full path) of SSL keystore for client certificates/private keys | false |
| SSLKeyStorePassword | | Password of SSL keystore for client certificates/private keys | false |
| SSLTrustStoreName | | Name (full path) of SSL keystore for trusted certificates | false |
| SSLTrustStorePassword | | Password of SSL keystore for trusted certificates | false |
| CMODedicated | FALSE | Indicates if sockets are dedicated to specific CMO clients. | false |
| UserName | | Default name of the user to be authorized | false |
| Password | | Default password of the user | false |
| GroupName | | Default name of the IMS group of the user | false |
| TraceLevel | 1 | Level of information to be traced. | false |
| TransactionResourceRegistration | dynamic | Type of transaction resource registration (enlistment). Valid values are either "static" (immediate) or "dynamic" (deferred). | false |
| MFSXMIRepositoryID | default | Unique identifier of MFS XMI Repository. | false |
| MFSXMIRepositoryURI | | Location of MFS XMI Repository. | false |
| Total 18 | | | |

Figure 9-15 IMS remote J2C connection factory custom properties

Note the following explanation of the properties listed in Figure 9-15 on page 312:

- DataStoreName
The name of the target IMS data store. It must match the ID parameter of the DataStore statement specified in the IMS Connect configuration member (Example 9-2 on page 300).
- HostName
The TCP/IP host name of the system on which the IMS Connect task is running.
- IMSConnectName
Is not used for remote J2C connection factories.
- PortNumber
The port on which the IMS Connect task is listening (Example 9-2 on page 300).
- TraceLevel:
 - 0: No tracing or logging occurs.
 - 1: Only errors and exceptions are logged.
 - 2: Errors and exceptions, plus the entry and exit of important methods, are logged.
 - 3: Errors and exceptions, the entry and exit of important methods, and the contents of buffers sent to and received are logged.
- Userid
A default user ID to be used for this connection if no other is provided.
- Password
A password to be used with the default user ID.
- GroupName
An RACF group to be used with the above user ID and password.

Tip: If you need to add additional J2C connection factories, you can click **J2C connection factories** at the top of the pane to redisplay the list of J2C connection factories configured for this adapter. Repeat this process. Otherwise, save the changes.

6. Since there are no other J2C connection factories to enter, we saved the configuration changes.

9.5 Cell scope J2C connection factories

J2C connection factories can be configured at the cell level as an alternative to configuring J2C connection factories on each node. To do this the IMS Connect resource adapter must be installed on each node in the cell. This ensures that the implementation code (Java Archive (JAR) files) of the resource adapter is available to the Java Virtual Machines (JVMs) running on that node.

1. On the WebSphere Administration Console, we expanded **Resource** and clicked **Resource Adapters**.
2. We set the scope to the Cell level.
3. On the Resource Adapter pane (Figure 9-16), we clicked **New** to start the configuration process.

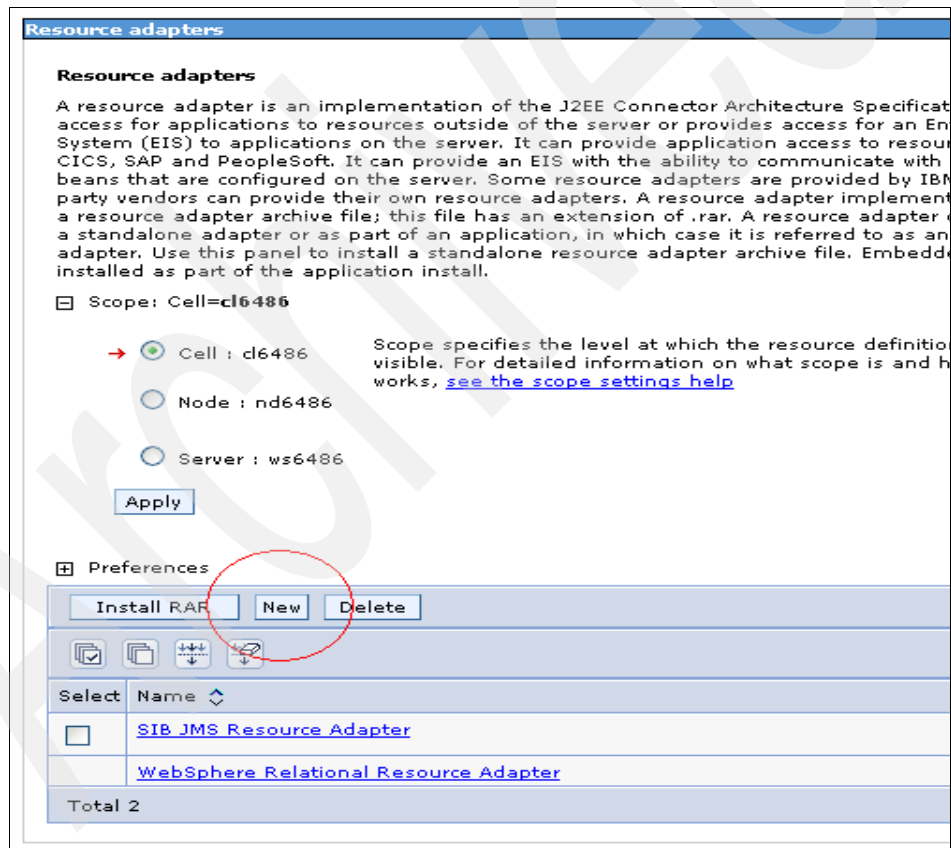


Figure 9-16 Install new adapter at the Cell scope

4. On the General Properties panel (Figure 9-17), for Name we entered IMS Connect 2.2. For Archive Path, we selected the **Choose an archive path from the list of installed RARs** option to display a list of previously installed resource adapters. We selected the **imscon.rar** file and clicked **OK**.

The screenshot shows the 'Configuration' dialog for a J2C connection factory. The 'General Properties' tab is active. The 'Scope' is set to 'cells:d6486'. The 'Name' field is empty. The 'Description' field is empty. The 'Archive path' section has two options: 'Choose an archive path from the list of installed RARs (recommended)' (selected) and 'Specify the archive path of an installed RAR'. Below the first option, a list of installed RAR files shows '\$\${CONNECTOR_INSTALL_ROOT}/ims9101.rar' selected. Below the second option, there is a text field for specifying the archive path. The 'Class path' and 'Native path' fields are empty. The 'Thread pool alias' is set to 'Default'. At the bottom, there are buttons for 'Apply', 'OK', 'Reset', and 'Cancel'. On the right side, there is a list of 'Additional Properties' including 'J2C Activation specification', 'J2C administrative objects', 'J2C connection factories', 'Custom properties', and 'View Deploy Descriptor'.

Figure 9-17 J2C connection factory Cell scope

The J2C connection factories that are installed at the Cell scope are now available to every node in the cell. The adapter is now displayed as being

available at the cell level. See 9.4.1, “Configuring a local J2C connection factory resource” on page 305, and 9.4.2, “Configuring a remote IMS J2C factory resource” on page 310, for details about configuring J2C factories.

9.6 Application deployment

The JNDI name specified in the application (see JNDI name in Figure 9-19 on page 317) needs to be resolved to a JNDI name of one of our J2C connection factories during deployment.

During the installation of the Trader application, a pane (Figure 9-18) is presented that lists the JNDI names found in the application. From the pull-down, a list of available J2C connection factories is displayed. We selected the desired factory and the desired modules. Click **Apply**.

Map resource references to resources

Each resource reference that is defined in your application must be mapped to a resource.

javax.resource.cci.ConnectionFactory

To set multiple existing resource JNDI names:

1. Select one or more checkboxes in the table

2. Select existing resource JNDI name

3. Click Apply

Specify existing Resource JNDI name:

Select...

Apply

To modify Resource Authentication method (if Authorization type is 'container'):

1. Select one or more checkboxes in the table

2. Select either 'none', 'default', or 'custom login configuration'

■ if 'none' is selected:

a. Select one or more checkboxes in the table

■ if 'default' is selected:

a. select an authentication data entry from the dropdown menu

b. Click Apply

■ if 'custom login configuration' is selected:

a. select a custom login configuration from the dropdown menu

b. Click Apply

c. To edit the properties of the custom login configuration, click Mapping Properties in the table

Specify authentication method:

☐ none

☒ Use default method

Select authentication data entry

Select...

☐ Use custom login configuration

Select application login configuration

Select...

Apply

Select

Module

EJB

URI

Reference binding

JNDI name

Login configuration

☐

TraderIMSJ2CEJB

Trader

TraderIMSJ2CEJB.jar,META-INF/ejb-jar.xml

IMS

eis/IMSRemote

Resource authorization:
Container

Authentication method:
DefaultPrincipalMapping

Figure 9-18 Mapping resource references to a J2C connection factory

We then continued with the installation and successfully executed the application.

Tip: You can modify the J2C connection factory later without re-installing the application.

9.7 Implementing application security

Each application that accesses a J2C resource has a resource deployment descriptor attribute called *res-auth* that determines the authentication behavior when accessing the resource. There are two options:

- ▶ Container authentication

When container authentication is specified (Figure 9-19), the user ID and password used on the connection are provided by the container. The user ID and password combination is provided to the container by the J2C connection factory in the Container-managed authentication alias entry in the Configuration - General Properties pane of the J2C connection factory.

- ▶ Application authentication

When application authentication is specified, the user ID and password used on the connection are provided explicitly by the application or by the J2C connection factory in the Component-managed authentication alias entry in the Configuration - General Properties pane of the J2C connection factory.

The screenshot shows the 'References' pane on the left with a tree view containing 'Trader' and 'ResourceRef IMS'. The main configuration area on the right has the following fields:

- Name: IMS
- Description: (empty)
- Type: javax.resource.cci.ConnectionFactory
- Authentication: Application
- Sharing scope: Shareable

Below these fields is the 'WebSphere Bindings' section, which includes the text 'The following are binding properties for the WebSphere Application Server.' and the following fields:

- JNDI name: eis/IMSRemote
- JAAS Login Configuration: ☒ None, ☐ Use Default Method
- Authentication Alias: (empty)
- ☐ Use Custom Login Configuration
- Login Configuration Name: (empty)

At the bottom left of the main area are 'Add...' and 'Remove' buttons.

Figure 9-19 EJB Deployment Editor - Authorization - Container

In this scenario, we wanted the application to provide the user ID and password. Therefore, we changed the Authentication deployment descriptor from Container to Application.

9.7.1 Connection properties

When the deployment descriptor is set to application-managed security, additional parameters (user ID and password) must be present in application connection requests to IMS. A J2C Java Bean by default does not provide access to the parameters. To add these parameters to a J2C Java Bean, we manually modified the J2C doclet tags (see Example 9-4) before the `runTraderbl` method in the J2C Java Bean. When the J2C Java Bean is saved, the code required to pass these parameters is automatically generated.

Example 9-4 Sample connection J2C doclet code

```
* @j2c.connectionSpec class="com.ibm.connector2.ims.ico.IMSConnectionSpec"
* @j2c.connectionSpec-property name="UserName" argumentBinding="ivuserid"
* @j2c.connectionSpec-property name="Password" argumentBinding="ivpassword" *
*@generated
```

The code that invokes the modified J2C Java Bean method (for example, `runTraderbl`) must be modified to pass the user ID and password on each invocation of the method as shown here:

```
outbuffer = proxy.runTrader(inbuffer,ivUserID,ivPassword);
```

Table 9-1 identifies the available connection properties that can be accessed using doclet tags.

Table 9-1 ConnectionSpec properties

| ConnectionSpec property name | Java type |
|------------------------------|-----------|
| UserName | string |
| Password | string |
| GroupName | string |
| ClientID | string |

9.7.2 Interaction properties

Interaction properties are not included by default in J2C Java Beans. The doclet tags must be manually added to the J2C Java Bean (see Example 9-5 on page 319) and when the class is saved, the required code is generated.

```
/**
 * @j2c.interactionSpec
 class="com.ibm.connector2.ims.ico.IMSInteractionSpec"
 * @j2c.interactionSpec-property name ="CommitMode"
 argumentBinding="ivcommit"
 * @j2c.interactionSpec-property name ="SocketTimeout"
 argumentBinding="ivT0"
 * @generated
 */
```

Table 9-2 identifies the available interaction properties that can be accessed using doclet tags.

Table 9-2 *InteractionSpec* properties

| InteractionSpec property name | Java type |
|-------------------------------|-----------|
| AsyncOutputAvailable | boolean |
| CommitMode | int |
| ConvEnded | boolean |
| ExecutionTimeout | int |
| ImsRequestType | int |
| InteractionVerb | int |
| LtermName | string |
| MapName | string |
| PurgeAsyncOutput | boolean |
| ReRoute | boolean |
| ReRouteName | String |
| SocketTimeout | int |

9.8 Implementing infrastructure security

Next we want to use SSL to protect our communications between WebSphere and the remote IMS Connect server. For our test we used RACF as our certificate authority (CA).

1. First, we created a RACF digital certificate for our ITSO IMS certificate authority.

Example 9-6 Generate an ITSO Certificate Authority (CA) certificate

```
racdcert certauth gencert subjectsdn(cn('ITSO IMS CertAuth') ou('ITSO') +  
o('IBM')) withlabel('ITSO IMS CertAuth') keyusage(certsig) +  
notafter(date(2010/12/31))
```

2. We next created a RACF keyring (1) and a client certificate (2) signed by our ITSO IMS CA certificate for the user ID associated with the IMS Connect task. We then connected our client certificate (3) and the ITSO IMS CA certificate (4) to this RACF keyring.

Note: The numbers between brackets associate to the numbers in Example 9-7 on page 320.

Example 9-7 Create IMS Connect client keyring and client certificate

```
racdcert id(stc) addring(IMSConnectKeyring) 1  
racdcert id(stc) gencert subjectsdn(cn('IMSConnect') ou('ITSO') o('IBM')) +  
withlabel('IMSConnect') signwith(certauth label('ITSO IMS CertAuth')) 2  
racdcert id(stc) connect(label('IMSConnect') ring(IMSConnectKeyring) default 3  
racdcert id(stc) connect(label('ITSO IMS CertAuth') ring(IMSConnectKeyring) +  
certauth 4
```

9.8.1 RACF keyrings

For RACF keyrings:

1. The client component is IMS Connector for Java or IC4J. IMS Connector for Java V2.2.3 and later supports both JSSE keystores and RACF keyrings. Initially, we wanted to use RACF, so we created a RACF keyring, as described in 9.8, “Implementing infrastructure security” on page 319, (5) for the user ID associated with the WebSphere servant region (ASSR1). We then connected our ITSO IMS CA certificate (6) to this RACF keyring.

Restriction: RACF does not allow a user to use or access another user's private key. Therefore, the RACF keyring used in a WebSphere on z/OS IMS J2C Connection factory must be owned by the servant's identity.

Example 9-8 Create an IMS Keyring for WebSphere

```
racdcert id(assr1) addring(IC4JKeyring) 5  
racdcert id(assr1) connect(label('ITSO IMS CertAuth') ring(IC4JKeyring)  
certauth) 6
```

2. Next, we wanted to enable mutual authentication between a WebSphere client and an IMS Connect server. The first step was to generate a personal

certificate for the client (7). We then connected this personal certificate to our WebSphere client keyring (8).

Example 9-9 Create a Websphere personal certificate for IMS connectivity

```
racdcert id(assr1) gencert subjectsdn(cn('IMSConnectClient') ou('ITS0') +  
o('IBM')) withlabel(IC4JKeyring') signwith(certauth label('ITS0 IMS +  
CertAuth')) 7  
racdcert id(assr1) connect(label('IC4JClient') + ring(IC4JKeyring) default 8
```

3. Finally, we specified the IMS Connect for Java keyring in the *SSLKeyStoreName* and *SSLTrustStoreName* custom properties in our IMS J2C Connection factory; see Figure 9-20 on page 322.

Note: The format of the keystore name is *Provider;keyring label;userid* (with each parameter separated by semicolons). The provider must be either JCERACFKS for software SSL encryption or JCE4758RACFKS for hardware SSL encryption. Any other value for the provider name indicates a JSSE keystore rather than a RACF keyring.

| Name | Value | Description | Require |
|---|---|---|-----------------------|
| IMSConnectName | | Name of the target IMS Connect - for Local Option only | false |
| HostName | wtsc48.itso.ibm.com | TCP/IP host name of the target IMS Connect | false |
| PortNumber | 6003 | Target TCP/IP port number of IMS Connect | false |
| DataStoreName | IMSA | Name of the target IMS datastore | false |
| SSLEnabled | true | Indicates if SSL is enabled for this connection factory | false |
| SSLEncryptionType | Strong | The type of cipher suite to be used for encryption | false |
| SSLKeyStoreName | JCERACFKS;IC4JKeyring;ASSR1 | Name (full path) of SSL keystore for client certificates/private keys | false |
| SSLKeyStorePassword | | Password of SSL keystore for client certificates/private keys | false |
| SSLTrustStoreName | JCERACFKS;IC4JKeyring;ASSR1 | Name (full path) of SSL keystore for trusted certificates | false |
| SSLTrustStorePassword | | Password of SSL keystore for trusted certificates | false |
| CMODedicated | FALSE | Indicates if sockets are dedicated to specific CM0 clients. | false |
| UserName | | Default name of the user to be authorized | false |
| Password | | Default password of the user | false |
| GroupName | | Default name of the IMS group of the user | false |
| TraceLevel | 3 | Level of information to be traced. | false |
| TransactionResourceRegistration | dynamic | Type of transaction resource registration (enlistment). Valid values are either "static" (immediate) or "dynamic" (deferred). | false |
| MFSXMIRepositoryID | default | Unique identifier of MFS XMI Repository. | false |
| MFSXMIRepositoryURI | | Location of MFS XMI Repository. | false |
| Total 18 | | | |

Figure 9-20 IMS J2C customer properties with RACF keyrings

9.8.2 JSSE keyrings

For JSSE keyrings:

1. IMS Connector for Java also supports JSSE keyrings. When we used JSSE keystores we needed to connect our ITSO IMS CA certificate to a JSSE keyring. As described in 9.10, “Performance and availability” on page 330, we exported (9) the ITSO IMS CA certificate and used the OPUT command (10) to move the exported data set to OMVS. We then imported (11) the ITSO IMS CA certificate into a new JSSE keystore. This CA certificate allows the verification of client certificate provided by IMS Connect.

Example 9-10 Export ITSO IMS CA certificate and import into keystore

```
racdcert certauth export(label('ITSO IMS CertAuth')) dsn(itsoims.der) +  
format(certder) 9  
oput itsoims.der '/u/johnson/itsoims.der' binary convert(no) 10  
keytool -import -v -trustcacerts -alias "ITSO Server CA" -file itsoims.der  
-keystore jsse.jks 11
```

Note: Importing the certificate created the keystore file. During the import we were prompted for a password and we entered password.

2. We next wanted to enable mutual authentication between WebSphere and IMS Connect. As shown in Figure 9-11, the first step is to generate (12) a personal certificate for our JSSE keystore. We then created a certificate request (13) using this personal certificate. We moved this request from OMVS to an MVST[™] data set. We next signed this personal certificate (14) with our ITSO IMS CA certificate and exported (15) the signed certificate and moved it to OMVS (16), where we imported the signed certificate (17) into our JSSE keystore.

Example 9-11 Create personal certificate request and sign it with ITSO IMS CA

```
keytool -genkey -alias "WebSphere/IMS" -dname "cn=was, ou=ITSO, o=IBM"  
-keystore jsse.jks -keyalg RSA 12  
keytool -certreq -alias "WebSphere/IMS" -file certreq.arm -keystore jsse.jks 13  
racdcert id(stc) gencert(certreq.arm) withlabel('WebSphere/IMS') +  
signwith(certauth label('ITSO IMS CertAuth')) notafter(date(2005/12/31)) 14  
racdcert id(stc) export(label('WebSphere/IMS')) dsn(client.der) format(certder)  
15  
oput client.der '/u/johnson/client.der' binary convert(no) 16  
keytool -import -v -alias "WebSphere/IMS" -file client.der -keystore jsse.jks  
17
```

Note: In order to move the certificate request from in the correct code page, we had to FTP the file to Windows in binary format and then FTP the file to MVS to a VB dataset in ASCII format.

3. Finally, we added a new IMS J2C Connection factory to WebSphere (see Figure 9-21 on page 325) and verified that we were not able to connect from WebSphere to IMS Connect unless the proper client and CA certificates were present in the IMS Connect keyring and the IMS Connector for Java keystore.

| Name | Value | Description | Required |
|---|-------------------------------------|---|----------|
| IMSConnectName | | Name of the target IMS Connect - for Local Option only | false |
| HostName | wtsc48.itso.ibm.com | TCP/IP host name of the target IMS Connect | false |
| PortNumber | 6003 | Target TCP/IP port number of IMS Connect | false |
| DataStoreName | IMSA | Name of the target IMS datastore | false |
| SSLEnabled | true | Indicates if SSL is enabled for this connection factory | false |
| SSLEncryptionType | Weak | The type of cipher suite to be used for encryption | false |
| SSLKeyStoreName | /u/johnson/isse.iks | Name (full path) of SSL keystore for client certificates/private keys | false |
| SSLKeyStorePassword | password | Password of SSL keystore for client certificates/private keys | false |
| SSLTrustStoreName | /u/johnson/isse.iks | Name (full path) of SSL keystore for trusted certificates | false |
| SSLTrustStorePassword | password | Password of SSL keystore for trusted certificates | false |
| CMODedicated | FALSE | Indicates if sockets are dedicated to specific CMQ clients. | false |
| UserName | | Default name of the user to be authorized | false |
| Password | | Default password of the user | false |
| GroupName | | Default name of the IMS group of the user | false |
| TraceLevel | 3 | Level of information to be traced. | false |
| TransactionResourceRegistration | dynamic | Type of transaction resource registration (enlistment). Valid values are either "static" (immediate) or "dynamic" (deferred). | false |
| MFSXMIRepositoryID | default | Unique identifier of MFS XMI Repository. | false |
| MFSXMIRepositoryURI | | Location of MFS XMI Repository. | false |
| Total 18 | | | |

Figure 9-21 IMS J2C SSL custom properties for JSSE keystores

9.8.3 IMS Connect and SSL

Enabling SSL in IMS Connect required configuration changes. We added SSLPORT and SSLENVAR to the IMS Connect parameters (see Example 9-12

on page 326) and specified our IMS Connect keyring and certificate label (see Example 9-13) as our SSL parameters.

Example 9-12 IMS Connect parameters

```
HWS (ID=IMSACONN,RACF=N,XIBAREA=20,RRS=Y)
TCPIP (HOSTNAME=TCPIP,PORTID=(6001,LOCAL),MAXSOC=2000,TIMEOUT=3000,
SSLPORT=(6003),SSELENVAR=HWSSL)
DATASTORE (ID=IMSA,GROUP=IMSAEXCF,MEMBER=HWS910A,TMEMBER=SCSIMS9A)
```

Example 9-13 IMS Connect SSL parameters

```
DEBUG_SSL=ON
GSK_CLIENT_AUTH_TYPE=GSK_CLIENT_AUTH_FULL_TYPE
GSK_SESSION_TYPE=GSK_SERVER_SESSION_WITH_CL_AUTH
#GSK_SESSION_TYPE=GSK_SERVER_SESSION
GSK_PROTOCOL_SSLV2=GSK_PROTOCOL_SSLV2_ON
GSK_PROTOCOL_SSLV3=GSK_PROTOCOL_SSLV3_ON
GSK_PROTOCOL_TLSV1=GSK_PROTOCOL_TLSV1_ON
GSK_KEYRING_FILE=IMSConnectKeyring
GSK_KEYRING_PW=
GSK_KEYRING_STASH_FILE=
GSK_KEYRING_LABEL=IMSConnect
GSK_V2_CIPHER_SPECS=6321
GSK_V3_CIPHER_SPECS=0906030201
```

For more information regarding SSELENVAR parameters see *IMS V9 Connect Guide and Reference*, SC18-9287-01.

9.9 Configuration automation using Jython scripts

We also used Jython scripts to automate the installation of the IMS Connect resource adapter and the configuration of J2C connection factories.

The Jython script in Figure 9-14 was used to install the IMS Connect Resource Adapter. We invoked the script using the **wsadmin** command in a batch environment using the JCL in Example 9-15 on page 327.

Example 9-14 Install an IMS Resource Adapter using wsadmin commands

```
#####
# Jython script to install an IMS Connect Resource Adapter
#####
# IMS HFS directory prefix
imsconnPrefix = '/usr/lpp/IMSIC0/J2C/'
# Additional classpath jar(s) - Optional
ClassPathJar = ''
```

```

# Resource Adapter Name
rarName = "IMS Connect 2.2"
symparmName = ['value',rarName]
attrs = [symparmName]
# Full path to rar file
rarFile = imskonPrefix + 'imsico.rar'
# Resource Adapter options
options = "-rar.name '" + rarName + "' -rar.classpath " + ClassPathJar
raOptions = '[' + options + ']'

# Access system property for line separator
import java.lang.System as sys
lineSeparator = sys.getProperty('line.separator')

# Build list of Cell names
cells = AdminConfig.list('Cell').split(lineSeparator)
for cell in cells:
    nodes = AdminConfig.list('Node',cell).split(lineSeparator)
    for node in nodes:
        nodeName = AdminConfig.showAttribute(node,'name')
        if nodeName != 'NetworkManager':
            print "Installing IMS Connect Resource Adapter " + rarName + " in Node " + nodeName +
                  " from " + rarFile
            ra = AdminConfig.installResourceAdapter(rarFile,nodeName,options)

print "Saving Configuration Changes"
AdminConfig.save()

```

Example 9-15 JCL used to invoke the wsadmin command

```

/*****
/* STEP 1 - Invoke wsadmin
/*****
//WSADMIN EXEC PGM=IKJEFT01,REGION=OM
//STDENV DD *
_CBCONFIG=/WebSphereAL6/V6R0/BS06
_ASDIR=AppServer
/*
//SYSTSPRT DD SYSOUT=*
//SYSTSIN DD *
BPXBATCH SH +
  $_CBCONFIG+
  /$_ASDIR+
  /bin/wsadmin.sh -conntype RMI -port 61809 -lang jython +
  -user JOHNSON -password password +
  -f /u/johnson/jac1/imsj2c.jython +
  1> /tmp/imsrar.out +
  2> /tmp/imsrar.err

```

```

/*
/*****
/* STEP 2 Copy - Copy script output back to joblog */
/*****
//SYSOUT EXEC PGM=IKJEFT01,DYNAMNBR=300
//SYSTSPRT DD SYSOUT=*
//JACLOUT DD PATH='/tmp/imsrar.out'
//JACLERR DD PATH='/tmp/imsrar.err'
//STDOUT DD SYSOUT=*,DCB=(LRECL=1000,RECFM=V)
//SYSPRINT DD SYSOUT=*
//SYSTSIN DD *
OCOPY INDD(JACLERR) OUTDD(STDOUT)
OCOPY INDD(JACLOUT) OUTDD(STDOUT)

```

We also used Jython scripts to configure our IMS J2C connection factories. The Jython code in Example 9-16 is a representation of the code we used. The complete scripts for installing an IMS resource adapter and configuring IMS J2C connection factories is provided with the additional materials.

Example 9-16 Installing a local IMS J2C Connection factory

```

#####
# Jython script to install an IMS J2C connection factories
#####
IMSCONN_RA = "IMS Connect 2.2"
#####
# IMS J2C Connection Factory Custom Properties
IMSCONNECTNAME = ['value','IMSACONN']
DATASTORENAME = ['value','IMSA']
HOSTNAME = ['value','wtsc48.itso.ibm.com']
TCPPORT = ['value',6001]
TRACE = ['value',3]

# Access system property for line separator
import java.lang.System as sys
lineSeparator = sys.getProperty('line.separator')

# Build list of Cell names
cells = AdminConfig.list('Cell').split(lineSeparator)
for cell in cells:
    nodes = AdminConfig.list('Node',cell).split(lineSeparator)
    for node in nodes:
        nodeName = AdminConfig.showAttribute(node,'name')
        cellName = AdminConfig.showAttribute(cell,'name')
        if nodeName != 'NetworkManager':

            # Retrieve the IMS Connect Resource Adapter
            getID = "/Cell:" + cellName + "/Node:" + nodeName +
                "/J2CResourceAdapter:" + IMSCONN_RA + "/"

```

```

ra = AdminConfig.getid(getID)

#####
print "Creating a Local IMS J2C Connection Factory in " + nodeName
name_attr = ['name','IMSLocal']
jndi_attr = ['jndiName','eis/IMSLocal']

attrs = [name_attr,jndi_attr]

# Create a Connection Factory
cf = AdminConfig.create('J2CConnectionFactory',ra,attrs)

# Parse the J2C factory's custom properties and set values
propSet = AdminConfig.showAttribute(cf,'propertySet')
rpsList = AdminConfig.showAttribute(propSet,'resourceProperties')
# Strip off the first and last bracket and split into an array
rps = rpsList[1:len(rpsList)-1].split(" ")
for rp in rps:
    name = AdminConfig.showAttribute(rp,'name');
    if name == 'IMSConnectName':
        AdminConfig.modify(rp,[IMSCONNECTNAME])
    if name == 'DataStoreName':
        AdminConfig.modify(rp,[DATASTORENAME])
    if name == 'TraceLevel':
        AdminConfig.modify(rp,[TRACE])

print "Saving Configuration Changes"
AdminConfig.save()

```

The table below identifies the IMS J2C connection factory properties that could be configured using this same technique.

Table 9-3 IMS J2C connection factory properties

| IMS J2C factory property name | Type |
|-------------------------------|---------|
| IMSConnectName | String |
| HostName | String |
| PortNumber | Integer |
| DataStoreName | String |
| SSLEnabled | Boolean |
| SSLEncryptionType | String |
| SSLKeyStoreName | String |

| IMS J2C factory property name | Type |
|---------------------------------|---------|
| SSLKeyStorePassword | String |
| SSLTrustStoreName | String |
| SSLTrustStorePassword | String |
| CMODedicated | Boolean |
| UserName | String |
| Password | String |
| TraceLevel | Integer |
| TransactionResourceRegistration | String |
| MFXSXMLRepositoryID | String |
| MFSXMLRepositoryURI | String |

9.10 Performance and availability

A *local* connection provides better performance than a remote connection since no network-related overhead is involved in processing a request. The implication is that a local connection has a single point of failure if the local IMS Connect task region is not available.

Use of a remote connection along with usage of Sysplex Distributor provides a solution for this single point-of-failure problem, as illustrated in Figure 9-22 on page 331. This is possible when an IMS connection factory is configured to access the IMS Connect task using a *Dynamic Virtual IP Address (DVIPA)* and sysplex distributor is used to select which IMS Connect task is assigned the request.

In the event of a failure of the logical partition (LPAR) or the IMS Connect task, Sysplex Distributor automatically selects an alternate LPAR and IMS Connect task. Another advantage of this configuration is that Sysplex Distributor uses Workload Manager performance data and its own algorithms to select which IMS Connect in the sysplex is best suited to service a request.

On the other hand, two-phase commit is not handled by this configuration in case of a communication failure. In the event of a communication failure during a two-phase commit transaction, sysplex distributor may route the recovery transaction to a different IMS Connect that causes problems, because the original IMS Connect knows about the transaction and not the newly accessed one.

For more details about architectural options, see *WebSphere for z/OS Connectivity Architectural Choices*, SG24-6365.

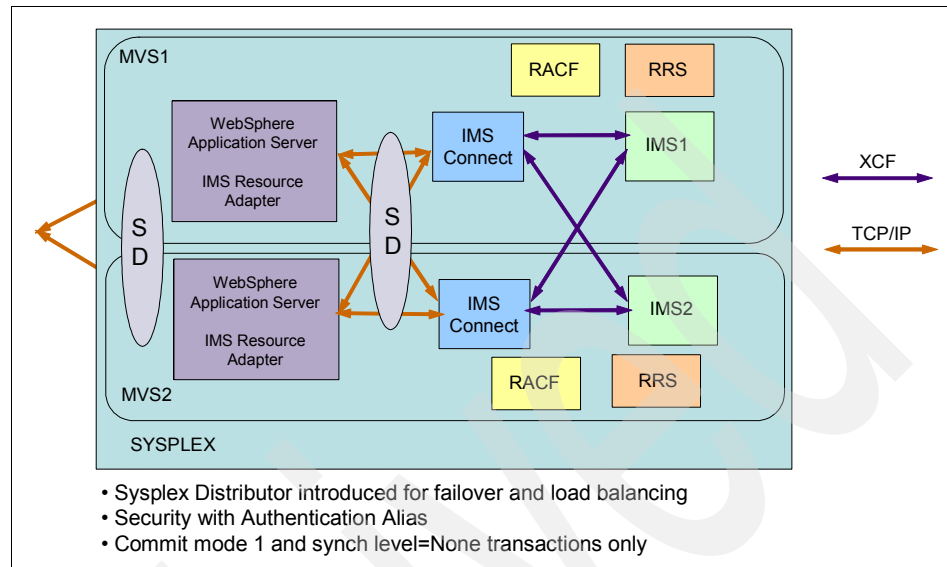


Figure 9-22 IMS Connect remote high availability option

9.11 Problem determination

To perform problem determination, we recommend that you enable the following WebSphere traces, as shown in Figure 9-23 on page 332:

- ▶ `com.ibm.ejs.j2c.*=all=enabled`
- ▶ `com.ibm.connector2.*=all=enabled`

You must set the trace level in the J2C connection factory customer property to 3.

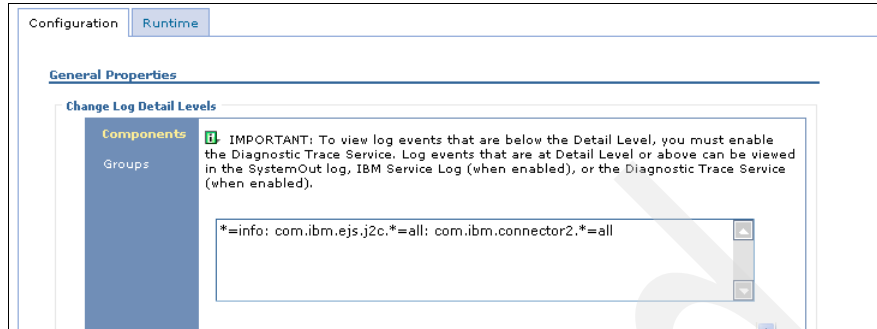


Figure 9-23 Diagnostic traces

9.11.1 Common errors

Common errors are:

- ▶ ICO0079E:com.ibm.connector2.ims.ico.IMSTCPIPManagedConnection@3b1bf125.getOutputData (InteractionSpec) error. IMS returned DFS™ message: DFS064 08:31:04 DESTINATION CAN NOT BE FOUND OR CREATED
 - Explanation: The first eight characters of the input are not recognized as a valid transaction, logical terminal name, or command.
 - Usual cause: The transaction name specified in the input request is not recognized by the target IMS system.
- ▶ ICO0001E:com.ibm.connector2.ims.ico.IMSTCPIPManagedConnection@d6fd946.processOutputOTMAMsg(byte [], InteractionSpec, Record) error. IMS Connect returned error: RETCODE=[8], REASONCODE=[SECFNPUI] Security failure; no password and no user ID.
 - Explanation: The application descriptor specifies res-auth Application but the application did not provide a user ID or password.
 - Solution: Consider changing res-auth to Container.
- ▶ ICO0003E:com.ibm.connector2.ims.ico.IMSTCPIPManagedConnection@5072da31.connect() error. Failed to connect to host [p390.raleigh.ibm.com], port [4000]. [java.net.ConnectException: Connection refused: connect]
 - Explanation: The IMS Connect task on the specified host is not accepting the connection request.
 - Solution: Verify that the IMS Connect task is active on the target host system and is listening on the specified port.
- ▶ ICO0064E:com.ibm.connector2.ims.ico.IMSLocalOptionManagedConnection@12d18e56.processSubject(javax.security.auth.Subject aSubject) error. Invalid security credential

- Explanation: IMS Connect was unable to validate the user ID and password passed on the request with the external security manager.
- Solution: An invalid user ID, password, or both was provided.

9.11.2 IMS Connect authorization and authentication support

IMS J2C connection factories support the assignment of the current thread identifier as the owner of a connection for authentication purposes when global security is enabled. Thread identity assignment is performed when the following conditions are met:

- ▶ Container-managed resource authority (res-auth=Container) is specified in the application deployment descriptors.
- ▶ The J2C connection factory uses a local connection (no TCP/IP) and no Container-managed authentication alias is specified.

The flow charts shown in Figure 9-24 on page 334 and Figure 9-25 on page 335 identify where the user ID and password used to access IMS are obtained when the deployment descriptor specifies Application (Figure 9-24 on page 334) or Container (Figure 9-25 on page 335).

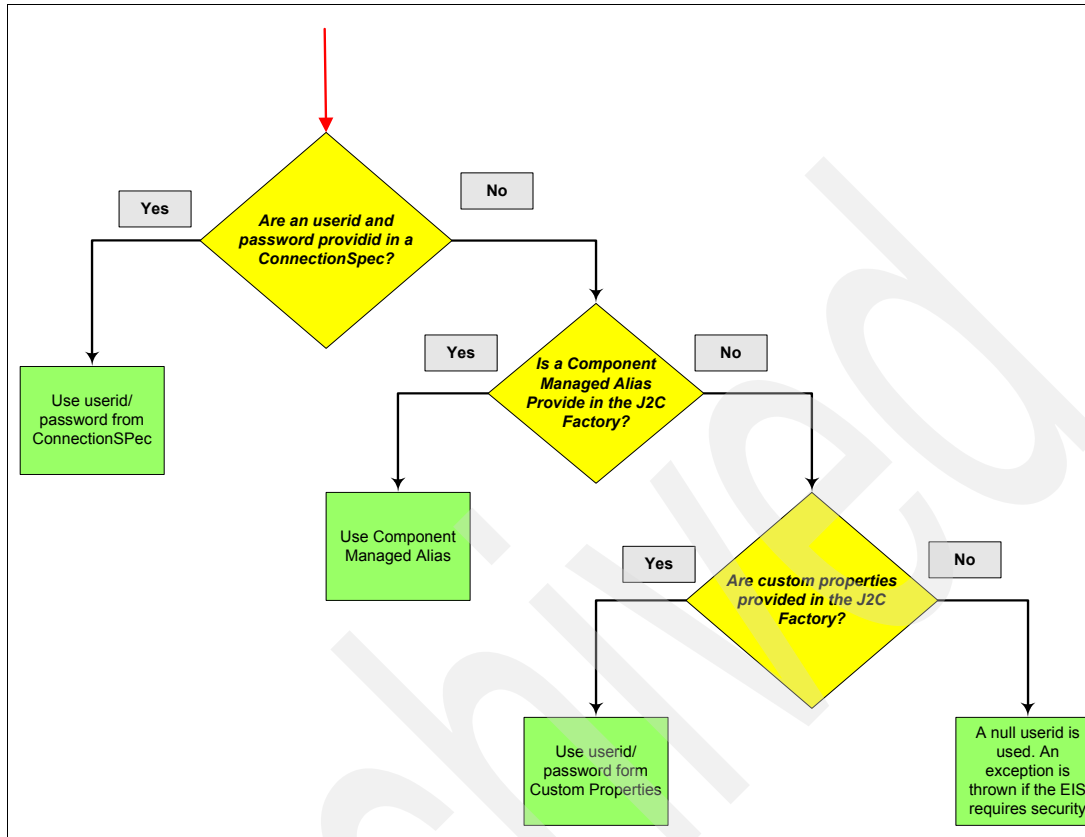


Figure 9-24 Application resource management (res-auth=Application)

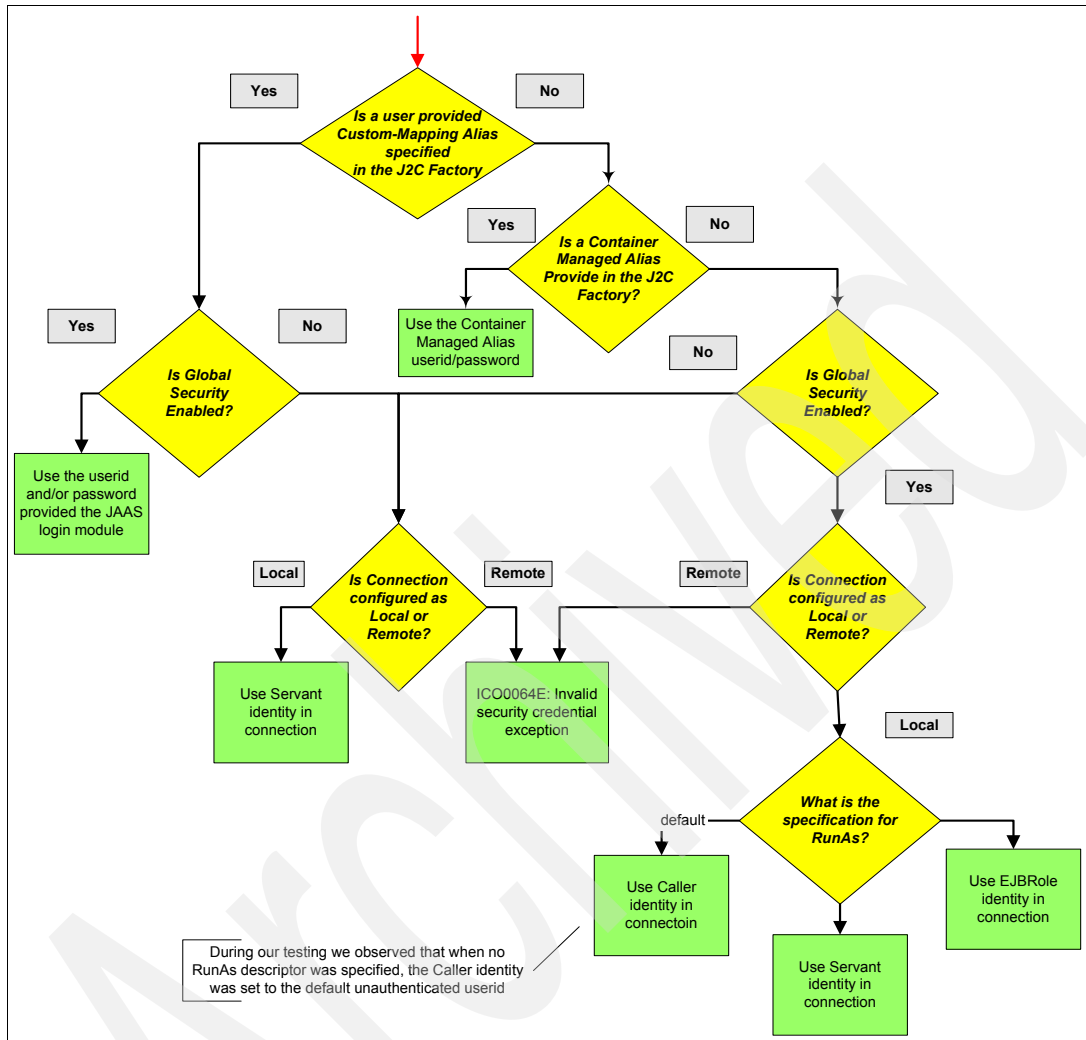


Figure 9-25 Container resource management (res-auth=Container)

9.12 Transaction management

If IMS is accessed via a local connector, WebSphere for z/OS works directly with Resource Recovery Services (RRS) and manages the two-phase commit (2PC) for IMS requests and resources.

If IMS is accessed via a remote connector (even on the same LPAR), IMS Connect manages the 2PC processing for IMS request and resources.

Note: During two-phase commit processing, RRS may communicate with other resource managers on the same LPAR (DB2, CICS, etc.) as necessary to preserve transactional integrity.

Introduction to Web services

This chapter focuses on some of the architectural concepts that need to be considered on a Web services project. We define and discuss *Service-Oriented Architecture (SOA)* and the relationship between SOAs and Web services.

We then take a closer look at *Web services*, a technology that enables you to invoke applications using Internet protocols and standards. The technology is called Web services because it integrates services (applications) using Web technologies (the Internet and its standards).

10.1 Introduction

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

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

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

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

10.2 Service-Oriented Architecture

This section is a short introduction to a Service-Oriented Architecture, its key concepts, and requirements. It should be observed that the presented architecture makes no statements about the infrastructure or protocols it uses. Therefore, the Service-Oriented Architecture can be implemented not only using Web technologies.

A Service-Oriented Architecture consists of three basic components:

- Service provider

- ▶ Service broker
- ▶ Service requestor

However, each component can also act as one of the two other components. For instance, if a service provider needs more information that it can only acquire from some other service, it acts as a service requestor while still serving the original request. Figure 10-1 shows the operations each component can perform.

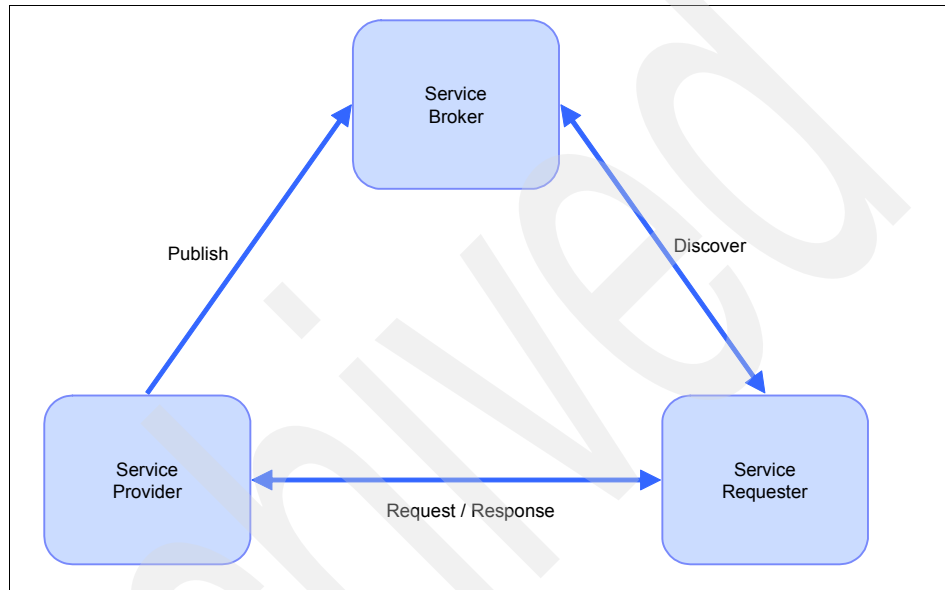


Figure 10-1 Web services components and operations

The components are:

- ▶ The *service provider* creates a Web service and possibly publishes its interface and access information to the service broker.
- ▶ 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* binds to the service provider in order to invoke one of its Web services, having optionally located entries in the broker registry using various find operations.

10.2.1 Characteristics

The presented Service-Oriented Architecture employs a loose coupling between the participants. Such a loose coupling provides greater flexibility:

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

10.2.2 Web services verses Service-Oriented Architectures

The Service-Oriented Architecture has been used under various guises for many years. It can and has been implemented using a number of different distributed computing technology, such as CORBA or messaging middleware. The effectiveness of service-oriented architectures in the past has always been limited by the ability of the underlying technology to interoperate across the enterprise.

Web services technology is an ideal technology choice for implementing a Service-Oriented Architecture:

- ▶ Web services are standards based. Interoperability is a key business advantage within the enterprise and is crucial in B2B scenarios.
- ▶ Web services are widely supported across the industry. For the very first time, all major vendors are recognizing and providing support for Web services.
- ▶ Web services are platform and language agnostic—there is no bias for or against a particular hardware or software platform. Web services can be implemented in any programming language or toolset. This is important because there will be continued industry support for the development of standards and interoperability between vendor implementations.
- ▶ This technology provides a migration path to gradually enable existing business functions as Web services are needed.
- ▶ This technology supports synchronous and asynchronous, RPC-based, and complex message-oriented exchange patterns.

Conversely, there are many Web services implementations that are not a Service-Oriented Architecture. For example, the use of Web services to connect two heterogeneous systems directly together is not a SOA. These uses of Web services solve real problems and provide significant value on their own. They may form the starting point of an SOA.

In general, an SOA has to be implemented at an enterprise or organizational level in order to harvest many of the benefits.

For more information about the relationship between Web services and Service-Oriented Architectures, or the application of IBM Patterns for e-business to a Web services project, refer to *Patterns: Service-Oriented Architecture and Web Services*, SG24-6303.

10.3 Web services

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

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

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

The W3C Services Architecture Working Group defines a Web service as follows:

A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically, WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, *typically conveyed using HTTP with an XML serialization* in conjunction with other Web-related standards.

It is important to note from this definition that a Web service is not constrained to use SOAP over HTTP/S as the transport mechanism. Web services are equally as at home in the messaging world.

10.3.1 Properties of a Web service

All Web services share the following properties:

- Web services are self-contained.

On the client side, no additional software is required. A programming language with XML and HTTP client support is enough to get you started. On the server side, only an HTTP server and a SOAP server are required.

- ▶ Web services are self-describing.

Using WSDL, all the information required to implement a Web service as a provider or invoke a Web service as a requester is provided.

- ▶ Web services can be published, located, and invoked across the Web.

This technology uses established lightweight Internet standards such as HTTP. It leverages the existing infrastructure.

- ▶ Web services are modular.

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

- ▶ Web services are language-independent and interoperable.

The client and server can be implemented in different environments. Theoretically, any language can be used to implement Web service clients and servers.

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

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

- ▶ Web services are loosely coupled.

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

- ▶ Web services provide programmatic access.

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

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

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

10.3.2 Core standards

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

SOAP: Simple Object Access Protocol

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

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

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

As of this writing, SOAP Version 1.2 is available though SOAP Version 1.1 and is still the recommended standard for interoperability. For more information, refer to:

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

WSDL: Web Services Description Language

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

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

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

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

XML

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

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

UDDI: Universal Description, Discovery, and Integration

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

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

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

10.3.3 Additional standards

Figure 10-2 attempts to provide a snapshot of the rapidly changing landscape of Web services-related standards and specifications. It is not intended to be a strictly correct stack diagram—it just attempts to show the various standards efforts in terms of the general category to which they belong.

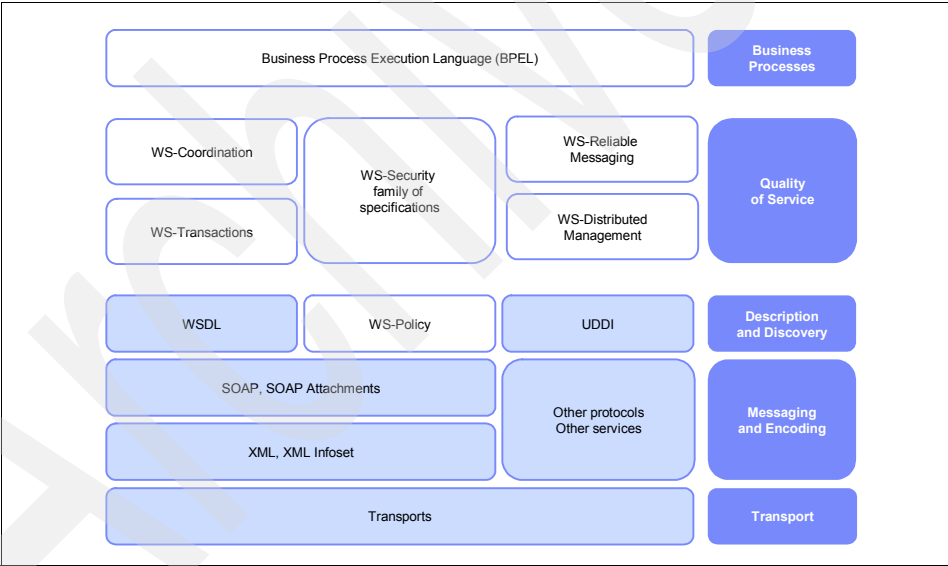


Figure 10-2 Web services standards

Given the current momentum behind Web services and the pace at which standards are evolving, it is also useful to refer to an online compilation of Web services standards. An online compilation is available on the IBM developerWorks® Web site at:

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

Of particular interest to those developing Web services in CICS are:

- ▶ WS-AtomicTransaction
- ▶ WS-Security

WS-AtomicTransaction (WS-AT)

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

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

WS-Security

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

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

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

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

10.4 Simple Open Access Protocol (SOAP)

In this section we introduce SOAP, the specification covering the exchange of XML-based messages between the three main actors in the Service-Oriented Architecture (SOA).

10.4.1 Overview

SOAP is a specification for the exchange of structured information in a decentralized, distributed environment. As such, it represents the main way of communication between the three main actors in SOA: The service provider, service requestor, and service broker. The main goal of its design is to be simple and extensible. Originally proposed by Microsoft®, it is now submitted to W3C as the basis of the XML Protocol Working Group by several companies, including

IBM. At the time of writing of this book, the current standard is 1.2. You can get more details at:

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

Because SOAP 1.2 is not included in WS-I interoperability Basic Profile 1.0 (WS-I BP 1.0), most Web service runtimes still support and recommend using SOAP 1.1. CICS and WebSphere, for example, have support for both SOAP 1.1 and SOAP 1.2. In this section we mainly discuss SOAP 1.1.

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

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

The way SOAP applications communicate when exchanging messages is often referred to as the Message Exchange Pattern (MEP). The communication can be either one-way messaging, where the SOAP message only goes in one direction, or two-way messaging, where the receiver is expected to send back a reply.

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

10.4.2 The three pillars of SOAP

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

Overall message format: Envelope with header and body

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

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

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



Figure 10-3 Example of a simplified SOAP message

Figure 10-3 shows a simplified SOAP request message based on our catalog application:

- ▶ The header tells *who* and *how* to deal with the message. The actor *next* (or omitted actor) is the default actor and declares the receiver as the server who has to do what the body says. Furthermore, the server must understand and process the application-defined <TranID> tag.
- ▶ The body tells *what* has to be done: Dispatch an order for quantityRequired 1 of itemRefNumber 0010 to customerID CB1 in chargeDepartment ITSO.

10.4.3 SOAP elements

In this section we discuss SOAP elements.

Namespaces

The use of *namespaces* plays an important role in SOAP message. A namespace is simply a way of adding a unique qualifier to an element name to ensure it is unique.

For example, we may have a message that contains an element <customer>. Customers are fairly common, so it is very likely that many Web services will have a customer element. To ensure we know what customer we are talking about we declare a namespace for it, for example, as follows:

```
xmlns:itso="http://itso.ibm.com/CICS/catalogApplication"
```

This identifies the prefix *itso* with the declared namespace. Then whenever we reference the element <customer> we prefix it with the namespace as follows: <itso:customer>. This identifies it uniquely as a customer type for our application. Namespaces can be defined as any unique string. They are often defined as URLs, as URLs are generally globally unique. These URLs do not have to physically exist though.

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

Table 10-1 shows the namespaces of SOAP and WS-I Basic Profile 1.0 used in this book.

Table 10-1 SOAP namespaces

| Namespace URI | Explanation |
|---|--------------------------------------|
| http://schemas.xmlsoap.org/soap/envelope/ | SOAP 1.1 envelope namespace |
| http://schemas.xmlsoap.org/soap/encoding/ | SOAP 1.1 encoding namespace |
| http://www.w3.org/2001/XMLSchema-instance | Schema instance namespace |
| http://www.w3.org/2001/XMLSchema | XML Schema namespace |
| http://schemas.xmlsoap.org/wsdl | WSDL namespace for WSDL framework |
| http://schemas.xmlsoap.org/wsdl/soap | WSDL namespace for WSDL SOAP binding |
| http://ws-i.org/schemas/conformanceClaim/ | WS-I Basic Profile |

SOAP envelope

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

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

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

Headers

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

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

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

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

```
<Header>
  <thens:TranID mustUnderstand="1">ABCD</thens:TranID>
</Header>
```

In the example, the header element specifies that a service invocation must fail if the service provider does not support the ability to run the TranID header.

A SOAP intermediary is an application that is capable of both receiving and forwarding SOAP messages on their way to the final destination. In realistic situations, not all parts of a SOAP message may be intended for the ultimate destination of the SOAP message, but, instead, may be intended for one or more of the intermediaries on the message path. Therefore, a second predefined header attribute, SOAP-ENV:actor, is used to identify the recipient of the header information. In SOAP 1.2 the actor attribute is renamed SOAP-ENV:role. The value of the SOAP actor attribute is the URI of the mediator, which is also the final destination of the particular header element (the mediator does not forward the header).

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

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

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

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

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

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

Body

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

All immediate child elements of the *body* element are called *body entries*, and each *body entry* is encoded as an independent element within the SOAP *body* element. In the most simple case, the body of a basic SOAP message consists of an XML message as defined by the schema in the types section of the WSDL document. It is legal to have any valid XML as the body of the SOAP message, but WS-I conformance requires that the elements be namespace qualified.

Error handling

One area where there are significant differences between the SOAP 1.1 and 1.2 specifications is in the handling of errors. Here we will focus on the SOAP 1.1 specification for error handling.

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

- ▶ *faultcode* is a code that indicates the type of the fault. SOAP defines a small set of SOAP fault codes covering basic SOAP faults:
 - *soapenv:Client*, indicating incorrectly formatted messages
 - *soapenv:Server*, for delivery problems
 - *soapenv:VersionMismatch*, which can report any invalid namespaces for envelope element
 - *soapenv:MustUnderstand*, for errors regarding the processing of header content
- ▶ *faultstring* is a human-readable description of the fault. It must be present in a fault element.
- ▶ *faultactor* is an optional field that indicates the URI of the source of the fault. It is similar to the SOAP actor attribute, but instead of indicating the destination of the header entry, it indicates the source of the fault. The value of the *faultactor* attribute is a URI identifying the source that caused the error. Applications that do not act as the ultimate destination of the SOAP message must include the *faultactor* element in a SOAP fault element.
- ▶ *detail* is an application-specific field that contains detailed information about the fault. It must not be used to carry information about errors belonging to header entries. Therefore, the absence of the detail element in the fault element indicates that the fault is not related to the processing of the body element (the actual message).

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

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

Communication styles

SOAP supports two different communication styles:

Document Also known as *message-oriented* style, this is a very flexible communication style that provides the best interoperability. The message body is any legal XML as defined in the types section of the WSDL document.

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

Encodings

In distributed computing environments, *encodings* define how data values defined in the application can be translated to and from a protocol format. We refer to these translation steps as *serialization* and *deserialization*, or, synonymously, *marshalling* and *unmarshalling*.

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

The following encodings are defined:

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

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

Messaging modes

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

- ▶ Document/literal—Provides the best interoperability between language environments. The WS-I Basic Profile states that all Web service interactions should use the Document/literal mode.
- ▶ RPC/literal—Possible choice between certain implementations. Although RPC/literal is WS-I compliant, it is not frequently used in practice. There are a number of usability issues associated with RPC/literal.
- ▶ RPC/encoded—Early Java implementations supported this combination, but it does not provide interoperability with other implementations and is not recommended.
- ▶ Document/encoded—Not used in practice.

10.5 Web Services Definition Language (WSDL)

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

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

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

10.5.1 WSDL document

The WSDL *document* contains the following main elements:

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

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

WSDL 1.1 introduces specific binding extensions for various protocols and message formats. There is a WSDL SOAP binding, which is capable of describing SOAP over HTTP. It is worth noting that WSDL does not define any mapping-to-programming languages; rather, the bindings deal with transport protocols. This is a major difference from interface description languages, such as CORBA Interface definition language (IDL), which has language bindings.

10.5.2 WSDL document anatomy

Figure 10-4 on page 355 shows the elements comprising a WSDL document and the various relationships between them.

The diagram should be read in the following way:

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

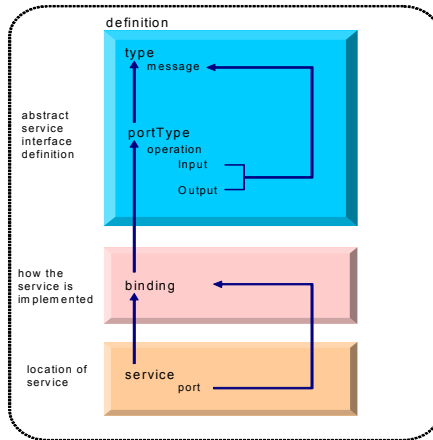


Figure 10-4 WSDL elements and relationships

Example

Let us now give an example of a simple, complete, and valid WSDL file. As we will see, even a simple WSDL document contains quite a few elements with various relationships to each other. Figure 10-1 contains the WSDL file example. This example is analyzed in detail later in this section.

Example 10-1 Complete WSDL document

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:reqns="http://www.exampleApp.dispatchOrder.Request.com"
  xmlns:resns="http://www.exampleApp.dispatchOrder.Response.com"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://www.exampleApp.dispatchOrder.com"
  targetNamespace="http://www.exampleApp.dispatchOrder.com">
  <types>
    <xsd:schema xmlns:tns="http://www.exampleApp.dispatchOrder.Request.com"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      attributeFormDefault="qualified"
      elementFormDefault="qualified"
      targetNamespace="http://www.exampleApp.dispatchOrder.Request.com"
      xmlns:reqns="http://www.exampleApp.dispatchOrder.Request.com">
      <xsd:element name="dispatchOrderRequest" nillable="false">
        <xsd:complexType mixed="false">
          <xsd:sequence>
            <xsd:element name="itemReferenceNumber" nillable="false">
              <xsd:simpleType>
                <xsd:restriction base="xsd:short">
                  <xsd:maxInclusive value="9999"/>
                  <xsd:minInclusive value="0"/>
                </xsd:restriction>
              </xsd:simpleType>
            </xsd:element>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:schema>
  </types>
  <binding name="dispatchOrderRequestBinding" type="tns:dispatchOrderRequest">
    <operation name="dispatchOrderRequest" input="request" output="response">
      <input message="tns:dispatchOrderRequest" use="literal"/>
      <output message="tns:dispatchOrderResponse" use="literal"/>
    </operation>
  </binding>
  <service name="dispatchOrderRequestService">
    <port name="dispatchOrderRequestPort" binding="dispatchOrderRequestBinding" location="http://www.exampleApp.dispatchOrder.com/dispatchOrderRequestPort">
    </port>
  </service>
</definitions>
```

```

        </xsd:restriction>
      </xsd:simpleType>
    </xsd:element>
    <xsd:element name="quantityRequired" nillable="false">
      <xsd:simpleType>
        <xsd:restriction base="xsd:short">
          <xsd:maxInclusive value="999"/>
          <xsd:minInclusive value="0"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>
</xsd:element>
</xsd:schema>
<xsd:schema xmlns:tns="http://www.exampleApp.dispatchOrder.Response.com"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  attributeFormDefault="qualified"
  elementFormDefault="qualified"
  targetNamespace="http://www.exampleApp.dispatchOrder.Response.com">
  <xsd:element name="dispatchOrderResponse" nillable="false">
    <xsd:complexType mixed="false">
      <xsd:sequence>
        <xsd:element name="confirmation" nillable="false">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:maxLength value="20"/>
              <xsd:whiteSpace value="preserve"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
</types>
<message name="dispatchOrderResponse">
  <part element="resns:dispatchOrderResponse" name="ResponsePart"/>
</message>
<message name="dispatchOrderRequest">
  <part element="reqns:dispatchOrderRequest" name="RequestPart"/>
</message>
<portType name="dispatchOrderPort">
  <operation name="dispatchOrder">
    <input message="tns:dispatchOrderRequest" name="DFH0XODSRequest"/>
    <output message="tns:dispatchOrderResponse" name="DFH0XODSResponse"/>
  </operation>
</portType>
<binding name="dispatchOrderSoapBinding" type="tns:dispatchOrderPort">

```

```

<soap:binding style="document"
    transport="http://schemas.xmlsoap.org/soap/http"/>
<operation name="dispatchOrder">
    <soap:operation soapAction="" style="document"/>
    <input name="DFHOXODSRequest">
        <soap:body parts="RequestPart" use="literal"/>
    </input>
    <output name="DFHOXODSResponse">
        <soap:body parts="ResponsePart" use="literal"/>
    </output>
</operation>
</binding>
<binding name="dispatchOrderSoapBindingMQ" type="tns:dispatchOrderPort">
    <soap:binding style="document"
        transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="dispatchOrder">
        <soap:operation soapAction="" style="document"/>
        <input name="DFHOXODSRequest">
            <soap:body parts="RequestPart" use="literal"/>
        </input>
        <output name="DFHOXODSResponse">
            <soap:body parts="ResponsePart" use="literal"/>
        </output>
    </operation>
</binding>
<service name="dispatchOrderService">
    <port binding="tns:dispatchOrderSoapBinding" name="dispatchOrderPort">
        <soap:address
            location="http://myserver:54321/exampleApp/services/dispatchOrderPort"/>
    </port>
</service>
</definitions>

```

Namespaces

WSDL uses the XML namespaces listed in Table 10-2.

Table 10-2 WSDL namespaces

| Namespace URI | Explanation |
|---------------------------------------|-------------------------------------|
| http://schemas.xmlsoap.org/wsdl/ | Namespace for WSDL framework. |
| http://schemas.xmlsoap.org/wsdl/soap/ | SOAP binding. |
| http://schemas.xmlsoap.org/wsdl/http/ | HTTP binding. |
| http://www.w3.org/2000/10/XMLSchema | Schema namespace as defined by XSD. |

| Namespace URI | Explanation |
|--------------------|---|
| (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 three namespaces are defined by the WSDL specification itself; the next definition references namespaces that are defined in the SOAP and XSD standards. The last one is local to each specification.

10.5.3 WSDL definition

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

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

Types

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

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

The XSD type system can be used to define the types in a message regardless of whether the resulting wire format is XML. In our example we have two schema sections, one to define the message format for the input and one for the output.

In our example, the type definition, shown in Figure 10-2, is where we specify that there is a complex type called *dispatchOrderRequest*, which is composed of two elements, a *itemReferenceNumber* and a *quantityRequired*.

Example 10-2 Types definition of our WSDL example for the input

```
<types>
  <xsd:schema xmlns:tns="http://www.exampleApp.dispatchOrder.Request.com"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    attributeFormDefault="qualified"
    elementFormDefault="qualified"
    targetNamespace="http://www.exampleApp.dispatchOrder.Request.com"
```

```

        xmlns:reqns="http://www.exampleApp.dispatchOrder.Request.com">
<xsd:element name="dispatchOrderRequest" nillable="false">
  <xsd:complexType mixed="false">
    <xsd:sequence>
      <xsd:element name="itemReferenceNumber" nillable="false">
        <xsd:simpleType>
          <xsd:restriction base="xsd:short">
            <xsd:maxInclusive value="9999"/>
            <xsd:minInclusive value="0"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:element>
      <xsd:element name="quantityRequired" nillable="false">
        <xsd:simpleType>
          <xsd:restriction base="xsd:short">
            <xsd:maxInclusive value="999"/>
            <xsd:minInclusive value="0"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
</xsd:schema>
.
.
</types>

```

Messages

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

```

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

```

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

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

In our example, the messages definition, shown in Figure 10-3, is where we specify the different parts that compose each message. The request message `dispatchOrderRequest` is composed of an element `dispatchOrderRequest` as defined in the schema in the parts section. The response message `dispatchOrderResponse` is similarly defined by the element `dispatchOrderResponse` in the schema. There is no requirement for the names of the message and the schema defined element to match; this was merely done for convenience in this example.

Example 10-3 Message definition in our WSDL document

```
<message name="dispatchOrderResponse">
  <part element="resns:dispatchOrderResponse" name="ResponsePart"/>
</message>
<message name="dispatchOrderRequest">
  <part element="reqns:dispatchOrderRequest" name="RequestPart"/>
</message>
```

Web services using SOAP over HTTP

The chapter uses a *Hello world* type of application called the SEWS application to show how to Web service-enable a Java application in WebSphere, an application in CICS, and how to call these Web services from CICS and WebSphere.

The focus is on the practical side of the Web service process, not the theory.

This chapter focuses on the use of SOAP over HTTP as the transport for the Web services. Chapter 12, “Web services and SOAP over JMS” on page 407, focuses on the use of SOAP over JMS as the Web service transport.

11.1 An end-to-end Web service example

The prospect of starting out on the path to using Web services may seem a little daunting at first. There are many new terms and concepts to grasp. Additionally, there are many sources of information about Web services available. There are also different areas of responsibility for different people when it comes to Web services. The application developers need to understand the Web services concepts and know how to use their development tools to build Web services. System programmers who look after WebSphere, CICS, IMS, and so on, need to know what they need to do in terms of configuration. They also need some understanding of how Web services are working within the product they look after, as sooner or later they will need to help resolve some problem to do with Web services.

But in essence, once you understand the “plumbing” of Web services, the rest is just how much application logic that gets added to the “plumbing” to do something useful.

This section uses a simple *Hello world* type example to demonstrate the basic plumbing of Web services. It does not cover topics such as security around Web services, for example.

The focus in this example is on WebSphere Application Server for z/OS Version 6.01 and CICS Transaction Server Version 3.1. The aim is to show how to have these products acting as both Web service providers and requestors. Also, the aim is to provide you with a simple application that you can install on your own system to allow you to try out some basic Web service connectivity.

In “Web service enabling SEWS for WebSphere” on page 369 we show an example of bottom-up development to Web service enable an existing Java application running in WebSphere.

In “Development of Web service invoker” on page 383 we show an example of how to develop a new application invoking a Web service that is running in both WebSphere itself and CICS. This invoker application will run in WebSphere.

In “CICS - Web service-enable business logic” on page 385 we show how we take a COBOL program and enable it as a Web service that is making it capable of processing Web service calls from within CICS itself or from outside CICS, specifically WebSphere on z/OS.

In “CICS - Performing a Web service request” on page 391 we show how to perform Web service calls out of a COBOL program in CICS.

“Web service definitions required in CICS” on page 396 describes the definitions required in CICS to support Web services.

Finally, in “Testing the SEWS application” on page 402 we test that we can actually perform Web service calls from WebSphere to WebSphere, WebSphere to CICS, CICS to WebSphere, and CICS to CICS.

11.2 Simple Example Web Services (SEWS) application

To show the fundamentals of Web services we first developed a simple application called the Simple Example Web Services.

SEWS consists of a part that runs in WebSphere and a part that runs in CICS. We use this concept to show how to make the simple business logic in each server available as a Web service and how each application can then be called as a Web service.

In the following sections we provide an overview of the components of SEWS and what the general idea of the application is.

11.2.1 WebSphere part of SEWS

On the WebSphere side, SEWS consists of the parts listed in Table 11-1.

Table 11-1 SEWS components

| Component type | Name | Purpose |
|----------------|-----------|--|
| Java Bean | SnoopInfo | Java Bean that returns some basic information. |
| Servlet | Snoop | Servlet that calls a method on the SnoopInfo Java Bean, and calls the SnoopEJB |
| EJB | SnoopEJB | EJB that calls a method on the SnoopInfo Java Bean |

The servlet, EJB, and Java Bean return the information shown in Table 11-2.

Table 11-2 Information returned by the SEWS application

| Field | Servlet | EJB | Java Bean |
|----------------|---------|-----|-----------|
| Java Principal | Y | Y | Y |
| Host | Y | Y | Y |
| Time | Y | Y | Y |

| Field | Servlet | EJB | Java Bean |
|----------------|---------|-----|-----------|
| Caller User ID | N | Y | N? |

Java principal

The Java principal identifies what user ID the process is running under in WebSphere. If the WebSphere server does not have Global Security enabled, then this value is non existent.

Host

The host value identifies the name of the host machine that the WebSphere server that ran your request is running on.

Time

Time is the time your request was run. Since time never stands still, the value returned in this field should always be different from the last time it was called, which is a handy way to check that the call is really working, since you should be getting a different value here each time.

Caller User ID

If calling an EJB, then this should identify the user ID of the process that called the EJB.

To run the servlet enter the following URL, updated to reflect the host TCP/IP address and port number of your server:

`http://9.12.4.38:9080/sewsWeb/Snoop`

The output will be similar to that shown in Figure 11-1 on page 365.

| ITSO Simple Example Web Services - Microsoft Internet Explorer | |
|--|--|
| File Edit View Favorites Tools Help | |
| address http://9.12.4.38:9080/sewsWeb/Snoop | |
| Servlet Snoop Information | |
| Java Principal | Subject: Principal: wtsc48.itso.ibm.com/WSGUEST Public Credential: com.ibm.ws.security.auth.WSCredentialImpl@2fcfd853 [com.ibm.ws.security.auth.zOSWSCredentialImpl@2ff75853; [com.ibm.ws.security.zos.PlatformCredential]] |
| Host | wtsc48.itso.ibm.com |
| Current Time | Fri Jun 17 12:31:46 GMT+00:00 2005 |
| Call to SnoopEjb Information | |
| Java Principal | Subject: Principal: wtsc48.itso.ibm.com/WSGUEST Public Credential: com.ibm.ws.security.auth.WSCredentialImpl@2fcfd853 [com.ibm.ws.security.auth.zOSWSCredentialImpl@2ff75853; [com.ibm.ws.security.zos.PlatformCredential]] |
| Host | wtsc48.itso.ibm.com |
| Current Time | Fri Jun 17 12:31:46 GMT+00:00 2005 |
| Principal of caller | WSGUEST |

Figure 11-1 Output from running SEWS servlet

11.2.2 CICS part of SEWS

The CICS part of SEWS consists of the programs shown in Table 11-3 on page 366.

Table 11-3 COBOL programs

| Program | Purpose |
|----------|----------------------|
| SEWSBMS | BMS map |
| SEWS0001 | Presentation Layer |
| SEWS0002 | Business logic layer |

The source code of the programs is supplied as part of the additional material of this book.

Define a transaction called SEWS that invokes program SEWS0001 in the CICS region. When you run SEWS, a screen is displayed. Select option 1, which tells the SEWS0001 program to LINK to the SEWS0002 program, and you will get output similar to that shown in Example 11-1.

Example 11-1 Information returned after running option 1 of SEWS

Simple Example Web Services

1. LINK to SEWS0002
2. Call the SnoopInfo Web Service
Enter Host Name or IP Address where SnoopInfo Web Service is located
1.2.3.4:9080

Option:

Returned Data:

EDMCAR EDWARD MCCARTHY/VILA

ERW1 SC38TC99 SEWS

Mon, 20 Jun 2005 16:57:01 GMT

The LINK to SEWS0002 returns similar information to that returned by the SnoopInfo Web service described in “WebSphere part of SEWS” on page 363. The first line under the heading Returned Data shows the user ID the transaction ran under, the second line shows a conglomerate of information about the CICS region, and the third line shows the time.

It is this call to SEWS0002 that we will make available as a Web service, as described in “CICS - Web service-enable business logic” on page 385.

Containers - A better commarea

Later in this chapter, when setting up CICS to be a provider of Web services, we use a new feature of CICS called *containers*. We do not, however, explain this concept in detail in this book.

Containers can be thought of as a new bigger and better commarea, that is, a way to pass data between programs in CICS.

More information about this new feature can be found at:

http://publib.boulder.ibm.com/infocenter/cicsts31/index.jsp?topic=/com.ibm.cics.ts.doc/lpaths/channels_lp_overview.htm

11.2.3 Our system layout

Throughout this chapter we show the various definitions and settings we used on our system to demonstrate the use of Web services via SOAP over HTTP. The diagram in Figure 11-2 shows the key components in our environment.

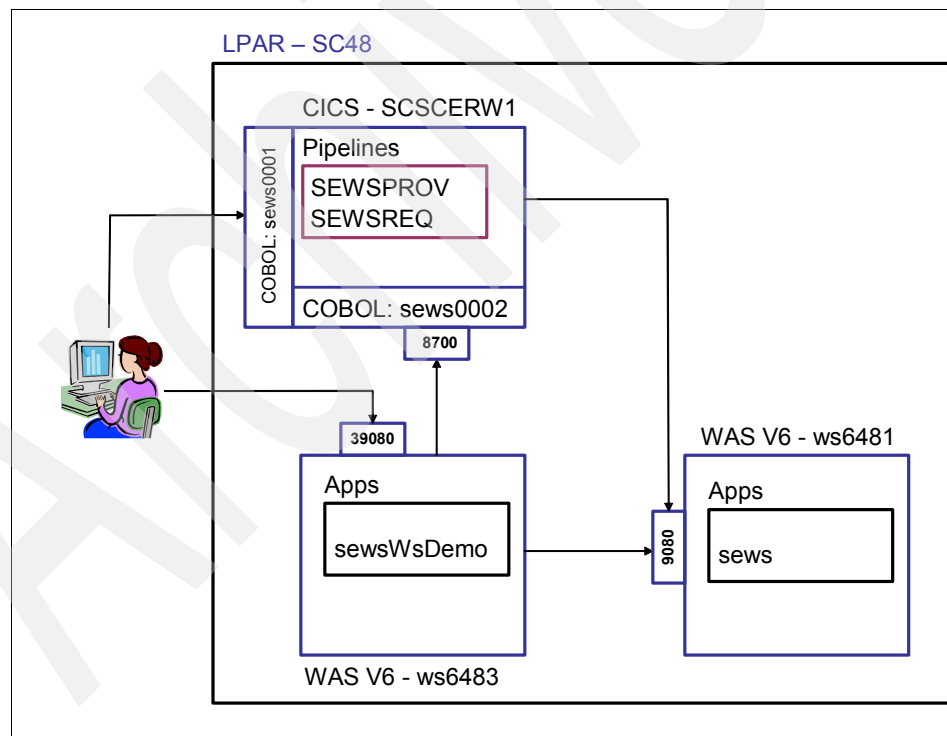


Figure 11-2 Environment used for Web services using SOAP over HTTP

WebSphere Server - ws6481

The ws6481 server is a WebSphere Application Server for z/OS Version 6.01 standalone server. It is listening on port 9080 for standard HTTP requests. It will have deployed into it the *sews* application, providing a Web service.

WebSphere Server - ws6483

The ws6483 server is a WebSphere Application Server for z/OS Version 6.01 standalone server. It is listening on port 39080 for standard HTTP requests. It will have deployed into it the *sewsWsDemo* application. This server did not have global security enabled.

CICS Region - SCSCERW1

This is a CICS Transaction Server Version 3.1 region.

11.2.4 Supplied material

Table 11-4 describes the additional material supplied with this book.

Table 11-4 *Supplied additional material*

| Name | Purpose |
|------------------------|---|
| sews-initial.zip | This is a RAD project interchange zip file. It contains the SEWS application before the process to enable it for Web services has begun. You can use this file if you want to re-create for yourself the steps described in “Web service-enable SnoopInfo Java Bean” on page 370. |
| sews-stage1.zip | This is a RAD project interchange zip file. It contains the sews project after the steps described in “Web service-enable SnoopInfo Java Bean” on page 370 have been completed. |
| sewsWsDemo-initial.zip | This is a RAD project interchange zip file. It contains the application that runs in WebSphere, which is used to invoke Web services. It is the starting point for the process described in “Development of Web service invoker” on page 383. |
| sewsWsDemo-stage1.zip | This is a RAD project interchange zip file. It contains the sewsWsDemo project after the steps described in “Development of Web service invoker” on page 383 have been completed. |
| sews-initial.ear | Initial version of the sews application, before any parts have been Web service-enabled. |

| Name | Purpose |
|-----------------------|---|
| sews-stage1.ear | Fully Web service-enabled version of the SEWS application that you can deploy into WebSphere to try out Web services. |
| sewsWsDemo-stage1.ear | Ear file that you can deploy into WebSphere to start calling the Web services in WebSphere or in CICS. |
| SEWS001 | COBOL source code for the program. |
| SEWS002 | COBOL source code for the program. |
| SEWSBMS | COBOL source code for the program. |

11.3 Web service enabling SEWS for WebSphere

We first provide an overview of what we want to demonstrate in this section and then describe the process.

11.3.1 Overview

This section is an example of bottom-up Web service development, where you are Web service-enabling an existing application.

The starting point is what is in sews-initial.zip, part of the additional material for this book. Start Rational Application Developer Version 6, set a new workspace when asked, and after startup has completed, import this file, selecting **Project Interchange** as the type of input source.

Make sure that you select all projects that are part of the zip file.

At this stage the application consists of:

| | |
|----------------------------|--|
| Snoop Servlet | Servlet to get information, also calls the EJB |
| SnoopEjb EJB | EJB to get information |
| SnoopInfo Java Bean | Java Bean that actually gets the information |

Attention: Note that when we created this project in Rational Application Developer Version 6, we first followed the steps described in Chapter 13, under the heading “Web services configuration settings,” of the *WebSphere Version 6 Web Services Handbook Development and Deployment*, SG24-6461. The steps there must be followed if you want to be able to do any work with Web services in your project in Rational Application Developer Version 6.

The SnoopInfo Java Bean is the piece of code that actually gathers the basic information described in 11.2, “Simple Example Web Services (SEWS) application” on page 363.

The Snoop servlet calls a method on this Java Bean to get information as seen by the servlet. The Snoop servlet then calls the SnoopEjb EJB, which also gets this information as seen by the EJB running in WebSphere.

In “Web service-enable SnoopInfo Java Bean” on page 370 we show how to Web service-enable the SnoopInfo Java Bean.

We then show that we can then do a Web service call on these newly created Web service from within Rational Application Developer Version 6 without having an invoker application yet in “Test SEWS in Rational Application Developer Version 6” on page 380.

We then show how we develop a new Java application invoking the function we have Web service-enabled in the SEWS application in “Development of Web service invoker” on page 383.

Finally, in “Testing the SEWS application” on page 402, we show how to run the SEWS application to verify the Web services are working correctly.

11.3.2 Web service-enable SnoopInfo Java Bean

Before we use the wizard to Web service-enable the SnoopInfo Java Bean we first need to set some preferences in the project in Rational Application Developer Version 6.

Set preferences

Click **Windows®** → **Preferences**, and select **Do not overwrite loadable Java Class**, as shown in Figure 11-3 on page 371. If you do not do this, then the Web service generation process will create interface classes of the same name in the target project, which could cause problems with class loading when you actually try to run the application. These additional classes will also cause compile errors in the Snoop servlet, as there will now be an interface class called SnoopInfo in the Web Project.

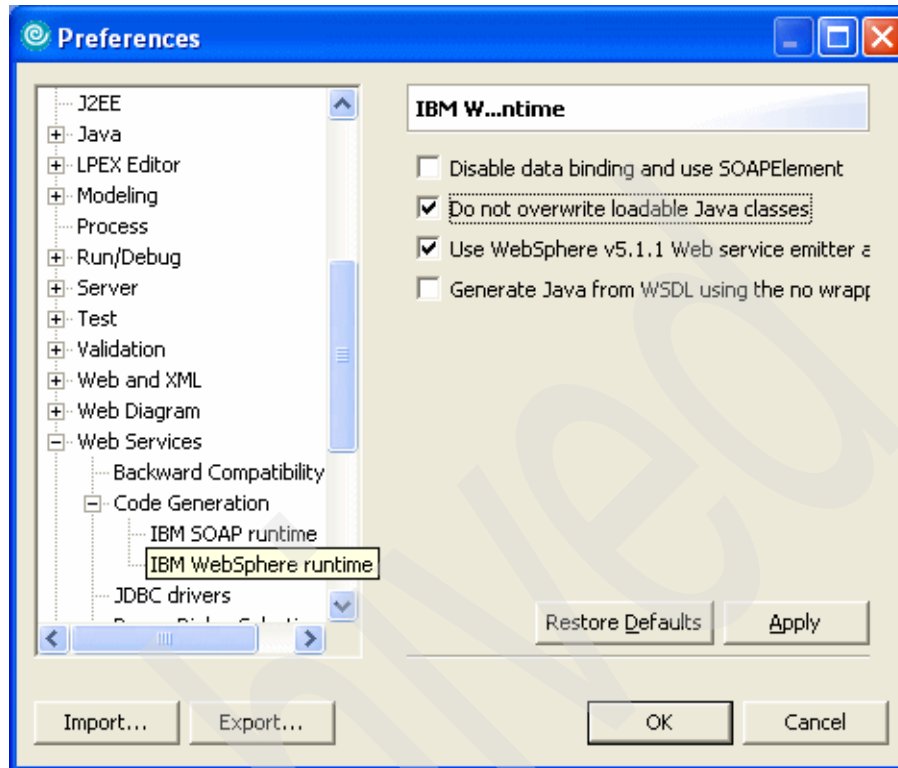


Figure 11-3 Setting preferences in the SEWS project

New or existing Web project

When generating the Web service, you have the opportunity to specify into what Web project the generated code should be placed. If you chose to save the code to a new Web project, the window you do this on has a message saying that Rational Application Developer Version 6 will do this for you. However, this will not work and you will get the error message shown Figure 11-4 on page 372.

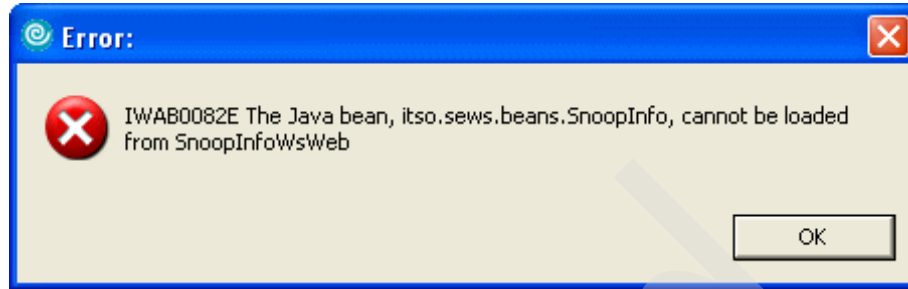


Figure 11-4 Error when trying to use new Web project

If you do indeed want to use a new Web Project for the generated code, then you need to create it first before starting the Web service generation process. Once you have created it, you then need to add the `sewsBean` project to the Java Build Path of the new Web project you have created. You will then be able to select this new Web project as the place to save the generated code during the Web service generation project.

For our purposes, we use the existing Web service project.

Create Web service

Expand **Other Projects** → **sewsBeans** → **itso.sews.beans**, then right-click **SnoopInfo.java**, then select **Web Services** → **Create Web Service**, as shown in Figure 11-5 on page 373.

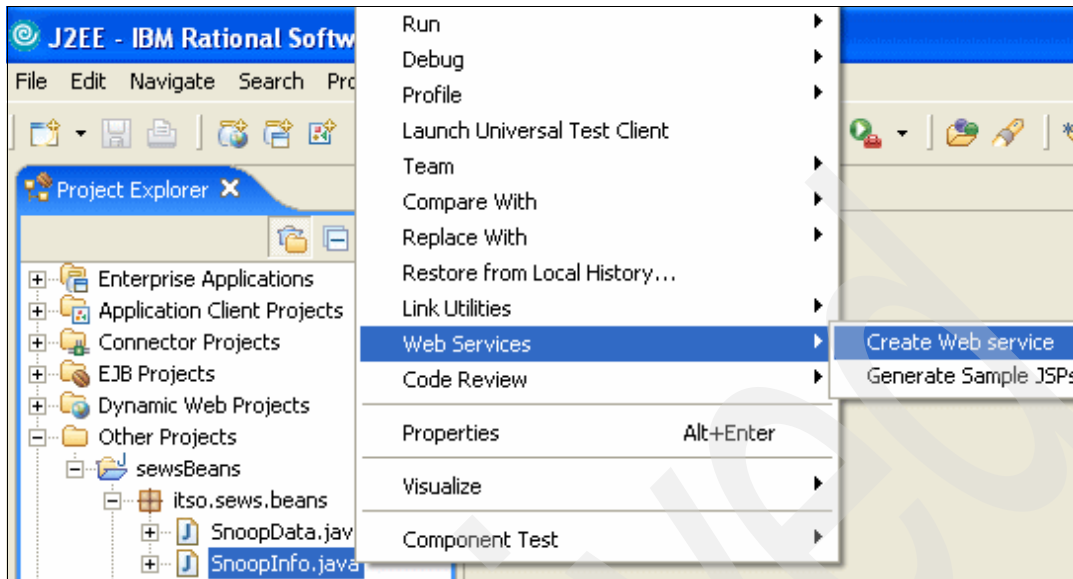


Figure 11-5 Starting process to Web service-enable SnoopInfo Java Bean

A new window is then displayed on which you set options regarding the generation of the Web service you want to create. As we are creating a Web service for a Java Bean, leave the Web service type as Java Bean Web service. Select the option to **Generate a Proxy**, as this will generate code that will enable you to test the generated Web service. Set the other options as shown in Figure 11-6 on page 374.

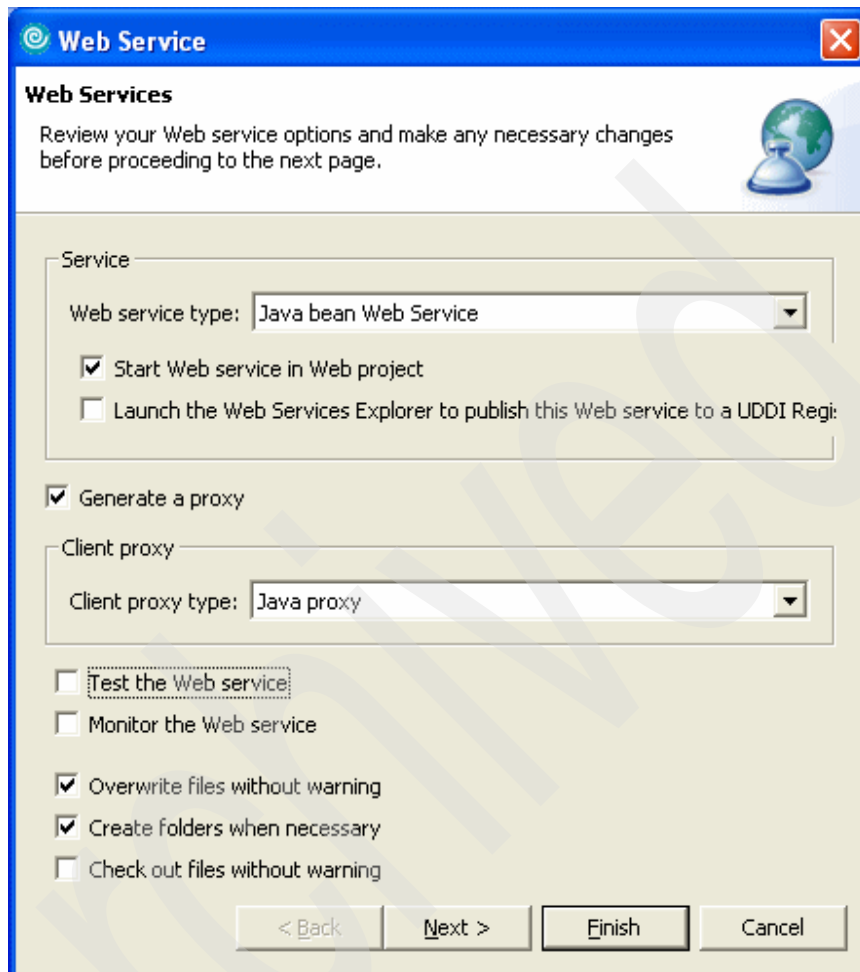


Figure 11-6 Setting options for generation of Web service

Click **Next**. The next window displayed is where you select the Java Bean class to be Web service enabled. This will already be set to the correct value, as shown in Figure 11-7 on page 375.

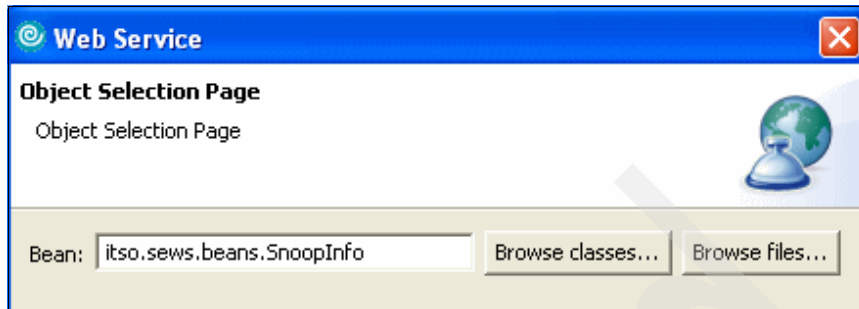
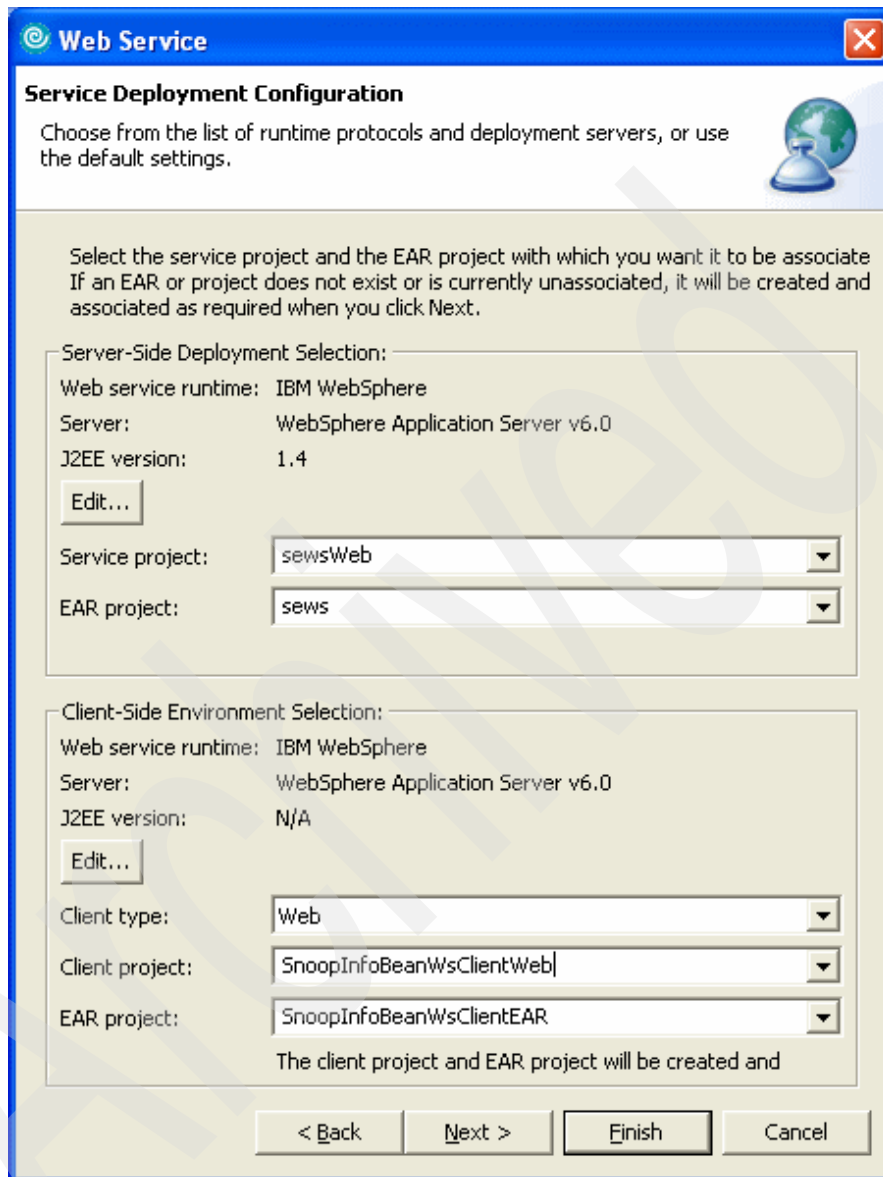


Figure 11-7 Selecting the Java Bean to Web service enable

Click **Next**. The next window displayed is where you specify where the generated code will be saved.



The image shows a 'Web Service' dialog box titled 'Service Deployment Configuration'. It contains instructions to choose runtime protocols and deployment servers. The dialog is divided into two sections: 'Server-Side Deployment Selection' and 'Client-Side Environment Selection'. Both sections show 'Web service runtime' as 'IBM WebSphere', 'Server' as 'WebSphere Application Server v6.0', and 'J2EE version' as '1.4' (for server) or 'N/A' (for client). In the server section, 'Service project' is 'sewsWeb' and 'EAR project' is 'sews'. In the client section, 'Client type' is 'Web', 'Client project' is 'SnoopInfoBeanWsClientWeb', and 'EAR project' is 'SnoopInfoBeanWsClientEAR'. A note states 'The client project and EAR project will be created and'. At the bottom are buttons for '< Back', 'Next >', 'Finish', and 'Cancel'.

Web Service

Service Deployment Configuration

Choose from the list of runtime protocols and deployment servers, or use the default settings.

Select the service project and the EAR project with which you want it to be associate
If an EAR or project does not exist or is currently unassociated, it will be created and associated as required when you click Next.

Server-Side Deployment Selection:

Web service runtime: IBM WebSphere
Server: WebSphere Application Server v6.0
J2EE version: 1.4
Edit...

Service project: sewsWeb
EAR project: sews

Client-Side Environment Selection:

Web service runtime: IBM WebSphere
Server: WebSphere Application Server v6.0
J2EE version: N/A
Edit...

Client type: Web
Client project: SnoopInfoBeanWsClientWeb
EAR project: SnoopInfoBeanWsClientEAR
The client project and EAR project will be created and

< Back Next > Finish Cancel

Figure 11-8 Identifying where the generated code will be saved

Click **Next**. The next window displayed deals with selecting what service endpoint to use. Leave it as shown in Figure 11-9 on page 377.

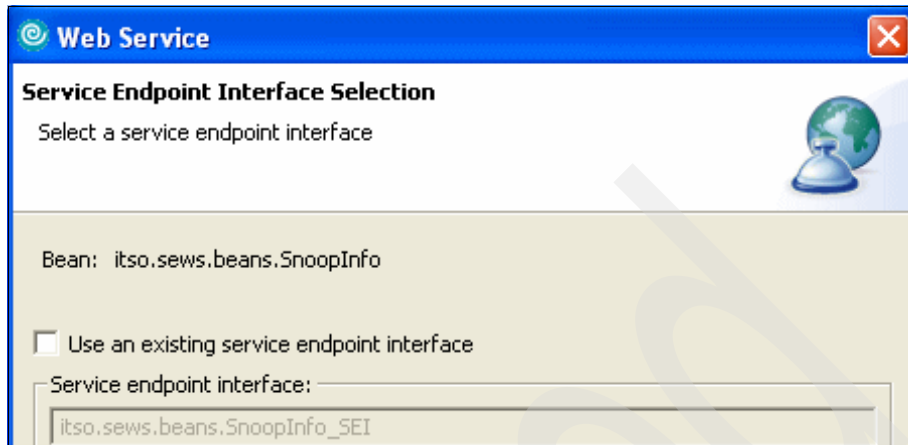


Figure 11-9 Let process generate service endpoint

Click **Next**. On the next window displayed, you select the methods of the Java Bean to be made available as Web services. For our application, only select the SnoopInfo method. Leave other settings as shown in Figure 11-10 on page 378.

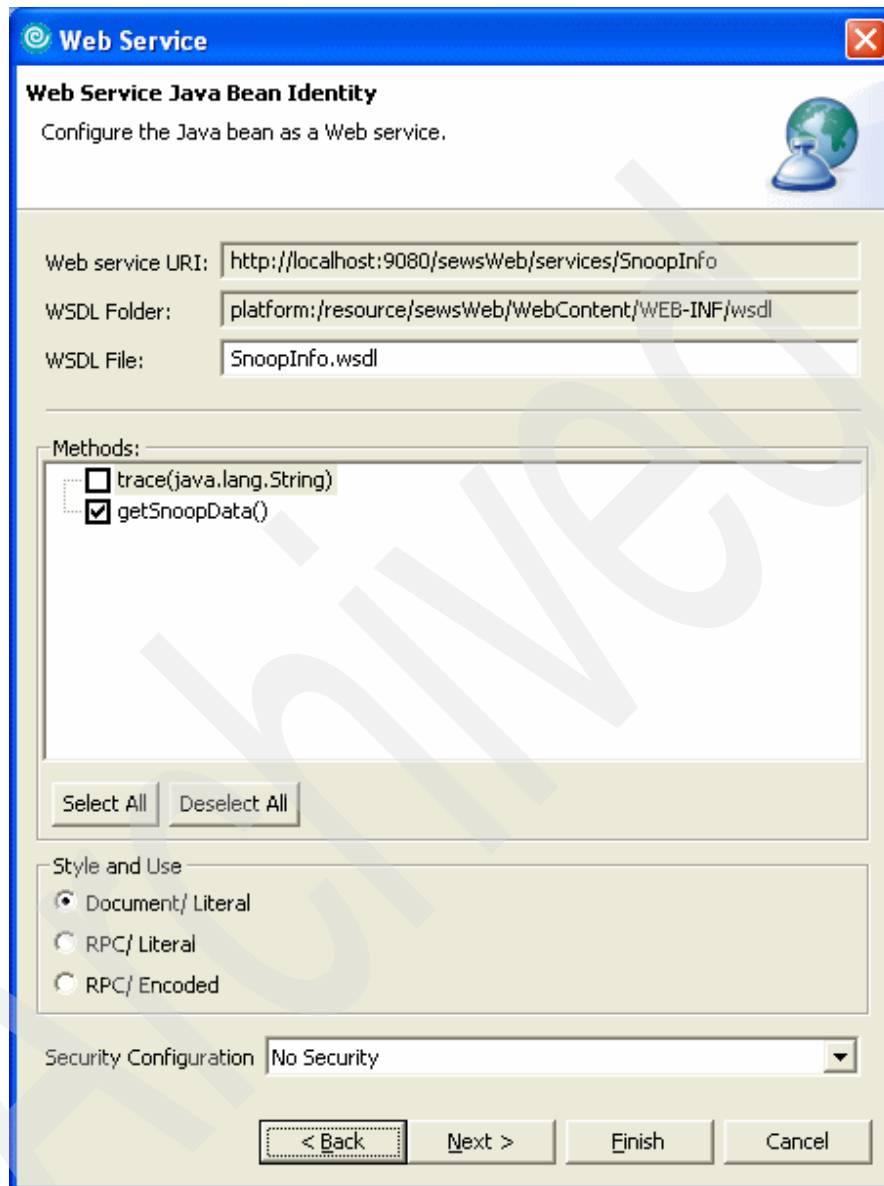


Figure 11-10 Web Service Wizard: Identity settings

Click **Next**. The various components to Web service-enable the Java Bean are then generated.

The next window displayed is used to generate a Web service proxy. This will generate Java code that calls the Web service. Set the options as shown in Figure 11-11.

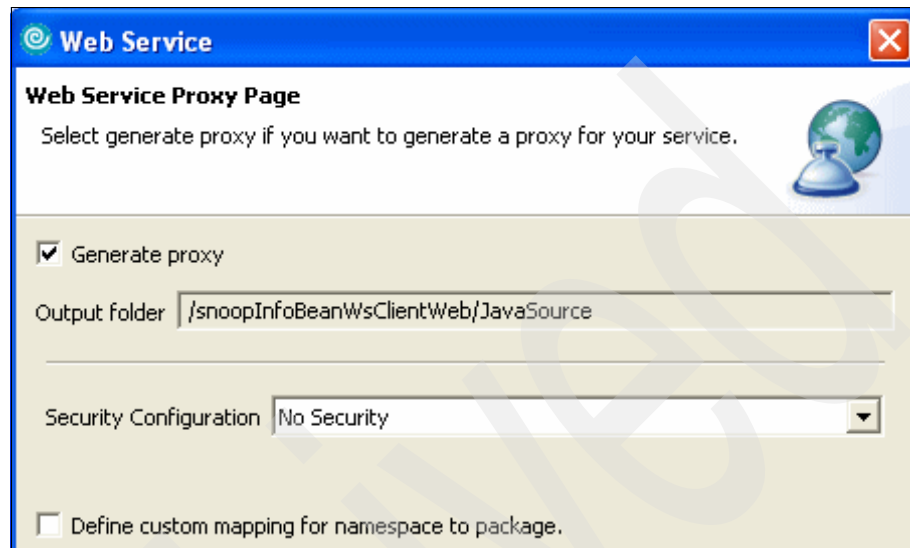


Figure 11-11 Generating a proxy for the Web service

Click **Next**. You may get a warning message similar to that shown in Figure 11-12.

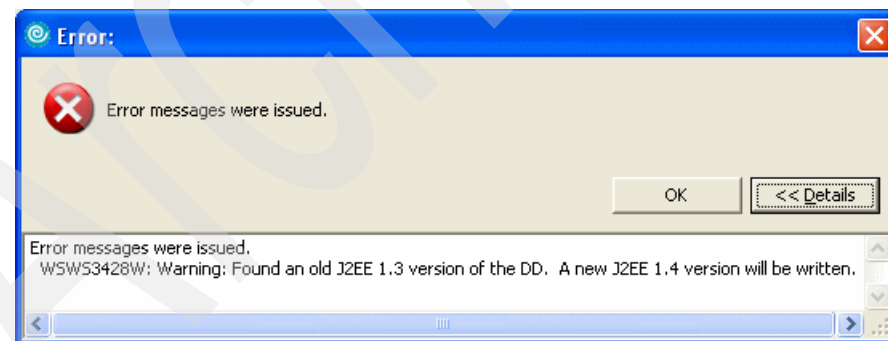


Figure 11-12 Warning message

Click **OK** to accept the warning message, and then click **Next** again.

On the next window leave the options as shown in Figure 11-13 on page 380.

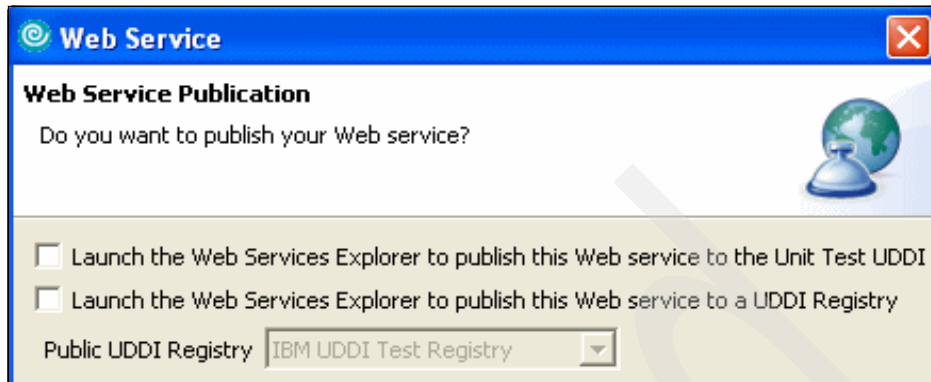


Figure 11-13 Setting options to do with publishing the Web service

Click **Finish** and the process is complete. The SnoopInfo Java Bean has now been Web service-enabled.

WSDL file

The WSDL file that describes this Web service is located under **Dynamic Web Projects** → **sewsWeb** → **WebContent** → **wsdl** and is called SnoopInfo.wsdl. It can also be viewed under **Web Services** → **Services** → **SnoopInfoService**.

11.3.3 Test SEWS in Rational Application Developer Version 6

We can now test our Web service using the Web Service Explorer facility of Rational Application Developer Version 6. In the project, select **Web Services** → **Services** → **SnoopInfoService** → **Test with Web Service Explorer**, as shown in Figure 11-14 on page 381.

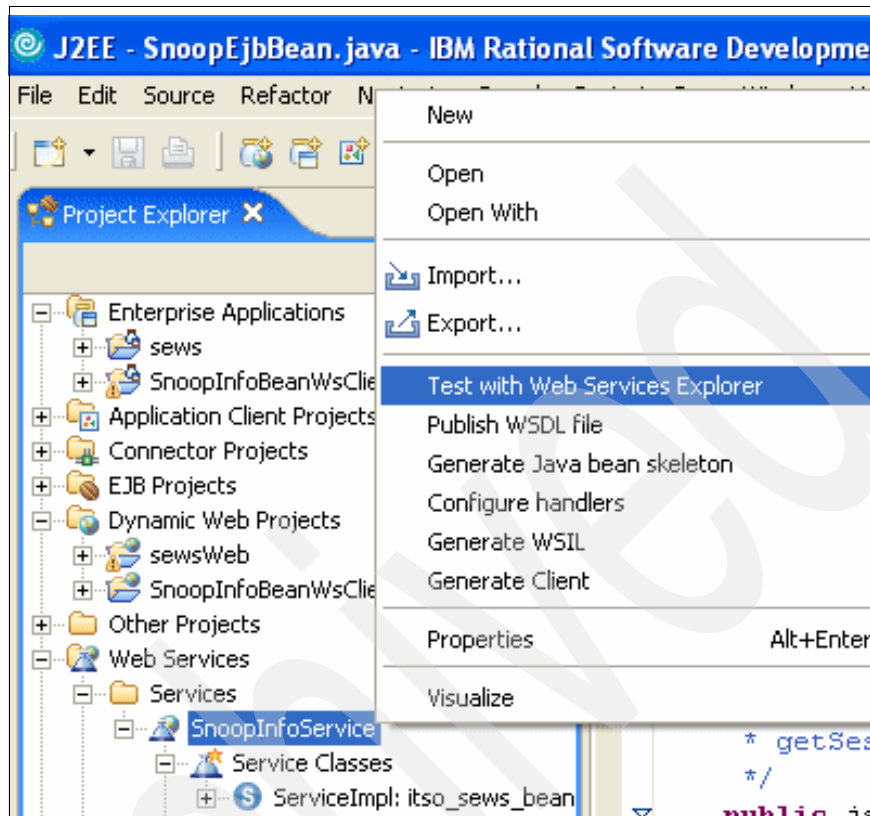


Figure 11-14 Using Web Services Explorer to test the SnoopInfo Web service

A Web browser view opens with the WSDL file selected. Expand **SnoopInfoSoapBinding**, click **getSnoopData**, then click **Go** in the window on the right to invoke the Web service. The result will be similar to that shown in Figure 11-15 on page 382.

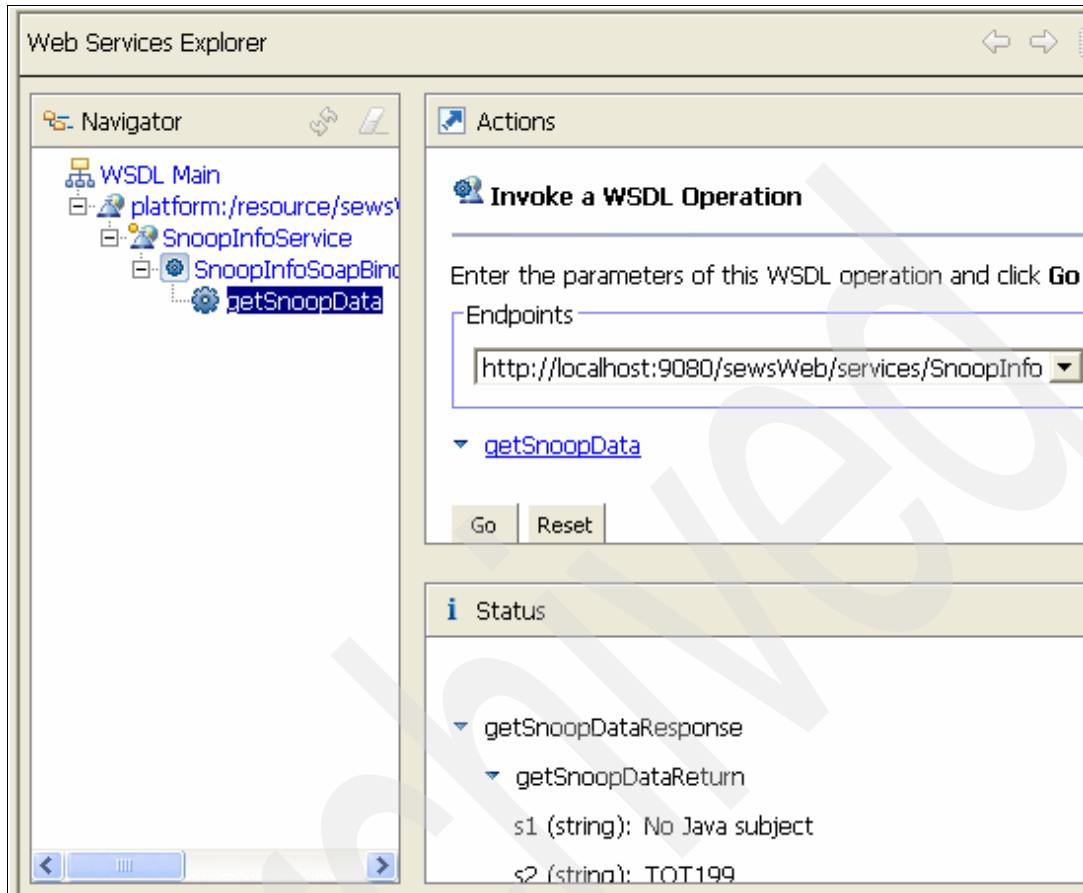


Figure 11-15 Result of invoking the Web service in Rational Application Developer Version 6

Now export the application as an EAR file and deploy it into a WebSphere server.

11.3.4 Export Web service client code

A by-product of the Web service generation process described in “Web service-enable SnoopInfo Java Bean” on page 370 is generated client code. This makes the process of developing a new application to call the Web service very straightforward.

In the project where the Web service was created, export SnoopInfoBeanWsClientWeb to a WAR file, as shown in Figure 11-16 on page 383. Save it as a name of your choosing. We called it SnoopInfoBeanWsClient.war.

In the next section, “Development of Web service invoker” on page 383, we import this war file into the project we will use to set up the application that will call the Web services.

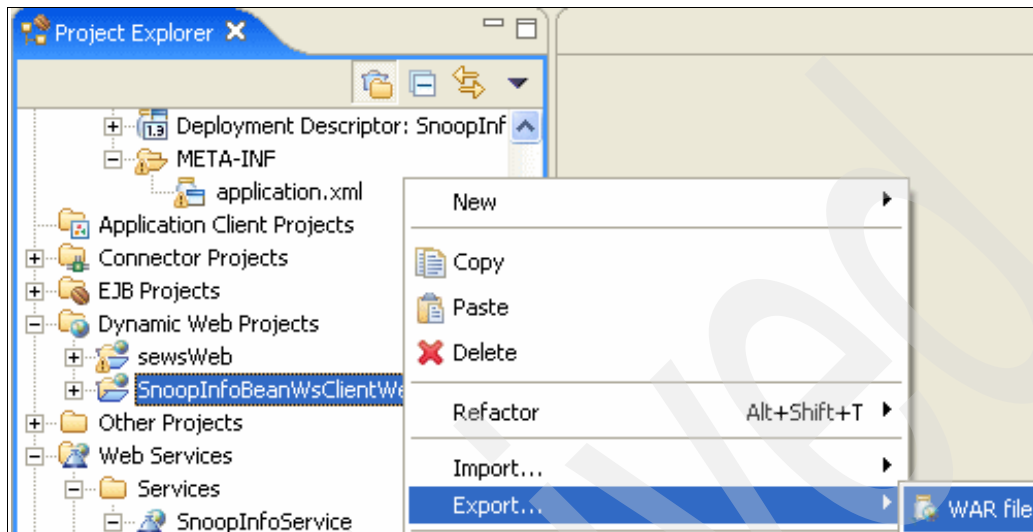


Figure 11-16 Exporting the client proxy code to a WAR file

11.4 Development of Web service invoker

Having created a Web service, we now want to create a new application that will call the Web service we have developed.

The starting point is what is in `sews-wsDemo-initial.ear`, part of the additional material for this book. Start Rational Application Developer Version 6, set a new workspace when asked, and after startup has completed, import this file, selecting the **Project Interchange** as the type of input source.

At this stage the application consists of:

| | |
|---------------------------|--|
| SewsWsDemo servlet | Servlet to run the Web service demonstration |
| sewsWsDemo JSP | JSP used by the servlet |

The imported application will have several compile errors, but these will be resolved when we add in the classes that provide the support to call the Web service.

11.4.1 Java Code that performs the WebSphere Web service call

The Java code in the SewsWsDemo servlet that calls the Web service that we will run in WebSphere is straightforward.

In SewsWsDemo.java, located in the Dynamic Web Projects folder under sewsDemoWeb, in the processRequest method. The following line creates a new instance of the SnoopInfoProxy class:

```
SnoopInfoProxy sip = new SnoopInfoProxy();
```

This class will be in the WAR file imported as explained later in “Import the SnoopInfoBean client WAR file” on page 385, and will handle the actual Web service call.

When the servlet is initially run, we display the default location of the Web service, which we obtain via this line:

```
wsHost1 = sip.getEndpoint();
```

The user running the servlet updates this value with the TCP/IP and port address of the server where the Web service is located. This is also referred to as the end point. In the Java code we set the end point with this line:

```
sip.setEndpoint(wsHost1);
```

We then call the Web service with this line:

```
sd = sip.getSnoopData();
```

The returned data is retrieved with the lines of code shown in Example 11-2.

Example 11-2 Retrieving the returned data

```
sInfo1 = sd.getS1();  
sInfo2 = sd.getS2();  
sInfo3 = sd.getS3();  
sInfo4 = sd.getS4();
```

11.4.2 Java code that calls the CICS Web service

The code to call the Web service in CICS is very similar. The line to obtain an instance of the Proxy class to handle the Web service call is:

```
SEWS0002PortProxy sip = new SEWS0002PortProxy();
```

The call to the Web service is then done with these lines:

```
Sews_Biz_Logic_Request_Data sdReq =  
    new Sews_Biz_Logic_Request_Data();  
Sews_Biz_Logic_Response_Data sd2 = sip.SEWS0002Operation(sdReq);
```

11.4.3 Import the SnoopInfoBean client WAR file

Import the SnoopInfoBeanWsClient.war file that was exported in “Export Web service client code” on page 382.

Then add the SnoopInfoBeanWsClientWeb project to the build path of the sewsWsDemoWeb project, as shown in Figure 11-17.

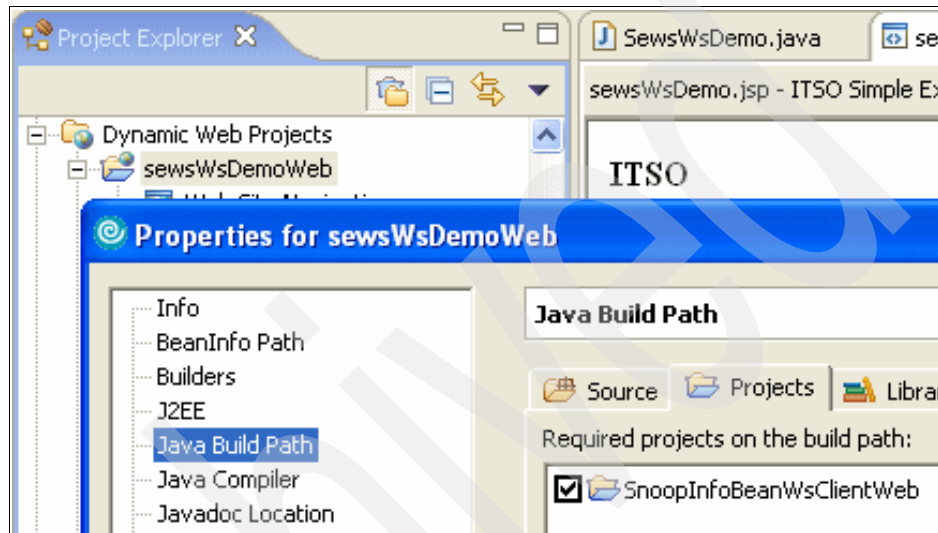


Figure 11-17 Importing the SnoopInfo Web service client WAR file

This will resolve the errors related to the SnoopInfoBean classes previously present in the project.

We now show how we create a WSDL file for the business logic that exists in COBOL program SEWS0002 to make it available as a Web service. We need this WSDL file to generate proxy code for use in the sewsWsDemo project.

11.5 CICS - Web service-enable business logic

In “CICS part of SEWS” on page 365 we described how the CICS part of the SEWS application works.

We now want to make the business logic represented by the SEWS0002 program available as a Web service. Clearly, the business logic represented here by the SEWS0002 program is very simple, but the purpose of this section is to demonstrate how to make existing CICS business logic available as a Web service.

11.5.1 Generate WSDL from a CICS business logic program

To make SEWS0002 callable as a Web service, we use a supplied CICS program called DFHLS2WS to generate binding information and a WSDL file.

The JCL we used to generate a WSDL to represent the SEWS0002 business logic is shown in Example 11-3.

Example 11-3 JCL to run DFHLS2WS

```
// JCLLIB ORDER=CICSTS31.CICS.SDFHINST
//*
//WS2LS      EXEC DFHLS2WS,
//  JAVADIR='java142s/J1.4',
//  USSDIR='cicsts31',
//  TMPFILE='cb',
//  PATHPREF=''
//INPUT.SYSUT1 DD *
PDSLIB=//EDMCAR.COPYLIB
LANG=COBOL
REQMEM=SEWSWP
RESPMEM=SEWSWR
PGMNAME=SEWS0002
URI=/service/snoopInfoCICS
PGMINT=COMMAREA
LOGFILE=/usr/cics/sews/wsprov/wsbinding/sewsWsCics.log
WSBIND=/usr/cics/sews/wsprov/wsbinding/sewsWsCics.wsbinding
WSDL=/usr/cics/sews/wsprov/wsd1/sewsWsCics.wsd1
*/
```

The input parameters are described in Table 11-5. Note that in the JCL above we hardcoded the location of Java and CICS. This should not normally be necessary.

Table 11-5 Description of input parameters to the DFHWS2LS program

| Input parameter | Purpose |
|-----------------|--|
| PDSLIB | Name of a PDS that contains the request and response data structures. |
| LANG | Identifies the programming language the generated code should be in. |
| PGMINT | Indicates whether the program is passed a COMMAREA or a container. |
| REQMEM | Name of PDS member that contains data structure that maps the request sent to the Web service. |

| Input parameter | Purpose |
|-----------------|---|
| RESPMEM | Name of PDS member that contains the data structure that maps the reply sent back in response. |
| PGMNAME | This is the name of the program in CICS that will be invoked to process the Web service requests when it is received by CICS. |
| URI | This is the URI that will become part of the default end point address where the Web service is located. |
| LOGFILE | Location where log information from the execution of this program will be written. |
| WSBIND | Location and name of file where the binding information will be written. |
| WSDL | Location and name of the WSDL file that will be created by running this JCL. |

Note that there more detailed descriptions of these and other parameters for the DFHWS2LS program can be found in the CICS Transaction Server Version 3.1 documentation at:

http://publib.boulder.ibm.com/infocenter/cicsts31/index.jsp?topic=/com.ibm.cics.ts.doc/dfhws/reference/dfhws_ws2ls.htm

The value referenced by the REQMEM keyword is the name of a member in a dataset that contains a COBOL data structure that maps the data expected by the CICS program. In our case the member SEWSWP contained the lines shown in Example 11-4.

Example 11-4 Contents of the SEWSWP member

```

01 SEWS-Biz-Logic-Request-Data.
   05 SEWS-bli-function  PIC X(8).
   05 SEWS-bli-fill      PIC X(78).
   05 SEWS-CICS-I1       PIC X(70).
   05 SEWS-CICS-I2       PIC X(70).
   05 SEWS-CICS-I3       PIC X(70).
   05 SEWS-CICS-I4       PIC X(70).

```

The value referenced by the RESPMEM keyword is the name of a member in a dataset, that contains a COBOL data structure, that maps the data the CICS

program returns. In our case the member SEWSWR contained the lines shown in Example 11-5.

Example 11-5 Contents of the SEWSWR member

```
01 SEWS-Biz-Logic-Response-Data.
   05 SEWS-blo-function PIC X(8).
   05 SEWS-blo-fill     PIC X(78).
   05 SEWS-CICS-01      PIC X(70).
   05 SEWS-CICS-02      PIC X(70).
   05 SEWS-CICS-03      PIC X(70).
   05 SEWS-CICS-04      PIC X(70).
```

11.5.2 Import WSDL into project

FTP the generated WSDL file to the workstation where you are running Rational Application Developer Version 6. The WSDL file is in EBCDIC on the z/OS system so it needs to be converted to ASCII for the distributed environment.

In case you are unable to generate the WSDL yourself on z/OS, you can use the WSDL supplied as part of the additional material for this book.

Important: In the generated WSDL is this line:

```
<soap:address location="http://my-server:my-port/service/snoopInfoCICS"/>
```

The string `my-server:my-port` must be replaced with a string to make it a valid URL, otherwise it will result in the Web service call not working. Change the line to the TCP/IP address and port number you are using for your CICS server.

```
<soap:address location="http://9.12.4.34:8700/port/service/snoopInfoCICS"/>
```

Import the WSDL file into the sews wsDemo project, into the `sewWsDemoWeb/WebContent` part of the project, as shown in Figure 11-18 on page 389.

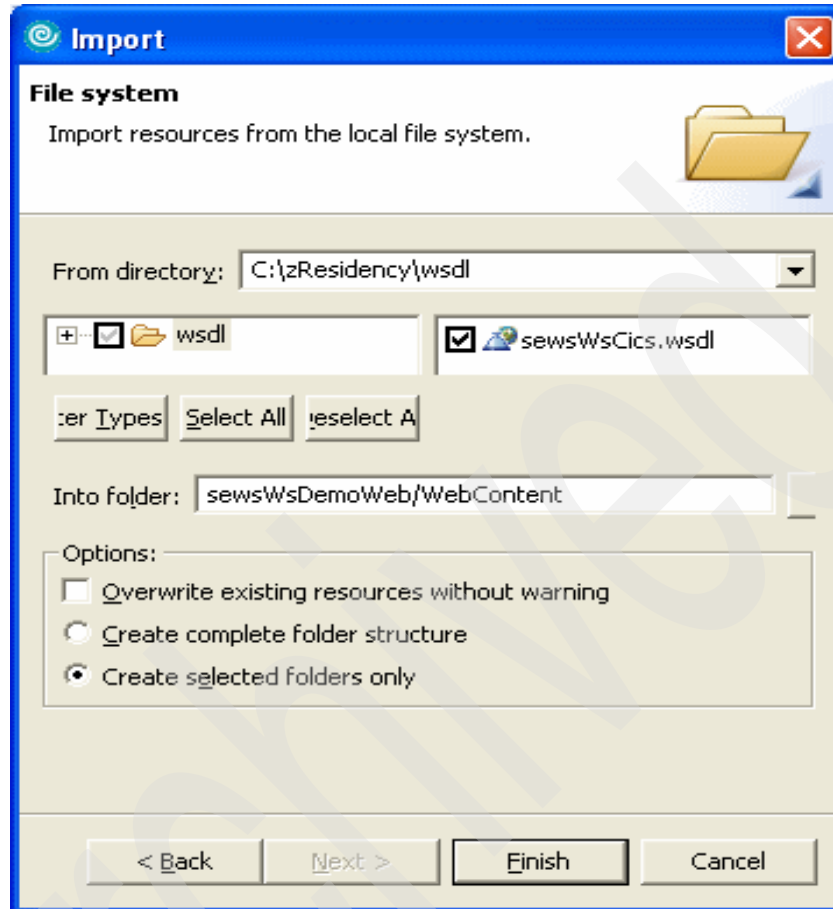


Figure 11-18 Importing the *snoopInfoCICS wsdl* file

Click **Finish** and the file is imported.

Select the WSDL file, then right-click it and select **Web Services** → **Generate Client**.

On the Web Services page click **Next**.

On the Web Service Selection page click **Next**.

On the Client Environment Configuration page, in the Client Side Environment Selection area, set Client Project to SnoopInfoCICSWsClientWeb and EAR project to SnoopInfoCICSWsClientEar. Click **Next**.

On the Web Service Proxy page click **Finish**.

On the Service Proxy Page click **Next**.

On the Web Service Publication click **Finish**.

Add the SnoopInfoCICSWsClientWeb project to the build path of the sewsWsDemoWeb project. This will resolve the errors in SewsWsDemo.java.

Update the application.xml

Before exporting the application as an ear file, we need to update the application.xml file, to add the two Web service client projects as modules in the application.

Expand **Enterprise Applications** → **sewsWsDemo** → **META-INF**, then double-click **application.xml**. Select the **Module** tab, then under Modules, click **Add**. Add SnoopInfoBeanWsClientWeb and SnoopInfoCICSWsClientWeb, as shown in Figure 11-19.

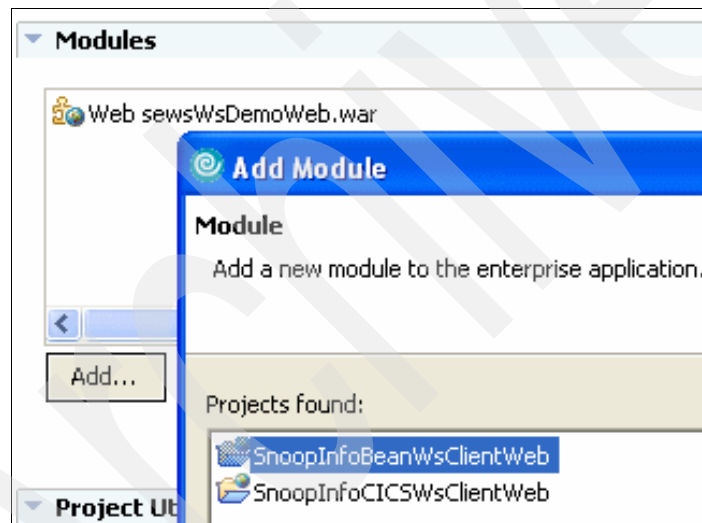


Figure 11-19 Adding the two Web service projects to the application

Then click the **Deployment** tab, and set the WAR class loading policy to application, as shown in Figure 11-20 on page 391.

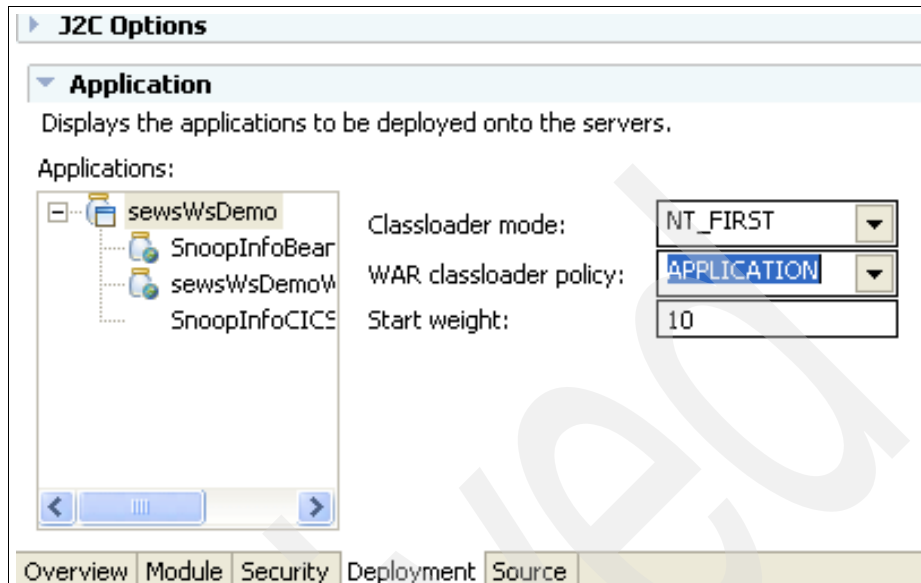


Figure 11-20 Setting the WAR class loader policy

Save the updated application.xml file. Export the application as an ear file and call it sewsWsDemo.ear. Deployment and testing of the application are explained in “Testing the SEWS application” on page 402.

Next we explain how the SEWS0002 program also performs a Web service call, as well as processing Web service requests, in “CICS - Performing a Web service request” on page 391. Then in “Web service definitions required in CICS” on page 396 we explain what other definitions are required in the CICS region so that it can be both a Web service requestor and provider.

11.6 CICS - Performing a Web service request

In the following sections we explain how a Web service enabled Cobol program can be called as a Web service and how a Cobol program can make a call to a Web service.

SEWS0002 - The Web service engine

The SEWS0002 program demonstrates the capability of CICS to be both a Web service provider and requestor.

Programs in CICS that work with Web services can use the existing commarea type approach that has always existed for passing data, or can use the new

concept of *channels* and *containers*. The major advantage of containers is that there is no 32 K limit as there is with the commarea.

A full discussion of channels and containers is beyond the scope of this book. A starting point in the CICS Transaction Server Version 3.1 Info Center at:

http://publib.boulder.ibm.com/infocenter/cicsts31/index.jsp?topic=/com.ibm.cics.ts.doc/lpaths/channels_lp_summary.htm

The SEWS0002 program demonstrates the use of both approaches. For the part of the example where we are showing how to Web service-enable existing business logic in a COBOL program, we use the commarea approach. For the part of the example where we are showing the SEWS0002 program performing a Web service call, we use the new container approach.

In Example 11-6 the COBOL statements that prepare a container to be used on the Web service call are shown.

Example 11-6 COBOL code to prepare a container for the Web service call

```
*-----*
* Set up the data for the web service call
*-----*
* 'DFHWS-DATA' is the name of the container that will store
* the data to make the webservice request
  MOVE 'DFHWS-DATA' TO WS-SERVICE-CONT-NAME

* 'sewsCh1' is the name of the channel we will pass to
* the web service call
  MOVE 'sewsCh1' TO WS-CHANNELNAME

* 'sews' is the name of the WEBSERVICE resource
* installed in this CICS region
  MOVE 'sews' TO WS-WEBSERVICE-NAME

* 'getSnoopData' is the name of the operation we are going to
* invoke on the remote service
  MOVE 'getSnoopData' TO WS-OPERATION

*-----*
* Place the request data into a container on a channel
*-----*
      EXEC CICS PUT CONTAINER(WS-SERVICE-CONT-NAME)
                  CHANNEL(WS-CHANNELNAME)
                  FROM(SEWS-DATA)

      END-EXEC
```

The code above was based on a supplied sample program in SDFHSAMP(DFH0XWOD).

Having prepared the container, the COBOL statement to invoke the Web service is shown in Example 11-7.

Example 11-7 CICS COBOL code to invoke a Web service

```
EXEC CICS INVOKE WEBSERVICE(WS-WEBSERVICE-NAME)
          CHANNEL(WS-CHANNELNAME)
          URI(WS-ENDPOINT-Uri)
          OPERATION(WS-OPERATION)
          RESP(RESP) RESP2(RESP2)
```

The various keywords in the WEBSERVICE API are explained in Table 11-6.

Table 11-6 Description of keywords on the WEBSERVICE API

| Keyword | Purpose | Value |
|------------|--|--|
| WEBSERVICE | Name of the Web service resource that CICS will use to marshal and demarshal the request | sews |
| CHANNEL | Name of the channel to pass to CICS | sewsChl |
| URI | End point where the Web service is located | Default value is http://localhost:9080/sewsWeb/services/SnoopInfo |
| OPERATION | The operation to be invoked in the Web service | getSnoopData |

When this sample transaction is run, you enter the TCP/IP address and port to be used in the URI, and this is passed through to the SEWS0002 program, which then sets the complete URI value.

11.6.1 Defining SnoopInfo Web service to CICS

The COBOL part of the SEWS CICS transaction is written to call the SnoopInfo Web service in WebSphere. There are a number of steps to follow to define the SnoopInfo Web service to CICS so that it can be called.

The WSDL file produced in “Web service-enable SnoopInfo Java Bean” on page 370 needs to be copied up to a directory on the z/OS system.

If you ftp the WSDL to the z/OS system as ASCII (binary transfer), then the file requires no further modification.

If you ftp the file to the z/OS system and do not use binary transfer, then the file will be in EBCDIC. In that case, prior to running DFHWS2LS you must change the first line in the file from:

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

To:

```
<?xml version="1.0" ?>
```

On our system we copied the WSDL to:

```
/usr/cics/sews/wsreq/wsd1/sews.wsd1
```

Run the DFHWS2LS program. This takes the WSDL file that describes the Web service and creates corresponding binding information that CICS then uses. The JCL we used to process the WSDL for the SnoopInfo Web service is shown in Example 11-8.

Example 11-8 JCL to run DFHWS2LS

```
// JCLLIB ORDER=CICSTS31.CICS.SDFHINST
//*
//WS2LS      EXEC DFHWS2LS,
//  JAVADIR='java142s/J1.4',
//  USSDIR='cicsts31',
//  TMPFILE='cb',
//  PATHPREF=''
//INPUT.SYSUT1 DD *
PDSLIB=//EDMCAR.COPYLIB
LANG=COBOL
PGMINT=CONTAINER
REQMEM=SEWSRQ
RESPMEM=SEWSRS
LOGFILE=/usr/cics/sews/wsreq/wsbind/sews.log
WSBIND=/usr/cics/sews/wsreq/wsbind/sews.wsbind
WSDL=/usr/cics/sews/wsreq/wsd1/sews.wsd1
BINDING=SnoopInfoSoapBinding
*/
```

The input parameters are described in Table 11-7 on page 395. Note that in the JCL above we hardcoded the location of Java and CICS. This should normally not be necessary.

Table 11-7 Description of input parameters to the DFHWS2LS program

| Input parameter | Purpose |
|-----------------|---|
| PDSLIB | Name of a PDS into which will be saved generated code |
| LANG | Identifies the programming language the generated code should be in |
| PGMINT | Indicates whether the program is passed a COMMAREA or a container |
| REQMEM | Name of PDS member that will store the generated code that will map the data structure passed out on the request |
| RESPMEM | Name of PDS member that will store the generated code that will map the data structure returned from the Web service call |
| LOGFILE | Location where log information from the execution of this program will be written |
| WSBIND | Location and name of file where the binding information will be written |
| WSDL | Location and name of the WSDL file that describes the Web service |
| Binding | The name of the binding element in the WSDL file to use |

The value set for the binding keyword is obtained from this line of the WSDL:

```
<wsdl:binding name="SnoopInfoSoapBinding" type="impl:SnoopInfo">
```

As a result of running the DFHWS2LS program, two dataset members are generated, as set by the REQMEM and RESPMEM keywords. These members contain COBOL data structures. The member referenced by REQMEM provides a data structure that maps the data passed to the CICS program. The RESPMEM provides a data structure that maps the data to be sent back by the CICS program.

In our case the result was a member called SEWSRS01 that contained the lines shown in Example 11-9.

Example 11-9 Generated COBOL data structure to map response

```
05 getSnoopDataResponse.
```

```

10 getSnoopDataReturn.
15 s1 PIC X(255).
15 s2 PIC X(255).
15 s3 PIC X(255).
15 s4 PIC X(255).
15 s5 PIC X(255).
15 s6 PIC X(255).
15 s7 PIC X(255).

```

We then copied this code into the SEWS0002 program and used it there to build the reply expected.

As our Web service in CICS did not expect any input parameters, we did not need to use the member generated for the REQMEM, since it contained no data structure anyway.

Note that more detailed descriptions of these and other parameters for the DFHWS2LS program can be found in the CICS Transaction Server Version 3.1 documentation at:

http://publib.boulder.ibm.com/infocenter/cicsts31/index.jsp?topic=/com.ibm.cics.ts.doc/dfhws/reference/dfhws_ls2ws.htm

11.7 Web service definitions required in CICS

CICS requires the following types of definitions for it to be able to support Web services:

- ▶ DFHPIPE
- ▶ TCPIP and TCPIP Service
- ▶ URIMAP
- ▶ Pipeline
- ▶ WEBS

In the following sections we give a brief explanation of each of them.

11.7.1 DFHPIPE

Check that the definitions in the supplied group DFHPIPE have been installed in the CICS region.

11.7.2 TCPIP and TCPIP Service

We do not explain in this book how to configure a CICS region to support TCP/IP. That is well covered in the CICS documentation available at:

<http://publib.boulder.ibm.com/infocenter/cicsts31/index.jsp>

Example 11-10 shows a CEMT display showing the TCPIP Service definition for the port we used on our system.

Example 11-10 CICS CEMT display showing TCPIP port used to receive Web services

```
I TCPIPS(SOAPPORT)
STATUS: RESULTS - OVERTYPE TO MODIFY
  TcpiPs(SOAPPORT) Ope Por(08700) Http Nos Tra(CWXN)
    Con(00009) Bac( 00010 ) Max( 000256 ) Urm( NONE )
```

11.7.3 URIMAP

A full description of the CICS URIMAP resource is beyond the scope of this book, but a starting point in the CICS Info Center is at:

http://publib.boulder.ibm.com/infocenter/cicsts31/index.jsp?topic=/com.ibm.cics.ts.doc/dfha4/resources/urimap/dfha4_overview.htm

Briefly, a URIMAP definition is required by CICS so that it can identify that this request received by the CICS region is in fact a request for a Web service, and not a standard CICS Web request.

The URIMAP definition identifies the CICS pipeline request to process the definition. Example 11-11 shows the definition we used on our system to define to the CICS region the SnoopInfo CICS Web service.

Example 11-11 CEMT display of URIMAP for the Snoop Info CICS Web service

```
I URIMAP
RESULT - OVERTYPE TO MODIFY
  Urimap($415050)
  Usage(Pipe)
  Enablestatus( Enabled )
  Analyzerstat(Noanalyzer)
  Scheme(Http)
  Redirecttype( None )
  TcpiPsService()
  Host(*)
  Path(/service/snoopInfoCICS)
  Transaction(CPIH)
  Converter()
  Program()
```

```
Pipeline(SEWSPROV)
Webservice(sewsWsCics)
Userid()
Certificate()
Ciphers()
Templatename()
```

Notice in the above definition that the pipeline name is SEWSPROV, which is the pipeline that we will define to handle processing of Web service requests received by CICS, explained in “Pipeline” on page 398. Also, the URIMAP definition identifies the corresponding Webservice definition, described in “WEBS (Webservice)” on page 401, which will identify to CICS the program that CICS is to invoke to handle the received request.

You may have noticed the odd name for the above URIMAP resource of \$415050. In fact, we did not manually define this URIMAP definition, but used a feature of the CICS pipeline resource definition that resulted in CICS defining the URIMAP automatically. This is discussed in more detail once we have described pipelines themselves.

Note though that a URIMAP definition does provide a number of useful settings for controlling how CICS processes a Web service request it receives. In particular, the Userid and the Transaction fields let you set what will be the transaction ID the request is processed under, and Userid lets you set what user ID it runs under. This then lets you implement greater security controls around the processing of Web Requests by CICS if desired.

11.7.4 Pipeline

A full description of the CICS pipeline resource is beyond the scope of this book, but a starting point in the CICS Info Center is at:

http://publib.boulder.ibm.com/infocenter/cicsts31/index.jsp?topic=/com.ibm.cics.ts.doc/dfha4/resources/pipeline/dfha4_overview.htm

Briefly, CICS pipelines are the mechanisms CICS uses to control the processing of Web service into and out of CICS.

Importantly, a pipeline only goes in one direction. A single pipeline can only handle either incoming or outgoing Web services, not both.

Example 11-12 on page 399 shows the settings to define a pipeline called SEWSPROV. This pipeline will be the one that handles Web service requests sent to CICS.

Example 11-12 Definition to define the SEWSPROV pipeline

```
OVERTYPE TO MODIFY                                CICS RELEASE = 0640
CEDA Alter Pipeline( SEWSPROV )
  Pipeline      : SEWSPROV
  Group         : ITS07064
  Description    ==>
  Status        ==> Enabled                Enabled | Disabled
  Configfile    ==> /usr/lpp/cicsts/cicsts31/samples/pipelines/basicsoap11prov
  (Mixed Case) ==> ider.xml
              ==>
              ==>
              ==>
  SHelf         ==> /var/cicsts/
  (Mixed Case) ==>
              ==>
              ==>
              ==>
  Wsdir         ==> /usr/cics/sews/wsprov/wsbind
  (Mixed Case) ==>
              ==>
              ==>
              ==>
```

Example 11-13 shows the settings to define a pipeline called SEWSREQ. This pipeline will be the one that handles Web service requests issued by programs in CICS.

Example 11-13 Definition to define the SEWSREQ pipeline

```
CEDA Alter Pipeline( SEWSREQ )
  Pipeline      : SEWSREQ
  Group         : ITS07064
  Description    ==>
  Status        ==> Enabled                Enabled | Disabled
  Configfile    ==> /usr/lpp/cicsts/cicsts31/samples/pipelines/basicsoap11req
  (Mixed Case) ==> ester.xml
              ==>
              ==>
              ==>
              ==>
  SHelf         ==> /var/cicsts/
  (Mixed Case) ==>
              ==>
              ==>
              ==>
              ==>
  SHelf         ==> /var/cicsts/
  (Mixed Case) ==>
              ==>
              ==>
```

```
==>
Wsdir      ==> /usr/cics/sews/wsreq/wsbind
(Mixed Case) ==>
==>
==>
==>
```

Note that in the above pipeline definitions for the requestor and provider have different Configfile and Wsdir settings. The Configfile pointed to by the pipeline definition is an XML file that describes the processing the pipeline is to perform on the requests it receives or the response it sends. CICS supplies sample files that should be adequate to begin with and can be extended as required.

The Wsdir setting points to a directory in the HFS where the binding files that describe the Web services are located. You should have separate directories for each pipeline.

After defining the above definitions using CEDA, they are installed into the CICS region.

When a pipeline definition is installed into a CICS region, CICS automatically scans the directory specified by the Wsdir setting. For an inbound pipeline definition, any Web service binding files found will result in the automatic creation of a corresponding URIMAP definition.

On our system, when we installed the above pipeline definitions, the URIMAP definition shown in Example 11-11 on page 397 was defined. The following command can also be used to have CICS update the URIMAP definitions should you add additional Web service binding files to the directory at a later date:

```
CEMT P PIPELINE(SEWSPROV) SCAN
```

The above definitions are all that are required in the CICS region to run the SEWS sample application.

Recommendation regarding location specified in Wsdir

In the examples in this chapter, we had the DFHWS2L and DFHLS2WS utilities write the wsbind files directly to the locations as specified in the Wsdir setting of the Pipeline definition.

However, in a real-world situation, this is not recommend. Better practice would be to have a set of directories reserved for use by the DFHLS2WS and DFHWS2LS programs, and a separate set of directories for use by the Pipeline definitions, and then set up a process to move the wsbind files between the two when ready to do so.

11.7.5 WEBS (Webservice)

The WEBS definition is used to define to CICS information about the Web services that programs in CICS can use or provide.

For the SnoopInfo Bean Web service called by the SEWS0002 program, the WEBS definition called *sews* is shown in Example 11-14.

Example 11-14 WebService definition for the SnoopInfo Bean Web service

```
I WEBS
RESULT - OVERTYPE TO MODIFY
  Webservice(sews)
  Pipeline(SEWSREQ)
  Validationst( Novalidation )
  State(Inservice)
  Urimap()
  Program()
  Pgminterface(Notapplic)
  Container()
  Datestamp(20050615)
  Timestamp(21:59:26)
  Wsdlfile()
  Wsbind(/usr/cics/sews/wsreq/wsbind/sews.wsbind)
  Endpoint(http://localhost:9080/sewsWeb/services/SnoopInfo)
  Binding(SnoopInfoSoapBinding)
```

Note that in the above definition the pipeline name is SEWSREQ, which is the pipeline we have defined to handle processing of Web service requests issued by programs in CICS.

For the Web service provided by the SEWS0002 program, the WEBS definition is shown in Example 11-15.

Example 11-15 Web service definition for the Web service provided by SEWS0002

```
I WEBS
RESULT - OVERTYPE TO MODIFY
  Webservice(sewsWsCics)
  Pipeline(SEWSPROV)
  Validationst( Novalidation )
  State(Inservice)
  Urimap($415050)
  Program(SEWS0002)
  Pgminterface(Commarea)
  Container()
  Datestamp(20050620)
  Timestamp(14:15:05)
  Wsdlfile()
```

```
Wsbinding(/usr/cics/sews/wsprov/wsbinding/sewsWsCics.wsbinding)
Endpoint()
Binding(SEWS0002HTTPSoapBinding)
```

Note in the above definition that the pipeline name is SEWSPROV, which is the pipeline we have defined to handle processing of Web service requests received by CICS. You can see that this definition identifies to CICS that it is the program SEWS0002 that is to be invoked to handle the request.

These definitions were automatically generated when the pipeline definitions were installed into the CICS region, or a scan operation done on the pipeline via the CEMT transaction. The definitions can be pre-defined via CEDA if required.

11.8 Testing the SEWS application

To test the SEWS application we will show the process using two separate WebSphere Application Server for z/OS Version 6.01 servers in different cells and one CICS region.

Both servers were located on the same LPAR with TCP/IP address of 9.12.4.38. Server A has security enabled and is listening for HTTP requests on port 9080. Server B has security disabled and is listening for HTTP requests on port 39080.

Deploy the sews-stage1.ear file into server A, either the one you have built yourself using the previous sections or the one supplied as part of the additional material for this book. This ear file contains the Web service.

Deploy the sewsWsDemo-stage1.ear into server B, either the one you have built yourself using the previous sections or the one supplied as part of the additional material for this book. This ear file contains the Web service requestor application.

On your own system you could deploy both ear files into one server if desired, but deploying the requestor application and the Web service itself into two separate servers should prove to any non-believers that a Web service call is really being done.

In the CICS region, use CEDA to define a transaction called SEWS that invokes the program SEWS0001. Compile SEWSBMS, SEWS0001, and SEWS0002, and put them in a load library that is part of the DFHRPL concatenation of the CICS region.

11.8.1 Running the Web service requestor application

First, access the Web service requestor application from your browser by entering the following address. However, use the TCP/IP address and port number of your own server.

`http://9.12.4.38:39080/sewsWsDemoWeb/SewsWsDemo`

The result will be as shown in Figure 11-21.



Figure 11-21 Initial display for the sewsWsDemo servlet

11.8.2 Test Web service call to SnoopInfo Bean in WebSphere

Select the radio button to the left of the box that invokes WebSphere SnoopInfo Bean (SOAP over HTTP), and update the TCP/IP address and port number that are preset in the URL for the Web service to reflect the values of your environment.

In our case we set the URL to:

`http://9.12.4.38:9080/sewsWeb/services/SnoopInfo`

Click **Invoke selected Web Service** and the result should be similar to that shown in Figure 11-22.

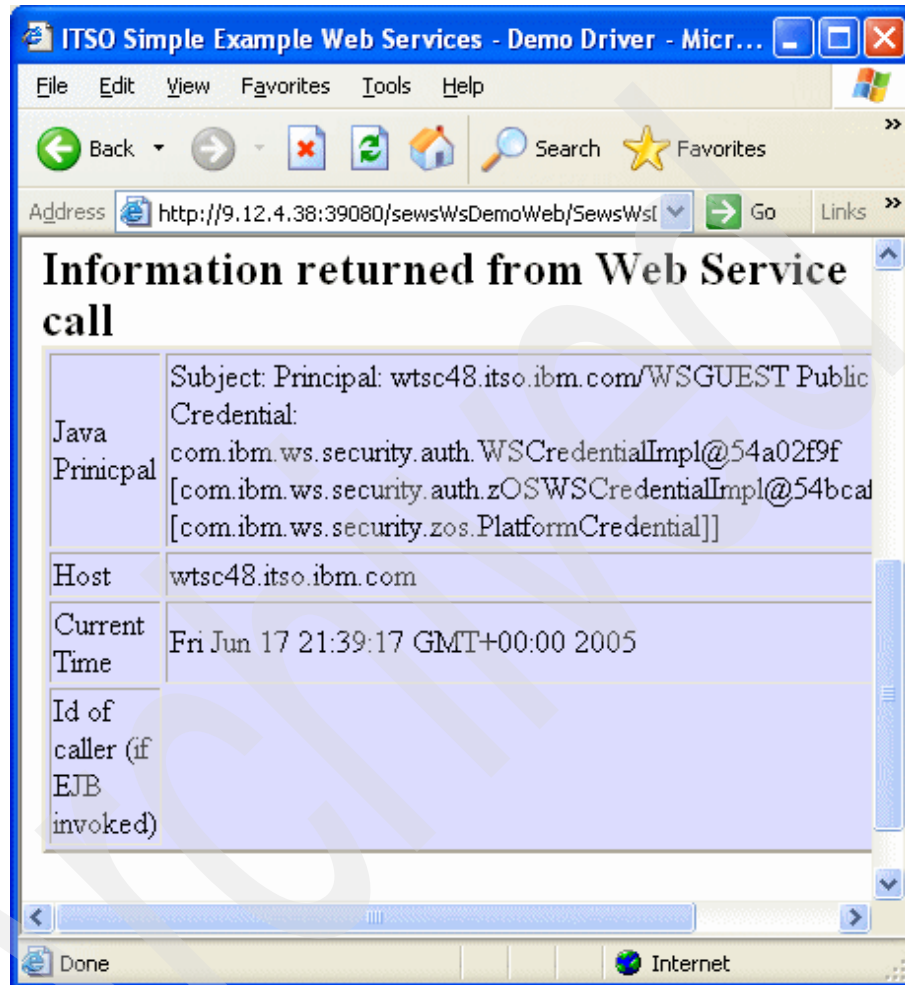


Figure 11-22 Result from calling the SnoopInfo Web service

This output shows we have been able to call the SnoopInfo Web service successfully.

11.8.3 Test Web service call to SnoopInfo in CICS

Select the radio button to the left of the box that invokes CICS Info Bean (SOAP over HTTP), and update the TCP/IP address and port number that are preset in the URL for the Web service to reflect the values of your environment.

In our case we set the URL to:

`http://9.12.4.38:8700/service/snoopInfoCICS`

This is the address of the CICS region running the Web service to be called.

Click **Invoke selected Web Service** and the result should be similar to that shown in Figure 11-23.

| Information returned from Web Service call | |
|--|----------------------------------|
| Userid | HAIMO HAIMO |
| Host | ERW1 CPH |
| Current Time | Mon, 20 Jun 2005 18:10:08 GMT |
| Id of caller (if EJB invoked) | |

Figure 11-23 Results of calling Web service in CICS

This output shows we have been able to call the Web service in CICS successfully.

11.8.4 Test Web service call from CICS to WebSphere

Log on to the CICS region and invoke transaction SEWS. Enter 2 as the option, and under the line Enter Host Name or IP Address where SnoopInfo Web Service is located, enter the value:

`http://9.12.4.38:9080/sewsWeb/services/SnoopInfo`

This value identifies the location of the Web service, in this case our WebSphere Application Server, and the Web service to invoke.

Press Enter, and the result should be similar to that shown in Example 11-16.

Example 11-16 Result of calling Web service from CICS

Simple Example Web Services

1. LINK to SEWS0002
2. Call the SnoopInfo Web Service

Enter Host Name or IP Address where SnoopInfo Web Service is located
http://9.12.4.38:9080/sewsWeb/services/SnoopInfo

Option:

Returned Data:

No Java subject

wtsc48.itso.ibm.com

Mon Jun 20 22:16:28 GMT+00:00 2005

Web Service API Response Codes Resp: 0000 Resp2: 0000

This output shows we have been able to call the SnoopInfo Bean Web service successfully from CICS.

11.9 Summary

In this chapter we have shown how to Web service-enable applications in WebSphere and CICS, and how to perform Web service calls into and out of WebSphere and CICS.

We have only dealt in this chapter with a simple example of Web services using SOAP over HTTP. *WebSphere Version 6 Web Services Handbook Development and Deployment*, SG24-6461, provides much more detail on the topics we have covered in this chapter, plus other more advanced features of the Web services available in WebSphere.

Web services and SOAP over JMS

This chapter uses the SEWS sample application as developed to the end of the previous chapter and shows how we extended it to call Web services using SOAP over JMS as the underlying transport. It then shows the definitions required to support SOAP over JMS when performing Web services in WebSphere, between WebSphere cells, and between WebSphere and CICS.

Important: The scenarios we have described in this chapter regarding the usage of SOAP over JMS between CICS and WebSphere are valid, but could not be tested successfully because of outstanding PTFs. To be able to successfully test the scenarios yourself, you have to make sure to install the required PTFs for this functionality once they become available.

12.1 Using SOAP over JMS

This chapter takes the theory of using SOAP over JMS for Web services and shows how to implement a basic setup.

In this chapter we show the following:

- ▶ In “Enabling SnoopEjb for SOAP over JMS” on page 410 we show how we extend the SEWS application to use SOAP over JMS as the transport mechanism for Web services.
- ▶ In “Web service using SOAP over JMS in one cell” on page 419 we show the setup required in a single WebSphere server to allow an application to call a Web service in the same server using SOAP over JMS.
- ▶ In “Web service using SOAP over JMS between cells” on page 441 we show the setup required to allow an application in a server in one WebSphere cell to invoke a Web service in a server in a different cell using SOAP over JMS.
- ▶ In “CICS to WebSphere using SOAP over JMS” on page 448 we show what setup is required to allow a program in CICS to invoke a Web service in WebSphere using SOAP over JMS.
- ▶ In “WebSphere to CICS using SOAP over JMS” on page 456 we show what setup is required to allow a program in WebSphere to invoke a Web service in CICS using SOAP over JMS.

Throughout the above sections, we use the values from our system when defining the various definitions. On your own system you would need to substitute values specific to your own system, such as CICS region name, TCP/IP addresses, and so on.

12.1.1 Our system layout

Throughout this chapter we show the various definitions and settings we used on our system to demonstrate the use of Web services via SOAP over JMS. The diagram in Figure 12-1 on page 409 shows the key components from our environment.

LPAR – SC48

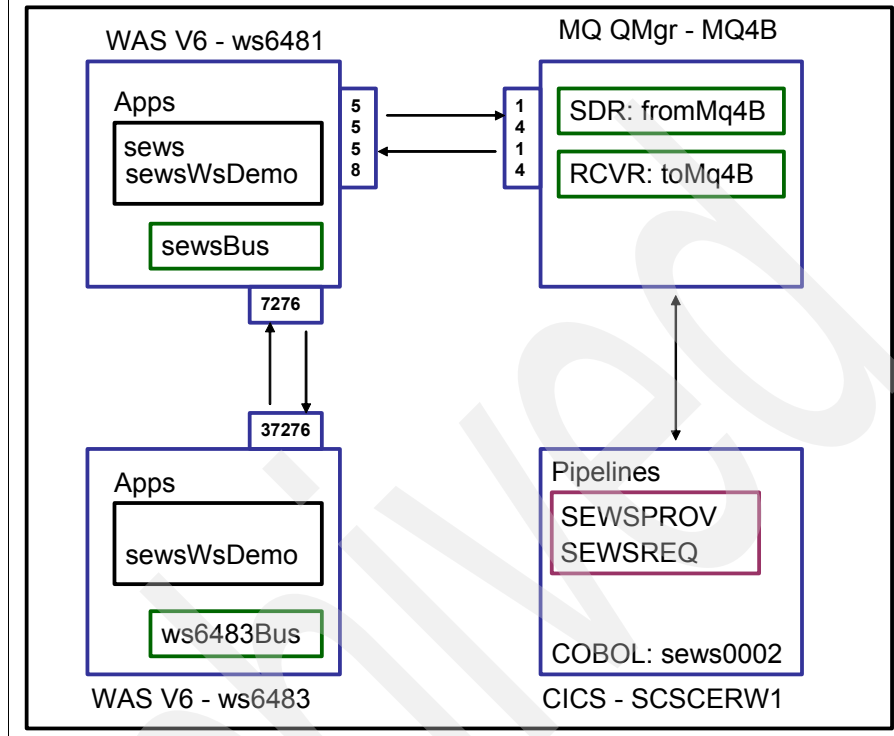


Figure 12-1 Environment used to describe example

WebSphere Server - ws6481

The ws6481 server is a WebSphere Application Server for z/OS Version 6.01 standalone server. It is listening on port 9080 for standard HTTP requests. It will have deployed into it the sews and sewsWsDemo applications.

WebSphere Server - ws6483

The ws6483 server is a WebSphere Application Server for z/OS Version 6.01 standalone server. It is listening on port 39080 for standard HTTP requests. It will have deployed into it the sewsWsDemo application. This server did not have global security enabled.

MQSeries Queue Manager - MQ4B

This is a MQSeries® V5.3.1 queue manager.

CICS Region - SCSCERW1

This is a CICS Transaction Server Version 3.1 region. It is set up to bind to the MQ4B queue manger.

Note that while the examples in this chapter use WebSphere Application Server for z/OS Version 6.01, the process would be the same from the WebSphere point of view regardless of platform.

12.1.2 Supplied material

Table 12-1 describes the additional material supplied with this book.

Table 12-1 Supplied additional material

| Name | Purpose |
|-----------------------|---|
| sews-stage2.zip | This is a project interchange zip file. It contains the sews project after the steps described in “Enabling SnoopEjb for SOAP over JMS” on page 410 have been completed. |
| sewsWsDemo-stage2.zip | This is a project interchange zip file. It contains the sewsWsDemo project after the steps described in “Update sewsWsDemo to invoke SnoopEjb Web service” on page 415 have been completed. |
| sews-stage2.ear | EAR file containing the sews application that is enabled to support SOAP over JMS. |
| sewsWsDemo-stage2.ear | Ear file containing updated version of the sewsWsDemo application that can call the SnoopEjb Web service. |

12.2 Enabling SnoopEjb for SOAP over JMS

When SOAP over JMS is used as the transport for invoking a Web service in WebSphere, only an EJB can be used as the underlying logic.

We show here how to enable SOAP over JMS for an existing application that has already had the EJB made available as a Web service via SOAP over HTTP.

Starting with the SEWS application as it was at the end of “Web service-enable the SnoopEJB” on page 411, we make the SnoopEjbBean EJB available as a Web service via SOAP over HTTP first. The process to do this is similar to that described to enable a Java Bean as a Web service. Once this is done we then show how to use Rational Application Developer Version 6 to add support so that the same EJB can be accessed via SOAP over JMS.

12.2.1 Web service-enable the SnoopEJB

Make sure you start with the workspace you used for Web service-enabling the SnoopInfo Java Bean, as described in 11.3.2, “Web service-enable SnoopInfo Java Bean” on page 370. If you did not perform that part of the scenario, you can also import the project interchange file from the additional material for this book, called `sews-stage1.zip`.

Create a new Web Project called `SnoopEjbWsWeb`. Add it as a module to the `sews` application.

In the SEWS application project, expand **EJB Projects** → **sewsEjb** → **ejbModule** → **itso.sews.ejb**. Then right-click **SnoopEjb.java**, and select **Web Services** → **Create Web Service**.

On the Web Services page, select **Ejb Web Service** as the Web service type and make sure you check **Generate a proxy**. Click **Next**.

On the Object Selection page, select **sews** as the EAR Project to work on, then select **SnoopEjb** as the EJB Bean to work with. Click **Next**.

On the Service Deployment Configuration page, in the Client-Side Environment Selection, set `SnoopEjbWsClientWeb` as the client project and `SnoopEjbWsClientEar` as the EAR project. Click **Next**.

On the Web Service EJB Configuration page, select **SnoopEjbWsWeb** as the Web Router project and select **SOAP over HTTP** as the transport. Click **Next**.

On the Web Service Java Bean Identity page click **Next**.

On the Web Service Proxy Page click **Next**.

On the Web Service Publication page click **Finish**.

Test that the Web service for the EJB is operational by expanding **EJB Projects** → **sewsEjb** → **ejbModule** → **META-INF** → **wsdl**, then right-click **SnoopEjb.wsdl**, select **Web Services** → **Test with Web Services Explorer**, and test the SnoopInfo operation.

Export `SnoopEjbWsClientWeb` in the Dynamic Web Projects folder to a war file and place it in a directory at your convenience. We will need this file again for the `sewsWsDemo` project.

Having now Web service-enabled the SnoopEJB using SOAP over HTTP, we now want to add support to use this Web service via SOAP over JMS.

12.2.2 Enabling SOAP over JMS support

Expand EJB Projects, then right-click **sewsEJB**, and select **Web Services** → **Endpoint Enabler**, as shown in Figure 12-2.

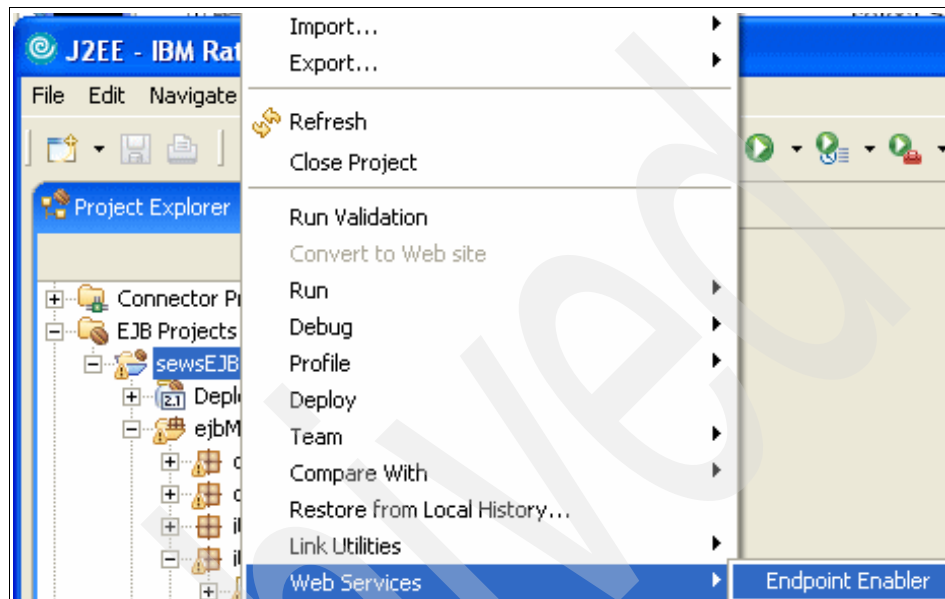


Figure 12-2 Using the Endpoint Enabler to add SOAP over JMS support

Select JMS as the transport type, set JNDI name of the ActivationSpec to `aspec/sewsEJB_ActivationSpec`, and leave other fields as shown in Figure 12-3 on page 413.

In this example we use the new Default messaging provider of WebSphere V6. The ActivationSpec is part of this new feature and is analogous to the listener port approach. The ActivationSpec is used to identify what queue messages are to be taken off to drive the MDB. Further information about the ActivationSpec concept can be found in “Activation specification” on page 103 and at:

http://publib.boulder.ibm.com/infocenter/wasinfo/v6r0/index.jsp?topic=/com.ibm.websphere.nd.doc/info/ae/ae/rj2c_asob.html

Endpoint Enabler Configuration

Select transports

JMS

The name of the HTTP router module:

sewsEJB_HTTPRouter

The context root associated with the HTTP router module:

sewsEJB

The name of the JMS router module:

sewsEJB_JMSRouter

The JMS destination type:

Queue

The JNDI name of the ActivationSpec to be associated with the MDB:

aspec/sewsEJB_ActivationSpec

☐ Use Listener Port

The name of the Listener Input Port to be associated with the MDB:

sewsEJB_ListenerInputName

OK Cancel

Figure 12-3 Selecting JMS as the transport type

Click **OK**. In the window shown in Figure 12-3, you have to choose between using the new ActivationSpec approach or the Listener Port approach; you cannot define both.

We have now completed the process to enable the sample application to support receiving Web service requests for the SnoopEjb via SOAP over JMS. By expanding **EJB Projects** you will notice that a new project called sewsEJB_JMSRouter has been created.

12.2.3 Exploring the new ejb-jar.xml

Before we move on, it is worth looking at what has been generated, in particular within the `sewsEJB_JMSRouter` project. Expand **sewsEJB_JMSRouter** → **ejbModule** → **META-INF** and double-click **ejb-jar.xml**.

Click the **bean** tab, and click **WebServicesJMSRouter**. Scrolling down, you will see information displayed about the activation specification, as shown in Figure 12-4.

The screenshot displays the configuration for the **WebServicesJMSRouter** bean in the WebSphere IDE. The configuration is organized into three main sections:

- Message-Driven Destination:** The destination type is set to `javax.jms.Queue`.
- Activation Configuration:** This section indicates that the following activation configurations are defined for the message-driven bean. A single configuration is shown with the name `destinationType` and the value `javax.jms.Queue`.
- WebSphere Bindings:** This section shows the binding properties for the WebSphere Application Server. The **JCA Adapter** is selected, and the **ActivationSpec JNDI name** is set to `aspect/sewsEJB_Activation`. The **ActivationSpec Authorization Alias** field is currently empty.

Figure 12-4 Display showing information about the *ActivationSpec*

Note that there is a field where you can specify the name of an authorization alias to be associated with the *ActivationSpec*. Note that here it is called an authorization alias, but in WebSphere these definitions are called authentication aliases.

In WebSphere Application Server for z/OS Version 6.01, where global security has been enabled, an authorization alias will be required to allow the consumer to start in the adjunct server, as explained later in “Defining a SIB in WebSphere” on page 420 and “Adjunct server” on page 423.

You either need to specify an authentication alias in the JMS Authentication specification you define in “Define JMS activation specification” on page 428, or you can specify the authentication alias here in the ejb-jar.xml.

However, specifying the value here in the ejb-jar.xml only works if it exactly matches the full name of an authentication alias defined in the server. Since this name consists of the actual node name of the server, it is not recommended to set an authentication alias here in the ejb-jar.xml, because you will then have a node-specific setting in the ejb-jar.xml, which will be of no use if you deploy this ear file into a different server.

Click the **References** tab, and you will see under WebServicesJMSRouter a resource definition called jms/WebServicesReplyQCF. This identifies the QCF definition that will be used by the MDB to write the reply message on. In “Define JMS queue connection factory” on page 426 we show how to define a QCF definition via the Administration GUI.

12.2.4 Update the application.xml

Before exporting the application as an EAR file, we need to update the application.xml file, to add the project that provides the SOAP over JMS support as a module in the application.

Expand **Enterprise Applications** → **sews** → **META-INF**, then double-click **application.xml**. Select the **Module** tab, then under Modules, click **Add**. Add SnoopInfoBeanWsClientWeb and sewsEJB_JMSRouter (in case it is not already included).

Save the change and export the application as an ear file. We called it sews-stage2.ear.

12.2.5 Update sewsWsDemo to invoke SnoopEjb Web service

In the additional material supplied with this book is sewsWsDemo-stage2.zip. This projectInterchange file contains the sewsWsDemo application updated so that it can invoke the SnoopEjb as a Web service.

To enable this, we did the following.

In the sewsWsDemo project we imported the SnoopInfoEjbWsClientWeb.war file you have previously exported in “Web service-enable the SnoopEJB” on page 411, as shown in Figure 12-5 on page 416. In case you did not perform the steps described in the previous sections, you can import the SnoopInfoEjbWsClientWeb.war file as supplied in the additional material for this book. Note that we added this module to the sewsWsDemo EAR project. Doing this means that the application.xml file is updated for the application.

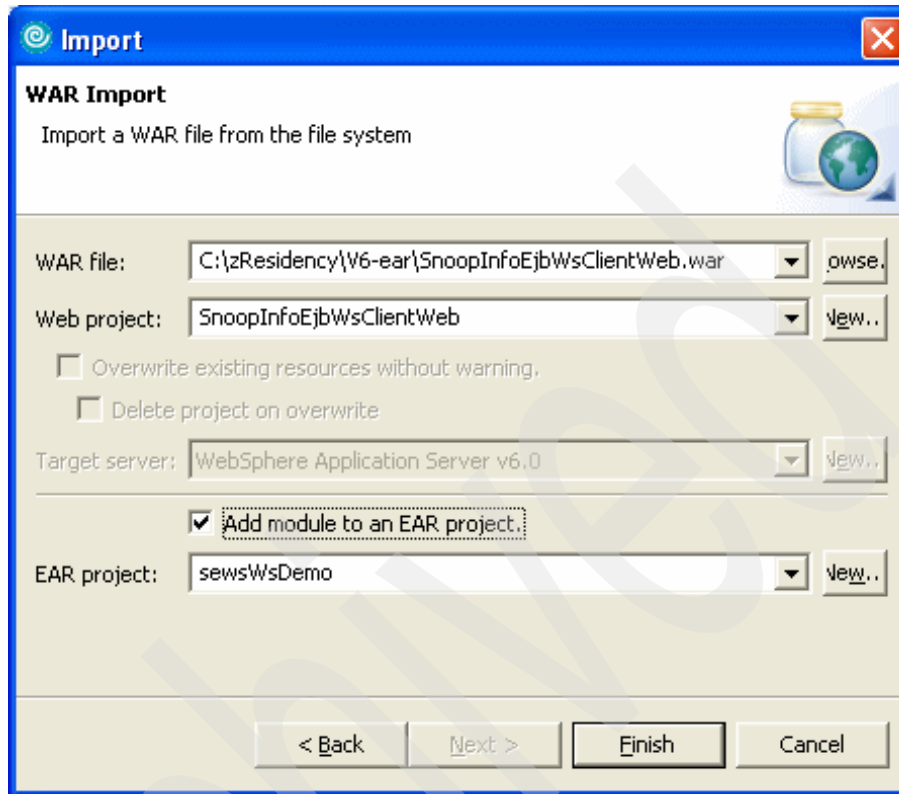


Figure 12-5 Importing the SnoopEjb client code into the sewsWsDemo application

We added SnoopInfoEjbWsClient as a project on the build path for sewsWsDemoWeb.

We then updated the sewsWsDemo.java servlet to have the additional code shown in Figure 12-1.

Example 12-1 Code to call the SnoopEJB Web service

```
// Option c - calls the SnoopEJB Web Service
uidString = "Java principal";
SnoopEjbProxy sip = new SnoopEjbProxy();
trace("got sip: " + sip);
wsHost3 = arg0.getParameter("wsHost3");
sip.setEndpoint(wsHost3);
trace("set endPoint: " + wsHost3);
sd = sip.snoopInfo();
sInfo1 = sd.getS1();
sInfo2 = sd.getS2();
sInfo3 = sd.getS3();
```

```
sInfo4 = sd.getS4();  
arg0.setAttribute("wsHost3", wsHost3);
```

The application was then exported as `sewsWsDemo.ear`.

12.2.6 Update the web.xml

The `web.xml` of the `sewsWsDemoWeb` project needs to be updated with additional logical resource references. A logical resource reference needs to be defined for each JMS resource the application will be referencing.

If this is not done, then later on when testing the application, you would find, as we did, that you would get messages similar to that shown in Example 12-2.

Example 12-2 Warning message about deprecated use of direct JNDI lookups

```
0Trace: 2005/06/27 13:12:25.713 01 t=7C9828 c=UNK key=P8 (0000000A)  
  Description: Log Boss/390 Error  
  from filename: ./bbojtr.cpp  
  at line: 901  
  error message: BB000221W: J2CA0294W: Deprecated usage of direct JNDI lookup  
of resource jms/WebServicesReplyQCF. The following default values are used:  
[Resource-ref settings]  
0 res-auth: 1 (APPLICATION)  
  res-isolation-level: 0 (TRANSACTION_NONE)  
  res-sharing-scope: true (SHAREABLE)  
  loginConfigurationName: null  
  loginConfigProperties: null  
[Other attributes]  
0 res-resolution-control: 999 (undefined)  
  isCMP1_x: false (not CMP1.x)  
  isJMS: false (not JMS)
```

The following link provides further information warning against the use of direct JNDI lookups:

http://publib.boulder.ibm.com/infocenter/wasinfo/v6r0/index.jsp?topic=/com.ibm.websphere.zseries.doc/info/zseries/ae/rdat_jnditips.html

Expand **Dynamic Web Projects** → **sewsWsDemoWeb** → **WebContent** → **WEB-INF** and double-click **web.xml** to open it for editing.

Click **References**, then the **Add** button, then select **Resource Reference** as the type of resource to add.

Add the following four definitions:

| | |
|----------------------------------|--------------------------------------|
| Name | jms/localWasQcf |
| Authentication | container |
| Type | javax.resource.cci.ConnectionFactory |
| JNDI Name | jms/localBus |
| JAAS Authentication Alias | nd6483/sewsdemo |
| Name | jms/snoopEjbInLocalWasQ |
| Authentication | container |
| Type | javax.jms.Queue |
| JNDI Name | jms/snoopEjbInLocalWasQ |
| JAAS Authentication Alias | not applicable |
| Name | jms/snoopEjbInRemoteWasQ |
| Authentication | container |
| Type | javax.jms.Queue |
| JNDI Name | jms/snoopEjbInRemoteWasQ |
| JAAS Authentication Alias | not applicable |
| Name | jms/sews0002InCicsQ |
| Authentication | container |
| Type | javax.jms.Queue |
| JNDI Name | jms/sews0002InCicsQ |
| JAAS Authentication Alias | not applicable |

Note that the type for the QCF definition is set to `javax.resource.cci.ConnectionFactory`, as we will be defining the JMS queue connection factory definition in the new WebSphere Default messaging provider. If you would set it to `javax.jms.QueueConnectionFactory`, which was the value to use if using the previous JMS 1.0 specification, then you would get during deployment of the application the warning message shown in Example 12-3.

Example 12-3 Warning message when QCF resource types do not match

ADMA0139W: Resource Assignment of name jms/sews0002InCicsQcf and type javax.jms.QueueConnectionFactory, with JNDI name jms/ws6483Bus is found within scope of module sewsWsDemoWeb with URI sewsWsDemoWeb.war,WEB-INF/web.xml deployed to target WebSphere:cell=c16483,node=nd6483,server=ws6483, but of wrong resource type JMS. The expected resource type is J2CResourceAdapter.

For the queue definitions, Rational Application Developer Version 6 does not yet support setting the type to reflect the new Default messaging provider feature of WebSphere V6. When you deploy the application you will see a warning message similar to that shown in Example 12-4. It is okay to ignore this warning message.

Example 12-4 Warning message when resource type for Queue does not match

ADMA0139W: Resource Assignment of name jms/snoopEjbInWasQ and type javax.jms.Queue, with JNDI name jms/snoopEjbInWasQ is found within scope of module sewsWsDemoWeb with URI sewsWsDemoWeb.war,WEB-INF/web.xml deployed to target WebSphere:cell=c16483,node=nd6483,server=ws6483, but of wrong resource type JMS. The expected resource type is J2CAdminObject.

12.3 Web service using SOAP over JMS in one cell

The SEWS sample application has now been updated to support the use of SOAP over JMS as the transport mechanism for Web services. We now need to configure the WebSphere server with the appropriate definitions to be able to actually provide the JMS support.

When we used SOAP over HTTP as the underlying infrastructure for Web services, we did not have to set up any additional WebSphere configuration as such, since this was already done when the servers were defined. Now that we want to show the use of SOAP over JMS, additional configuration is required.

We are going to show the use of the new Default messaging provider in WebSphere Application Server for z/OS Version 6.01. In this section we show how to configure a single WebSphere server with both parts of the SEWS sample application installed. We then test that we can invoke the SnoopEjb Web service using SOAP over JMS within a single server.

In “Web service using SOAP over JMS between cells” on page 441 we show how to extend this example to use two servers in different WebSphere cells.

12.3.1 Overview of definitions required

Table 12-2 lists the definitions required.

Table 12-2 Definitions required, for example, using single server

| Definition type | Described in section |
|--------------------------|---|
| Service Intergration Bus | “Defining a SIB in WebSphere” on page 420 |

| Definition type | Described in section |
|---|--|
| Authentication alias | "Define authentication alias" on page 424 |
| Destination queue | "Define queue" on page 424 |
| JMS queue | "Define JMS queue" on page 425 |
| JMS queue Connection factory | "Define JMS queue connection factory" on page 426 |
| JMS activation specification | "Define JMS activation specification" on page 428 |
| JMS queue Connection factory and queue for the sewsWsDemo application | "Definitions for the Web service driver application" on page 430 |

12.3.2 Defining a SIB in WebSphere

A Service Integration Bus (SIB) is an administrative concept that represents all the messaging resources in a WebSphere Application Server for z/OS Version 6.01 environment.

This is a new concept in WebSphere Application Server for z/OS Version 6.01; further information can be found in 4.2.1, "WebSphere Default messaging provider" on page 89; 4.7.2, "Configuring the Default messaging JMS provider" on page 149; and *WebSphere Version 6 Web Services handbook Development and Deployment*, SG24-6461, and at:

http://publib.boulder.ibm.com/infocenter/wasinfo/v6r0/index.jsp?topic=/com.ibm.websphere.express.doc/info/exp/ae/welc6tech_si.html

Using the WebSphere Administration GUI, expand **Service Integration**, and click **Buses**.

Click **New**, then enter `sewsBus` as the name for the SIB, as shown in Figure 12-6 on page 421.

Unselect the **Secure** box under the Security heading. Enabling the bus to be secure in the WebSphere server requires careful planning. Further information is available at:

http://publib.boulder.ibm.com/infocenter/wasinfo/v6r0/topic/com.ibm.websphere.pmc.zseries.doc/tasks/tjr0009_.html

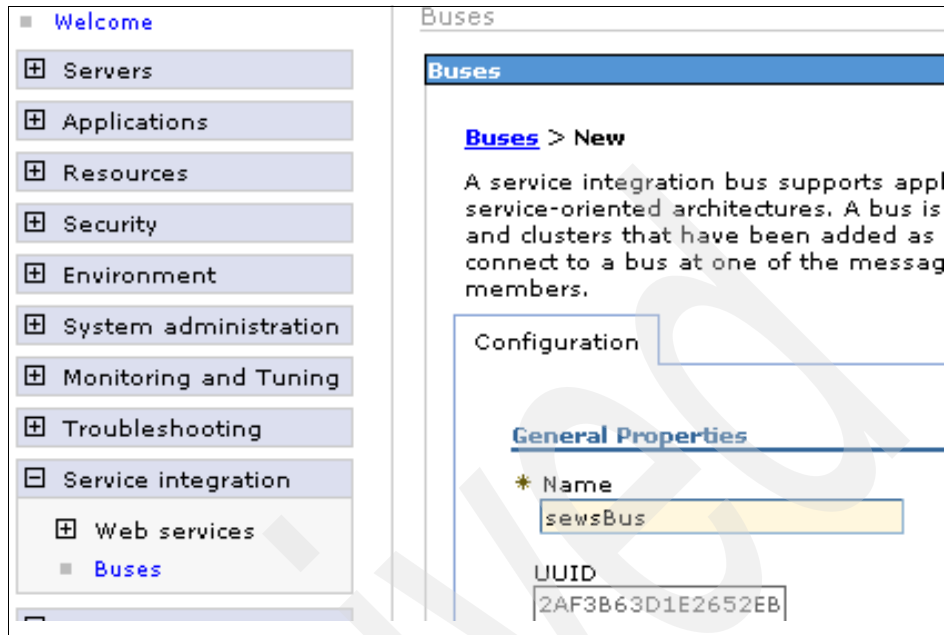


Figure 12-6 Defining the sewsBus

Click **Apply**. The options under Additional Properties and Related Items will now be available. The next step is now to set up a Messaging Engine in the SIB we have created.

Messaging Engine

Ultimately, something is required within WebSphere to handle the processing of messages. This something is what is called the *Messaging Engine (ME)*. The Messaging Engine runs in the WebSphere Application Server and has its own data store with which to store messages. As such, messages become persistent. A Messaging Engine also has the concept of queues associated with it.

A Messaging Engine is automatically created when you add a server to the SIB. Further information about this concept can be found in 4.2.1, “WebSphere Default messaging provider” on page 89, and at:

http://publib.boulder.ibm.com/infocenter/wasinfo/v6r0/index.jsp?topic=/com.ibm.websphere.pmc.zseries.doc/concepts/cjj0020_.html

In the Admin GUI, where we have just defined the sewsBus SIB, click **Bus Members**, then click **Add**. The Admin GUI will then display a page where you select the server you want to add to the SIB. As we are only using a single server setup, there is only one server that can be picked, as shown in Figure 12-7.

Buses

Add a new bus member

Add a server or server cluster as a new member of the bus.

→ **Step 1: Select server or cluster**

Step 2: Confirm the addition of a new bus member

Select server or cluster

Choose the server or cluster to add to the bus

☒ **Server**

Server
nd6481:ws6481

Data store

☒ Default

Figure 12-7 Adding a server to the sewsBus SIB

Click **Step 2**, then click **Finish**.

If you now go back to the display shown in Figure 12-6 on page 421, and click **Messaging Engines**, you will see that a Messaging Engine has been defined, as shown in Figure 12-8.

Buses > sewsBus > Messaging engines

A messaging engine is a component, running inside a server, that manages messaging resources for a bus member. Applications are connected to a messaging engine when accessing a service integration bus.

☒ Preferences

Stop mode: Immediate

| Select | Name | Description | Status |
|--------------------------|-----------------------|-------------|--------|
| <input type="checkbox"/> | nd6481.ws6481-sewsBus | | |

Total 1

Figure 12-8 Messaging Engine defined for the server

Save the changes.

Adjunct server

Before continuing to define other resources, restart the WebSphere server.

If this is the first Messaging Engine you have defined for the server, you will see a new STC started along with the Control Region and Servant Region when you restart the server. This new STC is referred to as the *Control Region Adjunct (CRA)*. The name of this STC will be the name of the Control Region with an “A” appended.

HTTP and HTTPS ports will not be opened by the Control Region until both the Servant and Adjunct STCs have completed startup. In the Adjunct STC you will see messages similar to those shown in Example 12-5, which identify the STC as being the adjunct STC.

Example 12-5 Message identifying the Adjunct server

```
BB000288I WEBSHERE FOR Z/OS ADJUNCT PROCESS 094
CL6481/ND6481/CLU6481/WS6481 IS STARTING.
BB000293I WEBSHERE FOR Z/OS ADJUNCT PROCESS cl6481/nd6481/ws6481 IS
STARTING.

BB000222I: WSVR0001I: Server ADJUNCT PROCESS ws6481 open for
e-business
BB000291I INITIALIZATION COMPLETE FOR WEBSHERE FOR Z/OS ADJUNCT
PROCESS WS6481.
BB000292I INITIALIZATION COMPLETE FOR WEBSHERE FOR Z/OS ADJUNCT
PROCESS CL6481/ND6481/CLU6481/WS6481.
```

In the Adjunct STC SYSPRINT, you should see a message showing that the SIB you defined has been successfully started, similar to this:

```
BB000222I: [sewsBus:nd6481.ws6481-sewsBus] CWSID0016I: Messaging engine
nd6481.ws6481-sewsBus is in state Started.
```

In the Administration GUI, the Messaging Engine should now show a running status indicator, as shown in Figure 12-9 on page 424.



Figure 12-9 Display showing Messaging Engine is running

12.3.3 Define authentication alias

In the Administration GUI, expand **Security**, and click **Global Security** → **JAAS Configuration** → **J2C Authentication data** → **New**. Set the name as **sews-alias**, and enter a valid user ID and password. Click **Apply** and save the change.

12.3.4 Define queue

A queue is required, as this will be the place that messages for the SnoopEjb Web service request will be sent for processing. When a message arrives on this queue, an MDB will be driven to process the message. This MDB in turn will invoke the method on the SnoopEjb, which has been Web service-enabled.

In the Default messaging provider of WebSphere Application Server for z/OS Version 6.01, queues are defined under *destinations*.

Further information can be found at:

http://publib.boulder.ibm.com/infocenter/wasinfo/v6r0/index.jsp?topic=/com.ibm.websphere.pmc.zseries.doc/concepts/cjo0001_.html

At the above link a Bus destination is described as:

A Bus destination is a virtual location within a Service Integration Bus, to which applications attach as producers, consumers, or both to exchange messages.

To define the queue, expand **Service Integration**, and click **Buses** → **Destinations** → **new**. In the panel displayed select **Queue** as the Destination type, and click **Next**.

Defining the queue is a three-step process. The first step is shown in Figure 12-10, where the queue name or identifier is set.

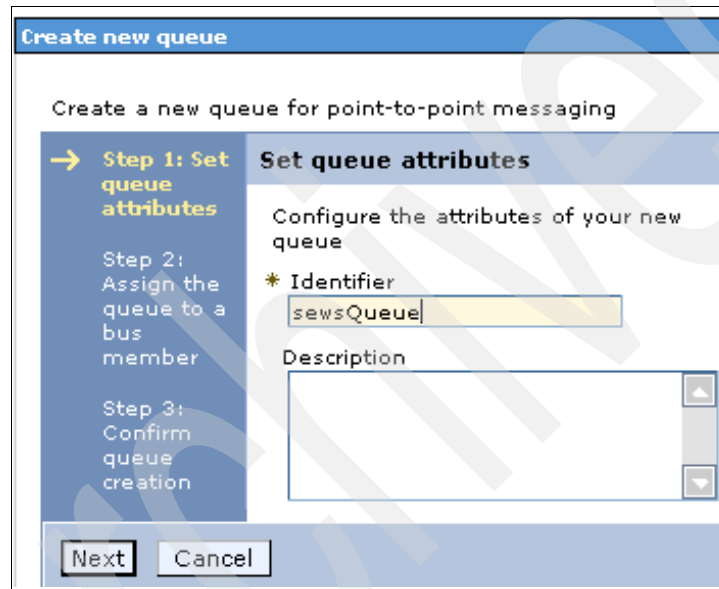
The screenshot shows a 'Create new queue' dialog box with a blue header. Below the header, it says 'Create a new queue for point-to-point messaging'. On the left, there is a vertical list of three steps: 'Step 1: Set queue attributes' (highlighted with a yellow arrow), 'Step 2: Assign the queue to a bus member', and 'Step 3: Confirm queue creation'. The main area is titled 'Set queue attributes' and contains the text 'Configure the attributes of your new queue'. There are two input fields: '* Identifier' with the text 'sewsQueue' and 'Description' which is an empty text area. At the bottom, there are 'Next' and 'Cancel' buttons.

Figure 12-10 Setting the queue identifier

Click **Next**, and another panel is displayed. This is where you identify the member on the SIB that will process messages on this queue. In a cell such as ours with only one server, there is only the one server to select. Click **Next** and then **Finish**. Save the change.

12.3.5 Define JMS queue

We now need a JMS queue definition that will point to the queue we have just defined.

To define the JMS queue, expand **Resources** → **JMS Providers**, click **Default Messaging**, click **JMS Queue**, and click **New**.

The filled out display is shown in Figure 12-11.

General Properties

Administration

- * Scope: cells:cl6481:nodes:nd6481
- * Name: sewsJmsQueue
- * JNDI name: jms/sewsQueue
- Description: [Empty text area]

Connection

- Bus name: sewsBus
- * Queue name: sewsQueue
- Delivery mode: Application

Figure 12-11 Defining the JMS queue

Click **Apply**, then save the changes.

12.3.6 Define JMS queue connection factory

We need a JMS queue connection factory definition. This definition is used by applications to create a connection to the JMS provider where target queues are stored.

Further information about JMS queue connection factory can be found at:

http://publib.boulder.ibm.com/infocenter/wasinfo/v6r0/index.jsp?topic=/com.ibm.websphere.pmc.zseries.doc/sibjmsresources/SIBJMSQueueConnectionFactory_CollectionForm.html

At the above link JMS Queue Connection Factory is defined as:

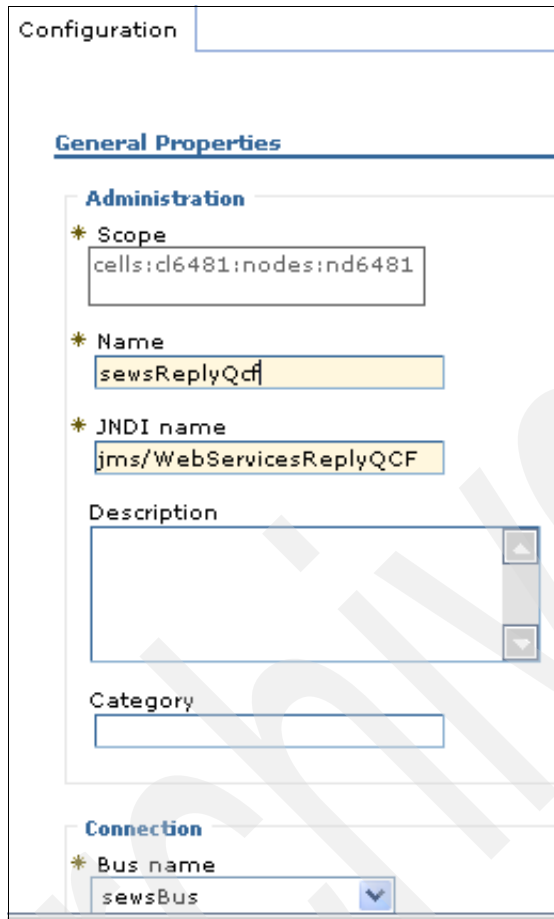
A JMS Queue Connection Factory is used to create connections to the associated JMS provider of JMS queues, for point-to-point messaging. Use Queue Connection factory administrative objects to manage JMS Queue Connection factories for the Default messaging provider.

To define the definition, expand **Resources** → **JMS Providers**, click **Default Messaging**, click **JMS Queue Connection factory**, and click **New**.

Part of the display where you define the definition is shown in Figure 12-12 on page 428.

Set the name of the definition to `sewsReplyQcf`, the JNDI name to `jms/WebServicesReplyQCF`, select the `sewsBus` as the Bus name to use, and select the `sews-alias` entry as the Component-managed authentication alias.

This JNDI name was the value we set in the References tab of the `ejb-jar.xml` in the `sewsEjb_JMSRouter` project, as explained in “Exploring the new `ejb-jar.xml`” on page 414.



Configuration

General Properties

Administration

* Scope
cells:cl6481:nodes:nd6481

* Name
sewsReplyQcf

* JNDI name
jms/WebServicesReplyQCF

Description

Category

Connection

* Bus name
sewsBus

Figure 12-12 Defining the JMS queue connection factory

Click **Apply** and save the change.

12.3.7 Define JMS activation specification

Information about JMS activation specifications can be found at:

http://publib.boulder.ibm.com/infocenter/wasinfo/v6r0/index.jsp?topic=/com.ibm.websphere.pmc.zseries.doc/sibjmsresources/SIBJMSActivationSpec_CollectionForm.html

At the above link a JMS activation specification is defined as:

A JMS activation specification is associated with one or more Message-Driven Beans and provides the configuration necessary for them to receive messages.

In “Exploring the new ejb-jar.xml” on page 414 we discussed how the ejb-jar.xml had a JNDI value to reference a JMS activation specification; it had a value of `aspec/sewsEJB_ActivationSpec`. When we create the definition this will be the value we set the JNDI name to.

This will tie together the application that will be providing the Web service via SOAP over JMS, and the WebSphere infrastructure that provides the messaging capability.

To define the JMS activation specification, expand **Resources** → **JMS Providers**, click **Default Messaging**, and then under activation specifications click **JMS Activation Specification**, then **New**.

Set the name to `sews-jmsActSpec`, the JNDI name to `aspec/sewsEJB_ActivationSpec`, the destination JNDI name as `jms/sewsQueue`, and select **sewsBus** as the bus name, as shown in Figure 12-13 on page 430.

Welcome

- Servers
- Applications
- Resources
 - JMS Providers
 - Default messaging
 - WebSphere MQ
 - Generic
 - V5 default messaging
 - JDBC Providers
 - Resource Adapters
 - Asynchronous
 - Schedulers
 - Cache instances
 - Object pool

General Properties

Administration

- * Scope: cells:cl6481:nodes:nd6481
- * Name: sews-jmsActSpec
- * JNDI name: jmspec/sewsEJB_ActivationSpec

Destination

- * Destination type: Queue
- * Destination JNDI name: jms/sewsQueue
- Message selector:
- Bus name: sewsBus

Figure 12-13 Settings for the JMS activation specification

Also set the authentication alias to sews-alias. Failure to do this will result in the error messages discussed in “Preventing deprecated JNDI lookup” on page 436.

Click **Apply** and save the change.

12.3.8 Definitions for the Web service driver application

The sewsWsDemo application is used to test the Web service available in the sews application. To use it to demonstrate Web services using SOAP over JMS, we need to define additional JMS definitions, as explained in the following sections.

JMS queue connection factory

Using the same process described in “Define JMS queue connection factory” on page 426, we now define a JMS queue connection factory to be used by the `sewsWsDemo` application.

When defining the definition, set the name to `localWasBus`, the JNDI name to `jms/localWasBus`, the bus name to `sewsBus`, and the Component-managed authentication alias to `sewsDemo-alias`. Save the change.

JMS queue

Using the same process as described in “Define JMS queue” on page 425, we now define a JMS queue definition that will point to the `sewsQueue` definition we defined in “Define JMS queue” on page 447.

When defining the definition, set the name to `snoopEjblnWasQ`, the JNDI name to `jms/snoopEjblnWasQ`, the bus name to `sewsBus`, and the queue name to `sewsQueue`. Save the change.

12.3.9 Testing in one server

Install the `sews.ear` and `sewsWsDemo.ear` files into the WebSphere server where the above definitions have been defined.

As a number of new definitions have been added to the server, it will need to be restarted to pick up the changes.

Verify consumer started

In the Adjunct STC, once startup has completed, you should see a message similar to that shown in Example 12-6. This message indicates that a consumer to process messages that arrive on the `sewsQueue` is active. If you do not get this message, then further investigation is required to determine why.

Example 12-6 Successful startup of MDB

```
Trace: 2005/06/22 23:31:48.329 01 t=7C7690 c=UNK key=P8 (0000000A)
Description: Log Boss/390 Error
from filename: ./bborjtr.cpp
at line: 901
error message: BB000222I: [sewsBus:nd6481.ws6481-sewsBus] CWSIV0764I: A
consumer has been created for a message-driven bean against destination
sewsQueue on bus sewsBus following the activation of messaging engine
nd6481.ws6481-sewsBus.
```

Invoke the SewsWsDemo servlet

Start the servlet that lets you test the Web services by entering a URL as follows. Make sure that you substitute the TCP/IP address and port number with the ones you are using.

`http://9.12.4.38:9080/sewsWsDemoWeb/SewsWsDemo`

The result will be similar to that shown in Figure 12-14.



Figure 12-14 Initial display after invoking the SewsWsDemo servlet

Test SnoopEjb using SOAP over HTTP

The first test is to invoke the SnoopEjb Web service using SOAP over HTTP. Select the **WebSphere SnoopInfo Ejb** option, update the localhost:9080 part of the URL to reflect the TCP/IP address and port number of the server the applications are running in, and click **Invoke selected Web Service**.

The result should be similar to that shown in Figure 12-15 on page 433.

Information returned from Web Service call

| | |
|-------------------------------|--|
| Java principal | Subject: Principal: wtsc48.itso.ibm.com/WSGUEST Public Credential: com.ibm.ws.security.auth.WSCredentialImpl@7433eec0 [com.ibm.ws.security.auth.zOSWSCredentialImpl@74ca6ec0; [com.ibm.ws.security.zos.PlatformCredential]] |
| Host | wtsc48.itso.ibm.com |
| Current Time | Thu Jun 23 13:33:14 GMT+00:00 2005 |
| Id of caller (if EJB invoked) | WSGUEST |

Figure 12-15 Result after calling SnoopEjb Web service using SOAP over HTTP

Test SnoopEjb using SOAP over JMS

Now finally the moment of truth—test the invocation of the SnoopEjb Web service using SOAP over JMS.

And this is finally where we see the beauty of Web services, as to have the Web service done via SOAP over JMS, all that is needed in the client program is a different end point value. Instead of using a value like this:

```
http://9.12.4.38:9080/SnoopEjbWsWeb/services/SnoopEjb
```

we simply enter as the URL the value:

```
jms:/queue?destination=java:comp/env/jms/snoopEjbInLocalWasQ&connectionFactory=java:comp/env/jms/localWasQcf&targetService=SnoopEjb
```

This value will be set as the end point in the SewsWsDemo.java program, before it does the Web service call. Example 12-1 on page 416 shows the Java code that performs the Web service call on the SnoopEjb.

Enter the URL above to invoke the SnoopEjb Web service using SOAP over JMS, and you should get a result similar to that shown in Figure 12-16 on page 434.

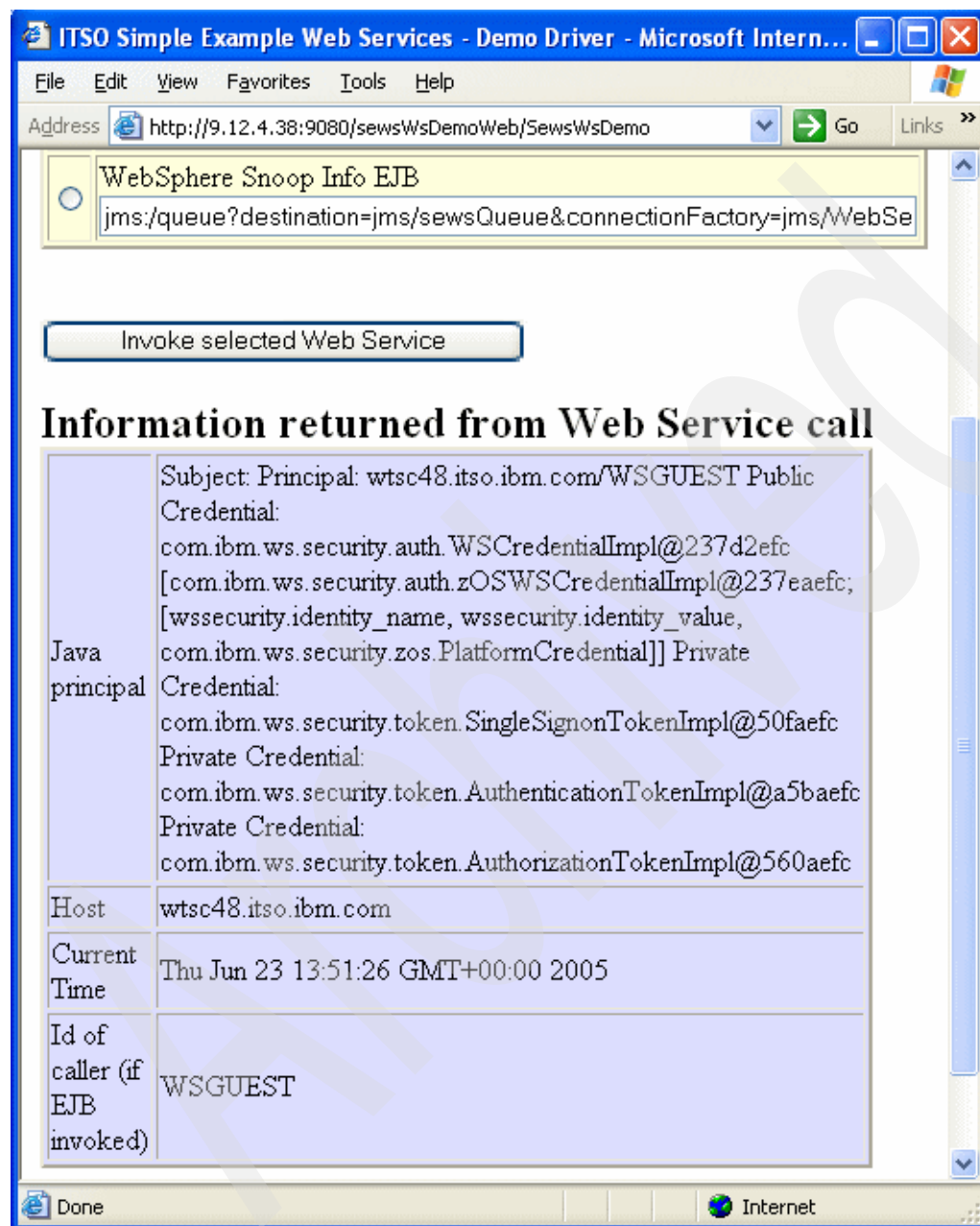


Figure 12-16 Result from calling SnoopEjb Web service using SOAP over JMS

Breakdown of message flow

The following explains the message flow of how the Web service using SOAP over JMS has occurred:

1. The SewsWsDemo servlet calls the SnoopEjbProxy class, with the URL set to indicate to use SOAP over JMS.
2. The SnoopEjbProxy class drives the classes in WebSphere that handle Web service processing.
3. A dynamic temporary reply queue name is generated and set in the message header.
4. Using the JNDI name for the JMS queue connection factory passed in on the URL, a connection to the SIB is obtained.
5. Using the JNDI name of the JMS queue passed in via the URL, the SOAP request is written as a message to the sewsQueue queue.
6. The consumer monitoring the sewsQueue detects a message has arrived and starts a Message-Driven Bean (MDB).
7. The MDB creates the SnoopEJB EJB and calls the SnoopInfo method.
8. The SnoopInfo method runs, and returns a SnoopData object to the MDB.
9. The MDB writes the SnoopData object as a message to the specified reply queue at the specified reply QCF name.
10. WebSphere Web service classes read the reply message.
11. The reply data is put into a SnoopData object, which is then passed back to the SewsWsDemo servlet and the result displayed.

Trace sample

We did a trace using `com.ibm.ws.sib.*=all=enabled` and examined the trace produced.

Example 12-7 shows a sample trace record produced when we ran the Web service using SOAP over JMS. In particular, this trace record shows the message to be written to the queue and you can see the name of the temporary dynamic reply queue.

Example 12-7 Sample trace record showing message to be written to queue

```
Trace: 2005/06/23 13:51:24.925 01 t=7CC0C0 c=9.2 key=P8 (13007002)
  ThreadId: 0000002b
  FunctionName: com.ibm.ws.sib.api.jms.impl.JmsMsgProducerImpl
  SourceId: com.ibm.ws.sib.api.jms.impl.JmsMsgProducerImpl
  Category: FINEST
  ExtendedMessage: [:] message :
  JMSMessage class: jms_bytes
```

```

JMSType:          null
JMSDeliveryMode:  2
JMSExpiration:    0
JMSPriority:      4
JMSMessageID:     null
JMSTimestamp:     0
JMSCorrelationID: null
JMSDestination:   null
JMSReplyTo:
queue://_Q_F7FCADA4C24EBC7B00000000002DC6C7?busName=sewsBus
JMSRedelivered:   false
  transportVersion: 1
  targetService: SnoopEjb
  endpointURL:
jms:/queue?destination=jms/sewsQueue&connectionFactory=jms/WebServicesReplyQCF&
targetService=SnoopEjb
  contentType: text/xml; charset=utf-8

3c736f6170656e763a456e76656c6f706520786d6c6e733a736f6170656e763d22687474703a2f2
f736368656d61732e786d6c736f61702e6f72672f736f61702f656e76656c6f70652f2220786d6c
6e733a736f6170656e633d22687474703a2f2f736368656d61732e786d6c736f61702e6f72672f7
36f61702f656e636f64696e672f2220786d6c6e733a7873643d22687474703a2f2f7777772e7733
2e6f72672f323030312f584d4c536368656d612220786d6c6e733a7873693d22687474703a2f2f7
77772e77332e6f72672f323030312f584d4c536368656d612d696e7374616e6365223e3c736f61
70656e763a4865616465722f3e3c736f6170656e763a426f64793e3c703137323a736e6f6f70496
e666f20786d6c6e733a703137323d22687474703a2f2f656a622e736577732e6974736f222f3e3c
2f736f61 70656e763a426f64793e3c2f736f6170656e763a456e76656c6f70653e

```

12.3.10 Preventing deprecated JNDI lookup

Instead of using the URL:

```
jms:/queue?destination=java:comp/env/jms/snoopEjbInLocalWasQ&connectionFactory=
java:comp/env/jms/localWasQcf&targetService=SnoopEjb
```

you could use the actual JNDI values of the defined JMS queue connection factory and queue definitions. To do this the URL would be:

```
jms:/queue?destination=jms/snoopEjbInLocalWasQ&connectionFactory=jms/localWasBu
s&targetService=SnoopEjb
```

However, using this approach results in the warning message shown in Example 12-2 on page 417 being issued in the server.

It is recommended not to use actual JNDI names of JMS resources in the URL.

12.3.11 Security errors without authentication alias

During our testing, we did at one stage have security enabled for the bus in the WebSphere Application Server. Some of the security errors we encountered with security enabled are outlined here, as they may be of assistance should you enable security in the bus.

No alias on JMS activation specification

The following error situation can occur if you do not set the authentication alias in the JMS activation specification in the application itself or in the Additional property called *Provide listener bindings*. For Message-Driven Beans, the consumer will not start in the Adjunct server and you will get error messages similar to those shown in Example 12-8.

Example 12-8 Consumer failing to start in the adjunct server

```
Trace: 2005/06/22 23:02:19.350 01 t=7C7690 c=UNK key=P8 (0000000A)
  Description: Log Boss/390 Error
    from filename: ./bborjtr.cpp
    at line: 901
    error message: BB000220E: [sewsBus:nd6481.ws6481-sewsBus] CWSII0050E: The
Platform Messaging Component can not authenticate the user ID null.
com.ibm.ws.sib.utils.ras.SibMessage com.ibm.ws.sib.utils.ras.SibMessage
Trace: 2005/06/22 23:02:19.357 01 t=7C7690 c=UNK key=P8 (13007002)
  ThreadId: 00000039
  FunctionName: com.ibm.ws.sib.utils.ras.SibMessage
  SourceId: com.ibm.ws.sib.utils.ras.SibMessage
  Category: SEVERE
  ExtendedMessage: BB000220E: [sewsBus:nd6481.ws6481-sewsBus] CWSIV0555E: The
exception javax.resource.ResourceException: CWSIV0954E
: The authentication exception
com.ibm.wsspi.sib.core.exception.SIAuthenticationException:
USER_NOT_AUTHORIZED_ERROR_CWSIP0303 was thrown while attempting to create a
connection on factory com.ibm.ws.sib.processor.impl.MessageProcessor@1ae6eba3.
was thrown when processing the startup of messaging engine
nd6481.ws6481-sewsBus on bus sewsBus for endpoint activation
[com.ibm.ws.sib.ra.inbound.impl.SibRaStaticDestinationEndpointActivation@5a1aeb
70 <active=true> <connections={}>
<messageEndpointFactory=com.ibm.ejs.container.MessageEndpointFactoryImpl@cdd882
a9>
<endpointConfiguration=[com.ibm.ws.sib.api.jmsra.impl.JmsJcaActivationSpecImpl$
JmsJcaEndpointConfigurationImpl@1574120304
<JmsJcaActivationSpecImpl.this=[com.ibm.ws.sib.api.jmsra.impl.JmsJcaActivationS
pecImpl@1107651440 <userName=null> <password=null> <xaRecoveryAlias=sews-alias>
<destination=queue://sewsQueue?busName=sewsBus> <durableSubscriptionHome=>
<destinationType=javax.jms.Queue> <messageSelector=>
<acknowledgeMode=Auto-acknowledge> <subscriptionDurability=NonDurable>
```

```
<shareDurableSubscriptions=InCluster> <clientId=> <subscriptionName=>
<maxBatchSize=1> <maxConcurrency=10> <busName=sewsBus> <shareDataSourceWithC
MP=false> <targetTransportChain=null> <readAhead=Default>]>]>
<endpointInvoker=com.ibm.ws.sib.api.jmsra.impl.JmsJcaEndpointInvokerIm
pl@5a6ceb70> <workManager=com.ibm.ejs.j2c.work.WorkManagerImpl@5bb42b70>
<endpointMethodTransactional=true> <remoteConnection=null>
<remoteDestination=false> <timer=java.util.Timer@58576b70>].
```

No alias on JMS queue connection factory

The following error situation can occur if you do not specify an authentication alias on the JMS queue connection factory.

When the SewsWsDemo servlet runs the SnoopEjb Web service via SOAP over JMS, this will try to get a connection to the SIB. If you do not specify an authentication alias on the JMS queue connection factory this will fail and you will see an error message similar to that shown in Example 12-9.

Example 12-9 Security exception when no alias defined on the queue connection

```
Trace: 2005/06/23 16:02:58.202 01 t=7C9828 c=1.1 key=P8 (13007002)
  ThreadId: 00000030
  FunctionName: com.ibm.ejs.j2c.poolmanager.FreePool
  SourceId: com.ibm.ejs.j2c.poolmanager.FreePool
  Category: SEVERE
  ExtendedMessage: BB000220E: J2CA0046E: Method
createManagedConnectionWithMCWrapper caught an exception during creation of the
ManagedConnection for resource jms/WebServicesReplyQCF, throwing
ResourceAllocationException. Original exception:
javax.resource.ResourceException: CWSJR1028E: An internal error has occurred.
The exception com.ibm.wsspi.sib.core.exception.SIAuthenticationException: C
WSIT0010E: A client request for messaging engine nd6481.ws6481-sewsBus in bus
sewsBus failed with reason: CWSIT0016E: The user ID null failed authentication
in bus sewsBus. was received in method createManagedConnection.
    at
com.ibm.ws.sib.api.jmsra.impl.JmsJcaManagedConnectionFactoryImpl.createManagedC
onnection(JmsJcaManagedConnectionFactoryImpl.java
:420)
    at
com.ibm.ejs.j2c.poolmanager.FreePool.createManagedConnectionWithMCWrapper(FreeP
ool.java:1502)
    at
com.ibm.ejs.j2c.poolmanager.FreePool.createOrWaitForConnection(FreePool.java:12
94)
    at com.ibm.ejs.j2c.poolmanager.PoolManager.reserve(PoolManager.java:2023)
    at
com.ibm.ejs.j2c.ConnectionManager.allocateMCWrapper(ConnectionManager.java:800)
```

```

        at
com.ibm.ejs.j2c.ConnectionManager.allocateConnection(ConnectionManager.java:576
)
        at
com.ibm.ws.sib.api.jmsra.impl.JmsJcaConnectionFactoryImpl.createConnection(JmsJ
caConnectionFactoryImpl.java:241)
        at
com.ibm.ws.sib.api.jms.impl.JmsManagedConnectionFactoryImpl.createConnection(Jm
sManagedConnectionFactoryImpl.java:197)
        at
com.ibm.ws.sib.api.jms.impl.JmsManagedQueueConnectionFactoryImpl.createQueueCon
nection(JmsManagedQueueConnectionFactoryImpl.java
:146)
        at
com.ibm.ws.sib.api.jms.impl.JmsManagedQueueConnectionFactoryImpl.createQueueCon
nection(JmsManagedQueueConnectionFactoryImpl.java
:110)
        at
com.ibm.ws.webservices.engine.transport.jms.JMSSender.sendMessage(JMSSender.jav
a:735)
        at
com.ibm.ws.webservices.engine.transport.jms.JMSSender.invoke(JMSSender.java:204
)
        at
com.ibm.ws.webservices.engine.PivotHandlerWrapper.invoke(PivotHandlerWrapper.ja
va:218)
        at
com.ibm.ws.webservices.engine.WebServicesEngine.invoke(WebServicesEngine.java:2
74)
        at
com.ibm.ws.webservices.engine.client.Connection.invokeEngine(Connection.java:73
0)
        at
com.ibm.ws.webservices.engine.client.Connection.invoke(Connection.java:651)
        at
com.ibm.ws.webservices.engine.client.Connection.invoke(Connection.java:612)
        at
com.ibm.ws.webservices.engine.client.Connection.invoke(Connection.java:447)
        at com.ibm.ws.webservices.engine.client.Stub$Invoke.invoke(Stub.java:732)
        at
itso.sews.ejb.SnoopEjbSoapBindingStub.snoopInfo(SnoopEjbSoapBindingStub.java:92
)
        at itso.sews.ejb.SnoopEjbProxy.snoopInfo(SnoopEjbProxy.java:64)
        at itso.sewsWsDemo.SewsWsDemo.processRequest(SewsWsDemo.java:102)
        at itso.sewsWsDemo.SewsWsDemo.doPost(SewsWsDemo.java:43)
        at javax.servlet.http.HttpServlet.service(HttpServlet.java:763)
        at javax.servlet.http.HttpServlet.service(HttpServlet.java:856)

```

```

        at
com.ibm.ws.webcontainer.servlet.ServletWrapper.service(ServletWrapper.java:1216
)
        at
com.ibm.ws.webcontainer.servlet.ServletWrapper.handleRequest(ServletWrapper.jav
a:630)
        at com.ibm.ws.webcontainer.webapp.WebApp.handleRequest(WebApp.java:2872)
        at com.ibm.ws.webcontainer.webapp.WebGroup.handleRequest(WebGroup.java:220)
        at com.ibm.ws.webcontainer.VirtualHost.handleRequest(VirtualHost.java:204)
        at
com.ibm.ws.webcontainer.WebContainer.handleRequest(WebContainer.java:1779)
        at
com.ibm.ws.webcontainer.channel.WCChannelLink.ready(WCChannelLink.java:77)
        at
com.ibm.ws.http.channel.inbound.impl.HttpInboundLink.handleDiscrimination(HttpI
nboundLink.java:466)
        at
com.ibm.ws.http.channel.inbound.impl.HttpInboundLink.handleNewInformation(HttpI
nboundLink.java:405)
        at
com.ibm.ws.http.channel.inbound.impl.HttpInboundLink.ready(HttpInboundLink.java
:283)
        at com.ibm.xmem.channel.ws390.XMemConnLink.ready(XMemConnLink.java:574)
        at com.ibm.xmem.ws390.XMemSRBridge.httpinvoke(XMemSRBridge.java:105)
        at com.ibm.ws390.orb.ServerRegionBridge.httpinvoke(Unknown Source)
        at com.ibm.ws390.orb.ORBE

```

12.3.12 Checking queue status

You can use the WebSphere administration GUI to check the status of a queue in the Default messaging provider of WebSphere. For example, to check on the `sewsQueue`, expand **Service Integration**, and click **Buses** → **sewsBus** → **Messaging engines** → **sewsBusQueue** → **Runtime**.

The display will be similar to that shown in Figure 12-17 on page 441. If there are messages on the queue, they can be viewed by clicking **Messages**.

[Buses](#) > [sewsBus](#) > [Messaging engines](#) > [nd6481.ws6481-sewsBus](#) > [Queue points](#) > [sewsQueue@nd6481.ws6481-sewsBus](#)

The message point for a queue, for point-to-point messaging.

Configuration Runtime

General Properties

Identifier
sewsQueue

Run-time ID
54F50B5FCF1DC2F54A8E836E_QUEUE_39

High message threshold
50000

☒ Send allowed

Current message depth
0

Additional Properties

■ [Messages](#)

Figure 12-17 Display showing runtime status of a queue

12.4 Web service using SOAP over JMS between cells

In “Web service using SOAP over JMS in one cell” on page 419 we described how to configure a single server so we could invoke the sample Web service using SOAP over JMS.

We now show a Web service call, using SOAP over JMS, being done between two WebSphere cells. We first show how to connect the SIBs in the servers of two cells to each other.

12.4.1 WebSphere servers used example

With reference to Figure 12-1 on page 409, our aim is to run the sewsWsDemo application in the ws6483 server and have it invoke the SnoopEjb Web service in the ws6481 server. As both these servers are base servers, they are in effect their own cells.

We show what definitions need to be added to the ws6483 and ws6481 servers to allow for the SIB in each server to connect to the other.

12.4.2 Second server configuration

This section describes the definitions added to the second server, ws6483. Note that this server is a standalone server, thus is its own cell, and not a member of the cell that contains the ws6481 server.

The definitions required are listed in Table 12-3.

Table 12-3 Definitions required for cell to cell example

| Definition type | Described in section |
|---|---|
| Authentication alias | "Define authentication alias" on page 442 |
| Service Integration bus | "Define SIB in ws6483" on page 442 |
| Define sewsBus as a foreign bus to ws6483 | "Define foreign bus in ws6483" on page 443 |
| Define ws6483Bus as a foreign bus to ws6481 | "Define foreign bus in ws6481" on page 444 |
| Destination queue | "Define sewsQueue in ws6483" on page 446 |
| JMS queue connection factory | "Define JMS queue connection factory" on page 447 |
| JMS queue | "Define JMS queue" on page 447 |

12.4.3 Define authentication alias

In the second server, ws6483, define an authentication alias. This will be needed, as the ws6481 server has global security enabled. The ws6483 server must authenticate ws6481 when establishing an inter-SIB link.

Using the Administration GUI, expand **Security**, and click **Global Security** → **JAAS Configuration** → **J2C Authentication data** → **New**. Set the name as sewsDemo, and enter a valid user ID and password. Click **Apply** and save the change.

12.4.4 Define SIB in ws6483

Define in the ws6483 server an SIB called ws6483Bus using the approach described in "Defining a SIB in WebSphere" on page 420.

12.4.5 Define foreign bus in ws6483

To allow the messages containing the SOAP requests to flow between the ws6481 and ws6483 servers, we need to interconnect the SIBs in each server with each other.

This is done by defining the SIB in each server to each other as a foreign bus. Further information about the foreign bus concept can be found at:

http://publib.boulder.ibm.com/infocenter/wasinfo/v6r0/index.jsp?topic=/com.ibm.websphere.pmc.zseries.doc/concepts/cjj0030_.html

To define the foreign bus, expand **Service integration**, select **Buses**, select **ws6483 bus**, under Additional properties click **Foreign buses**, and then click **New**.

Set the name of the foreign bus to `sewsBus`, as shown in Figure 12-18.

Create the routing definition for the foreign bus, to define the routing type and properties.

→ **Step 1: Foreign bus properties**

Step 2: Routing definition type

Step 3: Routing definition properties

Step 4: Summary

Foreign bus properties

* Name
sewsBus

Description

☒ Send allowed

Figure 12-18 Defining a foreign bus in the ws6483 server

Note that the value you set in the Name field must be the actual name of the bus in the remote server.

Click **Next**.

In Step 2, select **Direct, Service Integration Bus link** and click **Next**.

In Step 3, leave the user IDs fields blank and click **Next**.

Click **Finish** for Step 4 and the foreign bus definition is defined.

Define Service Integration Bus link in ws6483

We now need to define a Service Integration Bus link. This will be the definition that actually defines the location of the remote bus.

To define this definition, expand **Service integration**, select **Buses**, select **ws6483Bus**, click **Messaging Engines**, click the **Messaging Engine**, and under Additional properties, click **Service Integration Bus link**. Click **New**.

Set the name to ws6483BusTows6481BusLink. Note that later on, when we are defining the ws6483 bus as a foreign bus in the ws6481 server, the name we use there must be the same as the value we use here.

Set the foreign bus name to sewsBus.

Set the Remote Messaging Engine name to the actual name of the Messaging Engine in the other server. In our case this was nd6481.ws6481-sewsBus.

Set the Bootstrap endpoints to a value of <tcpipAddress:port>, where the TCPIP address is where the other server is located, and the port is the value of the SIB_ENDPOINT_ADDRESS port defined in the other server. In our case this was 9.12.4.38:7276.

Set the authentication alias to sewsDemo.

Click **Apply** and save the changes.

12.4.6 Define foreign bus in ws6481

We now need to define the ws6483 bus to the ws6481 server. This is the same process as described in “Define foreign bus in ws6483” on page 443.

Use the same process except set the name for the new definition to ws6483Bus.

Define Service Integration Bus link in ws6481

Now define a Service Integration Bus link to connect the bus in the ws6481 server to the bus in the ws6483 server. This is the same process as described in “Define Service Integration Bus link in ws6483” on page 444.

Set the name to the same value of ws6483BusTows6481BusLink.

Set the foreign bus name to ws6483Bus.

Set the Remote Messaging Engine name to the actual name of the Messaging Engine in the other server; in our case this was nd6483.ws6483-ws6483Bus.

Set the Bootstrap endpoint to a value that reflects the setup of your other server; in our case this was 9.12.4.38:37276.

As our ws6483 server did not have global security enabled, we left the authentication alias set to none.

Click **Apply** and save the changes.

12.4.7 Verify inter-bus connectivity

Restart both servers to bring the new definitions into effect.

In the ws6481 Adjunct STC, the message shown in Example 12-10 indicates that the ws6483 bus has connected to the sewsBus in the ws6481 server.

Example 12-10 ws6483 server message showing connection to remote bus up

```
Trace: 2005/06/23 18:45:14.472 01 t=7DCCF0 c=UNK key=P8 (13007002)
  ThreadId: 0000003a
  FunctionName: com.ibm.ws.sib.utils.ras.SibMessage
  SourceId: com.ibm.ws.sib.utils.ras.SibMessage
  Category: INFO
  ExtendedMessage: BB000222I: [:] CWSIT0032I: The inter-bus connection
ws6483BusTows6481BusLink from messaging engine nd6483.ws6483-ws6483Bus in bus
ws6483Bus to messaging engine nd6481.ws6481-sewsBus in bus sewsBus started.
```

Similarly, in the ws6481 server we see messages, shown in Example 12-11, that show the sewsBus in the ws6481 server has connected to the ws6483Bus in the ws6483 server.

Example 12-11 ws6481 server message showing connection to remote bus up

```
Trace: 2005/06/23 18:44:45.723 01 t=7DCE88 c=UNK key=P8 (13007002)
  ThreadId: 00000098
  FunctionName: com.ibm.ws.sib.utils.ras.SibMessage
  SourceId: com.ibm.ws.sib.utils.ras.SibMessage
  Category: AUDIT
  ExtendedMessage: BB000222I: [:] CWSII0060I: The Platform Messaging Component
successfully authenticated user ID edmcarr on bus sews Bus.
Trace: 2005/06/23 18:45:14.570 01 t=7C0360 c=UNK key=P8 (0000000A)
  Description: Log Boss/390 Error
  from filename: ./bborjtr.cpp
  at line: 901
  error message: BB000222I: [sewsBus:nd6481.ws6481-sewsBus] CWSIT0032I: The
inter-bus connection ws6483BusTows6481BusLink from messaging engine
```

```
nd6481.ws6481-sewsBus in bus sewsBus to messaging engine
nd6483.ws6483-sewsBus in bus ws6483Bus started.
Trace: 2005/06/23 18:45:18.835 01 t=7B4CF0 c=UNK key=P8 (13007002)
  ThreadId: 000000ab
  FunctionName: com.ibm.ws.sib.utils.ras.SibMessage
  SourceId: com.ibm.ws.sib.utils.ras.SibMessage
  Category: INFO
  ExtendedMessage: BB000222I: [sewsBus:nd6481.ws6481-sewsBus] CWSIP0382I:
messaging engine 6A45ED968BAFF849 responded to subscription request, Publish
Subscribe topology now consistent.
Trace: 2005/06/23 18:45:18.835 01 t=7B4CF0 c=UNK key=P8 (0000000A)
  Description: Log Boss/390 Error
  from filename: ./bborjtr.cpp
  at line: 901
  error message: BB000222I: [sewsBus:nd6481.ws6481-sewsBus] CWSIP0382I:
messaging engine 6A45ED968BAFF849 responded to subscription request, Publish
Subscribe topology now consistent.
```

12.4.8 Define sewsQueue in ws6483

In the ws6483 server, we now define a destination to refer to the sewsQueue in the remote ws6481 server. To define this definition, expand **Service integration**, select **Buses**, select **ws6483Bus**, click **Destinations**, then click **New**. Select **Foreign** as the type of destination to define, as shown in Figure 12-19.

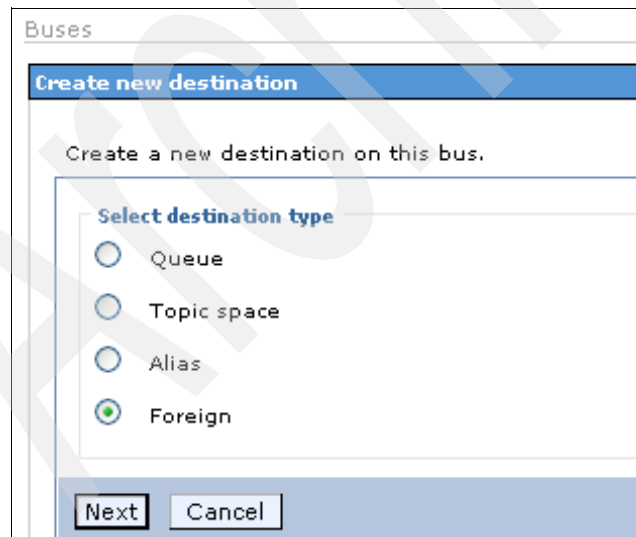


Figure 12-19 Start of process to define a foreign queue

Click **Next**, set the identifier to `sewsQueue`, and select **sewsBus** as the bus. Leave the other fields as shown in Figure 12-20.

Create a new foreign destination (a destination on a foreign bus).

→ **Step 1: Set foreign destination attributes**

Step 2: Confirm foreign destination creation

Set foreign destination attributes

Configure the attributes of your new foreign destination

* Identifier
sewsQueue

Description

* Bus
sewsBus

Default reliability
Assured persistent

Maximum reliability
Assured persistent

Figure 12-20 Setting attributes for the foreign queue

Click **Next**, then click **Finish** in the next step.

12.4.9 Define JMS queue connection factory

Using the same process described in “Define JMS queue connection factory” on page 426, we now define a JMS queue connection factory to be used by applications running in the `ws6483` server.

When defining the definition, set the name to `localWasBus`, the JNDI name to `jms/localWasBus`, the bus name to `ws6483`, and the Component-managed authentication alias to `sewsDemo`. Save the change.

12.4.10 Define JMS queue

Using the same process as described in “Define JMS queue” on page 425, we now define a JMS queue definition that will point to the `sewsQueue` foreign definition we defined in “Define JMS queue” on page 447.

When defining the definition, set the name to `snoopEjbInRemoteWasQ`, the JNDI name to `jms/snoopEjbInRemoteWasQ`, the bus name to `sewsBus`, and the queue name to `sewsQueue`. Save the change.

12.4.11 Test using two WebSphere cells

Start the servlet that lets you test the Web service by entering a URL similar to this:

```
http://9.12.4.38:39080/sewsWsDemoWeb/SewsWsDemo
```

In the input field for the WebSphere Snoop Ejb option, enter:

```
jms:/queue?destination=java:comp/env/jms/snoopEjbInRemoteWasQ&connectionFactory=java:comp/env/jms/localWasQcf&targetService=SnoopEjb
```

Click **Invoke selected web service**. The result should be similar to that shown in Figure 12-15 on page 433.

Result

This shows that we have been able to invoke the SnoopEjb Web service from a server in a different cell, using SOAP over JMS as the transport mechanism.

SnoopEjb using SOAP over HTTP

You can also invoke the SnoopEjb Web service via SOAP over HTTP between the two WebSphere servers by using the URL:

```
http://9.12.4.38:9080/SnoopEjbWsWeb/services/SnoopEjb
```

12.5 CICS to WebSphere using SOAP over JMS

In this section we demonstrate CICS calling a Web service in WebSphere using SOAP over JMS as the transport mechanism.

To do this we show how to connect the SIB in the WebSphere server with a standard external MQSeries Queue Manager, as this is the bridge that will allow CICS and WebSphere to pass messages between each other. Note that in this section we use the new default message provider of WebSphere for handling messaging in the WebSphere server.

12.5.1 CICS prereqs

CICS Transaction Server Version 3.1 APAR PK04615 must be installed to provide CICS the capability to perform Web services using SOAP over JMS.

Additionally, the MQSeries SCSQLOAD dataset must be added to the DFHRPL concatenation in the CICS region. If you do not do this and try a Web service using SOAP over JMS, you will get the following message in the CICS region:

DFHAP0900 SCSCERW1 MQ support for CICS Web Services is not available.

12.5.2 Define foreign bus

The external MQSeries Queue Manager is made known to the WebSphere server SIB, by defining it as a foreign bus. Define a foreign bus using the process shown in “Define foreign bus in ws6483” on page 443.

In Step 2, select the Routing type as a **Direct, WebSphere MQ link**.

As our MQSeries Queue Manager was called MQ4B, we named the foreign bus for it MQ4B.

12.5.3 Define WebSphere MQ Link

We now need to define a WebSphere MQ Link, which will define how the SIB in WebSphere can connect with the external MQSeries Queue Manager.

This definition is defined on the Messaging Engine of the SIB. Expand **Service Integration**, select **Buses**, select your bus, click **Messaging Engines**, click your Messaging Engine, and under Additional properties, click **WebSphere MQ Link**. Click **New**.

There are four steps in defining this definition. Figure 12-21 on page 450 shows how we filled out the fields in this first step for our systems.

Create new WebSphere MQ link

Wizard to create a new WebSphere MQ link

→ Step 1: General WebSphere MQ link properties

Step 2: Sender channel WebSphere MQ link properties

Step 3: Receiver channel WebSphere MQ link properties

Step 4: Summary of WebSphere MQ link properties

General WebSphere MQ link properties

* Name

mq4bLink

Description

* Foreign bus name

MQ4B

* Queue manager name

sewsBus

Batch size

50

Maximum message size

4194304

bytes

Heartbeat interval

300

seconds

Sequence wrap

999999999

Nonpersistent message speed

Fast

☒ Adoptable

Initial state

Started

Figure 12-21 Defining a WebSphere MQ link

Note that the Queue manager name field is set to sewsBus, the name of the bus sending the message; it is not the name of the queue manger the message is being sent to. This value becomes the reply to the queue manager name set in the message header sent in the message to the remote queue manager. This allows the remote queue manager, in this case MQ4B, to know where to send the reply message.

Click **Next** and the Step 2 page is displayed.

450 WebSphere for z/OS V6 Connectivity Handbook

Figure 12-22 shows how we filled out the fields in this second step for our systems.

The screenshot displays a wizard titled "Wizard to create a new WebSphere MQ link". On the left, a vertical sidebar lists four steps: "Step 1: General WebSphere MQ link properties", "Step 2: Sender channel WebSphere MQ link properties" (highlighted with a yellow arrow), "Step 3: Receiver channel WebSphere MQ link properties", and "Step 4: Summary of WebSphere MQ link properties". The main area is titled "Sender channel WebSphere MQ link" and contains the following fields:

- Sender MQ channel name:
- Host name:
- Port:
- * Transport chain: - Disconnect interval: seconds
- Short retry count:
- Short retry interval: seconds
- Long retry count:
- Long retry interval: seconds
- Initial state:

Figure 12-22 Defining sending channel to the MQSeries queue manager

Click **Next** and the Step 3 page is displayed.

Figure 12-23 on page 452 shows how we filled out the fields in this third step for our systems.

Wizard to create a new WebSphere MQ link

| | |
|--|--|
| Step 1: General WebSphere MQ link properties Step 2: Sender channel WebSphere MQ link properties → Step 3: Receiver channel WebSphere MQ link properties Step 4: Summary of WebSphere MQ link properties | Receiver channel WebSphere MQ link properties Receiver MQ channel name <input type="text" value="fromMq4b"/> Inbound nonpersistent message reliability <input type="button" value="Reliable"/> ▾ Inbound persistent message reliability <input type="button" value="Assured"/> ▾ Initial state <input type="button" value="Started"/> ▾ |
|--|--|

Figure 12-23 Defining the receiving channel from the MQSeries queue manager

Click **Next**, then click **Finish** in the Step 4 page. Save the changes and restart the server.

12.5.4 MQ definitions

In the MQSeries Queue Manager the following definitions are defined.

Xmit queue

Define a transmission queue using this command:

```
define qlocal(ws6481) usage(xmitq)
```

Sender channel

Define a sender channel from the MQ4B Queue Manger to the sewsBus in the ws6481 server using this command:

```
define channel(fromMq4b) chltype(sdr) +  
  conname('9.12.4.38(5558)') trptype(TCP) xmitq(ws6481)
```

Figure 12-24 on page 453 shows where in the definition of a server you can determine what TCP/IP ports are used to support connections from native MQSeries Queue Managers.

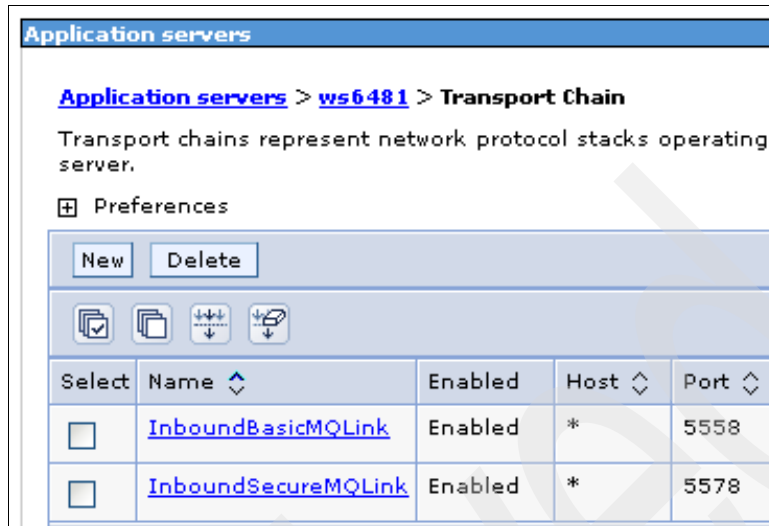


Figure 12-24 Transport chains used to support native MQ connections

Define a receiver channel

Define a receiver channel to the MQ4B Queue Manager from the sewsBus in the ws6481 server using this command:

```
define channel(toMq4b) chltype(RCVR) trptype(TCP)
```

Define remote queue

Define a remote queue definition in the MQ4B Queue Manager that points to the target queue in the bus in the ws6481 server:

```
define qremote('sewsQueue') rname('sewsQueue') xmitq(ws6481) +
    rqmname('sewsBus')
```

12.5.5 Define SnoopEjb Web service to CICS

The DFHWS2LS program needs to be run against the SnoopEJB wsdl file to make it known to CICS. However, the WSDL file needs to be manually edited prior to processing by DFHWS2LS.

Edit SnoopEjb.wsdl

In the wsdl file there are two schema elements. The first of these schema elements is shown in Figure 12-12 on page 454.

Example 12-12 First schema entry from SnoopEjb.wsdl

```
<schema targetNamespace="http://ejb.sews.itso"
xmlns="http://www.w3.org/2001/XMLSchema" xmlns:impl="http://ejb.sews.itso"
xmlns:intf="http://ejb.sews.itso" xmlns:tns2="http://beans.sews.itso"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <import namespace="http://beans.sews.itso"/>
  <element name="snoopInfoResponse">
    <complexType>
      <sequence>
        <element name="snoopInfoReturn" nillable="true" type="tns2:SnoopData"/>
      </sequence>
    </complexType>
  </element>
  <element name="snoopInfo">
    <complexType>
      <sequence/>
    </complexType>
  </element>
</schema>
```

Move the lines shown in Figure 12-12 after the second schema element in the file.

Then change this line:

```
<import namespace="http://beans.sews.itso"/>
```

to:

```
<import namespace="http://beans.sews.itso" schemaLocation="SnoopEjb.wsdl"/>
```

If you ftp the wsdl to the z/OS system as ASCII (binary transfer), then the file requires no further modification.

If you ftp the file to the z/OS system and do not use binary transfer, then the file will be in EBCDIC. Prior to running DFHWS2LS you must change the first line in the file from:

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

to:

```
<?xml version="1.0" ?>
```

Run DFHWS2LS

Run the DFHWS2LS program using the input cards shown in Example 12-13 on page 455.

Example 12-13 DFHWS2LS input cards to define SnoopEjb as a Web service to CICS

```
PDSLIB=//EDMCAR.COPYLIB
LANG=COBOL
PGMINT=CONTAINER
REQMEM=SEWS3Q
RESPMEM=SEWS3S
LOGFILE=/usr/cics/sews/wsreq/wsbind/SnoopEjb.log
WSBIND=/usr/cics/sews/wsreq/wsbind/SnoopEjb.wsbind
WSDL=/tmp/SnoopEjb.wsd1
BINDING=SnoopEjbSoapBinding
*/
```

Issue a CEMT P PIPELINE(SEWSREQ) SCAN command in the CICS region, and the SnoopEjb Web service will now show as a defined Web service to CICS, similar to that shown in Example 12-14.

Example 12-14 SnoopEjb Web service definition in CICS

```
WebService(SnoopEjb)
Pipeline(SEWSREQ)
Validationst( Novalidation )
State(Inservice)
Urimap()
Program()
Pgminterface(Notapplic)
Container()
Datestamp(20050624)
Timestamp(16:06:59)
Wsd1file()
Wsbind(/usr/cics/sews/wsreq/wsbind/SnoopEjb.wsbind)
Endpoint(http://localhost:9080/SnoopEjbWsWeb/services/SnoopEjb)
Binding(SnoopEjbSoapBinding)
```

12.5.6 Test Web service - CICS to WebSphere

Prior to running the SEWS transaction check that the channel from the MQSeries Queue Manager to the WebSphere server is started. If it is not, in the MQSeries Queue Manager issue command:

```
START CHANNEL(FROMMQ4B)
```

When we reached this stage during the residency we found that we could not successfully get CICS to invoke a Web service in WebSphere using SOAP over JMS. APAR PK09658 has been raised to correct the error preventing this function from working. You will need to apply the PTF for PK09658 to test this functionality.

When you have the PTF applied, you should be able to test the invocation of the SnoopEjb Web service in WebSphere from CICS using SOAP over JMS, log on to the CICS region, and run transaction SEWS.

The URL to be entered in the CICS screen should be similar to what is shown in Example 12-15. However, you must verify the correct layout of the URL when applying the required PTFs to your environment.

Example 12-15 Calling the SnoopEjb service from CICS using SOAP over JMS

1. LINK to SEWS0002
2. Call the SnoopInfo Web Service
Enter Host Name or IP Address where SnoopInfo Web Service is located
`http://localhost:9080/sewsWeb/services/SnoopInfo`
3. Call the SnoopEJB Web Service
Enter URL (endPoint) to access SnoopEJB Web Service
`jms:/queue?destination=sewsQueue@MQ4B&targetService=/SnoopEjb`

Option: 3

Enter 3 as the Option and press Enter.

12.6 WebSphere to CICS using SOAP over JMS

In this section we demonstrate WebSphere calling a Web service in CICS using SOAP over JMS as the transport mechanism.

12.6.1 WebSphere definitions

In addition to the definitions to allow the sewsBus in the ws6481 server to connect to the MQ4B queue manager, we need to define the following definitions to allow a message to be sent to the target queue in the MQ4B queue manger.

Define foreign queue

In the ws6481 server expand **Service integration**, select **Buses**, select **sewsBus**, click **Destinations**, then click **New**. Select **Foreign** as the type of destination. Set the Identifier as sews0002InCics, select **MQ4B** as the bus, and leave other fields as shown in Figure 12-25 on page 457.

Create a new foreign destination (a destination on a foreign bus).

→ **Step 1: Set foreign destination attributes**

Step 2: Confirm foreign destination creation

Set foreign destination attributes

Configure the attributes of your new foreign destination.

* Identifier

Description

* Bus

Default reliability

Maximum reliability

Figure 12-25 Defining the MQ queue as a remote queue in the WebSphere bus

Click **Next** and on the next page click **Finish**.

After clicking Finish, you will be returned to the display showing defined destinations. A property needs to be added to the destination that has just been defined. Click **sews0002InCics**, and on the next panel displayed, under the Additional properties heading, click **Context properties**. Add a property of type Boolean, called `_MQRFH2Allowed` with a value of true. Click **Apply** to save the property. Figure 12-26 on page 458 shows the filled-in panel.

[Buses](#) > [sewsBus](#) > [Destinations](#) > [sews0002InCics](#) > [Context properties](#) > [_MQRFH2Allowed](#)

Context information used to enable correct processing of messages. This information adds to the context information derived from processing the message header.

Configuration

General Properties

* Name

Context type

* Context value

Figure 12-26 Adding the MQRFH2Allowed property to the destination

The following URL provides a more detailed explanation about why this property is required:

http://publib.boulder.ibm.com/infocenter/wasinfo/v6r0/index.jsp?topic=/com.ibm.websphere.pmc.nd.doc/tasks/tjo0050_.html

This link also provides more information:

http://publib.boulder.ibm.com/infocenter/wasinfo/v6r0/index.jsp?topic=/com.ibm.websphere.pmc.zseries.doc/concepts/cjc0004_.html#DefineJMSQueue

Using the same process as described in “Define JMS queue” on page 425, we now define a JMS queue definition that will point to the sews0002InCics foreign definition.

When defining the definition, set the name to sews0002InCicsQ, the JNDI name to jms/sews0002InCicsQ, the bus name to MQ4B, and the queue name to sews0002InCics. Save the change.

12.6.2 MQSeries definitions

Assuming as a starting point that we have set up the WebSphere bus to MQSeries queue manager connection as described in “Define foreign bus” on page 449, we need to set up the following additional MQSeries definitions.

Define process

Define a process definition in the MQ4B queue manger using this command:

```
DEFINE                                     +
PROCESS('sews0002')                     +
APPLTYPE(CICS)                           +
APPLICID(CPIL)
```

This identifies the CICS transaction to trigger in the CICS region when the SOAP message arrives on the queue.

Define queue

Define a local queue in the MQ4B queue manager, which will be the target queue for the SOAP messages from WebSphere:

```
DEFINE                                     +
qlocal('sews0002InCics')                 +
DESCR('Queue to drive Web Service in CICS') +
PROCESS('sews0002')                       +
INITQ('CICS01.INITQ')                    +
TRIGGER                                   +
TRIGTYPE(FIRST)                           +
TRIGDATA('/service/snoopInfoCICS')        +
BOTHRESH(1)
```

The CICS region should have the queue set in the INITQ value above as its initiation queue.

12.6.3 CICS Web service

We will use the Web service provided by the SEWS0002 program, explained in Chapter 11, “Web services using SOAP over HTTP” on page 361. To invoke this Web service using SOAP over JMS, as opposed to SOAP over HTTP, does not require any further definitions in the CICS region.

12.6.4 Test Web service - WebSphere to CICS

Start the servlet that lets you test the Web services by entering a URL similar to this:

```
http://9.12.4.38:9080/sewsWsDemoWeb/SewsWsDemo
```

In the input field for the CICS Info Bean option, enter:

```
jms:/queue?destination=java:comp/env/jms/sews0002InCicsQ&connectionFactory=
java:comp/env/jms/localWasQcf&targetService=/service/snoopInfoCICS
```

Click **Invoke selected web service**. The result should be similar to that shown in Figure 12-15 on page 433.

Result

Again, we were not able to successfully complete this test because of outstanding fixes in CICS.

12.7 Summary

In this chapter we have shown how to extend the sample application to support the use of SOAP over JMS as a transport for Web services. We have shown the definitions necessary to support the use of SOAP over JMS in WebSphere and CICS and then tested this capability.

We have only dealt in this chapter with a simple example of the use of SOAP over JMS as a transport for Web services. *WebSphere Version 6 Web Services handbook Development and Deployment*, SG24-6461, provides much more detail on the topics we have covered in this chapter, plus other more advanced features of the Web services available in WebSphere.

Integrating WebSphere and DB2 using Web services

In Chapter 1, “Accessing DB2 using JDBC” on page 1 and Chapter 2, “Accessing DB2 using SQLJ” on page 25 we described how to use JDBC and SQLJ to access DB2 databases. Those technologies are straightforward to use and well-suited for most data access operations. However, when the data access logic implemented *in* DB2 becomes so complex that we can speak of business logic, it may start to make sense to call this as a *Web service*. Complex data access logic implemented in DB2 in this context can be either implemented as a complex SQL query launched by a J2EE application in WebSphere or a DB2 stored procedure.

In the above case DB2 becomes a *Web service provider* and can offer Web services to the outside world. In “Calling a DB2 stored procedure as a Web service” on page 462 we describe how a WebSphere J2EE client application calls a stored procedure in DB2 as a Web service.

Technically, it is also possible to let DB2 act as a *Web service consumer* requesting a Web service outside DB2, for example, in *WebSphere Application Server for z/OS Version 6.01*. Functions can be added to DB2 to make this possible. We discuss a sample scenario in “Calling a Web service from DB2” on page 486.

13.1 Calling a DB2 stored procedure as a Web service

Nowadays, most relational databases implement stored procedures. These procedures provide an execution environment within the database. They can be invoked in a variety of ways, including from a Java application.

Stored procedures implement complex data access logic or even business logic inside DB2. By *inside* DB2 we mean that stored procedures are managed by DB2 (many times with the help of WLM). There may be a requirement that the logic in a stored procedure needs to be accessible for a variety of clients. If those clients would be J2EE clients only, JDBC would be sufficient in most cases. However, if the requirement goes beyond J2EE clients only, for example, CICS or .Net clients, Web services technology should be the way to go. Now, it is technically possible, within certain limitations, to access a DB2 stored procedure as a Web service. In fact, it is even possible to launch an SQL query as a Web service.

Web services are described using a standardized self-defining language. The description language, known as Web Services Definition Language (WSDL), provides instructions on what functions the service provides and how consumers should interact with the service. Refer to “Web Services Definition Language (WSDL)” on page 353 for detailed information about WSDL.

Web services are generally accessed via an asynchronous message, formatted using the Simple Object Access Protocol (SOAP), an XML-based format that is sent over HTTP. The SOAP message allows requests for functions to be sent, and responses to be received by the client application, known as a consumer.

Since DB2 cannot implement its own Web service and does not have a *native SOAP listener*, an application server must host the Web service, which on its turn delegates the stored procedure call or SQL query to DB2 using standard DB2 access technology; in the case of J2EE this is JDBC. In our sample environment, this means that a client will call the service in WebSphere Application Server for z/OS Version 6.01, which in turn will use JDBC to call the stored procedure.

Attention: Due to a variety of problems we encountered with the build of Rational Application Developer Version 6 that we had access to at the time of writing this book, we have done the work for this chapter in WebSphere Studio Application Developer Integration Edition.

13.1.1 Why stored procedures

When used and designed correctly, a stored procedure can offer many benefits over keeping business logic inside the application. Table 13-1 explains these benefits in detail.

Table 13-1 Benefits of stored procedures

| Benefit | Description |
|------------------------------|--|
| Reduction in network traffic | A stored procedure can communicate directly with a database and handle the hassle of obtaining large result sets. Smaller results or simple answers can be returned to the caller, significantly reducing network traffic. |
| Code reuse | Business logic can be reused for new and updated applications. |
| Static SQL | As an alternative to SQLJ, a developer can code static calls in a stored procedure, increasing performance and security. |
| Data maintenance | Stored procedures can be called to run DB2 utilities or user-created data manipulation to make data maintenance as simple as call to a stored procedure. |

13.1.2 When not to use stored procedures

Stored procedures are not without their overhead. This overhead must be taken into consideration when exploring a stored procedure as a potential benefit. Most importantly, there are some CPU costs involved with switching contexts into the stored procedure environment. When DB2 and the application are running on the same LPAR this becomes an issue. As a result it must be measured against the reduction in network traffic.

There is also some concern with the creation of the stored procedures. Many application programmers may not be comfortable leaving the development environment to generate some of this business logic when it can be made to work functionally within the application. It is worth noting that DB2 Connect has tooling to create these stored procedures on the workstation to help reduce this problem.

13.1.3 Developing the Trader sample application

We used the Trader application as the base for our new DB2 stored procedure Web service. The Trader already had a `getQuotes()` function, which retrieves a stock quote from the database using regular Java Database Connectivity (JDBC).

In the following sample scenario we retrieve the stock quotes from within a stored procedure and return the result to the calling Java application in WebSphere Application Server for z/OS Version 6.01.

The topology of the scenario is shown in Figure 13-1.

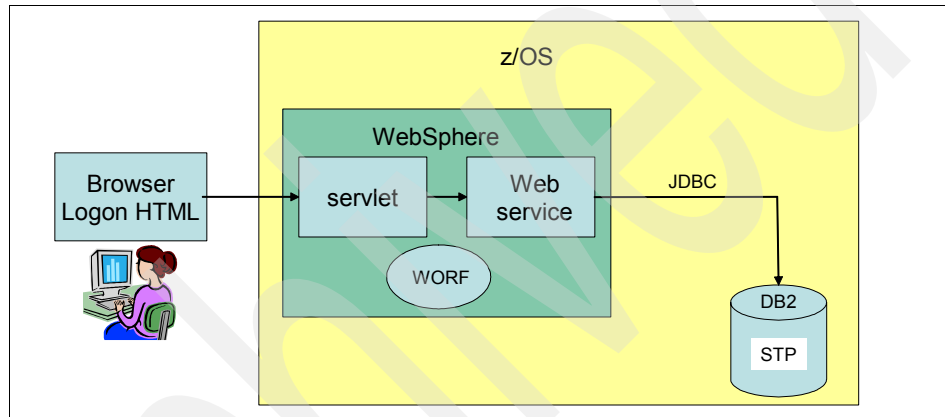


Figure 13-1 Calling a stored procedure as a Web service

Creating the Web service front end

WebSphere Studio Application Developer Integration Edition has tooling to perform almost all of the code generation needed to create a Web service front end for a stored procedure. This section will describe the process for creating the Web service.

Creating the Web service

The following steps show how to create the Web service:

1. Open WebSphere Studio Application Developer Integration Edition with a new workspace.
2. Begin the process by creating a new Dynamic Web Project where we will store the Web service. Go to **File** → **New** and select the **Other** category. Next, choose the **Dynamic Web Project** and click **Next** (Figure 13-2 on page 465).

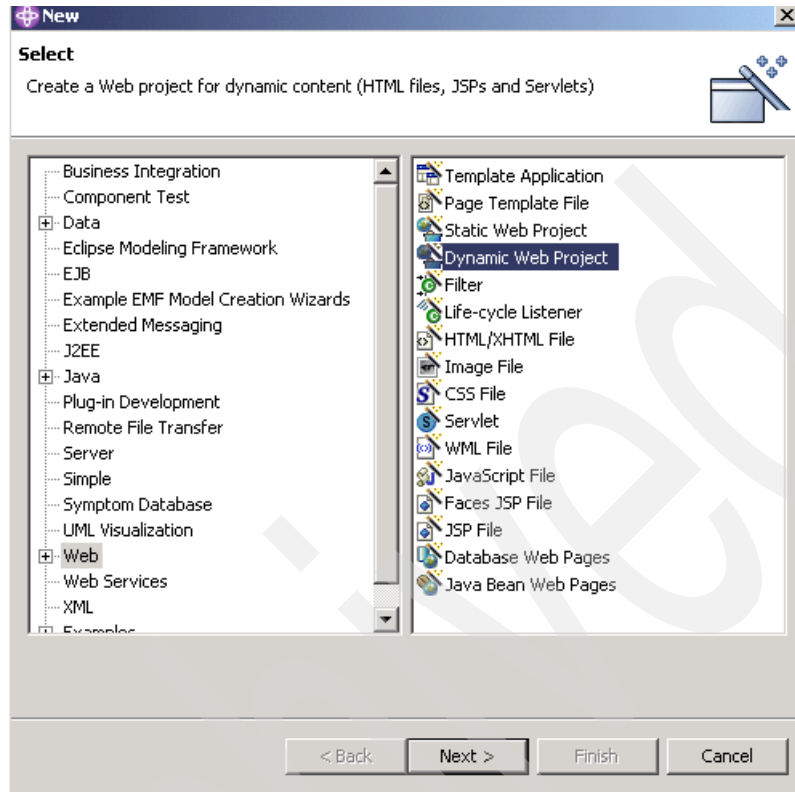


Figure 13-2 New dynamic Web project

3. Enter a name for the project on the next screen and click **Finish** to generate this project. To remain consistent with our example, use the name **TraderWebService** (Figure 13-3 on page 466).

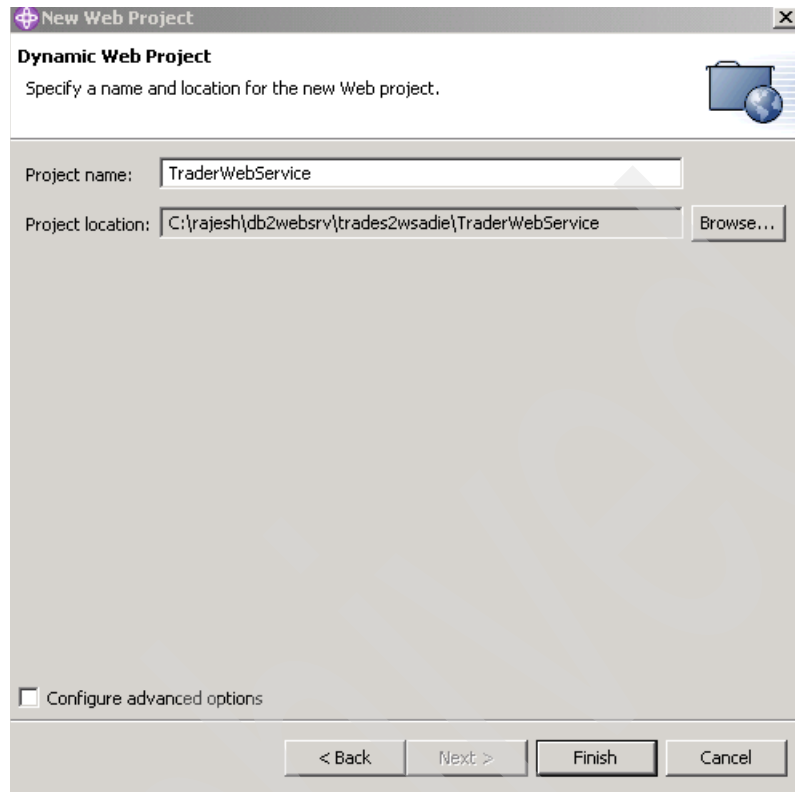


Figure 13-3 Naming the Web project

4. A DADX file will be required in order for the tooling to generate the proper WSDL and, eventually, a working client. We start the process of creating this file by creating Web Service DADX Group Configuration. As usual, there is a wizard for doing this. Click **File** → **New** → **Other** and select this wizard, as in Figure 13-4 on page 467.

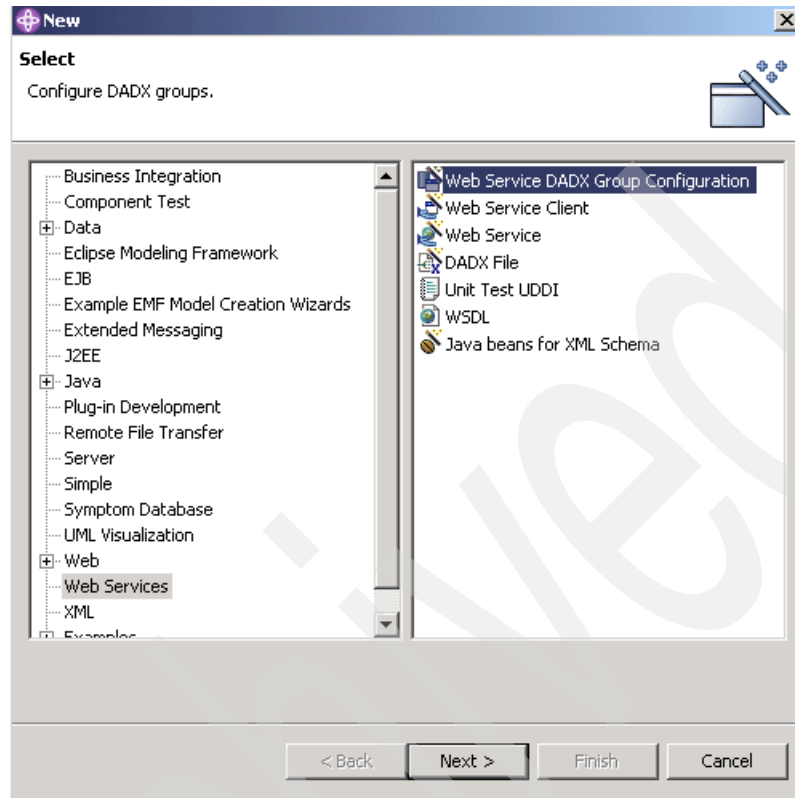


Figure 13-4 Create a DADX group configuration

5. Click **Next** to display the group configuration panel (Figure 13-5 on page 468). Add a new group and name it appropriately. *GetQuotes* is a good name because this is the function that this chapter implements.

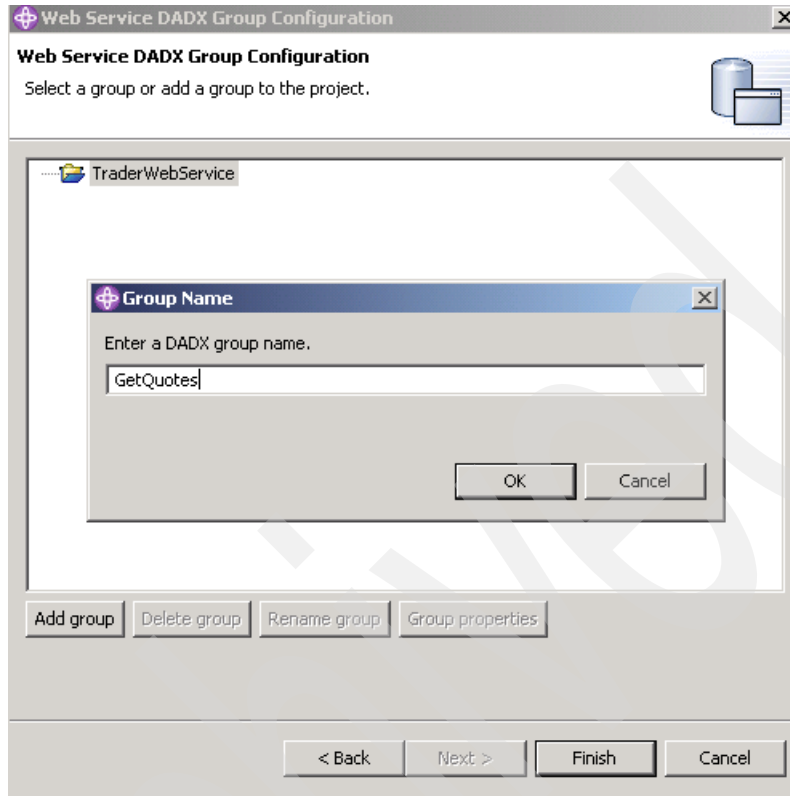


Figure 13-5 Enter a group name

6. Now, click the new group and click the **Group properties** button to enter the properties for DADX files in this group. Figure 13-6 on page 469 shows the settings that we used for this example.

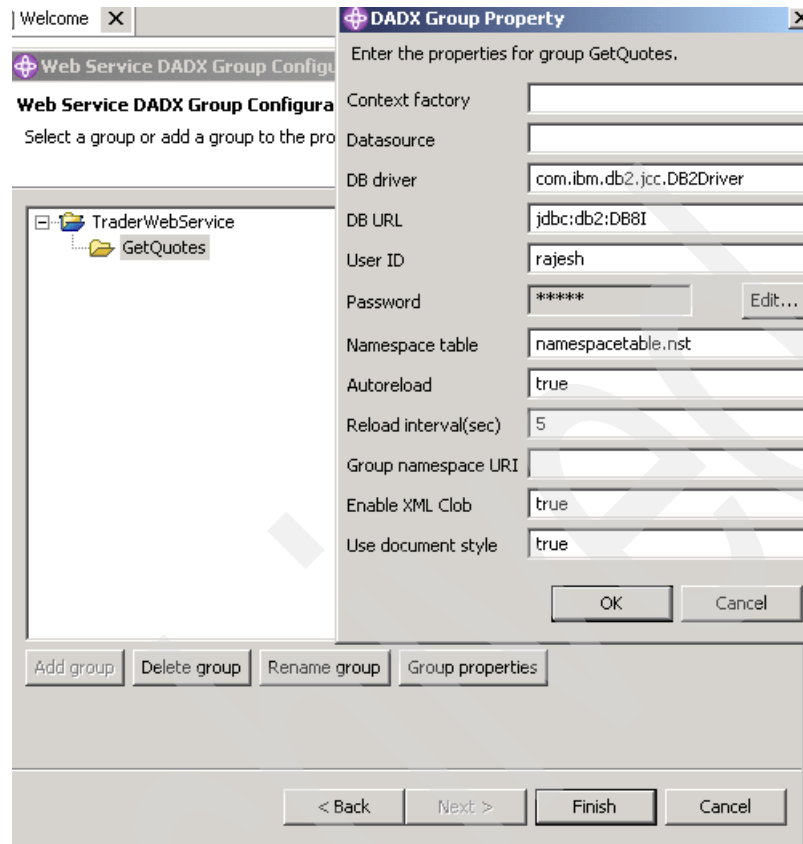


Figure 13-6 DADX group properties

7. Click **OK** and then **Finish** to set up these properties.
8. Now, the actual DADX file must be created (Figure 13-7 on page 470).

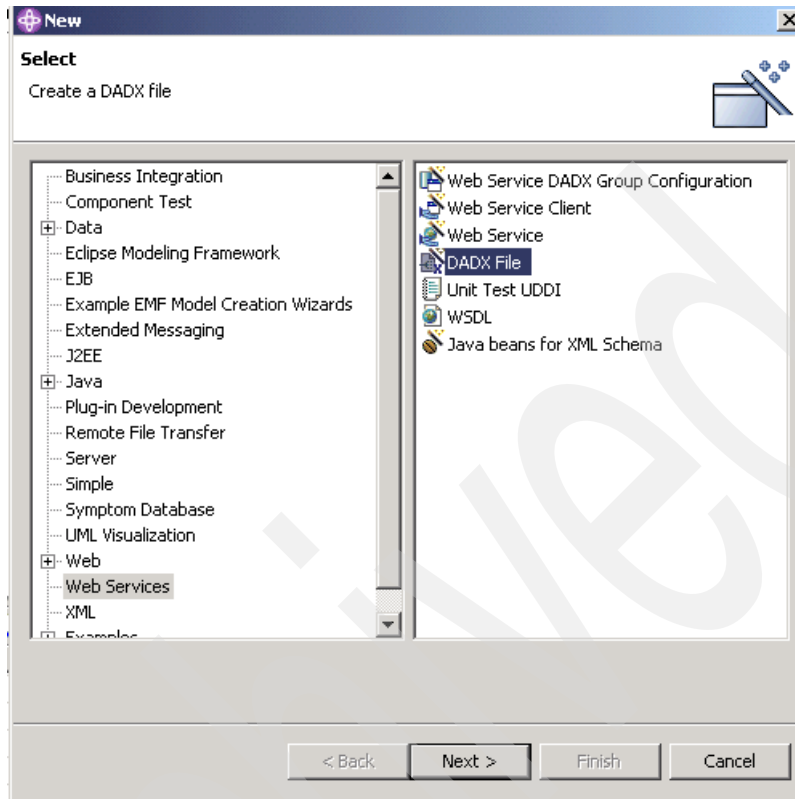


Figure 13-7 Create a new DADX file

9. Click **Next** to start this wizard.
10. At this point, the wizard would allow a connection to be made to a DB2 system to create the DADX file based on the stored procedure. Instead of going through the trouble of obtaining a connection, we choose to create our own DADX file. Click **Next** through the next two panels until you are presented with an opportunity to name the DADX file (Figure 13-8 on page 471).

Create DADX

DADX Generation

Generate a DADX file from a list of queries.

| Query: | Operation: | Description: |
|--------|------------|--------------|
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |

File name:
GetQuotes.dadx

Description:

Output folder:
/TraderWebService/JavaSource/groups/GetQuotes Browse...

< Back Next > Finish Cancel

Figure 13-8 Naming the DADX file

11. Name this file. Using the same name chosen for the DADX group makes sense. For this step we used the name *GetQuotes.dadx*.
12. Click **Finish** to create the file.
13. Since we did not base the DADX file on an actual stored procedure, we must now use the tooling to create the required information. Example 13-1 shows the DADX file that we generated for this example. Cutting and pasting it in place of the generated file will work fine. The finished ear file can also be found in the additional material for this book under the name *TraderWebServiceEAR.ear*. The outline view can also be used to generate this information.

Example 13-1 DADX file for the GetQuotes stored procedure

```
<?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">
```

```

<dadx:result_set_metadata name="resultSet1Metadata" rowName=
"resultSet1MetadataRow">
  <dadx:column name="COMPANY"
    type="CHAR"
    nullable="false"
    as="COMPANY" />
  <dadx:column name="SHARE_PRICE"
    type="REAL"
    nullable="true"
    as="SHARE_PRICE" />
  <dadx:column name="UNIT_VALUE_7DAYS"
    type="REAL"
    nullable="true"
    as="UNIT_VALUE_7DAYS" />
  <dadx:column name="UNIT_VALUE_6DAYS"
    type="REAL"
    nullable="true"
    as="UNIT_VALUE_6DAYS" />
  <dadx:column name="UNIT_VALUE_5DAYS"
    type="REAL"
    nullable="true"
    as="UNIT_VALUE_5DAYS" />
  <dadx:column name="UNIT_VALUE_4DAYS"
    type="REAL"
    nullable="true"
    as="UNIT_VALUE_4DAYS" />
  <dadx:column name="UNIT_VALUE_3DAYS"
    type="REAL"
    nullable="true"
    as="UNIT_VALUE_3DAYS" />
  <dadx:column name="UNIT_VALUE_2DAYS"
    type="REAL"
    nullable="true"
    as="UNIT_VALUE_2DAYS" />
  <dadx:column name="UNIT_VALUE_1DAYS"
    type="REAL"
    nullable="true"
    as="UNIT_VALUE_1DAYS" />
  <dadx:column name="COMM_COST_SELL"
    type="INTEGER"
    nullable="true"
    as="COMM_COST_SELL" />
  <dadx:column name="COMM_COST_BUY"
    type="INTEGER"
    nullable="true"
    as="COMM_COST_BUY" />
</dadx:result_set_metadata>
<dadx:operation name="GETQUOTES">
<dadx:documentation xmlns="http://www.w3.org/1999/xhtml">

```

```

<![CDATA[
]]>
</dadx:documentation>
<dadx:call>
  <dadx:SQL_call>
    <![CDATA[
      CALL RAJESHPR.GETQUOTES()
    ]]>
  </dadx:SQL_call>
  <dadx:result_set name="resultSet1" metadata="resultSet1Metadata"/>
</dadx:call>
</dadx:operation>
</dadx:DADX>

```

14. With the proper DADX file created, the Web service can be generated. Use WebSphere Studio Application Developer Integration Edition to create a new Web service by clicking **File** → **New** and selecting the Web Services category. Next choose Web Service as the type to create and click **Next** (Figure 13-9).

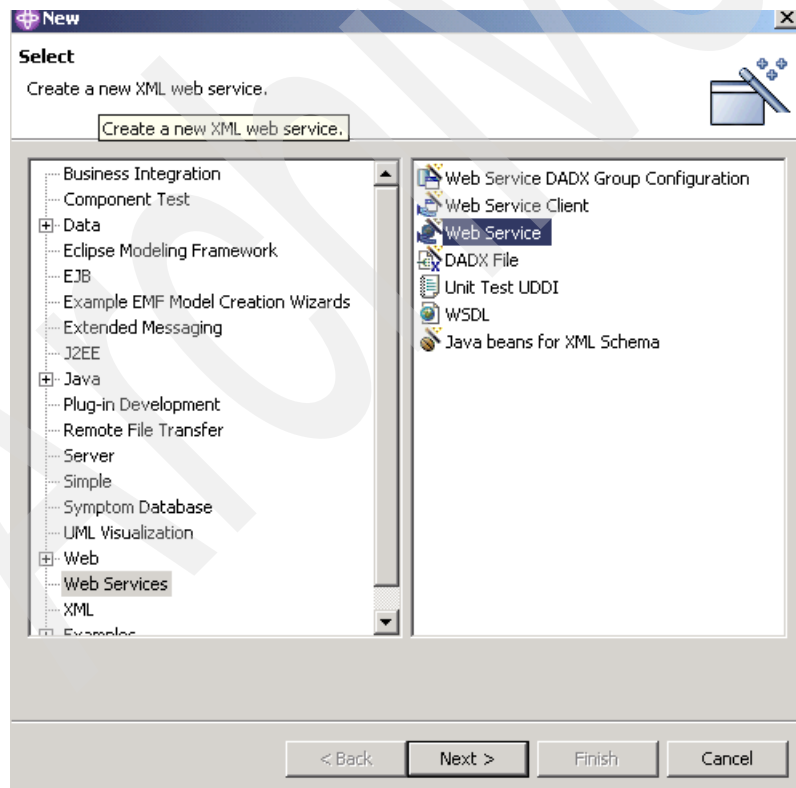


Figure 13-9 Create a new Web service

15. On the next panel, choose the DADX Web service type and do not select the option of generating the proxy. For now we just generate the service without a client. Figure 13-10 shows the setting that we used in this situation. Click **Next** when the proper changes are made.

Web Service

Web Services

Review your Web service options and make any necessary changes before proceeding to the next page.

Service

Web service type: DADX Web Service

☒ Start Web service in Web project

☐ Test the Web service

☐ Launch the Web Services Explorer to publish this Web service to a UDDI Registry

☐ Generate a proxy

Client proxy

Client proxy type: Java proxy

☐ Test the generated proxy

☐ Overwrite files without warning

☒ Create folders when necessary

☐ Check out files without warning

< Back Next > Finish Cancel

Figure 13-10 Configuring the Web service

16. Verify the proper runtime is selected (Figure 13-11 on page 475). Since we are using WSAD, our best choice is a Version 5.1 environment. We have tested this service on a Version 6 server without problems repeatedly. Click **Next** to continue.

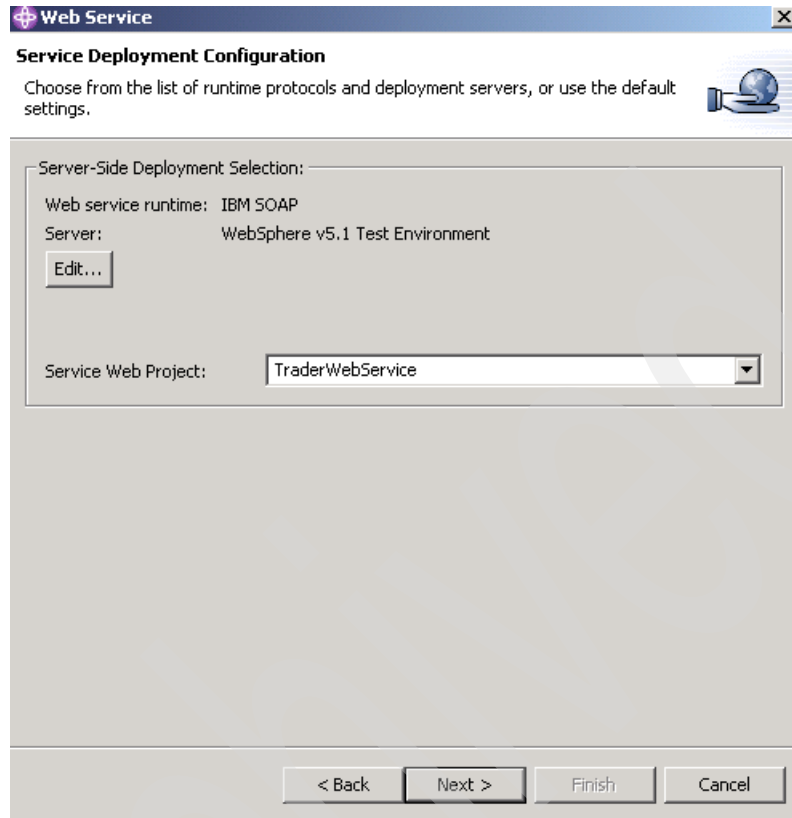


Figure 13-11 Deployment configuration

17. The next panel asks for the DADX file within this project to use to generate the WSDL and Web service code. We have only created one file. Browse to find the file and click **OK** and **Next** to continue. Figure 13-12 on page 476 shows where this file was in our environment.

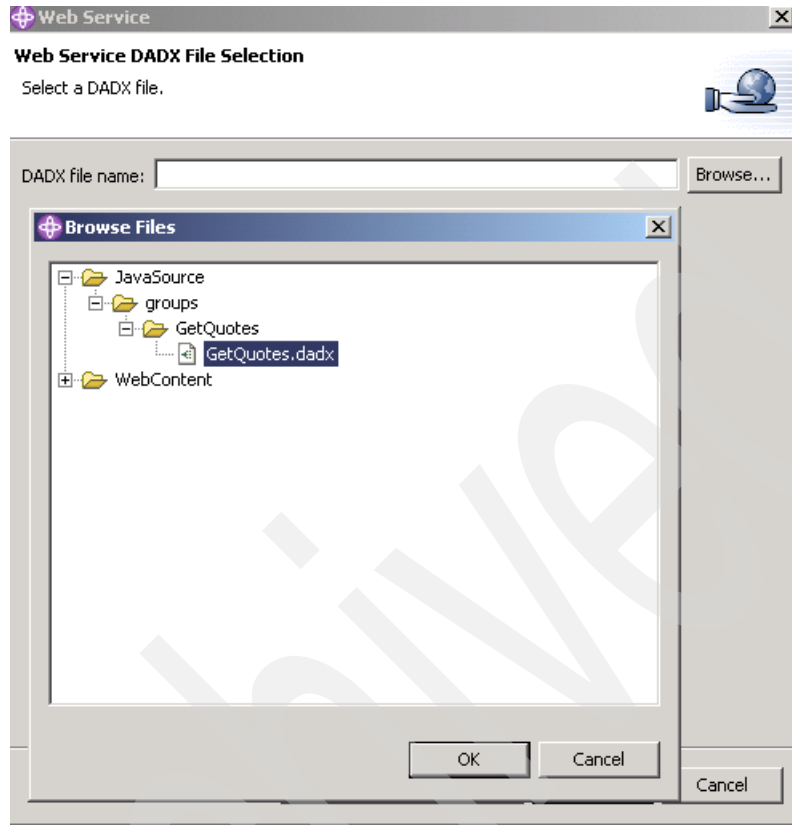


Figure 13-12 Selecting the DADX file

18. The next panel displays the properties that were chosen earlier. Click **Next** through this panel unless changes are desired. The **Finish** button can now be clicked to generate the Web service.

Creating the Web service client

With the exception of writing the DADX file, WebSphere Studio Application Developer Integration Edition has done all of the work of generating this Web service. It is possible to take this one step further and ask it to generate the client that we can call within our current Trader application to use this Web service.

1. As with the other wizards used in this chapter, this one can be started by clicking **File** → **New**, selecting the Web Services category, and choosing **Web Services Client**. Now, click **Next** to start this wizard.
2. The next panel allows for the client proxy type to be selected. Select the Java proxy because that is the type we need to use. Also check the Test the

generated proxy option so that the functionality can be tested. Figure 13-13 shows this panel with the chosen settings.

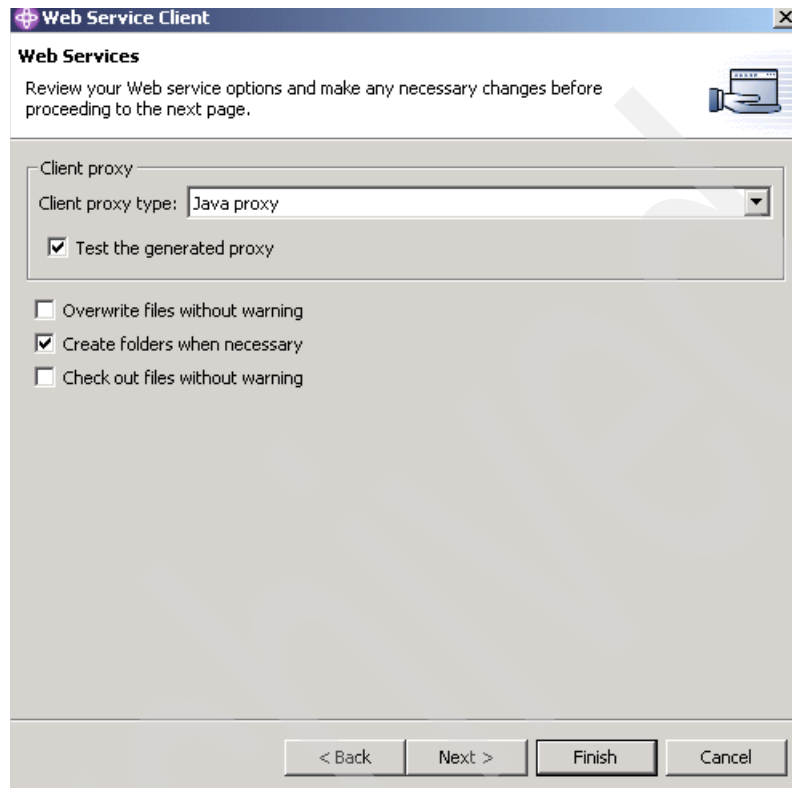


Figure 13-13 Select the client proxy type

3. Click **Next** to proceed.
4. Use the defaults in the next panel and then give the client a proper name. *TraderWebServiceClient* is the name that we used (Figure 13-14 on page 478). Now, click **Next**.

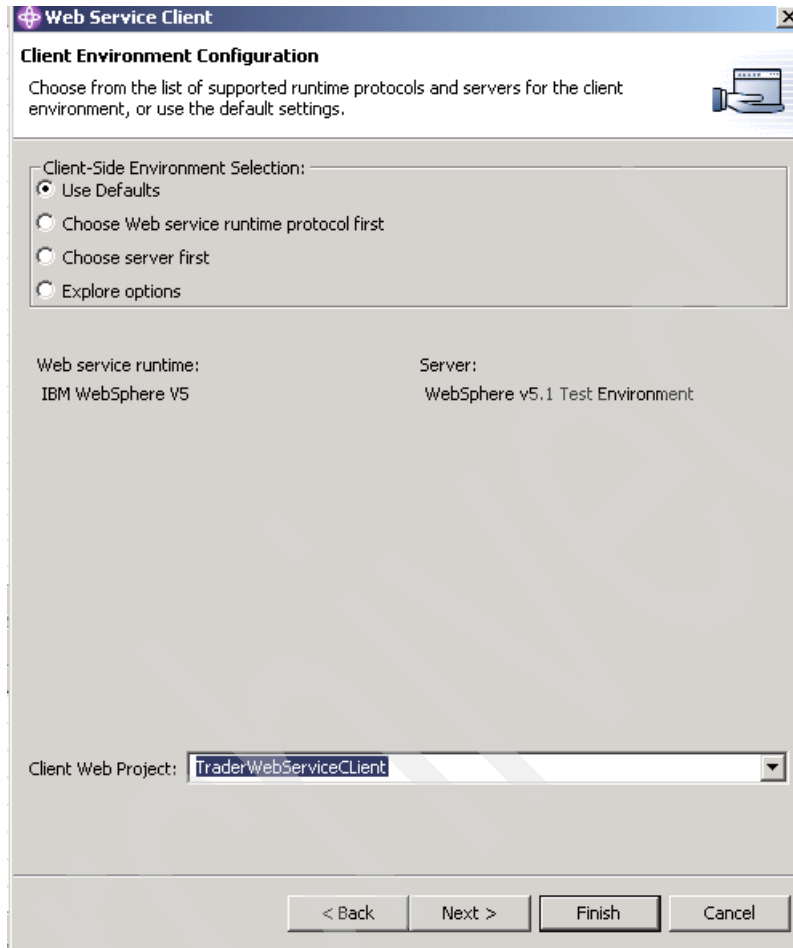


Figure 13-14 Environment configuration

5. The client needs to know which WSDL to use as information to create its client. Again, we have only one and the path should be similar to what is seen in Figure 13-15 on page 479. Once the path is selected, click **Next** to continue.



Figure 13-15 Enter WSDL file

6. We will need to have a proxy generated, and the next panel gives us this opportunity. As shown in Figure 13-16 on page 480, click the **Generate proxy** box and leave security off for the sake of this exercise.

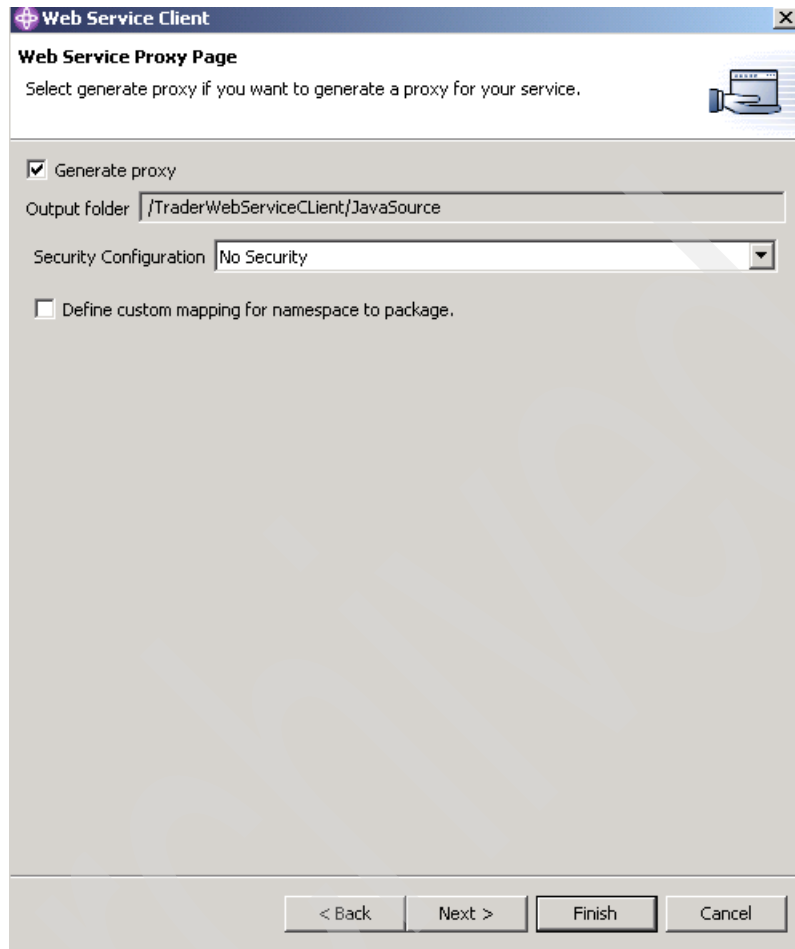


Figure 13-16 Generate a proxy

7. The last panel describes the options for testing the generated proxy. Be sure to select all methods here and to run the test on the server. Once selected, the panel should look exactly as it does in Figure 13-17 on page 481.

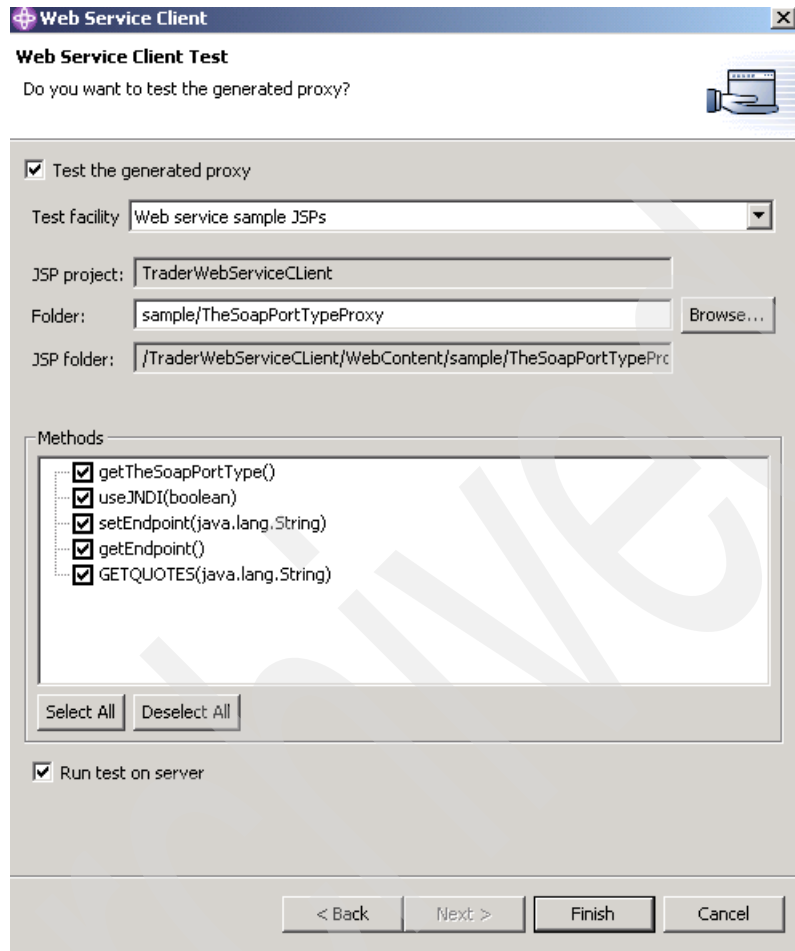


Figure 13-17 Testing the proxy

At this point a Web service has been created as well as a client proxy to call it. A new client could be created to use this service. What we chose to do for demonstration purposes was to take the TraderDB application and change it to use this service.

Calling the Web service client

The Web service is ready to be called by our code. In the same workspace, import TraderDBEAR.ear. Navigate in the J2EE perspective to find the code that returns the quotes to the application. It can be found in **EJBProjects** → **TraderDB** → **ejbModule** → **itso.db2.j2ee.trader** → **TraderDB2_UseJDBCBean.java**.

The `getQuotes` function returns a `QuotesBean` with the results for a specific company as well as holdings of a particular customer. The stored procedure that we have created and included with the additional materials gives us the first part of this information. Change the code to call the stored procedure using a SOAP call and interpret the results. The code is already there to process the individual customer information can be left unchanged, using JDBC. Example 13-2 contains the new version of the `getQuotes` code. Notice the URL points to the Web service created earlier. The Web service will call the stored procedure and return the results here.

Example 13-2 `getQuotes()` calling a DB2 stored procedure as a Web service

```

public QuotesBean getQuotes(String customer, String company) throws Exception {

    QuotesBean quotesBean = new QuotesBean();

    // get quotes from company table
    Connection con = connect();
    PreparedStatement sql;

    ResultSet rs,rs2;
    TheSoapPortTypeProxy qp = new TheSoapPortTypeProxy();

    qp.setEndpoint("http://wtsc48.itso.ibm.com:29080/TraderWebService/GetQuotes/GetQuotes.dadx/SOAP
");
    ResultSet1 quotes = qp.GETQUOTES(company);
    ResultSet1Metadata rsm = quotes.getResultSet1Metadata();
    ResultSet1MetadataRow rsr = rsm.getResultSet1MetadataRow(0);

    quotesBean.setUnitSharePrice(rsr.getSHARE_PRICE().toString());
    quotesBean.setUnitValue1Days(rsr.getUNIT_VALUE_1DAYS().toString());
    quotesBean.setUnitValue2Days(rsr.getUNIT_VALUE_2DAYS().toString());
    quotesBean.setUnitValue3Days(rsr.getUNIT_VALUE_3DAYS().toString());
    quotesBean.setUnitValue4Days(rsr.getUNIT_VALUE_4DAYS().toString());
    quotesBean.setUnitValue5Days(rsr.getUNIT_VALUE_5DAYS().toString());
    quotesBean.setUnitValue6Days(rsr.getUNIT_VALUE_6DAYS().toString());
    quotesBean.setUnitValue7Days(rsr.getUNIT_VALUE_7DAYS().toString());
    quotesBean.setCommissionCostSell(rsr.getCOMM_COST_SELL().toString());
    quotesBean.setCommissionCostBuy(rsr.getCOMM_COST_BUY().toString());

    // if the result has record, set the results to JavaBeans

    // get share amount from company table
    sql = con.prepareStatement
        ("SELECT * FROM " + fieldCustomerTable + " WHERE CUSTOMER='" + customer + "' AND
COMPANY='" + company + "'");
    rs2 = sql.executeQuery();

    int shares = 0;

```



```

        // if record is not exist, then create one
        if(!rs2.next()) {
            sql = con.prepareStatement
                ("INSERT INTO " + fieldCustomerTable + " VALUES ('" + customer + "','" +
company + "','0)");
            sql.executeUpdate();

            shares = 0;
            quotesBean.setNumberOfShares("0");

        } else {
            shares = (int) rs2.getFloat("NO_SHARES");
        }

        // share value is calculated here: value = shares * price
        float price = rsr.getSHARE_PRICE().floatValue();
        float value = shares * price;
        quotesBean.setTotalShareValue(new Float(value).toString());
        quotesBean.setNumberOfShares(new Integer(shares).toString());

        // close JDBC connection

        rs2.close();
        disconnect(con);

        return quotesBean;
    }

```

13.1.4 Enabling WebSphere to execute DB2 Web services

The application we just developed cannot be run in WebSphere Application Server for z/OS Version 6.01 without additional configuration. You will need to add the *Web Services Object Runtime Framework (WORF) for DB2* to your WebSphere runtime environment.

Web Services Object Runtime Framework for DB2

You will need the Web Services Object Runtime Framework for DB2 to deploy and execute Web services to DB2. WORF uses Apache Simple Object Access Protocol (SOAP) 2.2 or later and the Document Access Definition Extension (DADX). A DADX document specifies a Web service using a set of operations that are defined by SQL statements or XML Extender Document Access Definition (DAD) documents. Web services, or functions invoked over the Internet, specified in a DADX file are called DADX or DB2 Web services.

When we developed our sample using WebSphere Studio Application Developer Integration Edition we were implicitly using WORF in our development

environment. We now explain how to set up WORF in WebSphere Application Server for z/OS Version 6.01.

Configuring WORF in WebSphere

WORF for DB2 can only be used as of WebSphere Application Server for z/OS Version 5.1 and is provided as part of DB2 Version 8.1, WebSphere Studio Application Developer, and Rational Application Developer Version 6. It is not included in the WebSphere products for z/OS.

General information about how to enable WORF in WebSphere Application Server can be found in the DB2 Info Center at:

<http://publib.boulder.ibm.com/infocenter/db2help/index.jsp?topic=/com.ibm.db2.ii.doc/ad/ciiwascp.htm>.

However, we describe here the specific installation process for WebSphere Application Server for z/OS Version 6.01. Execute the following steps to enable your WebSphere server for DB2 Web services support. The DB2 Web services support package consists of a Java libraries, tools, schemas, and a sample program.

1. Create a directory named `worf` on z/OS and upload the file `worf.tar.Z` in this directory.
2. Stop the server now.
3. Copy the `worf.jar` file to the working directory of the application server. We placed the file in `/SC48/WebSphereAL1/V6R0/BS01/AppServer/lib/ext/`.

Note: It is a good practice to place Java libraries that are *not* part of WebSphere Application Server in the `/ext` subdirectory, and at the same time mount a separate HFS for this subdirectory. When WebSphere is upgraded, the HFS for the extensions can be remounted again and all Java extensions remain in tact.

4. You must make sure you have the JavaMail™ and Java Beans Activation Framework libraries available in the WebSphere server. The libraries are in `mail.jar` and `activation.jar`, respectively, and can be retrieved from:

<http://java.sun.com>

5. Now you must restart the server.

13.1.5 Deployment and testing

You will need to install the version of the Trader application containing the DB2 Web service. Also, you will need to make sure that you have DB2 set up for this sample. This is the checklist:

- ▶ Make sure JDBC is enabled. Refer to “Introduction to JDBC” on page 2 for more information about the JDBC infrastructure.
- ▶ Define the DB2 tables used by the Trader sample applications. The tables are described in “Trader DB2 table definitions” on page 575 and Figure A-2 on page 564.
- ▶ Install the stored procedure. The stored procedure is part of the additional material for this book.
- ▶ Deploy the Trader application including the DB2 Web service to WebSphere Application Server for z/OS Version 6.01, as described in “Deploying the Trader application to WebSphere” on page 485.

Deploying the Trader application to WebSphere

Information about deploying the TraderDB application using JDBC and SQLJ can be found in “The TraderDB application” on page 20. Deploying the Trader application using the DB2 Web service is similar.

Table 13-2 describes the files needing to be deployed for the application to function as it is described in this chapter.

Table 13-2 Applications to deploy

| Files to deploy | Description |
|----------------------|---|
| TraderWebService.war | Web service used to call the getQuotes stored procedure |
| TraderDBEAR.ear | EAR file containing the trader application |

Once deployed, the application will function the same as it does in its other forms, except that you have to enter another URL:

```
//http:<servername>:<port>/TraderDBWeb1/Logon.jsp
```

Entering a user ID on the main page and clicking **Go** starts the application. When the companies are listed, clicking the **Quote** button will run the stored procedure Web service through the SOAP connection.

13.2 Calling a Web service from DB2

In “Calling a DB2 stored procedure as a Web service” on page 462 we have seen how a Web service can be built accessing a stored procedure. This Web service is now commonly available to a variety of clients. Note, however, that you will need a J2EE application server to host this Web service and that it is not possible to call the stored procedure as a Web service directly in DB2.

In this section we show how DB2 can act as a consumer of a Web service implemented elsewhere.

13.2.1 Prerequisites for development and deployment

To be able to develop, deploy, and run such a scenario requires an additional component in the development environment and additional configuration in DB2.

The following sections describe what you need to do before being able to use the functionality.

DB2 prerequisites

In DB2 you will need additional tooling, in the form of *User Defined Functions (UDFs)*. The prerequisites for using the UDFs are:

- ▶ z/OS 1.2 or later.
- ▶ IBM XML Toolkit Version 1.6 for z/OS.
- ▶ The UDFs must run in a WLM-managed address space.

DB2 preparation

Perform the following steps:

1. Run job DSNTIJWS (provided in the SDSNSAMP data set). Customize the job to your needs.
2. Ensure that the XML Toolkit is added to the STEPLIB of the startup procedure for the WLM environment used for the UDFs.

13.2.2 Invoking the Web service consumer

The Web service consumer can be invoked in four different ways, depending on the size of the input data and the result data, as shown in Example 13-3. The UDFs are installed under the DB2XML schema.

Example 13-3 Ways of invoking the Web service consumer

```
DB2XML.SOAPHTTPV(IN endpoint_url VARCHAR(256),  
IN soap_action VARCHAR(256),
```

```
IN soap_body VARCHAR(3072))  
RETURNS VARCHAR(3072)
```

```
DB2XML.SOAPHTTPV(IN endpoint_url VARCHAR(256),  
IN soap_action VARCHAR(256),  
IN soap_body CLOB(1M))  
RETURNS VARCHAR(3072)
```

```
DB2XML.SOAPHTTPC(IN endpoint_url VARCHAR(256),  
IN soap_action VARCHAR(256),  
IN soap_body VARCHAR(3072))  
RETURNS CLOB(1M)
```

```
DB2XML.SOAPHTTPC(IN endpoint_url VARCHAR(256),  
IN soap_action VARCHAR(256),  
IN soap_body CLOB(1M))  
RETURNS CLOB(1M)
```

The variables in Example 13-3 on page 486 can be explained as follows:

| | |
|---------------------|---|
| endpoint_url | The URL of the Web service. |
| soap_action | Used as part of the SOAP request. Depending on the Web server, this parameter can be optional. If it is required, the required value can be found in the WSDL of the Web service. |
| soap_body | The operation name and the parameters to the Web service. The data in this parameter should be well-formed XML. |

13.2.3 The Trader sample

The way we are using the Trader application in our DB2 Web services usage scenarios is a bit unrealistic, but adequate to show the technology capabilities. In “Calling a DB2 stored procedure as a Web service” on page 462 we explained how to build a Web service calling a DB2 stored procedure. Now we are going to call that Web service from our DB2 environment using the installed UDFs. So, in fact, we use DB2 as a Web service client to a Web service installed in WebSphere Application Server for z/OS Version 6.01, which during its turn again calls a stored procedure in DB2.

For the invocation of the Web service consumer in DB2 we need some artifacts from the Web service previously created, which we then use as values for the variables, as explained in “Invoking the Web service consumer” on page 486:

endpoint_url

http://wtsc48.itso.ibm.com:29080/TraderWebService/GetQuotes/GetQuotes.dadx/SOAP

soap_action

http://tempuri.org/GetQuotes/GetQuotes.dadx

soap_body

<GETQUOTES
xmlns="http://tempuri.org/GetQuotes/GetQuotes.dadx" S
OAP-ENV:encodingStyle="http://schemas.xmlsoap.org/s
oap/encoding/"><COMPANY>IBM</COMPANY></GETQ
UOTES>

Combining the above variables will result in the statement shown in Example 13-4.

Example 13-4 Invoking the getQuotes Web service from DB2

```
SELECT
  DB2XML.SOAPHTTPV(
    'http://wtsc48.itso.ibm.com:29080/TraderWebService/GetQuotes/GetQuotes.d
adx/SOAP', 'http://tempuri.org/GetQuotes/GetQuotes.dadx',
    '<GETQUOTES xmlns="http://tempuri.org/GetQuotes/GetQuotes.dadx"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<COMPANY>IBM</COMPANY></GETQUOTES>') FROM SYSIBM.SYSDUMMY1;
```

Note: The part FROM SYSIBM.SYSDUMMY1 does not have any significance in this statement. Of course, we are not really selecting anything from a table, but need to satisfy the SQL syntax in some way. Using this construction is a generic way in DB2 to conform to the syntax of a SELECT statement that does not query a table.

If we would execute this statement from, for example, SPUFI, we will receive the SOAP body back as a reply, but without any formatting. The result received will be as shown in Example 13-5. Note that the response will come back as one continuous line, but in this book we had to split up the line in order to fit it on the page.

Example 13-5 Result from calling the getQuotes Web service without formatting

```
<GETQUOTESResponse xmlns="http://tempuri.org/GetQuotes/GetQuotes.dadx"
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">
```

```

<resultSet1><resultSet1Metadata
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<resultSet1MetadataRow> <COMPANY>IBM                </COMPANY>
<SHARE_PRICE>163.0</SHARE_PRICE>
<UNIT_VALUE_7DAYS>157.0</UNIT_VALUE_7DAYS>
<UNIT_VALUE_6DAYS>156.0</UNIT_VALUE_6DAYS>
<UNIT_VALUE_5DAYS>159.0</UNIT_VALUE_5DAYS>
<UNIT_VALUE_4DAYS>161.0</UNIT_VALUE_4DAYS>
<UNIT_VALUE_3DAYS>160.0</UNIT_VALUE_3DAYS>
<UNIT_VALUE_2DAYS>162.0</UNIT_VALUE_2DAYS>
<UNIT_VALUE_1DAYS>163.0</UNIT_VALUE_1DAYS>
<COMM_COST_SELL>10</COMM_COST_SELL> <COMM_COST_BUY>15</COMM_COST_BUY>
</resultSet1MetadataRow> </resultSet1Metadata></resultSet1>
</GETQUOTESResponse>

```

In our example we executed the statement from SPUFI, but it could also have been a stored procedure or any program using an SQL interface to DB2. If the statement would have been executed from a COBOL program, for example, the COBOL program would have received back the SOAP envelope with the results in XML.

Of course, under normal circumstances, the results become really usable after parsing the XML returned in the envelope. The statement shown in Example 13-6 is the same statement we explained earlier, except that we have now included an XML parser to parse the XML company name in the SOAP envelope.

Example 13-6 Invoking the getQuotes Web service from DB2 and parsing the results

```

WITH SOAPOUTPUT(OUT) AS (
  SELECT
    DB2XML.SOAPHTTPV(
      'http://wtsc48.itso.ibm.com:29080/TraderWebService/GetQuotes/GetQuotes.d
adx/SOAP', 'http://tempuri.org/GetQuotes/GetQuotes.dadx',
      '<GETQUOTES xmlns="http://tempuri.org/GetQuotes/GetQuotes.dadx"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<COMPANY>IBM</COMPANY></GETQUOTES>') FROM SYSIBM.SYSDUMMY1 )
SELECT DB2XML.EXTRACTCHAR(DB2XML.XMLVARCHAR(OUT),
  '/GETQUOTESResponse/resultSet1/resultSet1Metadata/resultSet1MetadataRow/
COMPANY') from soapoutput

```

13.3 Summary

We have seen in this chapter that Web services technology can be used between WebSphere Application Server for z/OS Version 6.01 and DB2 in both

directions. WebSphere Application Server for z/OS Version 6.01 and DB2 can be either a Web service consumer or a provider. Using DB2 as a consumer of a Web service installed elsewhere may seem unrealistic, but it is interesting to see how a Web service can be called using an SQL statement.

Connecting to WebSphere from IMS using SOAP

This chapter discusses the current possibilities to access a Web service from IMS.

IMS currently does not provide out-of-the-box support for outbound calls to WebSphere, including Web services. However, when coding IMS transactions there is still the possibility to use all the functionality availability that a certain programming language provides. This is also true for Java, in which case the Java classes provided by the J2EE programming model can be used to call an EJB as if making the call from a J2EE client container. Similarly, it is also possible to use the J2EE programming model to call Web services.

While using the J2EE Java classes, this means that the functionality of calling an EJB or a Web service is only supported directly by Java or indirectly by a language that is capable of calling Java (for example, Enterprise COBOL for OS/390 and z/OS).

In this chapter we provide a description on how to use IMS Java and Rational Application Developer Version 6 to call a Web service from an IMS Java transaction. WebSphere Studio Application Developer Version 5 or later can also be used, but the explanation in this chapter is based on Rational Application Developer Version 6. This means that we develop our code in an Eclipse-based development platform and then transfer the code to z/OS to run within IMS.

However, there is currently no support for testing the IMS Java transactions on a workstation. There is still the requirement after compiling the Java code in Rational Application Developer Version 6 or WebSphere Studio Application Developer to upload the class files or packaged jar files to z/OS and test the code by invoking an IMS transaction using any supported call method (for example, IMS Connect, 3270 terminal, APPC, MQSeries, and so on).

The sample will work as shown in Figure 14-1:

- ▶ The IMS Java transaction can be invoked from an application that calls the IMS Java transaction through IMS Connect, a 3270 terminal, a browser that invokes a servlet/EJB that calls the IMS Java transaction (top down order of arrows in Figure 14-1), or using any other method that is supported by IMS.
- ▶ The Java code itself will invoke a Java Proxy for the Web service. The Java Proxy itself contains the code to call the Web service, which can be located and run on the same LPAR, on a different LPAR, or run in an application server on any supported platform that supports Web service invocation. Figure 14-1 shows the call method.

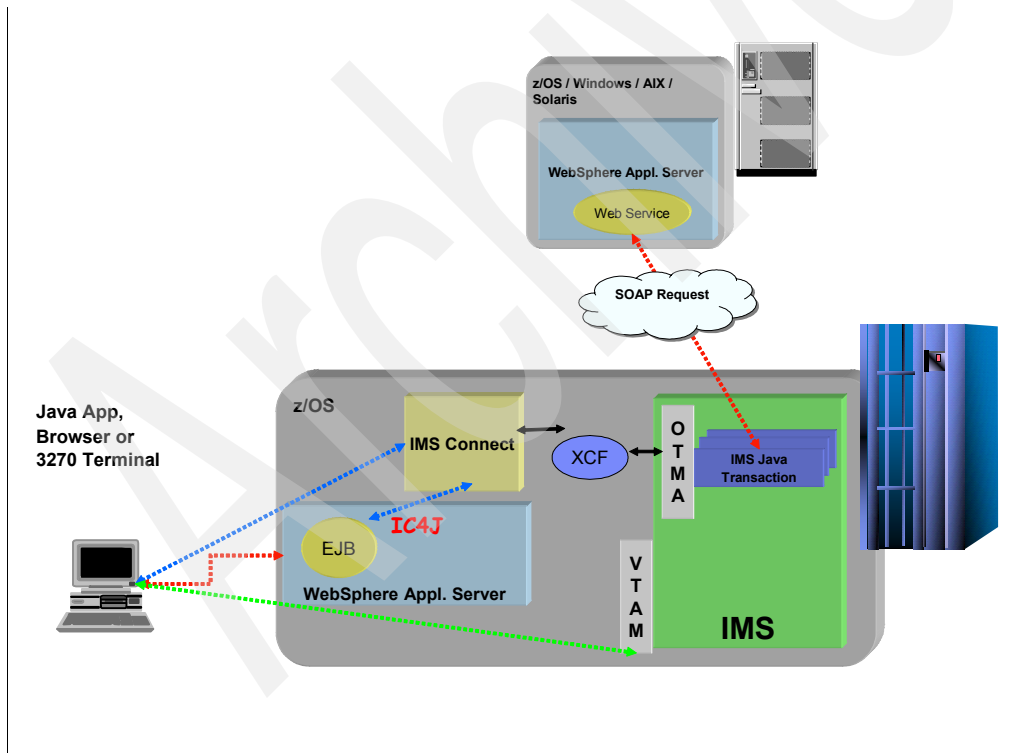


Figure 14-1 Architecture of calling an IMS Java transaction that calls a Web service

14.1 Setup

The following sections describe the software and configuration requirements.

14.1.1 WebSphere requirements

It is required that the WebSphere server instance is up and running for the Web service that is going to be called by the IMS transaction. In our sample we use the *SnoopInfo* Web service, which we described earlier in Chapter 11, “Web services using SOAP over HTTP” on page 361. It is interesting to see how the same Web service we called earlier from a CICS region can be called using the same techniques and protocols from an IMS transaction. Since the *SnoopInfo* Web service is already installed and running there is no further action required to access this Web service. If the Web service would not have been installed yet, the Web service must be installed first into WebSphere. The *SnoopInfo* Web service is part of the additional material for this book.

In addition, for IMS Java it is required to have a WebSphere Application Server for z/OS Version 6.01 installation at hand, since the runtime jar files, libraries, and the SBBLOAD datasets are required. However, it is not required that the WebSphere Application Server for z/OS Version 6.01 server is up and running, in case it is actually providing the Web service.

14.1.2 z/OS requirements

A z/OS JDK™ later than JDK 1.4.1 should be used with IMS Java. Initially, the sample IMS Java configuration members are tailored for use with JDK 1.3.1S, which was the first JDK that exploited the persistent reusable JVM technology.

Note: There can be more than one JDK installed in z/OS and be used at the same time, mounted to different HFS paths.

It is recommended to use the latest JDK level, which is 1.4.2 SR2 (PTF UK04987 Build date 20050623); refer to informational APAR II13519.

In addition, since throughout the chapter FTP transfers are used to transfer files from and to the z/OS system, it is required that there is a RACF user ID allowed to do FTP file transfers. Also, the z/OS FTP server must be configured and up and running.

Since IMS Java transactions require more memory than transactions written in traditional languages, make sure that a possible IEFUSI exit in your installation allows you to acquire at least 512 MB of memory and that there are no LE

options such as ALL24Bit in place, which might cause OutofMemoryErrors, or do not allow starting the IMS Java regions at all.

14.1.3 IMS requirements

The following steps assume that the IMS Java FMID is installed (SMP/E apply) and the IMS Java Installation Verification Procedure (IVP) has been completed successfully. The IMS system (V8 or later) should be enabled for IMS Java usage.

The following is a checklist for the setup that is required to have a development environment for running the sample to access a Web service. However, most of the required setup is available in the IMS System literature, for example, *IMS Version 9: IMS Java Guide and Reference*, SC18-7821, or *IMS Version 7 Java Update*, SG24-6536.

These are the basic requirements:

- ▶ Define a transaction code for a new transaction or use an existing one.
- ▶ If no generated Program Specification Block (GPSB) is used, a PSB for the transaction is required with LANG=JAVA.
- ▶ For the RACF user ID an OMVS segment is required for HFS access, both for development and runtime.

Important: Each IMS Java user or RACF user using IMS Java transactions requires an OMVS segment for HFS access. If that is not possible or too expensive to maintain, refer to 14.1.4, “Creating a default OMVS segment” on page 495, which will be used for all users that do not have an OMVS Segment defined in RACF.

- ▶ An HFS directory is required to store/upload the jar files created in the workstation development tool.
- ▶ Make sure the HFS that contains the IMS Java classes and samples is mounted.
- ▶ The DFSJMP PROCLIB procedure should be tailored to suit the environment's needs.
- ▶ The JCL for an IMS Java Message Processing Region (JMP) should be tailored. Make sure that the XPLINK parameter is set to Y when using JDK 1.4.1 or later and make sure that the XPLINK library <CEEHLQ>.SCEERUN2 is concatenated in STEPLIB or LINKLIST.
- ▶ The Java PROCLIB members for the Environment and the Master JVM should be tailored for use with a JDK later than 1.4.1.

14.1.4 Creating a default OMVS segment

To configure default OMVS segments, perform the following steps:

1. Define a Group ID (GID) to the system to be used as an anchor for a default OMVS group segment. Issue the following command:

```
ADDGROUP OEDFLTG OMVS(GID(777777))
```

Make the GID unique so that it is easily identified. The GID can be either very high or very low. The other fields related to the GID are not likely to be used for anything.

2. Define a user ID (UID) to be used as an anchor for the default OMVS user segment. Issue the following command:

```
ADDUSER OEDFLTU DFLTGRP(OEDFLTG) NAME('OE DEFAULT USER')OMVS(UID(999999)  
HOME('/') PROGRAM('/bin/sh'))
```

When defining a UID, consider the following issues:

- UID: Make the UID unique so that it is easily identified. The number can be very high or very low. To avoid giving super user authority, do not use 0 as the UID.
 - HOME: Select one of the following options when defining the home directory for the default user:
 - Define the HOME directory as the ROOT(/). The users do not have write access, but do not need to update their home directory.
 - Define the HOME directory in the /tmp directory.
 - Define a directory as you would for any other user. This directory is then used concurrently by many users that do not have an OMVS segment. (We do not recommend this directory option.)
 - PROGRAM: Define the default shell in this field. The other fields related to this UID are not likely to be used.
3. Set up a default value for the USER OMVS segment or set up a default UID and GID.

To set up a default value for the USER OMVS segment only, create a facility class profile named BPX.DEFAULT.USER, and then specify the default UID in the application data field. Issue the following commands:

```
RDEFINE FACILITY BPX.DEFAULT.USER APPLDATA('OEDFLTU') SETROPTS  
RACLIST(FACILITY) REFRESH
```

You cannot set up a default GROUP OMVS segment alone.

To set up a default UID and GID, create a facility class profile named BPX.DEFAULT.USER, and then specify the default UID and GID in the application data field.

Issue the following commands:

```
RDEFINE FACILITY BPX.DEFAULT.USER APPLDATA('OEDFLTG') SETROPTS  
RACLIST(FACILITY) REFRESH
```

The FACILITY class must be activated. In addition, the USER profile of the default UID and the GROUP profile of the default GID must exist, and must contain an OMVS segment with a UID and GID, respectively.

A similar process is followed to obtain a GID, when the user default group does not have an OMVS segment.

The RACF utility does not check to ensure that the application data points to a valid UID or UID and GID, or that the USER and GROUP profiles contain OMVS segments with the required UID and GID.

The following process explains how the BPX.DEFAULT.USER facility class profile works:

- a. You request a UNIX service, which is serviced by the kernel.
- b. The kernel calls the security product to extract the UID, GID, HOME, and PROGRAM information.
- c. The security product attempts to extract the OMVS segment associated with the user. If the user is not defined, the security product attempts to extract and use the OMVS segment for the default user that was listed in the BPX.DEFAULT.USER profile.

14.2 Generating Java code to access the Web service

For this sample, we rely on the Web service-enabled SnoopInfo Bean that was installed and configured throughout Chapter 11, “Web services using SOAP over HTTP” on page 361.

We use the WSDL file from that Web service as input for Rational Application Developer Version 6 to generate the Java classes that are required to call that Web service.

Follow the steps listed below to create the Java classes required to access an existing Web service:

1. Start Rational Application Developer Version 6. Usually you need to get the WSDL file from the Web service provider. In our case we had the WSDL from our previous sample for CICS. You can find the WSDL in the additional material for this book.
2. Switch to the Project Explorer view and create a new Dynamic Web Project, as shown in Figure 14-2 on page 497. This is done by selecting **File** → **New** → **Dynamic Web Project**.

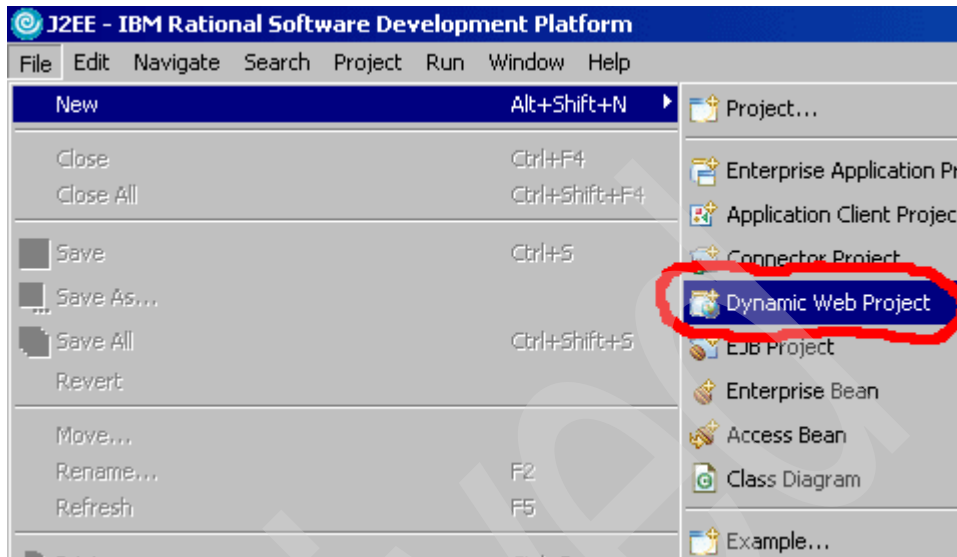


Figure 14-2 Create a new Dynamic Web Project

We called our project SnoopWebServiceClient, as shown in Figure 14-3.

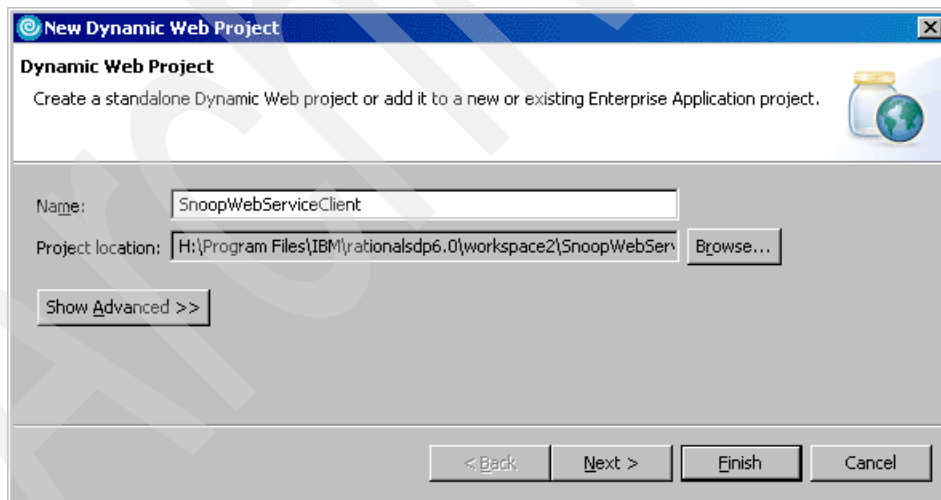


Figure 14-3 New Dynamic Web Project wizard

In the new project wizard enter the name and accept all defaults by clicking **Finish**.

Do not switch to the Web perspective if asked, as shown Figure 14-4 on page 498.

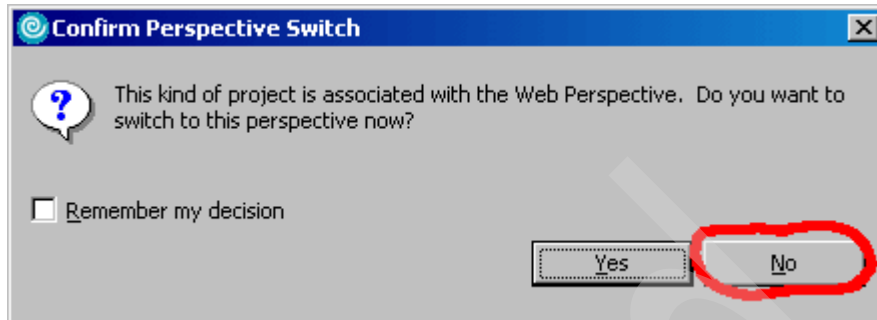


Figure 14-4 No switch to Web perspective

3. Expand the project and the folder **Java Resources**. Expand the **JavaSource** folder and right-click to select **Import** to import the WSDL file from the pop-up menu, as shown in Figure 14-5.

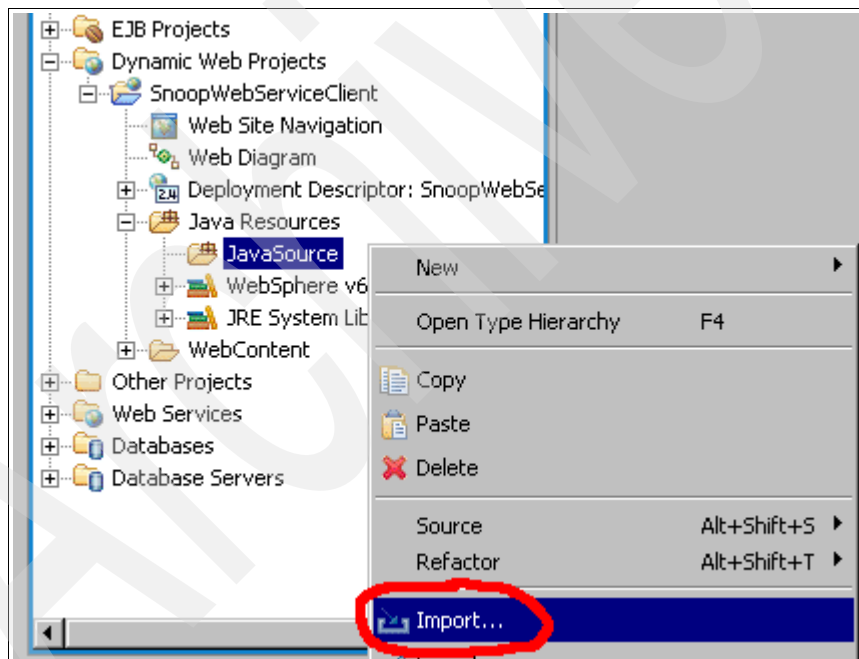


Figure 14-5 Import menu selection

Choose **File System** as import source, as shown in Figure 14-6 on page 499. Click **Next**.

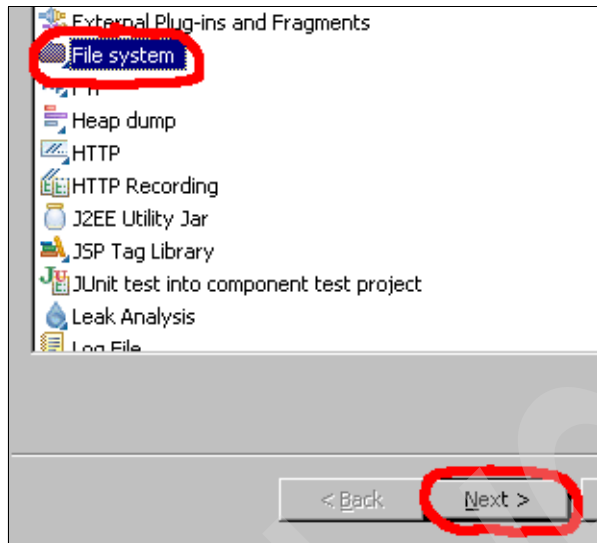


Figure 14-6 Import from File system wizard

Browse and select the folder where the WSDL file was stored. Click the folder name on the left side, select the file from the listing on the right side of Figure 14-7 on page 500 (in our case it was SnoopInfo.wsdl), and click **Finish** when selected.

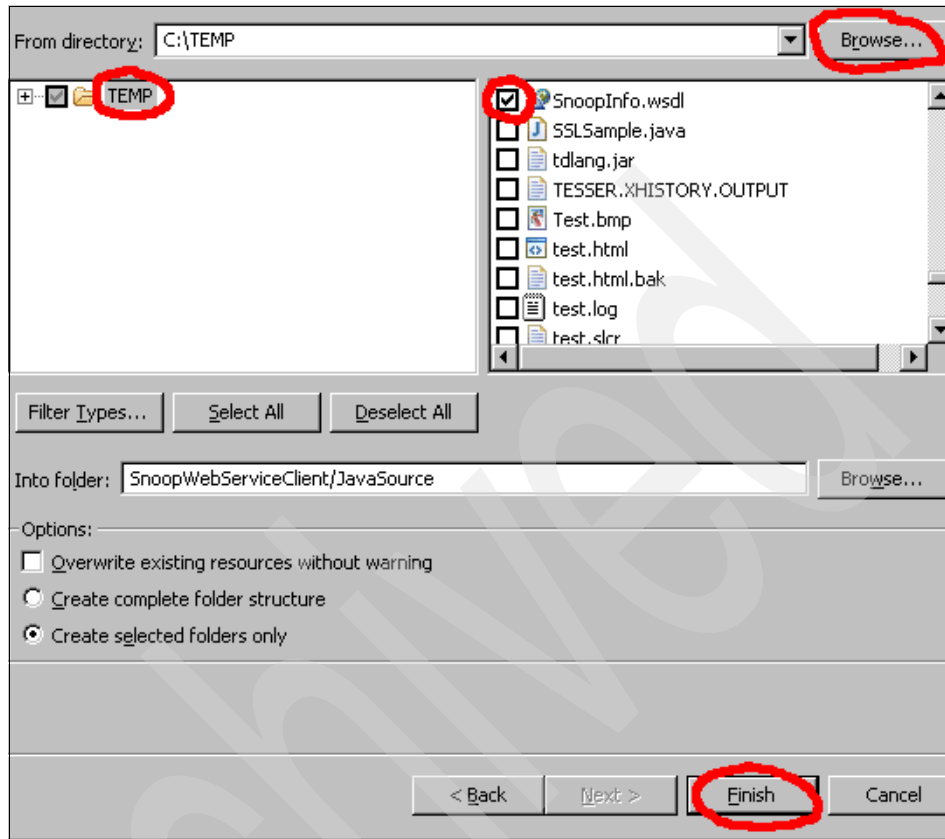


Figure 14-7 Import WSDL file from file system

Once the WSDL is imported, SnoopInfo.wsdl can be double-clicked to be viewed. At the end of the source listing, it should contain a valid URL to call the Web service, as shown in Figure 14-8.

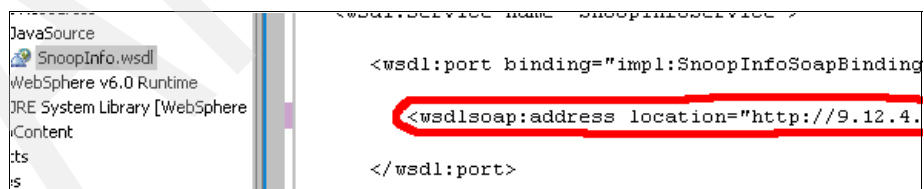


Figure 14-8 Check the URL of the web service

4. In the JavaSource folder create a new package called itso.ims.ws.snoop. This can be done by selecting the JavaSource folder with a right mouse click and using context menu **New** → **Package**, as shown in Figure 14-9 on page 501.

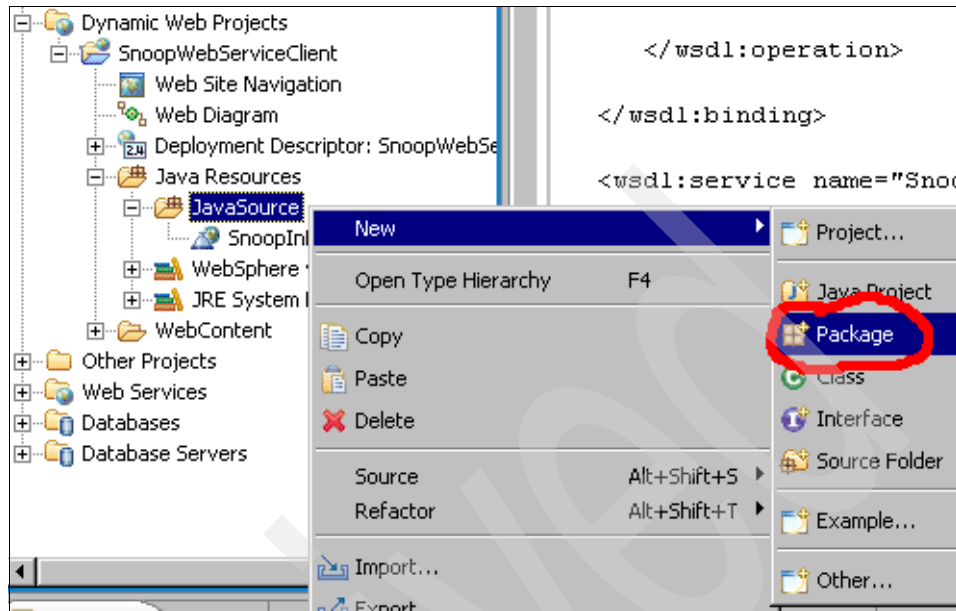


Figure 14-9 Menu selection new package

Another possibility is selecting JavaSource with one left mouse click and using the menu **File** → **New** → **Other**, and looking for Java, and selecting **Package**.

In the appearing New Java Wizard enter the name `itso.ims.ws.snoop`, as shown in Figure 14-10 on page 502. Click **Finish**.

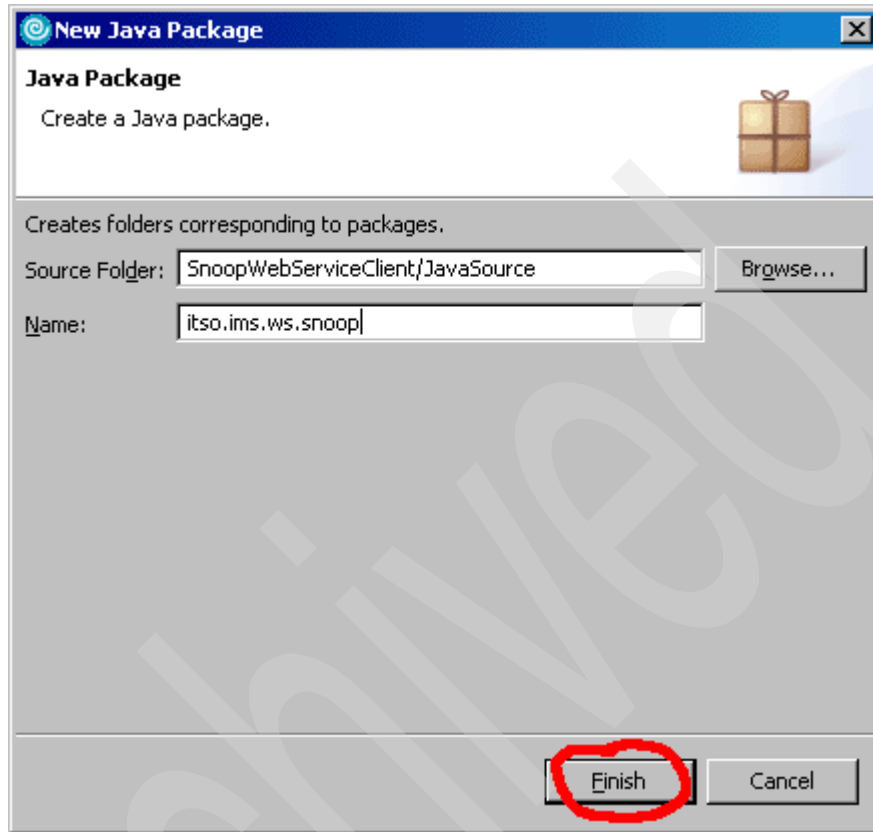


Figure 14-10 New Java Package wizard

The project tree after importing the WSDL looks as shown in Figure 14-11.

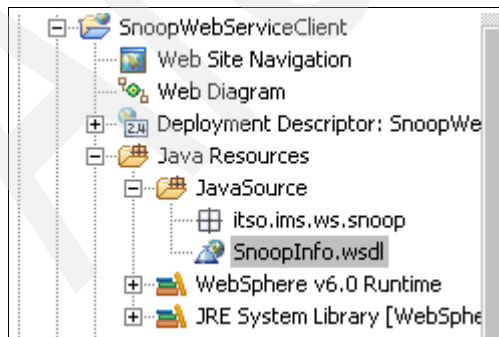


Figure 14-11 The project tree after importing the WSDL file

5. Mark the WSDL file by clicking it. Right-click and in the context menu select **Web Services** → **Generate Client**, as shown in Figure 14-12.

Important: We noticed that during the development, when switching between different workspaces, there were cases where the Web services menu was missing. Refer to 14.2.1, “Case of missing Web services menu” on page 507, if you encounter a similar situation.

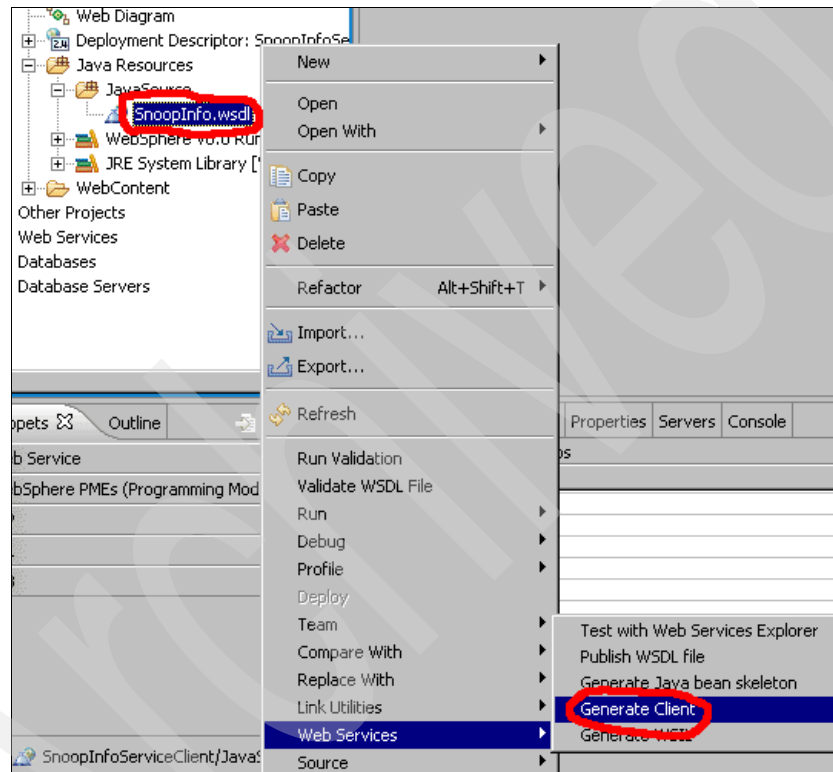


Figure 14-12 Menu selection of Web Services Generate Client

In the appearing Web Service Client wizard, select **Java Proxy** as the Client Proxy Type, as shown in Figure 14-13 on page 504.

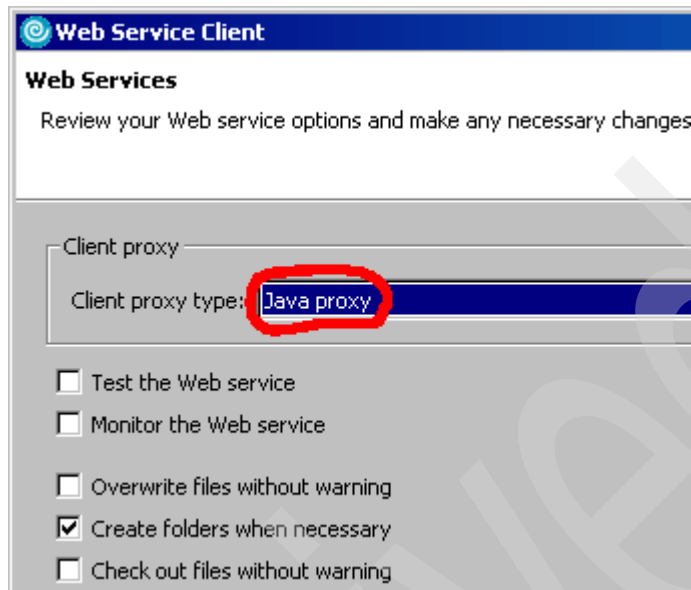


Figure 14-13 Web Service Client wizard Client proxy type selection page

Click **Next**.

At the Web Service Selection Page make sure that the correct WSDL file is selected, as shown in Figure 14-14.

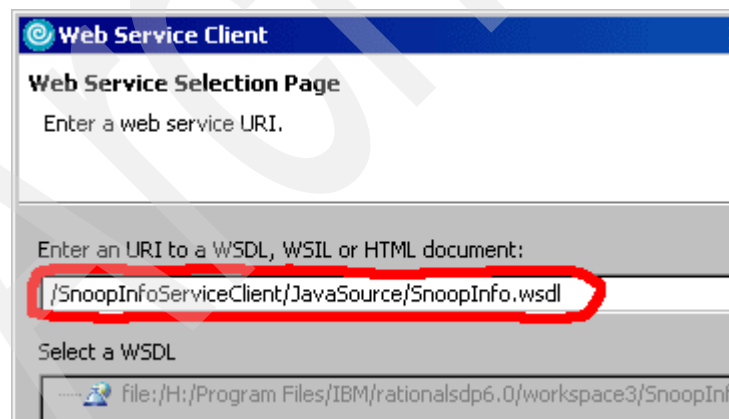


Figure 14-14 Web Service Client Wizard WSDL file selection page

Click **Next**.

As shown in Figure 14-15, at the Client Environment Configuration page make sure the client type is Web, the client project is the SnoopWebServiceClient, and the EAR project is SnoopWebServiceClientEAR (which is currently not there, but it will be created once the action is completed), and click **Next**.

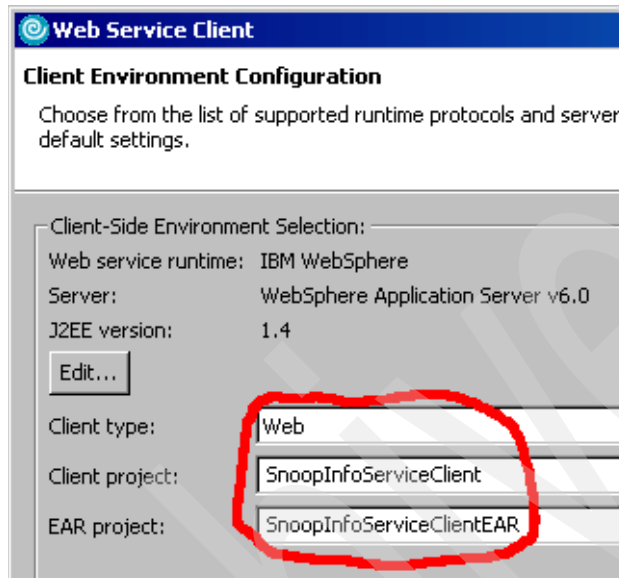


Figure 14-15 Web Service Client wizard Client Environment Configuration page

As shown in Figure 14-16 on page 506, at the Web Service Proxy Page make sure that No Security is selected and click **Finish**.

If there is a message complaining that file overwriting is not enabled, click **Yes** to enable file overwriting.

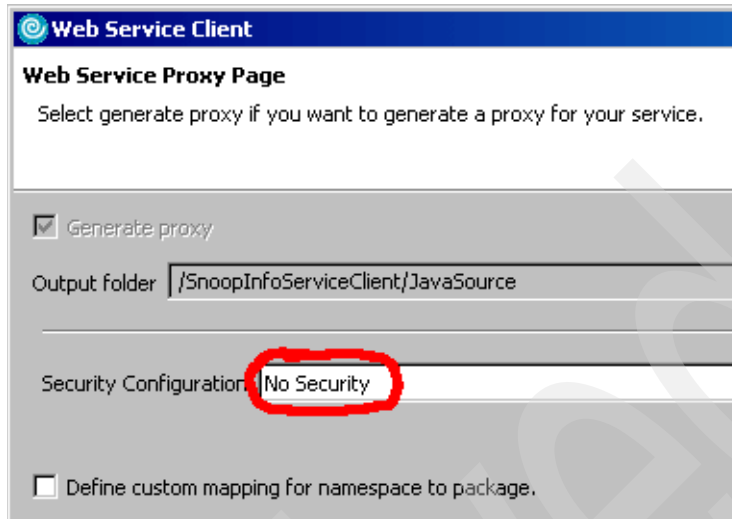


Figure 14-16 Web Service Client wizard security configuration

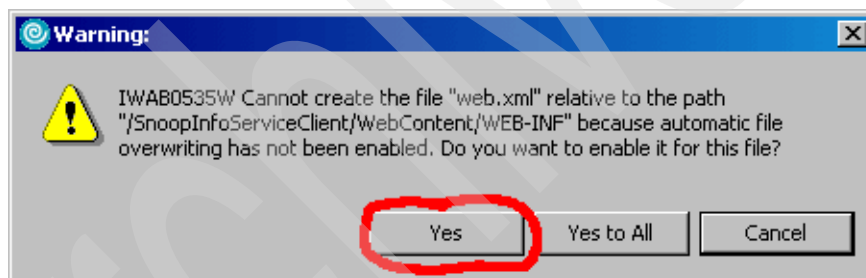


Figure 14-17 Enable automatic file overwriting

6. After completing this step, a new package called itso.sews.beans according to our SnoopInfo.wsdl has been generated. It contains all classes that are required to call the Snoop Web service. See Figure 14-18 on page 507.

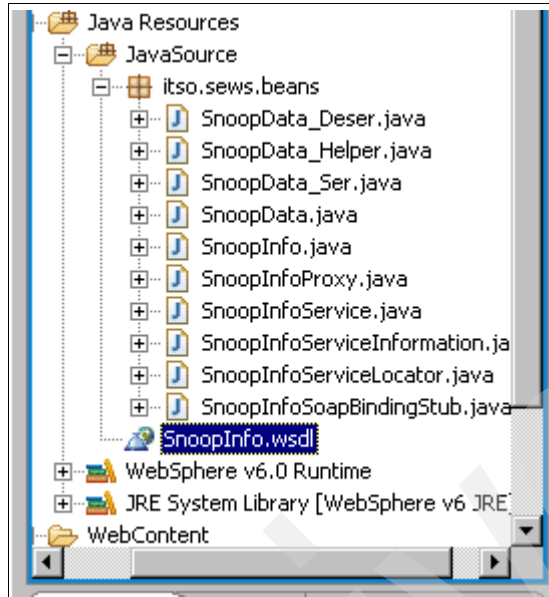


Figure 14-18 Web Service Proxy classes and itso.sews.beans package successfully created

At this point we have successfully created a Java proxy for the Web service. Testing this proxy is covered in “Test the Web service” on page 510.

14.2.1 Case of missing Web services menu

If the Web services context menu is missing, do the following. With the left mouse button click any WSDL file (for example, SnoopInfo.wsdl), and select **File** → **New** → **Other**, as shown in Figure 14-19 on page 508.

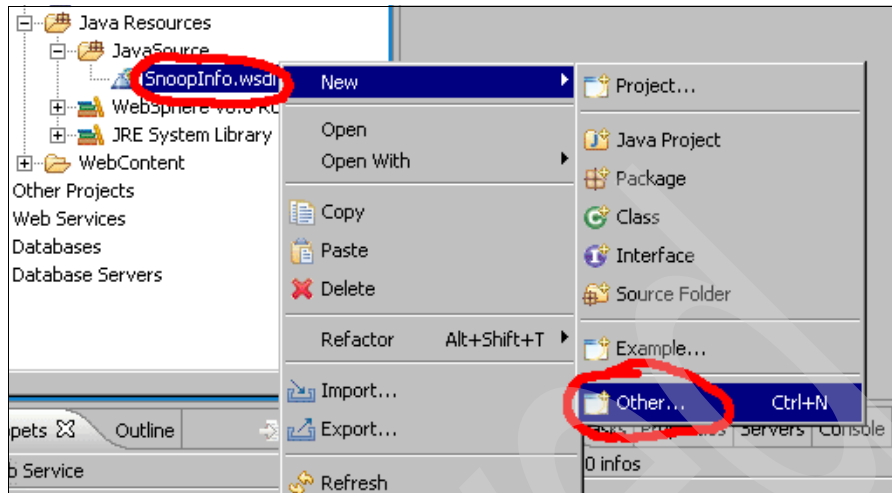


Figure 14-19 Menu selection - New -> Other

As shown in Figure 14-20 on page 509, in the appearing New Wizard selection select **Web Service** (the fourth selection from the top) and click **Next**.

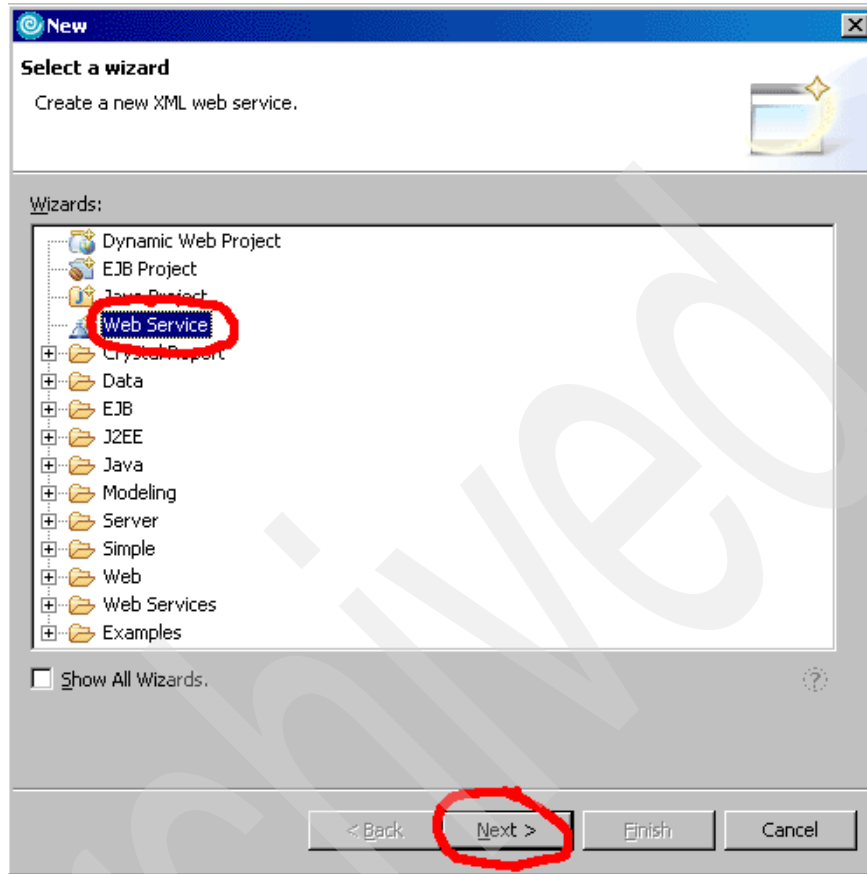


Figure 14-20 New other wizard Web Service selection

There will be a window asking if the Web Services Development capability should be enabled, as shown in Figure 14-21 on page 510.

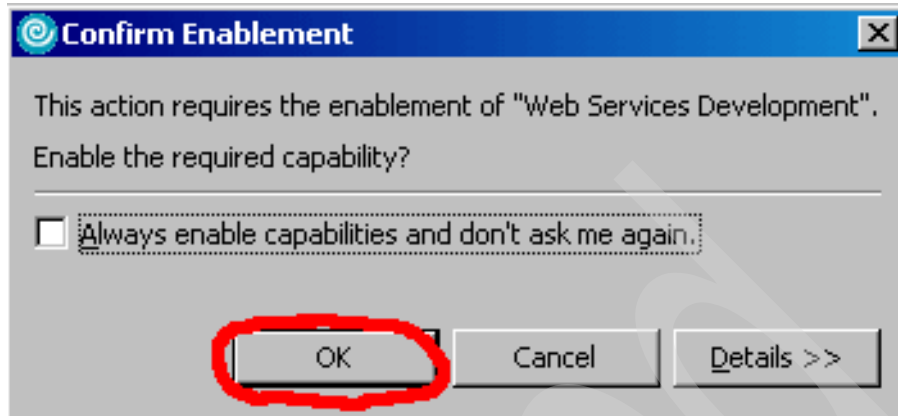


Figure 14-21 Confirm enablement of Web Services Development capability

Click **OK** to enable the Web Services Development and then click **Cancel** to close the appearing Web Service wizard.

As soon as the steps above have been completed, there should be the Web services menu available to be used in Rational Application Developer Version 6.

14.3 Test the Web service

Before we continue to use the generated classes in IMS, we will test if the Web service is working. This is required to prevent us from having problems while being in the IMS environment. It is always a good idea to do separate function level tests.

In order to test the Java proxy we create a little Java application that calls the Java proxy. Follow the steps below:

1. Select the package **itso.sews.beans** by clicking on the package name. Right-click and choose **New** → **Class**. The New Java class wizard opens. As the name, enter SnoopInfoProxyTestApp, check to create a main method, and click **Finish** to create the new class.
2. The source of the newly created class is opened in the Editor window. It should contain an empty main method. In order to invoke the Java Proxy for the SnoopInfo Web service, the following code should be added to the main method (Example 14-1).

Example 14-1 Contents of the main method to test the SnoopInfoProxy

```
try {  
    SnoopInfoProxy proxy = new SnoopInfoProxy();
```

```

        SnoopData output = proxy.getSnoopData();
        System.out.println( "\nS1: " + output.getS1() +
            "\nS2: " + output.getS2() +
            "\nS3: " + output.getS3() +
            "\nS4: " + output.getS4() +
            "\nS5: " + output.getS5() +
            "\nS6: " + output.getS6() +
            "\nS7: " + output.getS7());
    } catch (Exception e) {
        System.out.println("\nCaught exception is: " + e);
        e.printStackTrace();
    }
}

```

3. Save the changes by pressing Ctrl+S or selecting **File** → **Save**. Make sure that there are no compiler errors. The editor window can be closed now.
4. In the project tree select the file **SnoopInfoProxyTestApp.java**. Right-click to open the context menu and select **Run** → **Java Application**. If the Web service call was successful there will be some output in the Console window, as shown in Figure 14-22.

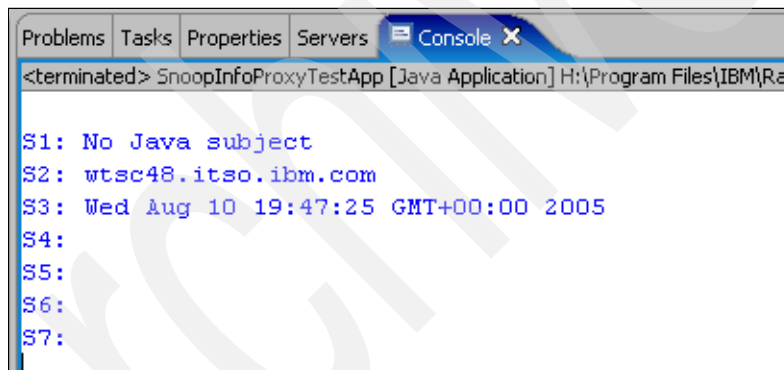


Figure 14-22 Output from testing the generated Java proxy

Now that we successfully tested the Java proxy is working we can continue to include that code in an IMS Java transaction.

14.4 Calling Web service from within IMS Java transaction

Now that we have the Web service access class working, we take the Java code for a very simple IMS Java transaction and call the Web service. The following sections describe the steps and the setup that is required to run that transaction.

14.4.1 Preparing RAD for use of ANT scripting

ANT scripting makes the deployment of IMS Java applications easier. It is a Java utility that has the power of make files. So an ANT script can contain all the required steps to create an application, for example, compile the Java source classes, delete the old jar file, create a new jar file with the compiled Java classes, and upload the jar file to a directory on the host. We use ANT build scripts throughout the following sections.

There is one thing that has to be considered. The ANT version that is delivered with our build of Rational Application Developer Version 6 does not contain the classes for FTP. There is a jar file missing, called commons-net.jar, which has to be added to the ANT classpath. This is done by downloading the Jakarta commons net package, extracting the commons-net-v.r.m.jar into the ANT plugins lib directory, and adding an entry into the ANT plug-ins configuration file. This file may be included in future builds of Rational Application Developer Version 6.

Important: Please make sure that either Rational Application Developer is closed before making those changes or it is required to restart RAD before the changes are active.

The Jakarta commons net package can be downloaded from the apache.org project Web page. For direct access the following URL can be used:

http://jakarta.apache.org/site/downloads/downloads_commons-net.cgi

Important: All official Apache commons-net-V.R.M contain a severe bug, which creates a java.lang.NullPointerException when accessing a directory that contains an external link from a HFS directory to a PDS or other MVS file system. So make sure to use the fixed commons-net-1.4.0.denis.jar, which is part of the additional material for this book.

One of the binary builds of the commons-net project should be downloaded and the commons-net.jar should be extracted into the ANT plugin lib directory, which is (assuming RAD was installed on drive C):

C:\Program Files\IBM\Rational\SDP\6.0\eclipse\plugins\org.apache.ant_1.6.2\lib

Once that file is downloaded and extracted, the commons-net.jar has to be added with the following lines into the ant configuration xml file:

C:\Program
Files\IBM\Rational\SDP\6.0\eclipse\plugins\org.apache.ant_1.6.2\plugin.xml

For our environment the current version was 1.4, so the file extracted was named commons-net-1.4.0.jar. Add an entry for that file at the end of the plugin.xml configuration file. The end of the file should look like Example 14-2.

Example 14-2 Changed plugin.xml for RAD Ant installation

```
<library name="lib/commons-net-1.4.0.denis.jar">
  <export name="*" />
</library>
</runtime>
</plugin>
```

Save the file.

According to the Rational Application Developer help there is a more convenient way to add a jar file to the Ant classpath. The following instructions are taken from Ant help:

When using an optional or custom task it is usually necessary to add extra libraries to the classpath. The Ant classpath can be modified globally or per launch configuration.

To modify the ANT classpath globally:

- a. Click **Window** → **Preferences**.
- b. Expand **Ant** and select **Runtime**. If you are not on it already, click the **Classpath** tab.
- c. To add a JAR file to the classpath, click **Add Jar** and select the JAR file.
- d. To add a folder to the classpath, click **Add Folder** and select the folder.
- e. To remove an item from the classpath, select it and click **Remove**.
- f. To restore the classpath to the default, click **Restore Defaults**.

Both possibilities will work but are required to run the ftp ant task successfully.

14.4.2 Importing the IMS Java classes into the project classpath

In order to write an IMS Java transaction it is required to import the IMS Java classes into the project classpath. Since the required classes are delivered with IMS in a jar file, it is necessary to download this jar file to the workstation.

Attention: According to the checklist in “IMS requirements” on page 494 make sure that the IMS Java HFS is mounted or ask the systems programmer about the location and the mount point of the IMS Java HFS.

Feel free to use any other method or ftp client to download imsjava.jar. The following description shows how to use the built-in functions of the Eclipse platform to achieve this.

Downloading imsjava.jar with an Ant script

Important: After writing this script we found a bug in Ant. Due to an error in the ant code, the download will not work as long as the bug is not fixed. Please proceed to “Workaround for the bug in Ant FTP task” on page 519. This is a download problem only; it does not apply to FTP uploads. However, this section shows how it is supposed to work if there were no bug. The description in “Preparing RAD for use of ANT scripting” on page 512 was updated to use a fixed version of the commons-net.1.4.0.jar.

The Ant source code was downloaded and the error fixed. A bug report at the Apache bug home page was opened and a patch donated. The description is available at:

http://issues.apache.org/bugzilla/show_bug.cgi?id=36258

After you have installed either our fixed version of the commons-net 1.4.0.jar file or a fixed version of Rational Application Developer you can proceed with downloading the imsjava.jar using an FTP client of your choice.

On our test system the IMS Java classes are located in the default installation directory, which is /usr/lpp/ims/imsjava91.

There are many different ways and clients in order to download a file from a FTP server using the FTP protocol. For the tests it was found to be most convenient when using Ant scripting to achieve this.

In order to execute an Ant script it is required to create an XML document that describes the actions to be done.

Mark the project name (SnoopInfoServiceClient) with the left mouse button and right-click to select menu **New** → **Other**, as shown in Figure 14-23 on page 515.

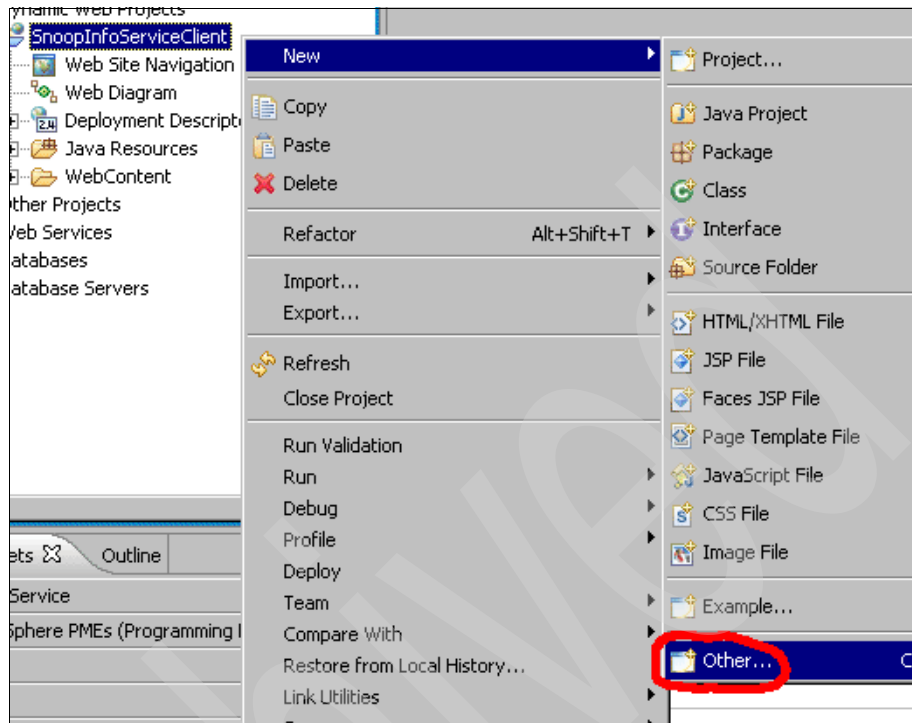


Figure 14-23 Create New Other menu selection

In the appearing New Wizard expand **Simple** and select **File**, as shown in Figure 14-24.

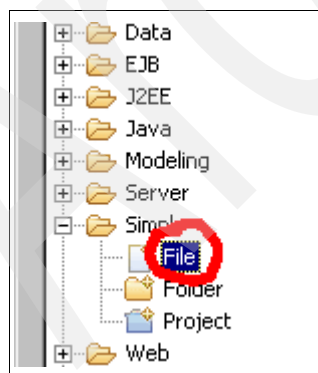


Figure 14-24 New wizard simple file selection

Click **Next**.

Make sure that for the parent folder the project SnoopInfoServiceClient is selected, as shown in Figure 14-25.

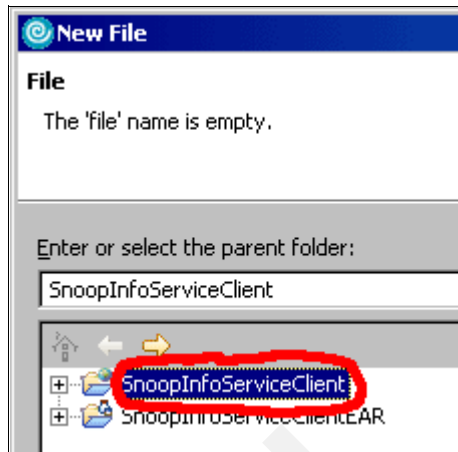


Figure 14-25 New File wizard parent folder selection

Enter download_imsjava_jar.xml as file name, as shown in Figure 14-26.

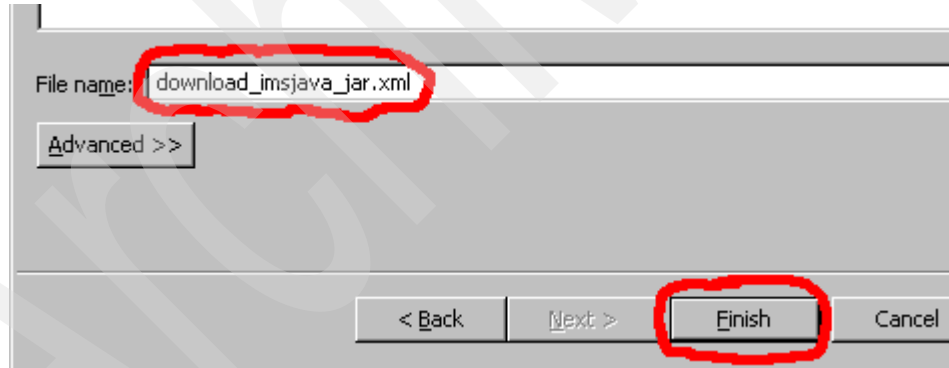


Figure 14-26 New File wizard file name entry

Click **Finish**.

In the appearing editor window enter the code as shown in Example 14-3.

Example 14-3 download_imsjava_jar.xml

```
<?xml version="1.0"?>
<project name="WebServiceProjectClient" default="all" basedir=".">
  <property name="target_dir" value="C:\temp"/>
```

```

<property name="server" value="zoshostname"/>
<property name="source_dir" value="/usr/lpp/ims/imsjava91"/>
<property name="userid" value="userid"/>
<property name="password" value="password"/>
<target name="all">
  <ftp action="get" server="${server}" userid="${userid}"
      password="${password}" remotedir="${source_dir}">
    <fileset dir="C:\temp">
      <include name="imsjava.jar"/>
    </fileset>
  </ftp>
</target>
</project>

```

Next, change the property entries in the first section of the file to suit your environment needs, such as:

- ▶ `target_dir` to represent the directory where `imsjava.jar` should be downloaded to on your local PC
- ▶ `server` to represent the host name of the z/OS system containing `imsjava.jar`
- ▶ `source_dir` to represent the directory on the z/OS machine where `imsjava.jar` is accessible
- ▶ `userid` to represent a valid RACF user ID that is allowed to access the `source_dir` where
- ▶ `password` to represent the corresponding password for the RACF user ID mentioned above

Use the keys **Ctrl+S** or menu **File** → **Save** to save the changes you made.

To execute the script mark the file `download_imsjava_jar.xml` and right-click to select the context menu **Run** → **Ant Build**, as shown in Figure 14-27 on page 518.

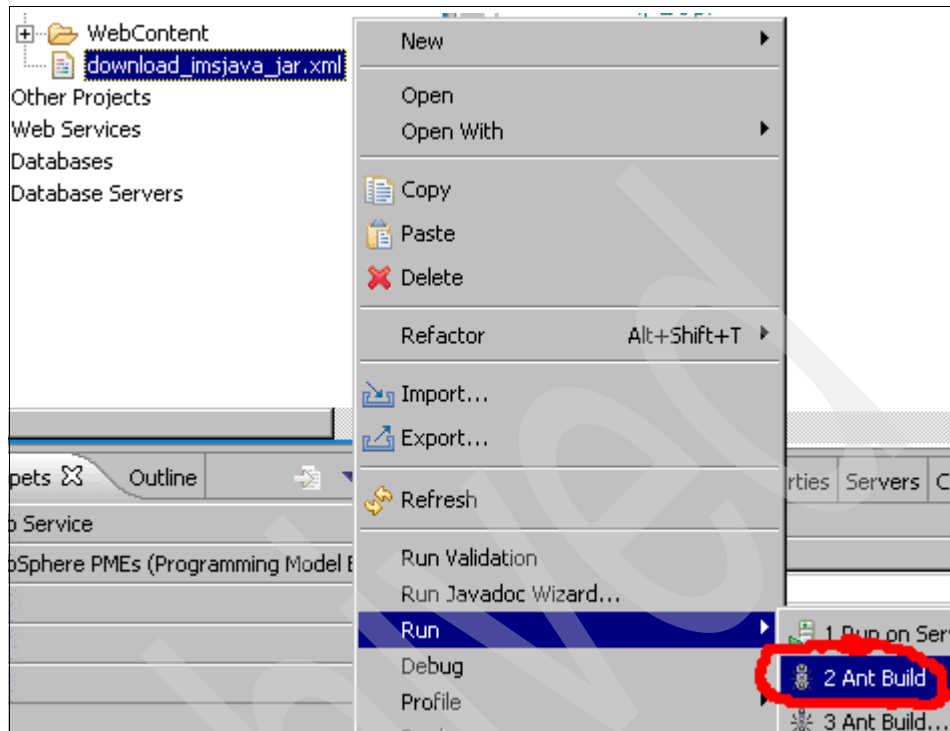


Figure 14-27 Run the Ant script to download imsjava.jar

The Ant script is now being launched and Rational Application Developer will switch to the console view in order to display the output of the ant script. In case there are any errors, those errors will display the console output in the color red. A successful run will look as shown in Figure 14-28.

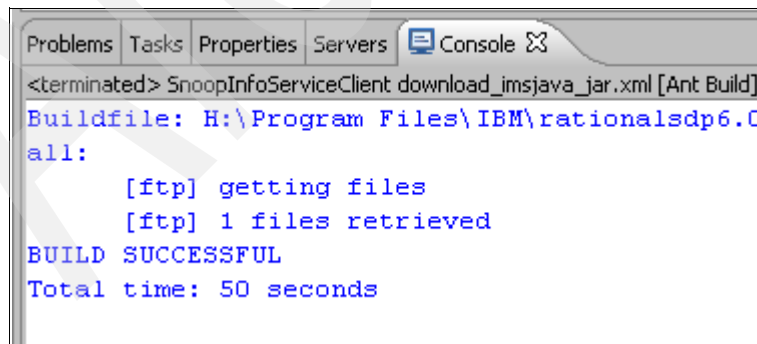


Figure 14-28 Console output of successful ant script run

Now the file is downloaded to the directory that was specified to be the target_dir. In our case the chosen target directory was C:\temp.

Workaround for the bug in Ant FTP task

Since the built-in FTP client does not work for our case, we simply use the FTP client that windows provides. As long as Rational Application Developer is installed on a Windows machine, the following description will work.

We need to create two files—the XML file with the Ant build script (download_imsjava_jar.xml) and the input file with the parameters for the Windows FTP client (ftp.in). An illustrated version on how to create a new file in the workspace can be found in “Downloading imsjava.jar with an Ant script” on page 514.

Select the project name **SnoopInfoServiceClient** by clicking it once with the left mouse button. Use the right mouse button to select **New** → **Other** from the context menu or use the main menu selection **File** → **New** → **Other**. In the appearing New wizard, expand the entry **Simple** and select **File**. Click **Next** and make sure that SnoopInfoServiceClient is selected. Enter download_imsjava_jar.xml for the file name and click **Finish**.

The editor window opens. Insert the following Ant script and change the server property to your z/OS hostname where IMS is installed, as shown in Example 14-4.

Example 14-4 Ant script to invoke the windows FTP client

```
<?xml version="1.0"?>
<project name="SnoopInfoServiceClient" default="all" basedir=".">
  <property name="server" value="zoshostname"/>
  <target name="all">
    <exec executable="ftp" dir=".">
      <arg line="-s:ftp.in ${server}"/>
    </exec>
  </target>
</project>
```

Use the keys Ctrl+S or select **File** → **Save** to save the changes you made.

Create the second file ftp.in and insert the code, as shown in Example 14-5.

Example 14-5 ftp.in file containing the FTP commands

```
userid
password
lcd C:\temp
bin
```

```
cd /usr/lpp/ims/imsjava91
get imsjava.jar
quit
```

Change the entries in ftp.in to suit your environments needs. The entries in order are:

1. userid should be changed to a valid RACF user ID that is allowed to access the HFS directory where imsjava.jar resides.
2. password should be changed to the corresponding password for the RACF user ID mentioned above.
3. The **lcd** command will change to the local directory where imsjava.jar should be downloaded to.
4. Leave bin, as we require a binary download.
5. **cd** changes to the host directory where imsjava.jar is stored. The default directory is /usr/lpp/ims/imsjava91.
6. **get imsjava.jar** is the command to receive the file.
7. quit ends the ftp utility.

Use the keys Ctrl+S or select **File** → **Save** to save the changes that were made.

To execute the script, mark the file download_imsjava_jar.xml and right-click to select the context menu **Run** → **Ant Build**, as shown in Figure 14-29 on page 521.

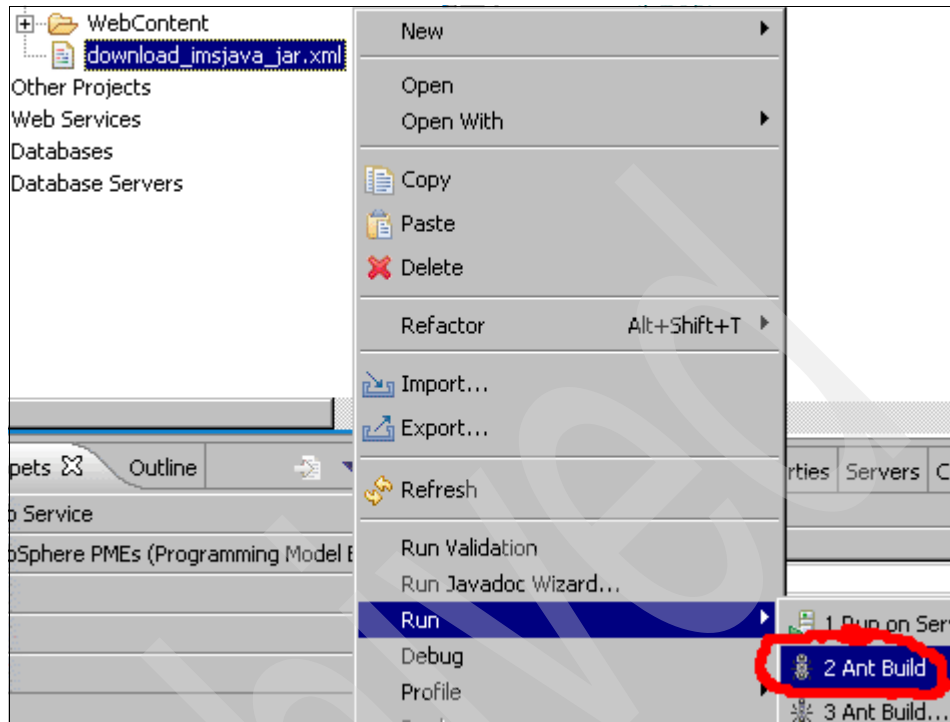


Figure 14-29 Run the Ant script to download *imsjava.jar*

The Ant script is now being launched, and Rational Application Developer will switch to the console view in order to display the output of the Ant script. In case there are any errors, those errors will display in the console output in the color red. A successful run will look like what is shown in Figure 14-30 on page 522.



Figure 14-30 Console output of successful windows ftp ant script run

Now the file is downloaded to the directory that was specified to be the target_dir. In our case the chosen target directory was C:\temp.

Adding imsjava.jar to the project classpath

Now the last thing is to add this external jar file to the project classpath.

To open the classpath settings select the name of the project **SnoopInfoServiceClient**, right-click, and select the bottom most menu, **Properties**. In the appearing Properties window for SnoopInfoServiceClient click **Java Build Path** in the left window, and after that select **Libraries** in the right window, as shown in Figure 14-31.

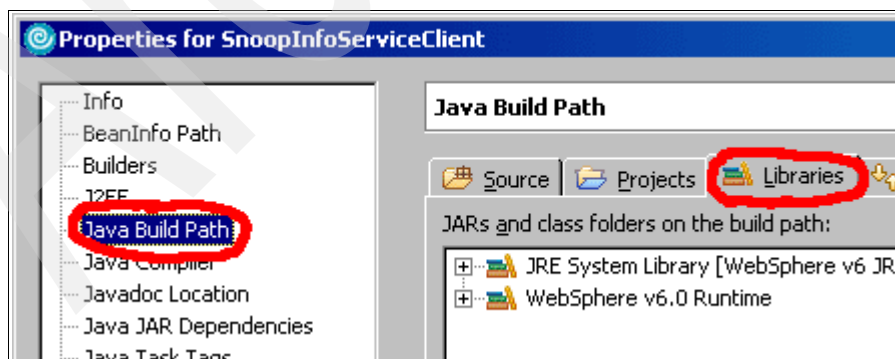


Figure 14-31 Java Build Path selection in the project properties

Click **Add External JARs** on the right side of the window, and in the appearing JAR selection wizard, select the directory where `imsjava.jar` was downloaded to (for example, `C:\temp`). Select the **imsjava.jar** file and click **Open**. The `imsjava.jar` should now be in the list of JARs and class folders on the build path, as shown in Figure 14-32.

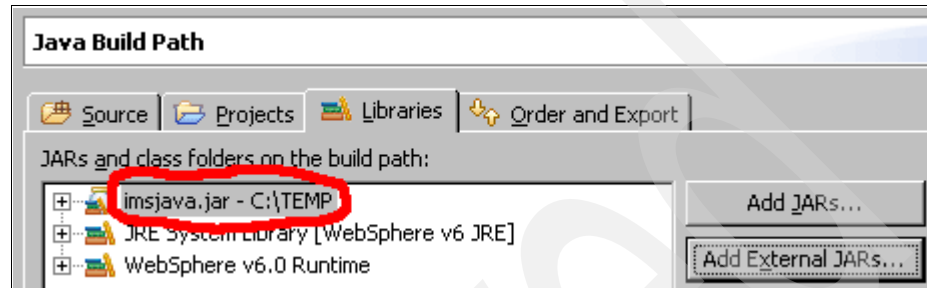


Figure 14-32 Successfully imported `imsjava.jar`

Confirm the changes by clicking **OK** and the Properties windows will close. We are now ready to start coding the IMS Java transaction that will invoke a Web service.

14.4.3 Build the IMS Java transaction to call the Web service

If not already done, create a new package called `itso.ims.ws.snoop`. A description how this is done can be found after Figure 14-8 on page 500.

For the most simple IMS Java transaction, a minimum of two classes is required—one class to contain the Input and Output message definition for the IMS Java transaction and one class to contain the code for invoking the Web service.

IMS Java input and output message definition

All IMS Java transactions require definitions for the input and output messages, similar to the copybook definitions in traditional languages. For the easiest example we create an 80-byte character string definition for the input and output messages.

Create a new class called `IMSJavaMessage` by selecting the `itso.ims.ws.snoop` folder of project `SnoopInfoServiceClient` created earlier and right-clicking to select **New** → **Class**, as shown in Figure 14-33 on page 524.

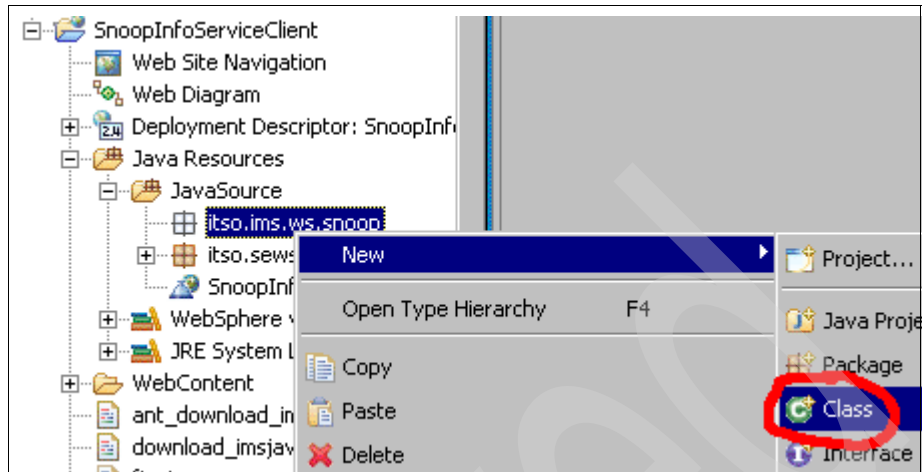


Figure 14-33 Menu selection new class

In the appearing New Java Class wizard make sure that the correct project and package are selected and enter the name of the new class `IMSJavaMessage`, as shown in Figure 14-34.

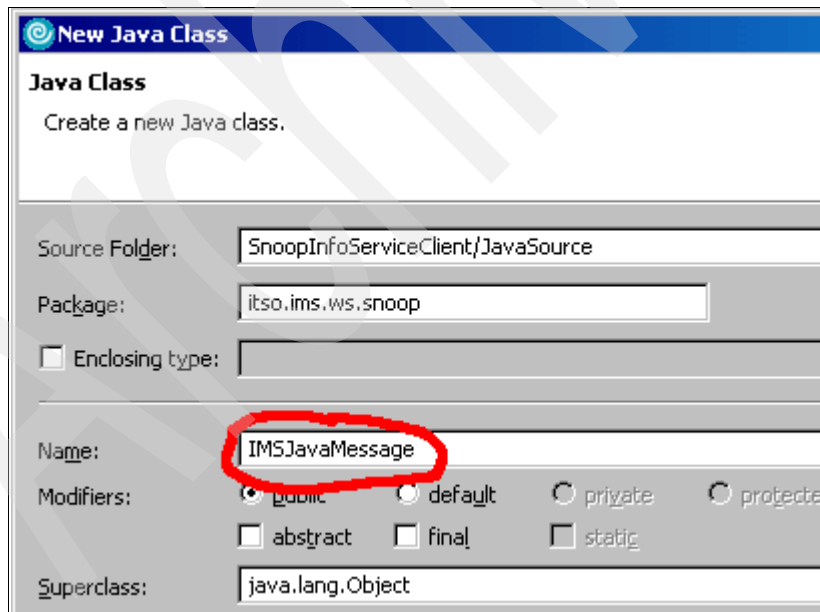


Figure 14-34 Create new `IMSJavaMessage` class

Leave all other fields and selections as defaulted and click **Finish**.

The editor window should now open with an empty class definition. Replace the generated class code as shown in Example 14-6 with the code as shown in Example 14-7.

Example 14-6 Generated IMSJavaMessage class definition to be replaced

```
public class IMSJavaMessage {  
  
}
```

Example 14-7 IMSJavaMessage class definition for IMS Java input/output message

```
import com.ibm.ims.base.*;  
import com.ibm.ims.application.*;  
  
public class IMSJavaMessage extends IMSFieldMessage {  
    static DLTypeInfo[] fieldInfo =  
    {  
        new DLTypeInfo("Message",DLTypeInfo.CHAR,1,80)  
    };  
  
    public IMSJavaMessage() { super(fieldInfo, 80, false); }  
}
```

Press Ctrl+S or click **File** → **Save** to save the changes that were made.

And finally, the structure for a 80-byte character input or output message has been created, similar to the COBOL PIC X(80) definition.

IMS Java transaction coding

Now the main IMS Java transaction class must be created. It simply uses the same code as was used for testing the Web service in. The output strings of the Web service will be displayed using multi-segmented output. In fact there will be six 80-byte output messages containing potential output from the Web service. The input message can be empty or chosen as one likes, since it will not be checked on transaction invocation.

Create a new class called IMSJavaMessage by selecting the **itso.ims.ws.snoop** folder of project SnoopInfoServiceClient created earlier and right-clicking and selecting **New** → **Class**, as shown in Figure 14-35 on page 526.

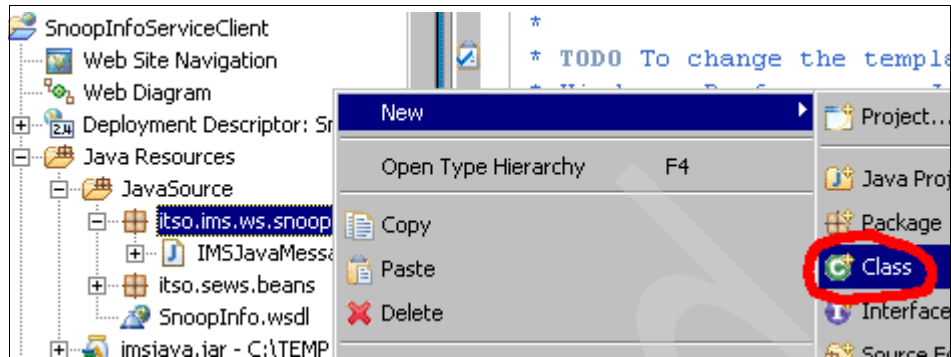


Figure 14-35 Menu selection new class

In the appearing New Java Class wizard make sure that the right project and package are selected and enter the name of the new class, IMSJavacallsWebService.

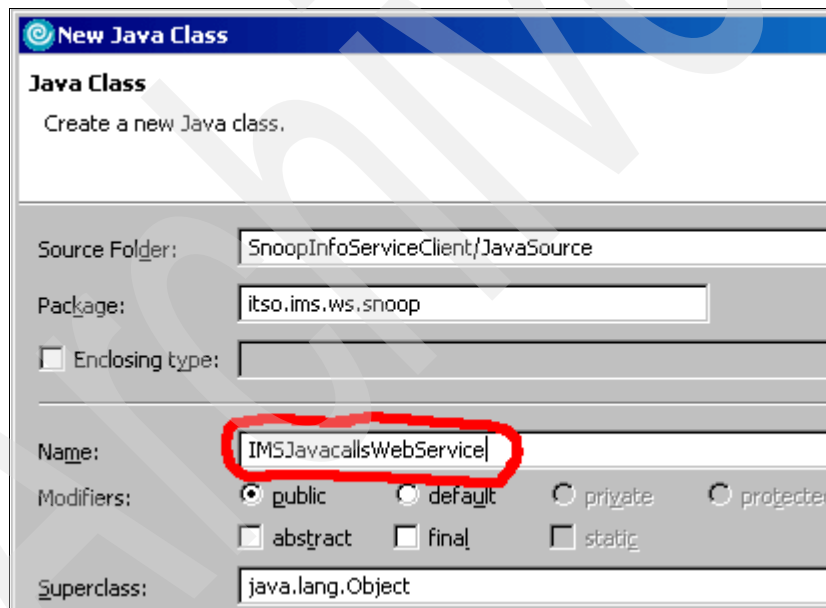


Figure 14-36 Create new IMSJavacallsWebService class

Leave all other fields and selections as defaulted and click **Finish**.

The editor window should now open with an empty class definition. Replace the generated class code shown in Example 14-8 on page 527 with the code shown in Example 14-9 on page 527.

Example 14-8 Generated IMSJavacallsWebService definition to be replaced

```
public class IMSJavacallsWebService {  
  
}
```

Example 14-9 IMSJavacallsWebService Java Transaction code to call the Web service

```
package itso.ims.ws.snoop;  
  
import com.ibm.ims.base.*;  
import com.ibm.ims.application.*;  
import itso.sews.beans.*;  
  
public class IMSJavacallsWebService extends IMSApplication {  
    public IMSJavacallsWebService() {}  
    public void doBegin() throws DLIException, IMSEException {  
        IMSMessageQueue messageQueue = new IMSMessageQueue();  
        IMSJavaMessage inputMessage = new IMSJavaMessage();  
        IMSJavaMessage outputMessage = new IMSJavaMessage();  
        messageQueue.getUniqueMessage(inputMessage);  
        do {  
            try {  
                SnoopInfoProxy proxy = new SnoopInfoProxy();  
                SnoopData output = proxy.getSnoopData();  
                outputMessage.setString("Message", "S1: " + output.getS1());  
                messageQueue.insertMessage(outputMessage);  
                outputMessage.setString("Message", "S2: " + output.getS2());  
                messageQueue.insertMessage(outputMessage);  
                outputMessage.setString("Message", "S3: " + output.getS3());  
                messageQueue.insertMessage(outputMessage);  
                outputMessage.setString("Message", "S4: " + output.getS4());  
                messageQueue.insertMessage(outputMessage);  
                outputMessage.setString("Message", "S5: " + output.getS5());  
                messageQueue.insertMessage(outputMessage);  
                outputMessage.setString("Message", "S6: " + output.getS6());  
                messageQueue.insertMessage(outputMessage);  
            } catch (Exception e) {  
                System.out.println("\nCaught exception is: " + e);  
                e.printStackTrace();  
                outputMessage.setString("Message", "\nCaught exception is: " + e);  
                messageQueue.insertMessage(outputMessage);  
            }  
            IMSTransaction.getTransaction().commit();  
        } while (messageQueue.getNextMessage(inputMessage));  
    }  
  
    public static void main(String args[]) {  
        IMSJavacallsWebService test = new IMSJavacallsWebService();  
    }  
}
```

```

        test.begin();
    }
}

```

The code in Example 14-9 on page 527 does the following:

- ▶ It creates objects for the input and output messages.
- ▶ It then creates the MessageQueue object for the IOPCB, which is required for GetUnique, GetNext, and Insert DL/I calls for the input and output messages.
- ▶ It contains a loop for processing more than one message per schedule. Within the loop it calls the Web service by using the code from the SnoopInfoProxy class used to test the Web service in “Test the Web service” on page 510 and inserts the six result strings as a multisegment message to the IOPCB.

Generating the application jar file and upload

The next step is to create and run an Ant script that creates a jar file out of the IMS Java transaction and uploads it to a host HFS directory.

Select the project name SnoopInfoServiceClient by clicking it once with the left mouse button. Right-click and select **New** → **Other** from the context menu or use main menu selection **File** → **New** → **Other**. In the appearing New wizard expand entry Simple and select **File**. Click **Next** and make sure that SnoopInfoServiceClient is selected. Enter build_imsjava_transaction_jar.xml for the file name and click **Finish**.

The editor window opens. Insert the following Ant script (Example 14-10) and change the properties on top of the file to suit your environment's needs:

- ▶ server to the z/OS hostname where IMS is installed
- ▶ remote_dir to the remote HFS directory where the jar file should be uploaded to
- ▶ userid and password to the RACF's user ID and password

Example 14-10 Ant script to create and upload the IMS Java transaction classes jar file

```

<?xml version="1.0"?>
<project name="SnoopInfoServiceClient" default="all" basedir=".">
  <property name="server" value="zoshostname"/>
  <property name="userid" value="userid"/>
  <property name="password" value="password"/>
  <property name="remote_dir" value="/u/imsjava"/>
  <target name="all">
    <delete file="imsjava_transaction.jar" />
    <jar destfile="imsjava_transaction.jar"
basedir="./WebContent/WEB-INF/classes" includes="*" excludes="*.jar" />

```

```

        <ftp server="${server}" userid="${userid}" password="${password}"
            remotedir="${remote_dir}">
            <fileset file="imsjava_transaction.jar"/>
        </ftp>
    </target>
</project>

```

Press Ctrl+S or select **File** → **Save** to save the changes that were made.

To execute the script mark the file `build_imsjava_transaction.jar.xml` with a mouse click and right-click to select **Run** → **Ant Build** from the context menu.

The Ant script is now being launched, and Rational Application Developer will switch to the console view in order to display the output of the Ant script. In case there are any errors, those errors will display in the console output in the color red. A successful run will look as shown in Figure 14-37.

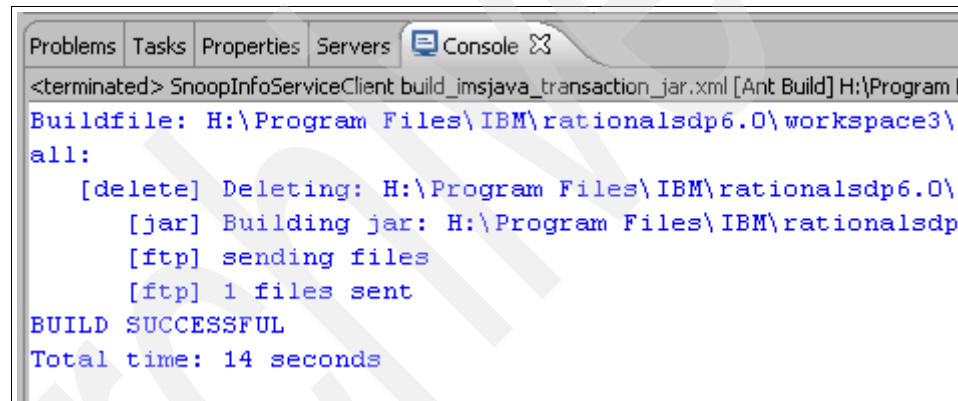


Figure 14-37 Successful run of Ant script uploading jar file with IMS Java code

Now the jar file containing the IMS Java transaction code is successfully uploaded.

14.4.4 Required WAS classes for calling a Web service

To call a Web service using the code that was generated by Rational Application Developer Version 6 it is required to have some classes from the WebSphere V6 runtime in the classpath. For a dynamic Web project, all required classes to call a Web service are already in the classpath. This is not true for IMS Java. IMS Java does not contain the runtime for calling a Web service or for any other operations such as calling EJBs or sending e-mails.

To be able to call a Web service from IMS Java, it is required to have a WebSphere Application Server for z/OS installed or at least a copy of the runtime HFS and the BBOLOAD dataset. It is not required for calling a remote Web service to have a running WebSphere for z/OS at hand—only if the Web service is hosted by WebSphere on z/OS.

Due to the 256-character classpath restriction in IMS Java, we tried to put all jar files from the WebSphere V6 runtime into one jar file. But this approach was not successful. It led to a `NoClassDefFoundError: Invalid Implementation Key` error that could not be resolved directly. The recommended solution is to use the `-Djava.ext.dirs` parameter to include all the directories that contain jar files from the WebSphere Application Server for z/OS Version 6.01 runtime. The `-Djava.ext.dirs` parameter is required to be in both the master and worker JVM configuration member.

The two required directories are `lib` and `installedChannels`, as we show later in “Application settings in IMS PROCLIB” on page 530. In addition, the BBOLOAD dataset has to be in STEPLIB concatenation for the IMS Java region.

It should also be mentioned that the WebSphere runtime classes from a distributed WebSphere do not work due to problems with codepage conversion.

14.4.5 Application settings in IMS PROCLIB

To be able to call the transaction some changes to the IMS Java PROCLIB members are required. This chapter lists the members we used in our system. Make sure that CAPS is set to OFF in ISPF when creating these members or making changes to them.

It is required to add the two jar files created in the previous steps to the `trusted.middleware.classpath` and to the `shareable.application.classpath`, as shown in Example 14-11. This is done by editing the master JVM PROCLIB member, which for our system is `DFSJVMMS`. After editing the member, it looks as shown in Example 14-11.

Example 14-11 Master JVM PROCLIB member DFSJVMMS

```
-Dibm.jvm.shareable.application.classpath=>
/u/denis/imsjava_transaction.jar
-Dibm.jvm.trusted.middleware.classpath=>
/usr/lpp/imsv9/imsjava91/imsjava.jar
-Djava.ext.dirs=/usr/lpp/java/J1.4/lib/ext:>
/usr/lpp/zWebSphereAL1/V6R0/lib:>
/usr/lpp/zWebSphereAL1/V6R0/installedChannels
-Xinitacsh4M
-Xinitsh4M
-Xinitth8M
```



```
-Xmaxf0.6
-Xmine8M
-Xminf0.3
-Xms8M
-Xmx64M
-Xoss400k
```

The worker JVM settings can be left as is. On our test system it look as shown in Example 14-12.

Example 14-12 Worker JVM PROCLIB member DFSJVMWK

```
-Djava.ext.dirs=/usr/lpp/java/J1.4/lib/ext:>
/usr/lpp/zWebSphereAL1/V6R0/lib:>
/usr/lpp/zWebSphereAL1/V6R0/installedChannels
-Xmaxf0.6
-Xminf0.3
-Xms128M
-Xmx1024M
-Xoss400k
-Xinith512M
```

The DFSJVMAP member contains the settings for the mapping between the transaction code in up to eight bytes length and the Java package and class name containing the IMS Java application code, which can be much longer than eight bytes. Example 14-13 shows the member from our test system.

Example 14-13 Java mapping PROCLIB member DFSJVMAP

```
TESTJAVA=itso/ims/ws/snoop/IMSJavacallsWebService
```

In our test system we used the transaction code TESTJAVA with the definitions shown in Example 14-14 in the IMS Sysgen.

Example 14-14 Transaction definition for TESTJAVA in STAGE1 source

```
APPLCTN PSB=TESTJAVA,PGMTYPE=TP,SCHDTYP=PARALLEL
      TRANSACT CODE=TESTJAVA,MODE=SNGL,EDIT=(ULC),
      MSGTYPE=(MULTSEG,RESPONSE,1),MAXRGN=0,
      PARLIM=0,PROCLIM=(10,100)
      SPACE 2
```

The PSB definition, as shown in Example 14-15, was used for TESTJAVA.

Example 14-15 PSB TESTJAVA

```
ALTERPCB PCB TYPE=TP,MODIFY=YES
PSBGEN LANG=JAVA,PSBNAME=TESTJAVA
```

To be complete with the listings, the PROCLIB member containing the settings for the Unix System Services environment that IMS Java is running under is shown in Example 14-16. Please note that the path to the dynamic link library path of the WebSphere Application Server for z/OS Version 6.01 runtime is included, since the jar files of the WebSphere Application Server for z/OS Version 6.01 runtime use some functionality through JNI from DLLs that are in the lib directory.

Example 14-16 Environment PROCLIB member DFSJVMEV

```
PATH=/usr/lpp/java/J1.4/bin:./bin:>
/u/denis
LIBPATH=/usr/lpp/imsv9/imsjava91:>
/usr/lpp/java/J1.4/bin:>
/usr/lpp/java/J1.4/bin/classic:>
/usr/lpp/zWebSphereAL1/V6R0/lib
```

After applying the required changes, make sure that the application server hosting the accessed Web service is up and running.

In addition, start or restart the IMS Java Region (JMP) to make sure that the transaction runs with the updated application jar file and the updated classpath.

For reference, the JCL for the Java region startup is listed in Example 14-17. Please make sure that for testing purposes the DFSJMP procedure in IMS PROCLIB is edited to comment out the SYSUDUMP DD statement, since Java exceptions lead to U101 abends and the JES Spool will get filled.

Example 14-17 IMS Java region startup JCL

```
//IMS9JAV1 JOB ACTINF01,
// 'TMCCIMS9',
// CLASS=A,MSGCLASS=X,MSGLEVEL=(1,1),
// REGION=OM
/*JOBPARM L=9999,SYSAFF=*
//PROC04 JCLLIB ORDER=(IMS910A.PROCLIB)
//*
//IMS9M1 EXEC PROC=DFSJMP,TIME=(1440),
//      AGN=IVP,          AGN NAME
//      NBA=6,
//      OBA=5,
//      TLIM=10,          MPR TERMINATION LIMIT
//      IMSID=IMSA,       IMSID OF IMS CONTROL REGION
//      JVMOPMAS=DFSJVMMS, MASTER JVM MEMBER
//      JVMOPWKR=DFSJVMWK, WORKER JVM MEMBER
//      ENVIRON=DFSJVMEV,  USS ENVIRONMENT MEMBER
```

```
//          XPLINK=Y,                XPLINK FOR JDK 1.4
//          RGN=OM
// *
//DFSCTL DD DISP=SHR,
//          DSN=IMS910A.PROCLIB(DFSSBPRM)
//DFSSTAT DD SYSOUT=*
//SYSOUT DD SYSOUT=*
//SYSABOUT DD SYSOUT=*
// *
//JAVAOUT DD PATH='/tmp/imsjavaout1',PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
//          PATHMODE=(SIRWXO,SIRWXG,SIRWXU)
// *
//JAVAERR DD PATH='/tmp/imsjavaerr1',PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
//          PATHMODE=(SIRWXO,SIRWXG,SIRWXU)
// *
//RESETEV DD PATH='/tmp/resetevents1.txt',
//          PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
//          PATHMODE=(SIRWXO,SIRWXG,SIRWXU)
```

At this point, all the configuration in IMS should be in place to run the IMS Java transaction.

14.4.6 Testing the IMS Java transaction from a 3270 terminal

When first trying to execute the application on our system we got the following error:

```
ICH408I USER(DENIS  ) GROUP(SYS1  ) NAME(DENIS GAEBLER  ) 488
/tmp/IBM_JVM_GLOBAL_MONITOR_0084674641
CL(FSOBJ  ) FID(00000000000000000000ED400000000)
INSUFFICIENT AUTHORITY TO OPEN
ACCESS INTENT(RW-) ACCESS ALLOWED(GROUP  R--)
EFFECTIVE UID(0000006803) EFFECTIVE GID(0000000000)
```

The JVM creates a file, which is used for communication between master and worker JVM. The monitor files are created at region startup, thus using the user ID assigned to the IMS region (in most cases a started task user ID). The files are created, allowing write access only to the creator. The code running in an IMS region always runs with the caller's user ID, so this leads to the error message. This problem is a JDK issue and was fixed with JDK 1.3 with PTF for APAR PK07167.

To resolve this, it is required to have PTF for APAR PK07167 applied to JDK 1.4 (as soon as it is available, PTF likely comes with SR3 for JDK 1.4) or to run the transaction with a user ID that has UID 0. We did test the application with a user ID with UID 0.

The sequence for testing the application is as follows:

1. Log on to IMS.
2. Enter the transaction code for the IMS Java transaction. In our case it is TESTJAVA.



Figure 14-38 Run the IMS Java transaction from a 3270 terminal

3. The output result should be similar to the output in Rational Application Developer Version 6 when testing the Java proxy. The output is likely truncated due to the 80-character limitation per output message.

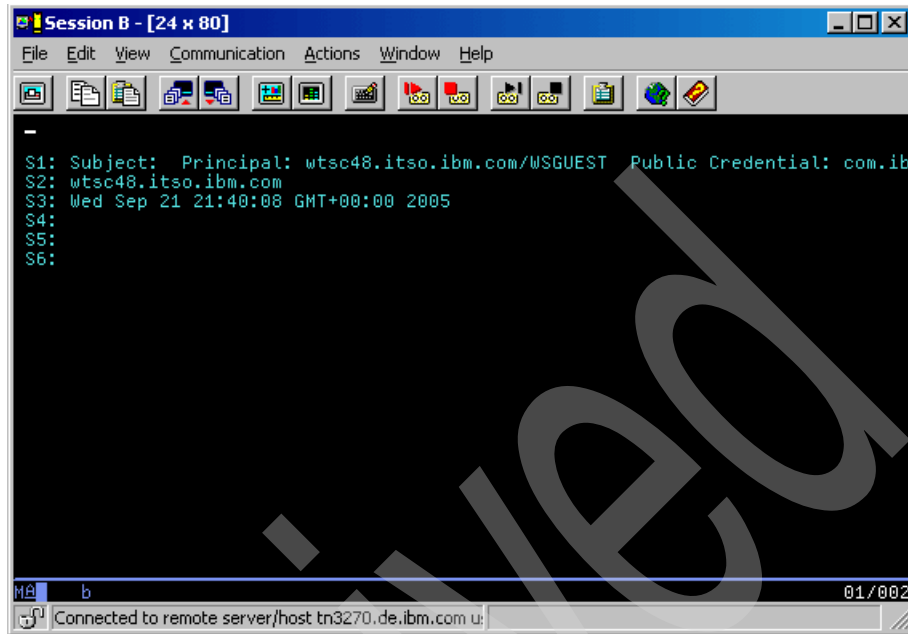


Figure 14-39 Output result from calling the Web service in an IMS Java transaction

Now the Web service was successfully called from an IMS Java transaction that was invoked from a 3270 terminal. One could easily add a MFS screen to have a better screen format or use other options to call the transaction, as explained in “Test IMS Java transaction” on page 535.

14.4.7 Test IMS Java transaction

It was also tested to invoke the IMS Java transaction using a Java program that uses IMS Connector for Java and IMS Connect to call the transaction through OTMA. However, since this is just another way of calling an IMS transaction, just like using WebSphere MQ middleware to call an IMS transaction, it is not discussed here.

Note: It should be noted that an IMS Java transaction can be called just like any other transaction using any LU capable protocol, APPC, OTMA (for example, WebSphere MQ, IMS Connect), and from VTAM® terminals. Furthermore, it is possible to create an MFS screen to use 3270 screen masks for input and output of the IMS Java transaction.

14.4.8 Automatic class reloading for testing

IMS Java has no option to support automatic class reloading. Thus, if there is a new jar file uploaded from your IMS Java development environment, it is required to recycle the IMS Java region in order to get the jar files refreshed. But there is a trick. If the classpath of the worker JVM is coded with a `-Djava.class.path` statement and there are all jar files added with application code (not the jars one should put into the `trusted.middleware.classpath`), they will be loaded on every invocation of the transaction. However, those classes need to be removed from the `shareable.application.classpath` in the master JVM settings since this classpath is first in search order and if the classes are found in the master JVM, automatic class reloading will not work.

Important: This option for automatic class reloading should only be used in development environments, since reloading the classes with every invocation of a transaction will result in very poor throughput.

Using the RMI-IIOP protocol

In this chapter we use a hello world type application to show how to use the J2EE RMI over IIOP protocol to call an EJB.

RMI over IIOP is a synchronous communication protocol that can be used only between a client and a server or between a server and another server that are both J2EE-capable. If you use RMI-IIOP directly from a client, the client must be a full J2EE client; it cannot be a browser. Using RMI-IIOP for connectivity is an architecturally totally different approach as using messaging technology. The following table illustrates the main differences.

Table 15-1 Main differences between RMI-IIOP and messaging

| | J2EE RMI-IIOP | Messaging (JMS) |
|--------------------|--|--|
| Client | Must be J2EE client | Must be J2EE client |
| Target server | Must be J2EE server | Can be any server serviced by WebSphere MQ |
| Nature | Synchronous | Asynchronous |
| Transport protocol | Native J2EE | |
| Transactionality | Fully transactional with implicit 2-PC | Transactionality only guaranteed until MQ server |

| | J2EE RMI-IIOP | Messaging (JMS) |
|----------------------|----------------------|------------------------|
| Suitable for ESB/SOA | No | Yes |

In this chapter we show two scenarios, both based on a simple sample, called `ejbMagic`:

- ▶ Calling an EJB from a client within the same cell, described in “Running `ejbMagic` in one server” on page 547
- ▶ Calling an EJB from a client in different cells, described in “Running `ejbMagic` between two cells” on page 551

In “Calling EJBs and IIOP” on page 539 we show the development aspects and in “CSI” on page 559 we give some information about the infrastructure used by the RMI-IIOP protocol.

15.1 Calling EJBs and IIOP

In the following sections we describe how we built our ejbMagic sample application using RMI-IIOP, but first we explain a few more basics.

15.1.1 Basics of remote EJB invocation

EJBs are a key component of the J2EE specification. Many applications typically consist of a servlet and one or more EJBs, with the application deployed into a WebSphere server. As the application is completely contained within the server, when the servlet calls the EJB, it is a local call that occurs within the server. No communications occurs over TCP/IP, for example.

A key concept of the J2EE specification, however, is that methods in the EJB can be called from remote clients. A remote client can be a stand-alone Java program or it can be a servlet or EJB in another WebSphere server or cell. When remote clients invoke a remote EJB, the communication between the client and the server is done over TCP/IP using a protocol referred to as RMI over IIOP.

As an application programmer who wants to invoke remote EJBs, the good news is that you do not need to understand anything at all about the inner mysteries of the RMI over IIOP protocol, as the communication part of the application is all handled by generated Java classes. As a developer you call a remote method as if it were locally available.

Furthermore, if you are an application developer who has a servlet and/or EJB in one WebSphere server or cell, that wants to call a method in a remote EJB in another WebSphere server, then the process is the same as calling a method in a local EJB in the same server. In this case, the only additional requirements to make this all work are for your WebSphere Administrators to configure the “plumbing” between the two participants. “Running ejbMagic in one server” on page 547 and “Running ejbMagic between two cells” on page 551 show how to set up this plumbing and how to simplify the calling of remote EJBs.

15.1.2 Supplied material

Table 15-2 describes the additional material supplied with this book.

Table 15-2 Supplied additional material

| Name | Purpose |
|--------------|---|
| ejbMagic.zip | This is a project interchange zip file. It contains the ejbMagic application. |

| Name | Purpose |
|------------------------|---|
| sews-stage1.zip | This is a project interchange zip file. It contains the sews project after the steps described in “Web service-enable SnoopInfo Java Bean” on page 370 have been completed. |
| sewsWsDemo-initial.zip | This is a project interchange zip file. It contains the application that runs in WebSphere, which is used to invoke Web Services. It is the starting point for the process described in “Development of Web service invoker” on page 383. |
| sewsWsDemo-stage1.ear | This is the ear file that you can deploy into WebSphere to start calling the Web services in the sews.ear or in CICS. |
| SEWS001 | COBOL source code for the program. |
| SEWS002 | COBOL source code for the program. |
| SEWSBMS | COBOL source code for the program. |

15.1.3 ejbMagic application

To demonstrate invoking methods on remote EJBs, we will a simple application, which we refer to throughout this chapter as ejbMagic.

The ejbMagic application consists of the following components (Table 15-3).

Table 15-3 *ejbMagic components*

| Component | Purpose |
|-----------------------|---|
| Java Bean - SnoopBean | Java Bean that returns some basic information |
| Servlet - EjbDemo | Servlet that calls method on the SnoopInfo Java Bean, and calls the SnoopMagicEJB |
| EJB - SnoopMagicEJB | EJB that calls method on the SnoopBean Java Bean |

When the servlet is invoked it displays in the browser a form. You then select on the form various options to control what you want the ejbMagic application to do.

When the form button is pressed the servlet will perform the following actions:

1. The EjbDemo servlet creates a SnoopBean and calls the getSnoopData method, which will return some basic information.

2. The EjbDemo servlet then calls the SnoopMagicEjb locally, invoking a nominated method. This method will create a SnoopBean, call the getSnoopData method, and return that basic information.
3. The local SnoopMagicEjb EJB will then call a remote SnoopMagicEjb EJB and invoke on this remote EJB a nominated method. This method will create a SnoopBean, call the getSnoopData method, and return the basic information.
4. The EjbDemo servlet then also calls the remote SnoopMagicEjb EJB, and invokes on this remote EJB a nominated method. This method will create a SnoopBean, call the getSnoopData method, and return the basic information.
5. The EjbDemo servlet then displays the results in the browser.

The above means that each time you run the servlet in EjbMagic, the servlet calls a local EJB and a remote EJB, and the local EJB calls a remote EJB.

The calls to the remote EJBs can in fact just be calls to a local EJB. The application lets you provide input to control where the remote EJB call actually goes to.

This application can thus be used to help you set up inter-WebSphere settings to allow remote EJB calls to function, check out security settings, and verify all is working before moving on to implement actual applications that have real business logic in them.

The ejbMagic application can be deployed into multiple WebSphere cells, and then used to call itself to help you verify inter-WebSphere cell communication.

15.1.4 SnoopMagicEjb methods

The SnoopMagicEjb EJB consists of a method called snoopInfo. This is the method that gathers the information such as Java principal, etc., and calls the remote EJB. However, this method is not called directly. Rather, four other methods are defined in the EJB, all with names that correspond to different runAs capabilities. In the ejb-jar.xml, these methods have been configured to run with the runAs attribute set to a value corresponding to their name. Table 15-4 shows these methods.

Table 15-4 Methods of the SnoopMagicEjb

| Method name | Description |
|---------------|--|
| siNoRunAs | Method has no runAs attribute set |
| siRunAsServer | Method has runAs attribute set to Server |
| siRunAsCaller | Method has runAs attribute set to Caller |

| Method name | Description |
|-------------------|--|
| siRunAsRoleWorker | Method has runAs attribute set to run as EJBROLE of Worker |

15.1.5 Understanding logical and direct JNDI lookups

It is important that you understand the difference between an application program performing a logical lookup for a resource and a direct lookup for a resource.

To invoke a method on an EJB, whether the EJB is local or remote, your application first needs to locate the EJB. This is done in the ejbMagic application using these two lines:

```
Context ic = new InitialContext();
oHome = ic.lookup(localJndiName);
```

An EJB in WebSphere is a resource, and it has a JNDI name associated with it. In the second line, localJndiName contains the lookup value. WebSphere will use this value passed to it to try and locate a resource that corresponds to it.

Direct lookup

If you pass, for example, the value:

```
ejb/itso/em/ejb/SnoopMagicEjbHome
```

then this represents an actual JNDI name of a resource, in this case an EJB. WebSphere will look through its namespace to find a match; if it does find a match then it will return a reference to that resource.

Logical lookup

It is not, however, best practice to use a direct lookup for a resource. Doing so can mean that your application has hard coded references to resources. If the JNDI names for these resources change, then you need to update the application to reflect that.

The recommended approach is to use a logical lookup value. All logical lookup values start with the string *java:comp/env*. The value we use in the ejbMagic application to locate the local ejbMagic EJB is:

```
java:comp/env/ejb/localEjbMagic
```

You may be wondering where the ejb/localEjbmagic part of this value comes from. This is the other key part having to do with using logical lookups. In your application, you define logical resource references in the a file called the deployment descriptor.

An application that has a servlet part and an EJB part will have a deployment descriptor file called `web.xml` for the servlet part, and a deployment descriptor file called `ejb-jar.xml` for the EJB part.

The `web.xml` deployment descriptor

Figure 15-1 shows the location of the `web.xml` file in the `ejbMagic` project.

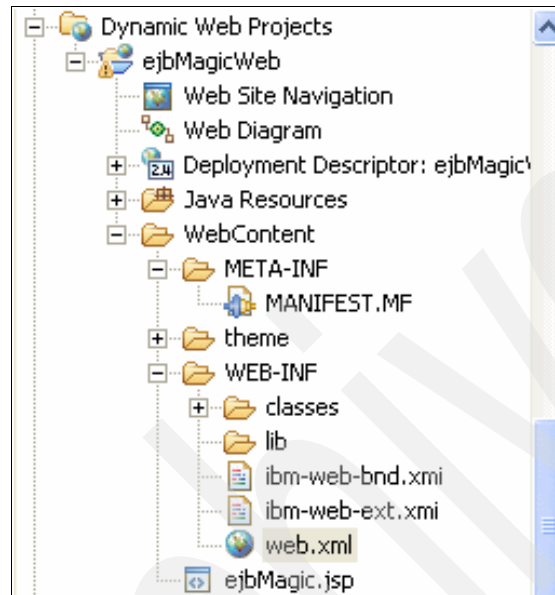


Figure 15-1 Location of `web.xml` file

In Rational Application Developer Version 6, double-clicking the `web.xml` file will open it up for updating. Click the **References** tab and you will see a window similar to that shown in Figure 15-2 on page 544.

References

This web application references the following resources:

EjbRef ejb/localEjbMagic

EjbRef ejb/remoteEjbMag

Add...

Remove

Name: ejb/localEjbMagic

Description:

Link: ejbMagicEJB.jar#SnoopMagicEjb

Type: Session

Home: itso.em.ejb.SnoopMagicEjbHome

Remote: itso.em.ejb.SnoopMagicEjb

WebSphere Bindings

The following are binding properties for the WebSp

JNDI name: ejb/itso/em/ejb/SnoopMagicEjbHome

Figure 15-2 web.xml - Reference tab

The value for Name is the logical name for the resource. The resource selected in Figure 15-2 has a name of ejb/localEjbMagic. This is the logical reference to the snoopMagicEjb EJB. It is this value that then gets combined with java:comp/env to form the lookup value of:

```
java:comp/env/ejb/localEjbMagic
```

If, for example, we changed the name in the web.xml to a value of someEjb/itso/redbookEjb, then the lookup value would be:

```
java:comp/env/someEjb/itso/redbookEjb
```

Also, in Figure 15-2 you can see a heading called WebSphere Bindings. This is an IBM extension to the standard structure of an ear file. This is where you can set the actual JNDI name of the SnoopMagicEjb EJB that will be associated with the EJB in the WebSphere namespace. Setting the value here saves you having to supply it during deployment of the ear file into WebSphere. This JNDI name is saved in a file called *ibm-web-bnd.xml*.

EJB JNDI name

Each EJB deployed into WebSphere is a resource, in the same way a JMS destination is a resource. Each EJB must have a JNDI name associated with it.

During deployment of an application with EJBs, WebSphere will prompt you to set a JNDI name for any EJBs it finds.

In Rational Application Developer Version 6 you can preset the JNDI name you want to be associated with the EJB. Figure 15-3 shows the Bean tab of the ejb-jar.xml file.

The screenshot shows the 'Bean' configuration window for 'SnoopMagicEjb'. The 'Bean Type' is 'Session 2.x', 'Type options' is 'Stateless', 'Transaction type' is 'Container', 'Display name' is empty, and 'Description' is empty. Under the 'WebSphere Bindings' section, the 'JNDI name' is set to 'ejb/itso/em/ejb/SnoopMagicEjbHome'.

| | |
|--|-----------------------------------|
| Bean Type: | Session 2.x |
| Type options: | Stateless |
| Transaction type: | Container |
| Display name: | |
| Description: | |
| WebSphere Bindings | |
| The following are binding properties for the WebSphere | |
| JNDI name: | ejb/itso/em/ejb/SnoopMagicEjbHome |

Figure 15-3 ejb-jar.xml - Bean tab

Under the heading WebSphere Bindings is where you can preset the JNDI name you want to have associated with the EJB resource. Rational Application Developer Version 6 saves this information in a file called ibm-ejb-jar.xmi.

Binding - Logical to actual resource

Now when the application performs the lookup passing the logical lookup value, WebSphere needs to resolve this to an actual resource. How does WebSphere know to which actual resource the logical resource defined in the web.xml corresponds to?

WebSphere knows the answer to this question, as it occurred during deployment of the application. When the application was deployed, WebSphere detects any logical resource references in the web.xml and ejb-jar.xml deployment descriptors. In the WebSphere administration GUI, WebSphere will indicate that it has found these logical resource references and ask you to select a corresponding existing resource to *bind* the logical resource definition to.

This binding process is how (later on when the application runs and performs a logical resource lookup) WebSphere is able to locate the actual resource that the logical resource represents.

15.1.6 ejbMagic application - Logical EJB references

In the ejbMagic application, the web.xml defines two logical resource references as follows:

ejb/localEjbMagic Logical reference to local snoopMagicEjb

ejb/remoteEjbMagic Logical reference to remote snoopMagicEjb

These are used by the servlet to call the snoopMagicEjb locally in the server, and call the same EJB either in the same server or remotely in some other server. We will see later in this chapter how we can update the actual JNDI name associated with the ejb/RemoteEjbMagic logical resource, to have the servlet call to the EJB become a remote call.

In the ejb-jar.xml is defined one logical resource reference as follows:

ejb/remoteEjbMagic Logical reference to remote snoopMagicEjb

This definition is used by the locally called snoopMagicEjb to also call the snoopMagicEjb. What JNDI name we actual set for this logical resource will control whether the call is to another instance of the EJB in the same server, or to an instance of the EJB in some other server. We will see later in this chapter how we control this by way of the value we set for the actual JNDI name associated with this resource.

15.1.7 Our system layout

Throughout this chapter we will show the various definitions and settings we used on our system to demonstrate the fundamentals of IIOP and EJBs.

WebSphere server - ws6481

The ws6481 server is a WebSphere Application Server for z/OS Version 6.01 stand-alone server. It is listening on port 9080 for standard HTTP requests. It will have the ejbMagic application deployed into it. This server does have global security enabled; Java 2 security is not enabled.

WebSphere server - ws6483

The ws6483 server is a WebSphere Application Server for z/OS Version 6.01 stand-alone server. It is listening on port 39080 for standard HTTP requests. It will have the ejbMagic application deployed into it. This server does have global security enabled; Java 2 security is not enabled.

15.2 Running ejbMagic in one server

As a first step, we run the ejbMagic application in a single WebSphere server. We deployed the application into the ws6481 server. Deployment involved nothing more than locating the ear and clicking through to the final step and saving the changes. The server was then restarted. The purpose of running the application within a single server to begin with is just to ensure that it is running, before trying to use it to invoke another copy of the ejbMagic application installed in a different cell.

15.2.1 URL to invoke the servlet

Enter the following URL to invoke the servlet:

`http://9.12.4.38:9080/ejbMagicWeb/ejbDemo`

When invoked, the servlet will display the output shown in Figure 15-4 on page 548.

EJBMagic - Demo

| Select JNDI value to use to call remote EJB | |
|--|-------------------------------------|
| <input checked="" type="radio"/> | Use logical EJB reference resource |
| <input type="radio"/> | Use JNDI override value |
| <input type="text"/> | |
| Select method in the local EJB to invoke | |
| <input type="radio"/> | method with no Run As |
| <input checked="" type="radio"/> | method with runAs of server |
| <input type="radio"/> | method with runAs of caller |
| <input type="radio"/> | method with runAs role of worker |
| Select method in the remote EJB that local EJB will invoke | |
| <input type="radio"/> | method with no Run As |
| <input type="radio"/> | method with runAs of server |
| <input checked="" type="radio"/> | method with runAs of caller |
| <input type="radio"/> | method with runAs of Role of worker |

Let's do some magic

Figure 15-4 Initial display of the ejbDemo servlet

When running the servlet there are three input areas where an action is required.

Input area 1

Under the heading “Select JNDI value to use to call remote EJB” are two choices:

- ▶ Selecting the choice **Use logical EJB reference resource** will result in a logical lookup for the remote EJB from the servlet and local EJB being done with a value of:

```
java:comp/env/ejb/remoteEjbMagic
```

- ▶ Selecting the choice **Use JNDI override value** allows you to enter the JNDI value to be used for the lookup of the remote EJB by the servlet and local EJB. Later in this chapter we will be using values here to invoke the EJB in remote servers.

Input Area 2

Select the runAs of the method to invoke in the local EJB called by the servlet.

Input Area 3

Select the runAs of the method to be invoked in the remote EJB from the servlet and local EJB.

15.2.2 Testing the ejbMagic application

Select the following choices:

- ▶ Use logical EJB reference resource.
- ▶ Method with runAs of server to invoke in the local EJB.
- ▶ Method with runAs of caller to invoke in the remote EJB.

Excerpts of the result are as shown in Table 15-5 on page 550.

Servlet Snoop Information

| | |
|-------------------|---|
| Java Principal | Subject: Principal: wtsc48.itso.ibm.com/WSGUEST Credential: com.ibm.ws.security.auth.WSCredentialImpl@11637 [com.ibm.ws.security.auth.zOSWSCredentialImpl@ |
|-------------------|---|

Servlet call to local SnoopMagicEjb Information

| | |
|-------------------|---|
| Java Principal | Subject: Principal: wtsc48.itso.ibm.com/ASSR1 Pu Credential: com.ibm.ws.security.auth.WSCredentialImpl@576 [com.ibm.ws.security.auth.zOSWSCredentialImpl@ [wssecurity.identity_name, wssecurity.identity_valu com.ibm.ws.security.zos.PlatformCredential]] Priva com.ibm.ws.security.token.SingleSignonTokenImpl@ |
|-------------------|---|

Local EJB call to remote SnoopMagicEjb Information

| | |
|-------------------|--|
| Java Principal | Subject: Principal: wtsc48.itso.ibm.com/ASSR1 Public Credential: com.ibm.ws.security.auth.WSCredentialImpl@5767701 [com.ibm.ws.security.auth.zOSWSCredentialImpl@50cb70 [wssecurity.identity_name, wssecurity.identity_value, com.ibm.ws.security.zos.PlatformCredential]] Private Creden com.ibm.ws.security.token.SingleSignonTokenImpl@11b9a2 |
|-------------------|--|

Servlet call to remote SnoopMagicEjb Information

| | |
|-------------------|--|
| Java Principal | Subject: Principal: wtsc48.itso.ibm.com/WSGUEST P Credential: com.ibm.ws.security.auth.WSCredentialImpl@1163b7 [com.ibm.ws.security.auth.zOSWSCredentialImpl@11 |
|-------------------|--|

Figure 15-5 Result of running ejbMagic in single server

15.3 Running ejbMagic between two cells

In this section we look at using the ejbMagic application to call another copy of itself in a WebSphere server in a different cell.

The ejbMagic application is deployed into the ws6481 and ws6483 servers. Each of these servers is in its own cell. We do not discuss the use of a Network Deployment cell in this book, but the principles that apply here with these single server cells are the same for a Network Deployment cell of any number of servers.

15.3.1 Call remote EJB test

Invoke the ejbDemo servlet in the ws6481 server.

We selected the following choices:

- ▶ Use JNDI override value.
- ▶ Method with runAs of server to invoke in the local EJB.
- ▶ Method with runAs of caller to invoke in the remote EJB.

We are using the Use JNDI override value option (as we want to pass a value to WebSphere from our program) that it can use to locate the SnoopMagicEjb in a remote server.

The value we use is:

```
corbaname::9.12.4.38:32809/NameServiceServerRoot#cell/nodes/nd6483/servers/  
ws6483/ejb/itso/em/ejb/SnoopMagicEjbHome
```

This value tells the ejbMagic application running in the ws6481 server to pass this lookup value to WebSphere to locate the remote EJB in the ws6483 server. The 32809 port is the Bootstrap port of the ws6483 server.

You may be wondering how we worked out the structure of this value for the URL. It is part of the way WebSphere implements the lookup process for initial contexts. Further information can be found at:

http://publib.boulder.ibm.com/infocenter/wasinfo/v6r0/index.jsp?topic=/com.ibm.websphere.base.doc/info/aes/ae/rnam_example_prop1.html

CBIND prevents access

We ran the request but found that it failed. Looking in the job log of the ws6483 control region we saw the messages shown in Example 15-1.

Example 15-1 RACF violation when servlet in ws6481 calls EJB in ws6483

```
ICH408I USER(WSGUEST ) GROUP(WSCLG ) NAME(WAS DEFAULT USER
```

```
CB.CLU6483 CL(CBIND )
INSUFFICIENT ACCESS AUTHORITY
ACCESS INTENT(READ ) ACCESS ALLOWED(NONE )
BBOS0002E CBIND CHECK FAILED WITH SAF RETURN CODE=00000008,
RACF RETURN CODE=00000008, RACF REASON CODE=00000000.
```

The servlet is running in the ws6481 server as unauthenticated, since we have not set any security profile around the URL used to run the servlet. This means that WebSphere assigns the default user ID set for the ws6481 server as the Java principal for the servlet.

In the ws6481 WebSphere Administrative console, expand **Security**, click **Global security**, under User registries click **LocalOS**, under Additional Properties click **z/OS SAF properties**, and the value of the default user ID can be viewed. In our ws6481 server it is set to WSGUEST.

Thus our servlet runs under this default user ID of WSGUEST. When the servlet tries to call the remote EJB in the ws6483 server, the exception shown in Example 15-1 on page 551 occurs.

Updating CBIND rule

The BIND RACF class is used to control who can connect to a WebSphere server via the IIOP protocol.

We issued this command to allow the WSGUEST user ID to bind to the WS6483 server:

```
PERMIT CB.CLU6483 CLASS(CBIND) ID(WSGUEST) ACC(READ)
```

A RACF SETROPTS was then required.

We then reran the servlet.

CBIND violation 2

Re-running the servlet produced a new violation in the ws6481 server, shown in Example 15-2.

Example 15-2 Another CBIND violation

```
ICH408I USER(ASCR1 ) GROUP(WSCFG1 ) NAME(WAS APPSVR CR
CB.CTFMVS09.CLU6481 CL(CBIND )
INSUFFICIENT ACCESS AUTHORITY
FROM CB.* (G)
ACCESS INTENT(READ ) ACCESS ALLOWED(NONE )
BBOS0002E CBIND CHECK FAILED WITH SAF RETURN CODE=00000008,
RACF RETURN CODE=00000008, RACF REASON CODE=00000000.
```

We issued this command to define a rule in RACF to allow the access being prevented:

```
RDEFINE CBIND CB.CTFMVS09.CLU6481 UACC(NONE)
PERMIT CB.CTFMVS09.CLU6481 CLASS(CBIND) ID(ASCR1) ACC(READ)
PERMIT CB.CTFMVS09.CLU6481 CLASS(CBIND) ID(WSGUEST) ACC(READ)
```

A restart of the server was required to pick up these new definitions.

The CTFMVS09 part of the this RACF definition was needed because the ws6481 server had been set up to have its own security domain. The idea of a security domain is that it allows you to set up a set of specific RACF rules that are applicable only for that cell. The concept of a security domain is further explained in:

http://publib.boulder.ibm.com/infocenter/wasinfo/v6r0/index.jsp?topic=/com.ibm.websphere.zseries.doc/info/zseries/ae/tins_planningsecdomain.html

If a cell has a security domain defined, the setting can be found by expanding **Security**, clicking **Global Security**, then clicking **Custom properties**. In the list look for an entry called security.ZOS.domainName.

Successful run

We then reran the servlet and the servlet ran without error. The key parts of the output displayed, which show the Java principal each servlet or EJB method ran under, as shown in Figure 15-6 on page 554.

Servlet Snoop Information

| | |
|-------------------|--|
| Java Principal | Subject: Principal: wtsc48.itso.ibm.com/WSGUEST Credential: com.ibm.ws.security.auth.WSCredentialImpl@6c65 [com.ibm.ws.security.auth.zOSWSCredentialImpl@ |
|-------------------|--|

Servlet call to local SnoopMagicEjb Information

| | |
|-------------------|--|
| Java Principal | Subject: Principal: wtsc48.itso.ibm.com/WSGUEST Credential: com.ibm.ws.security.auth.WSCredentialImpl@6c65 [com.ibm.ws.security.auth.zOSWSCredentialImpl@ |
|-------------------|--|

Local EJB call to remote SnoopMagicEjb Information

| | |
|-------------------|--|
| Java Principal | Subject: Principal: wtsc48.itso.ibm.com/ASSR1 Public Credential: com.ibm.ws.security.auth.WSCredentialImpl@9ab1dda [com.ibm.ws.security.auth.zOSWSCredentialImpl@9b09dd |
|-------------------|--|

Servlet call to remote SnoopMagicEjb Information

| | |
|-------------------|--|
| Java Principal | Subject: Principal: wtsc48.itso.ibm.com/ASSR1 Public Credential: com.ibm.ws.security.auth.WSCredentialImpl@9ab1dda [com.ibm.ws.security.auth.zOSWSCredentialImpl@9b09dd |
|-------------------|--|

Figure 15-6 Successful run of the ejbMagic application

15.3.2 Handling corbanames

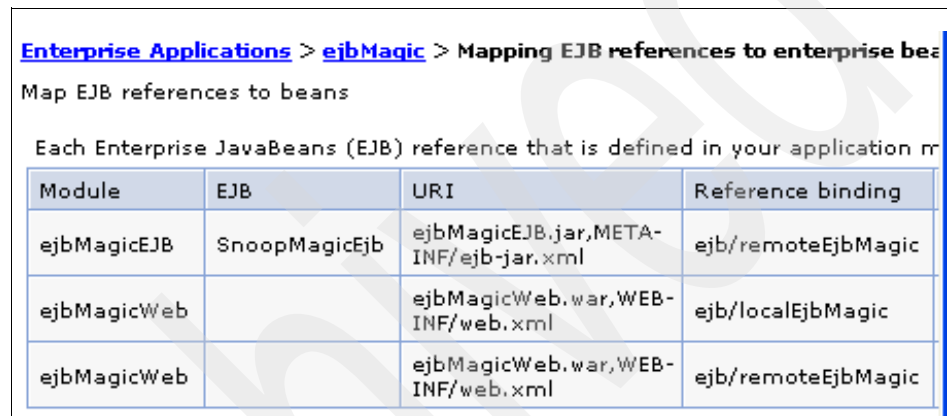
When we ran the ejbMagic application, we used the following value to locate the remote EJB:

```
corbaname::9.12.4.38:32809/NameServiceServerRoot#cell/nodes/nd6483/servers/  
ws6483/ejb/itso/em/ejb/SnoopMagicEjbHome
```


In a real-world application, you would not be entering such a string via a field on a browser as we do in the `ejbMagic` application. Rather, you would have it set as the binding for the logical resource in the deployment descriptors of the application.

Updating EJB references

We can test this by updating the value currently set for the application. In the WebSphere administrative console, expand **Applications**, click **ejbMagic**, then click **Map EJB references to beans**, which will result in the display shown in Figure 15-7.



| Enterprise Applications > ejbMagic > Mapping EJB references to enterprise beans | | | |
|---|---------------|--------------------------------------|--------------------|
| Map EJB references to beans | | | |
| Each Enterprise JavaBeans (EJB) reference that is defined in your application module | | | |
| Module | EJB | URI | Reference binding |
| ejbMagicEJB | SnoopMagicEjb | ejbMagicEJB.jar,META-INF/ejb-jar.xml | ejb/remoteEjbMagic |
| ejbMagicWeb | | ejbMagicWeb.war,WEB-INF/web.xml | ejb/localEjbMagic |
| ejbMagicWeb | | ejbMagicWeb.war,WEB-INF/web.xml | ejb/remoteEjbMagic |

Figure 15-7 Logical EJB resource references for the `ejbMagic` application

Currently, the logical reference bindings called `ejb/remoteEjbMagic` are both set to `ejb/itso/em/ejb/SnoopMagicEjbHome`.

Replace that value with:

```
corbaname::9.12.4.38:32809/NameServiceServerRoot#cell/nodes/nd6483/servers/
ws6483/ejb/itso/em/ejb/SnoopMagicEjbHome
```

Click **OK** and save the change, and stop and start the application.

We then reran the application, except we now selected the option Use logical EJB reference resource. The application ran successfully, producing the same output as shown in Figure 15-6 on page 554.

15.3.3 Name space bindings

We now have the application using logical JNDI lookups again to locate the remote EJB. The `corbaname` value for the URL can be set in the deployment

descriptors of the applications by the developers, so that during deploy time no further action is required.

However, from an application developer point of view, there are a few issues here.

Host-specific information

First of all, a corbaname value such as this:

```
corbaname::9.12.4.38:32809/NameServiceServerRoot#cell/nodes/nd6483/servers/  
ws6483/ejb/itso/em/ejb/SnoopMagicEjbHome
```

has a lot of specific information in it. For example, it has the TCP/IP address and port of the remote server. It also has the name of the node and server in the remote cell where the EJB resides.

Having a hardcoded value like this in the deployment descriptor of an application ear file is clearly not going to be best practice. In a typical situation where you have development, test, and production environments, the deployment descriptor would need to be updated for each environment the application was deployed into. This leads to mistakes, delays, etc.

Another consideration is that application developers should not really have to concern themselves with having to determine these sort of corbaname values, since their primary role is to develop application code, not worry about the intricacies of locating remote EJBs.

The solution

WebSphere provides a way to assist with this issue, and it is called *name space bindings*.

Name space bindings are a way to provide a layer of indirection into the JNDI name space lookup area.

You define a name space binding in the WebSphere cell where the application calling the remote EJB runs. This creates a new local JNDI value. This definition identifies where the remote EJB is located.

The logical resource definitions in the deployment descriptors of the application are set to this new local JNDI value.

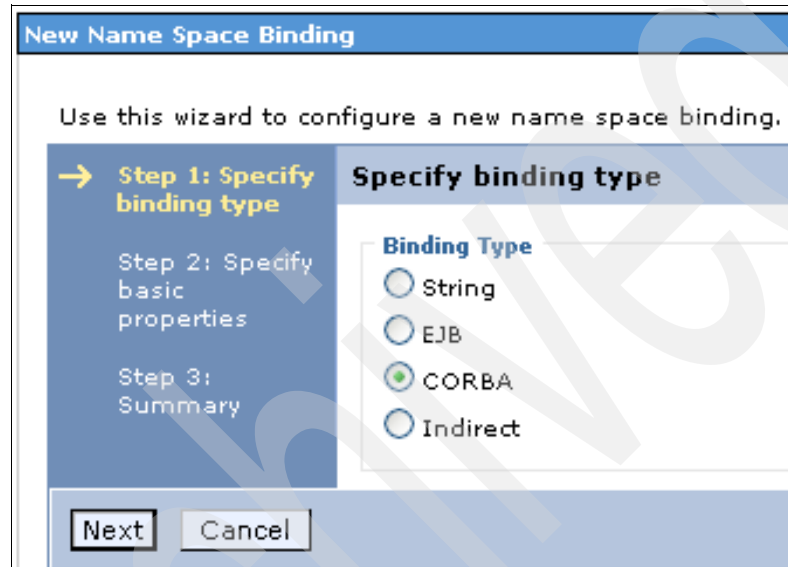
Typically, it would be the WebSphere administrator who defines these name space bindings.

Define name space binding

As way of example, we now define a name space binding to use with the ejbMagic application.

In the WebSphere administrative console of the ws6481 cell, expand **Environment**, then **Naming**, then click **Name Space Binding**, then **New**.

On the page displayed, select the type **CORBA**, as shown in Figure 15-8.



New Name Space Binding

Use this wizard to configure a new name space binding.

→ **Step 1: Specify binding type**

Step 2: Specify basic properties

Step 3: Summary

Specify binding type

Binding Type

☐ String

☐ EJB

☒ CORBA

☐ Indirect

Next **Cancel**

Figure 15-8 Creating a CORBA name space binding

On the next page, you set the properties specific to the new definition.

Set binding identifier to remoteMagicEjb, and set name in name space to ejb/remoteMagicEjb. Set Corbaname URL to:

```
corbaname::9.12.4.38:32809/NameServiceServerRoot#cell/nodes/nd6483/servers/  
ws6483/ejb/itso/em/ejb/SnoopMagicEjbHome
```

The page should then look as shown in Figure 15-9 on page 558.

New Name Space Binding

Use this wizard to configure a new name space binding.

Step 1: Specify binding type
→ Step 2: Specify basic properties
 Step 3: Summary

Specify basic properties

Binding Type

* Binding Identifier

* Name in Name Space

* Corbaname URL

☐ Federated Context

Figure 15-9 Setting properties for the new name space binding definition

Click **Next**, then **Finish**, and save the change.

Test new name space binding definition

Select the **Use JNDI override value** option, and enter the value:

```
cell/persistent/ejb/remoteMagicEjb
```

Run the servlet. The result is the same as shown in Figure 15-6 on page 554.

Update logical references

We can now update the bindings of the logical resource references of the ejbMagic application.

We do this in the same way described in “Update logical references” on page 558, except set the logical reference bindings called ejb/remoteEjbMagic to the value:

```
cell/persistent/ejb/remoteMagicEjb
```

Stop and start the application.

Rerun the servlet, except now select the option **Use logical EJB reference resource**. The application runs successfully, producing the same output as shown in Figure 15-6 on page 554.

Summary

Using name space bindings means that the application developer can code a value of cell/persistent/ejb/remoteMagicEjb in the deployment descriptor files of the application. They can use the same value in a development, test, or production environment. The WebSphere administrator in each environment defines a name space binding, which provides the JNDI name of a resource the application will be looking for, and tells the local WebSphere the location of the remote EJB.

15.4 CSI

When a client connects to WebSphere to invoke a method on an EJB, it does this using the RMI over IIOP protocol. If we were discussing HTTP requests that arrive at WebSphere to be processed, we would be discussing issues around how these HTTP requests are authenticated and whether we were going to use SSL.


The same issues of authentication and protocol apply for RMI over IIOP requests into a WebSphere cell. The mechanism WebSphere provides to control these aspects is called Common Secure Interoperability (CSI).

It is this feature of WebSphere that controls, first, what protocol will be used, either TCP/IP or SSL, and secondly what authentication controls will be in place.

Further information about CSI can be found at:

http://publib.boulder.ibm.com/infocenter/wasinfo/v6r0/index.jsp?topic=/com.ibm.websphere.nd.doc/info/ae/ae/csec_corba.html

It is by manipulation of the CSI settings in WebSphere that you can control whether java principals of applications running in the client can be propagated to applications invoked in the server by the client.



The Trader application explained

This appendix provides some background information about the Trader applications used in most of the chapters as a sample application.

Overview of Trader application

Trader is a sample application that provides different incarnations that show how to integrate WebSphere with other applications on the z/OS platform. It is a simple application that mimics trading stocks in four different companies.

The Trader application consists of four major components (Figure A-1):

- ▶ A back end
- ▶ A data store
- ▶ A middle tier providing access to the back end
- ▶ A front end that is implemented as a Web application

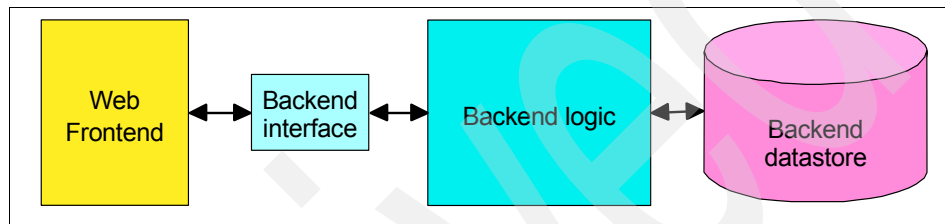


Figure A-1 Trader major components

The Web front end is a regular Java 2 Platform, Enterprise Edition (J2EE) Web module. The middle tier (back-end interface) is based on Enterprise JavaBeans™ (EJBs).

The following technologies are used:

- ▶ CICS ECI J2CA resource adapter.
This provides direct access from the back-end interface to the back-end logic hosted in CICS, using the CICS Transaction Gateway.
- ▶ IMS J2CA resource adapter.
This provides direct access from the back-end interface to the back-end logic hosted in IMS.
- ▶ JDBC connection using either straight Java Database Connectivity (JDBC) from a session EJB or using Container Managed Persistence (CMP) entity EJBs.
- ▶ SQLJ to provide a static SQL alternative to regular JDBC.
- ▶ WebSphere MQ JMS Provider.
This provides access to either IMS transactions via the WebSphere MQ IMS bridge or CICS via the WebSphere MQ CICS bridge.
- ▶ WebSphere Default messaging.

As of WebSphere Application Server for z/OS Version 6.01, an internal Service Integration Bus (SIB) is provided, which can route messages or service requests within WebSphere.

- ▶ Web services technology is used to show access to a DB2 Stored Procedure.

We created a Trader application for each of the following connectivity scenarios used:

- ▶ TraderCICS for access to CICS
- ▶ TraderIMS for access to IMS
- ▶ TraderDB for access to DB, using:
 - JDBC
 - SQLJ
 - Web services to call stored procedures
- ▶ TraderMQ for access to WebSphere Default messaging and WebSphere MQ

Trader IMS and CICS applications and data stores

Basically, these consist of a transaction that can process the trade of shares and a data store. The data stores for all Trader applications share the same basic structure as described for DB2 in Figure A-2 on page 564.

| TRADER.COMPANY | | | |
|------------------|-----------|-----|-------|
| Column name | Type | Len | Nulls |
| COMPANY | CHARACTER | 20 | No |
| SHARE_PRICE | REAL | 4 | Yes |
| UNIT_VALUE_7DAYS | REAL | 4 | Yes |
| UNIT_VALUE_6DAYS | REAL | 4 | Yes |
| UNIT_VALUE_5DAYS | REAL | 4 | Yes |
| UNIT_VALUE_4DAYS | REAL | 4 | Yes |
| UNIT_VALUE_3DAYS | REAL | 4 | Yes |
| UNIT_VALUE_2DAYS | REAL | 4 | Yes |
| UNIT_VALUE_1DAYS | REAL | 4 | Yes |
| COMM_COST_SELL | INTEGER | 4 | Yes |
| COMM_COST_BUY | INTEGER | 4 | Yes |
| | | | |
| TRADER.CUSTOMER | | | |
| Column name | Type | Len | Nulls |
| CUSTOMER | CHARACTER | 60 | No |
| COMPANY | CHARACTER | 20 | No |
| NO_SHARES | INTEGER | 4 | Yes |

Figure A-2 DB2 table definition of the Trader application

For TraderDB, a DB2 database is used directly. For TraderCICS, the CICS application uses a VSAM file as the data store. For TraderIMS, the IMS application uses DL/1 as the data store. TraderMQ uses the same CICS and IMS application or transaction as TraderCICS and TraderIMS.

Trader Web front-end user interface

All the Trader Web modules provide the same basic user interaction (see Figure A-3 on page 565). The entry page is the logon page. The logon page provides a field to enter a user name and one or more buttons that take you into the applications. The number of buttons depends on the actual Trader application. For example, TraderMQ provides a choice between using CICS or IMS as the processor of the MQ messages. There is also the possibility to modify the way the connector is used. For example, TraderDB provides both straight JDBC and JDBC encapsulated in entity EJBs using CMP as well as SQLJ.

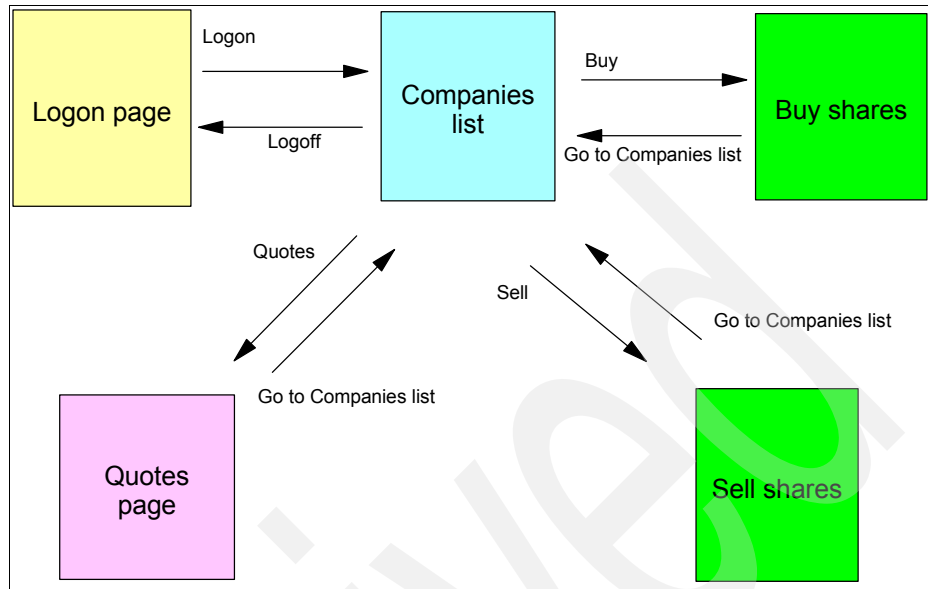


Figure A-3 Trader screen flow

If the logon is successful, you see a list of companies on the next page (Figure A-4). For each company, there are buttons to access quotes and holdings status, and to buy and sell shares. This list is obtained from the back-end data store.

| SQLJDBC | | | |
|----------------------|--------|-----|------|
| COMPANY SELECTION | | | |
| Company | Quotes | Buy | Sell |
| Casey_Import_Export | Quotes | Buy | Sell |
| Glass_And_Luget_Plc | Quotes | Buy | Sell |
| Headworth_Electrical | Quotes | Buy | Sell |
| IBM | Quotes | Buy | Sell |

Logoff

Figure A-4 Trader companies list

Clicking the Logoff button takes you back to the logon page of the Trader application.

Clicking Buy or Sell takes you to a page with a field where you can enter the number of shares you want to buy or sell. It includes a button to start the transaction. When the transaction is done, you see the Companies list again (Figure A-4 on page 565).

To see the result of a transaction, go to the Quotes page (Figure A-5). You do this by clicking the **Quotes** button on the Companies list page (Figure A-4 on page 565).

IBM

SQLCMP

QUOTES

| | | | |
|-----------------|----------------------------|----------------------------|---|
| User Company | ddd Glass_And_Luget_Plc | Comission Cost | |
| | | For Selling | 2 |
| | | For Buying | 2 |
| Share Values | | Number of shares held 13 | |
| Now | 19.0 | Value of shares held 247.0 | |
| 1 week ago | 17.0 | | |
| 6 days ago | 22.0 | | |
| 5 days ago | 20.0 | | |
| 4 days ago | 16.0 | | |
| 3 days ago | 20.0 | | |
| 2 days ago | 25.0 | | |
| 1 day ago | 22.0 | | |

Go to companies list

Figure A-5 Trader company quotes page

Trader Web front-end and back-end interface

The overall architecture of the Trader Web application is presented in Figure A-6 on page 567. It is a classic Model-Volume-Controller (MVC) approach. The *TraderServlet* contains the control logic, providing a method for each user interaction. Because of time constraints, we decided not to use the Command pattern. We recommend that you use the Command pattern for applications that

are larger than the Trader application. It gives a better separation of control and command logic, which makes the application easier to maintain.

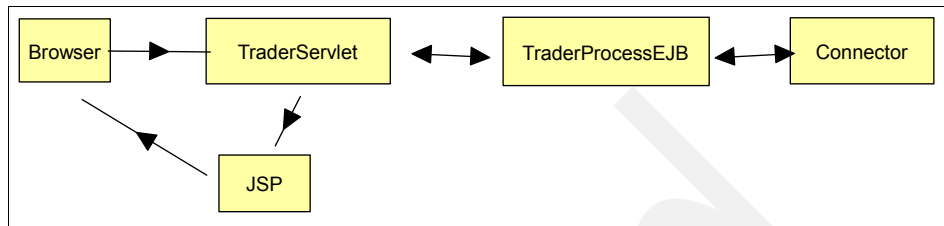


Figure A-6 Main component diagram of the Trader Web applications

The *TraderProcessEJB* contains the front-end business logic: Buy, sell, getCompanies, etc. The implementation is divided into two:

- ▶ An interface (*TraderProcess*), which is used and seen by the *TraderServlet*
- ▶ The actual implementation, which depends on the connector used

The JavaServer™ Pages™ (JSPs) format the output for the browser.

To simplify the implementation of the same base application for different variations, we use the simplified class diagram in Figure A-7 on page 568 as a basis. The *TraderSuperServlet* contains all the control and command logic of the application. The only methods implemented by the actual servlets are a method to create the *TraderProcess* instance (*createTrader*) and the *init()* method of the servlet. This initializes text strings for the construction of the Uniform Resource Locators (URLs) in the applications and displays the type of connector that is used.

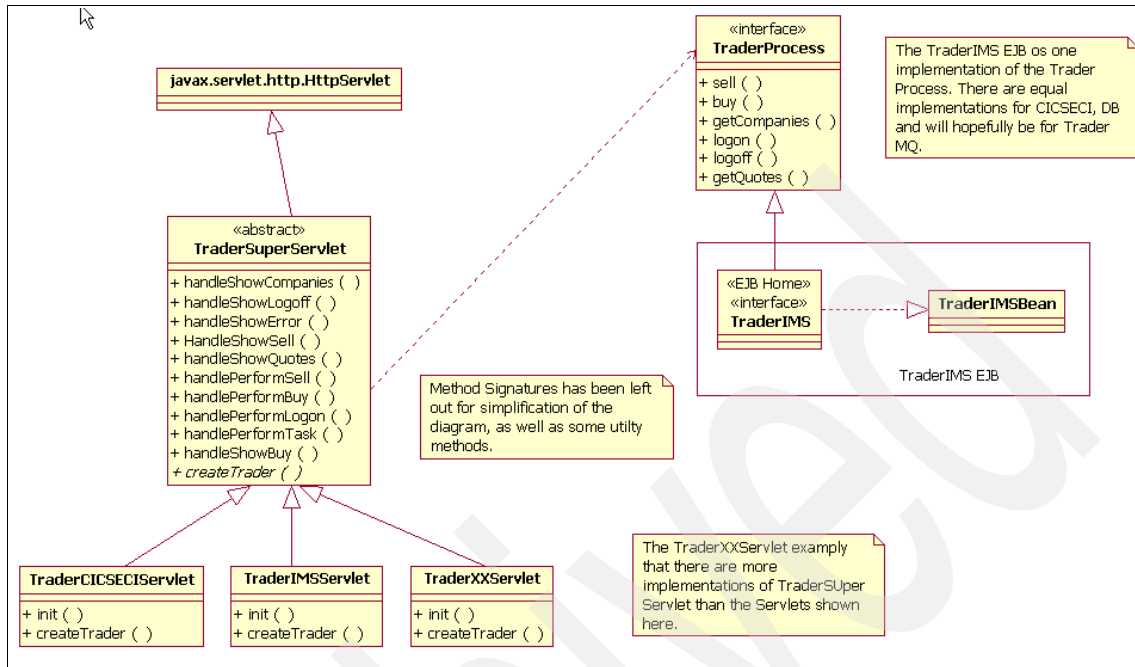


Figure A-7 Trader class diagram (simplified overview)

The TraderProcess implementations are specialized according to the connectivity technology being used. The specific connectivity issues include:

- ▶ CICS J2CA ECI connector

Uses the J2EE Connection (J2C) Architecture based CICS ECI connector. The connector uses the CICS Transaction Gateway Java client. The code to access the CICS J2CA ECI connector is generated by WebSphere Studio Application Developer Integration Edition or Rational Application Developer (RAD). The generated code consists of a Web service that is implemented as an EJB. It also consists of classes for setting and getting information in the ECI CommArea, based on the object definitions used in the CICS programs.

- ▶ IMS J2CA connector

Uses the a J2EE Connection (J2C) Architecture based IMS connector. It works the same way as the CICS ECI connector. The code is generated the same way.

- ▶ Java Database Connectivity (JDBC)

When using JDBC, we do not have a back-end CICS or IMS transaction with business logic. Therefore, this business logic is implemented in the WebSphere application, using EJBs. The data resides in a DB2 database.

Data is accessed in two ways:

- Using traditional straight JDBC from a session EJB
- Using CMP entity EJBs

► SQLJ

In the case of SQLJ, basically the same logic is performed as with JDBC. No new or different functions are being used, but rather another way of accessing the database (static instead of dynamic).

► WebSphere MQ

Instead of going straight to IMS and CICS from the application using J2CA connectors, there is the option to use WebSphere MQ. The TraderMQ application sends a message via WebSphere MQ to the back-end business logic in CICS or IMS (the TRADERBL program). The message receiver is the MQ bridge, either the WebSphere MQ IMS bridge or the WebSphere MQ CICS bridge. When the transaction is completed in either IMS or CICS, the reply is returned via WebSphere MQ to the Trader application in WebSphere. This is a quasi-synchronous solution to front end any traditional business logic in CICS or IMS.

There is an option to use a Message-Driven Bean (MDB) as the receiver in the Trader Web-application instead of a session EJB that queries the reply queue. When using this option, select the **MDB** option on the TraderMQ logon panel and start the message listener ports on the server.

Attention: When the MDB listeners are enabled, the normal TraderMQ scenarios do not work (non-MDB case), since the MDB listener picks up the messages from TRADER.CICS.REPLYQ or TRADER.IMS.REPLYQ regardless of whether the check box was selected. The XA (two-phase commit) feature has to be enabled on the WebSphere MQ connection factory for this to work.

If the message listeners are started when TraderMQ is executed and the MDB box is *not* selected on the logon panel, TraderMQ waits for the message to return from CICS or IMS. However, it never receives the reply (you have to click the Abort button). The reason for this is that the MDB already picked up the message from the TRADER.CICS.REPLYQ (or IMS) reply queue and placed it in TRADER.PROCESSQ. Since the MDB check box was not selected, the EJB business logic does not receive the message from TRADER.PROCESSQ.

Restriction: Trader was not implemented with the purpose of being a fully production-qualified application. Because of this, the screen flow is based on the need to be there. The fault tolerance is limited. The application cannot be expected to run in parallel without flaws. Plus, all resources are not externalized using the java:comp/env context. This results in a lack of transactional control and part of the implementation not being in compliance with best practices and recommended implementation patterns.

However, the application or applications assist in verifying that a WebSphere application can connect to something else. They act as an example of how a certain connectivity technology can be used in an application.

Packaging

The different Trader applications are packaged in Enterprise Application Repository (.ear) files and/or Project Interchange files (.zip):

- ▶ CICS J2CA ECI connector
 - TraderCICS2005.ear
 - TRADERCICS2005.zip
- ▶ IMS J2CA connector
 - TraderIMSJ2C.ear
 - TraderIMSJ2C.zip
- ▶ Messaging:
 - WebSphere MQ V5 (JMS 1.0) application:
 - TraderMQ2004.ear
 - WebSphere MQ V6 (JMS 1.0) application:
 - TraderMQ2005.ear
 - WebSphere MQ V6 (JMS 1.1) application:
 - TraderMQ2005_JMS1_1.ear
 - TraderMQ2005_JMS11.zip
 - WebSphere Default messaging:
 - TraderMQ2005_DM_JMS1_1.ear
 - TraderMQ2005_JMS11_DM.zip

- ▶ DB2 access:
 - Data access using JDBC only (V5 version):
 - TraderDB2004.ear
 - Data access using both JDBC and SQLJ (V6 version):
 - TraderDB2005.ear
 - TraderDB2005.zip
 - Calling Stored Procedures using Web services:
 - TraderDBWebServiceEAR.ear

Figure A-8 on page 572 shows the Trader EAR file content.

The *TraderLib.jar* file is shared between all Trader applications. It contains the `TraderSuperServlet`, `TraderProcess`, and some utility classes.

The Trader Web module contains the servlet or servlets that are sub-classed from `TraderSuperServlet` and the JSPs used in the Web application. Because of the way J2EE 1.3 works, it is impossible to share the JSPs in the way done with the `TraderSuperServlet`. Therefore, each Web module contains its own copy of the JSPs. The `Logon.html` is different for each Trader application, but the JSPs are not.

The Trader EJB JAR contains the EJBs used by the servlets. In the case of the `TraderDB`, it also contains the EJBs used for communication with the database and the business logic implementation.

The Trader Connectors JAR contains the Web service that provides access to the J2CA connectors, including the EJB that connects to the J2CA connector and the generated classes used for getting and setting data on the J2CA transaction object or objects. In CICS, this is the `ECI CommArea`. In IMS, they are the `InputHandler` and `OutputHandler` objects.

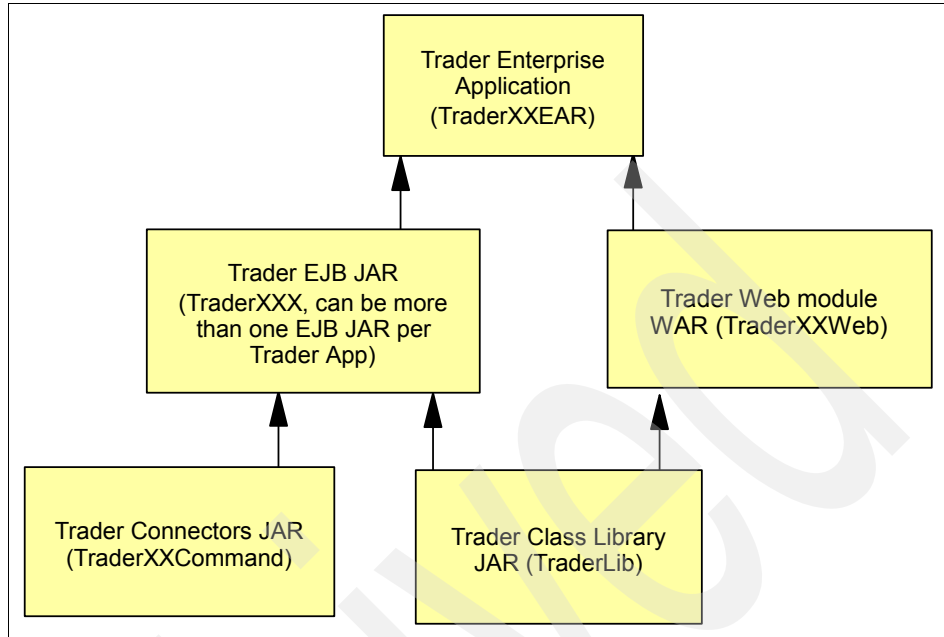


Figure A-8 Trader EAR file contents

Dependencies

Each Trader application depends on some external resources to be available for the application to be deployable and work. All the resources are, if possible, specified by their Java Naming and Directory Interface (JNDI) name and a type.

Trader MQ using WebSphere MQ JMS Provider

For TraderMQ, the necessary external resources are:

- ▶ jms/TraderQCF: WebSphere MQ JMS provider queue connection factory
- ▶ jms/TraderCF: WebSphere MQ JMS (unified) connection factory
- ▶ jms/TraderCICSReqQ: JMS request destination for CICS
- ▶ jms/TraderCICSRepQ: JMS reply destination for CICS
- ▶ jms/TraderIMSReqQ: JMS request destination for IMS
- ▶ jms/TraderIMSRepQ: JMS reply destination for IMS
- ▶ jms/TraderProcessQ: JMS postprocessing destination for the MDB case
- ▶ TraderMQCICSListener: MDB EJB listener (when a message is received on a Queue listened to, the corresponding MDB is executed)
- ▶ TraderMQIMSListener: MDB EJB Listener

Trader MQ using WebSphere Default Messaging Provider

Refer to “Using the Default messaging JMS provider” on page 148 and “The TraderMQ application environment” on page 166 for all the definitions you need to have in WebSphere and WebSphere MQ to use the TRADERMQ application with Default messaging.

Other dependencies

Depending on your local environment, you may also need to define a Java Authentication and Authorization Services (JAAS) user ID and password to be used by the MQ bridge.

If you want TraderMQ to work, you need to set up WebSphere MQ for z/OS, the proper queues, and the MQ bridge for CICS or IMS. You can learn more about this in “The TraderMQ application environment” on page 166.

For TraderDB, the necessary external resource is jdbc/TraderDB2. This is the JDBC data source, which is not specific to DB2 even if the name indicates so.

For TraderCICS, the necessary external resource is itsso/cics/eci/j2ee/trader/TraderCICSECICCommandCICSECIServiceTraderCICSECICCommandCICSECIPort. This is an ECI J2CA connector to CICS.

For TraderIMS, the necessary external resource is itsso/ims/j2ee/trader/TraderIMSCCommandIMSServiceTraderIMSCCommandIMSPort. This is an IMS J2CA connector to IMS.

Restriction: Not all of the necessary resources are externalized in the Web deployment and EJB deployment descriptors. You must look up some of the resources directly and not indirectly using the `java:comp/env` context. This also means that there is the possibility of setting up transaction control and redirection is limited.

Figure A-9 on page 574 shows an overview of the different connector paths that are implemented in the Trader applications.

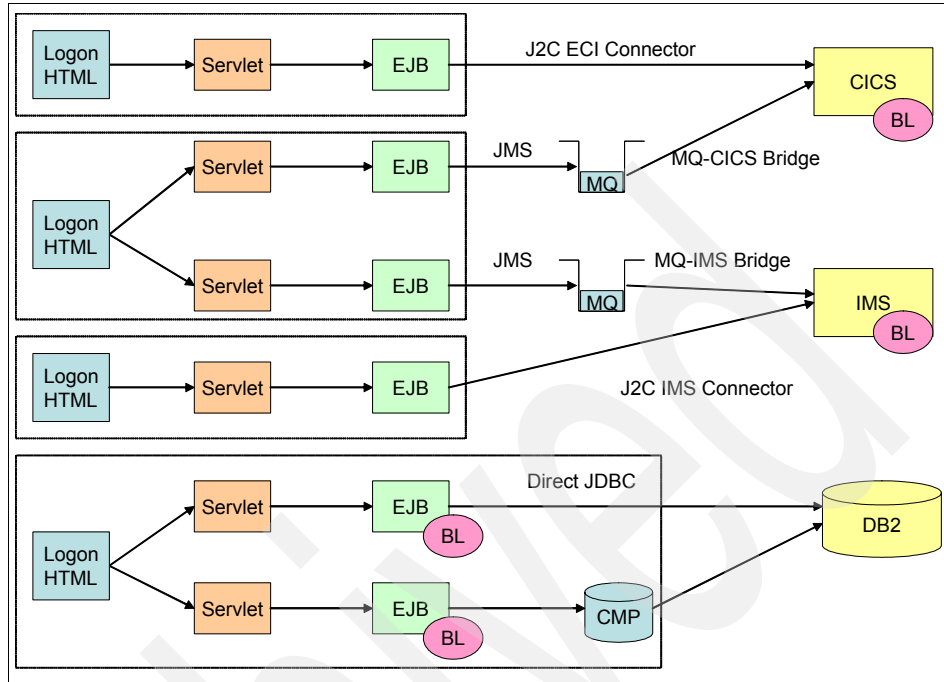


Figure A-9 Trader application connection overview

Deploying Trader applications to WebSphere for z/OS

When using the Trader samples, we assume you should already have installed and configured WebSphere for z/OS on your system. The Trader applications can be installed in both a single server or Network Deployment configuration.

You begin by accessing the WebSphere Administrative console using a Web browser. The URL for the Admin Console is typically:

`http://<servername>:<port>/admin/`

Here, *port* should be the TCP/IP port number of your Admin Console, which is specified when you set up WebSphere for z/OS.

Each application can be deployed by itself and you can deploy all applications together. The deployment procedure consists of four steps:

1. Plan for where you want to install the application.

This step is shared between all the applications. You decide where to install the application, which application server to use, and which virtual host to use.

2. Prepare for the installation.

This step is specific to each application in that you set up the resources needed for the actual Trader application. The outline of the work done is the same. That is, you set up the authentication credentials needed by the resources and subsequently set up the resources needed by the application.

3. Install the application ear files.

The installation of the application EAR file is mainly the same for all applications. There are some variations based on the mapping of resources from the `java:comp/env` context to actual resources provided by the server. The difference is in which resources are used by the actual Trader application.

4. Verify the installation.

The last step is always the same. That is, you must verify that the application is installed and performs properly.

Running the Trader applications

If the applications are installed and configured as explained throughout this book, they can be launched from a Web browser as follows:

```
http://<servername>:<port>/TraderCICSWeb/Logon.html  
http://<servername>:<port>/TraderIMSWeb/Logon.html  
http://<servername>:<port>/TraderMQWeb/Logon.html  
http://<servername>:<port>/TraderDBWeb/Logon.html  
http://<servername>:<port>/TraderRRSWeb/Logon.html
```

Servername is the host name of the node with the application server to which you deployed the Trader application. *Port* is port number of HTTP port of the application server.

Additional configuration information

In this section we provide Trader-generic information regarding the set up for the back-ends. Additional information for each specific scenario can be found in the “Project environment” section of the chapters that use the Trader application.

Trader DB2 table definitions

Figure A-10 on page 576 shows how the Trader sample application is accessing DB2. The TraderDB2 JDBC application has the business logic. The data is stored

in DB2 tables. The CMP component is similar, with the only difference being the use of a CMP entity bean to access DB2.

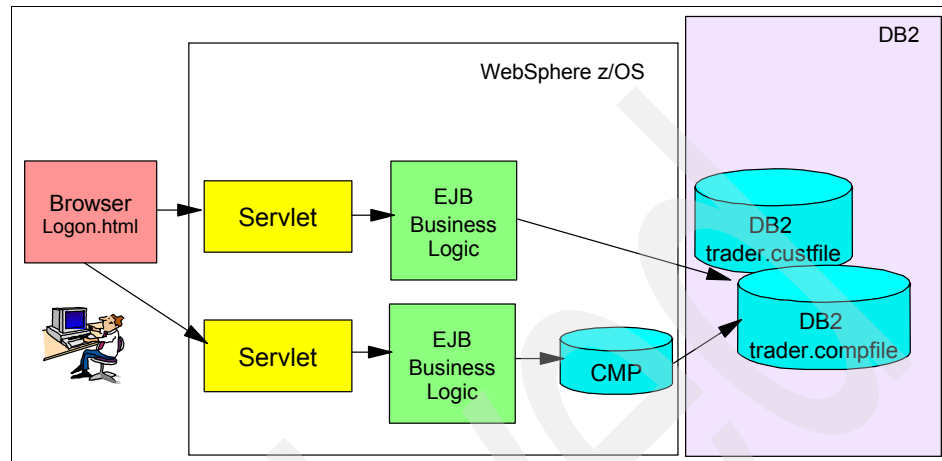


Figure A-10 Trader application using DB2

The Trader application uses two DB2 tables. The first table is the company table, and the second is the customer table. Example A-1 shows the commands used to create the Trader database, tablespace, tables, and unique indices.

Example: A-1 Trader sample database and table creation

```

DROP DATABASE TRADERDB ;
COMMIT;
CREATE DATABASE TRADERDB ;
CREATE TABLESPACE TRADERTS IN TRADERDB ;
COMMIT;
CREATE TABLE TRADER.COMPANY
( COMPANY CHAR(20) NOT NULL PRIMARY KEY,
  SHARE_PRICE REAL,
  UNIT_VALUE_7DAYS REAL,
  UNIT_VALUE_6DAYS REAL,
  UNIT_VALUE_5DAYS REAL,
  UNIT_VALUE_4DAYS REAL,
  UNIT_VALUE_3DAYS REAL,
  UNIT_VALUE_2DAYS REAL,
  UNIT_VALUE_1DAYS REAL,
  COMM_COST_SELL INT,
  COMM_COST_BUY INT )
IN TRADERDB.TRADERTS ;

CREATE, TABLE TRADER.CUSTOMER
( CUSTOMER CHAR(60) NOT NULL,

```

```

        COMPANY CHAR(20) NOT NULL,
        NO_SHARES INT,
        PRIMARY KEY (CUSTOMER,COMPANY))
    IN TRADERDB.TRADERTS ;

CREATE TYPE 2 UNIQUE INDEX TRADER.COMPANYX1
    ON TRADER.COMPANY (COMPANY) ;

CREATE TYPE 2 UNIQUE INDEX TRADER.CUSTOMERX1
    ON TRADER.CUSTOMER (CUSTOMER,COMPANY) ;

COMMIT;

```

Trader VSAM file definitions

Example A-2 shows the IDCAMS delete and define the TraderCICS VSAM files. It copies the example data into the file.

Example: A-2 Trader sample delete and define of VSAM files

```

/***** JOB USER=*****,CLASS=?,NOTIFY=*****,MSGCLASS=?
/*****
/*
/* THIS JOB WILL:
/*
/* 1. DELETE AND DEFINE THE TRADER 'COMPFILE'.
/* 2. REPRO THE REQUIRED DATA INTO THE 'COMPFILE'.
/*
/*****
//STEP01 EXEC PGM=IDCAMS
//SYSPRINT DD SYSOUT=*
//SYSIN DD *
DELETE h1q.SAMPLE.COMPFILE CLUSTER
SET MAXCC = 0
DEFINE CLUSTER -
    (NAME(h1q.SAMPLE.COMPFILE) -
    INDEXED -
    TRACKS(1) -
    SHAREOPTIONS(3 4)) -
DATA -
    (NAME(h1q.SAMPLE.COMPFILE.DATA) -
    KEYS(20 0) -
    RECORDSIZE(90 90) -
    CONTROLINTERVALSIZE(4096)) -
INDEX -
    (NAME(h1q.SAMPLE.COMPFILE.INDEX))
/*

```

```
/*  
//STEP02 EXEC PGM=IDCAMS  
//OUTFILE DD DSN=h1q.SAMPLE.COMPFILE,DISP=OLD  
//INFILE DD DSN=tradercodata.txt,DISP=OLD  
//SYSPRINT DD SYSOUT=*  
//SYSIN DD *  
        REPRO INFILE(INFILE) OUTFILE(OUTFILE)  
/*  
//
```

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/SG247064>

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, SG247064.

Using the Web material

The additional Web material that accompanies this redbook includes the following files:

| <i>File name</i> | <i>Description</i> |
|---------------------|-----------------------------------|
| SG247064.zip | All code samples in zipped format |

System requirements for downloading the Web material

There are no specific system requirements for your workstation to download the supplied materials. However, to be able to use the supplied samples you will need to adhere to product specifications as described throughout this book.

How to use the Web material

Create a subdirectory (folder) on your workstation, and unzip the contents of the Web material zip file into this folder.

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

IBM Redbooks

For information about ordering these publications, see “How to get IBM Redbooks” on page 582. Note that some of the documents referenced here may be available in softcopy only.

- ▶ *DB2 for z/OS and OS/390: Ready for Java*, SG24-6435
- ▶ *WebSphere for z/OS V5 Connectivity Handbook*, SG24-7064-01
- ▶ *WebSphere for z/OS Connectivity Architectural Choices*, SG24-6365
- ▶ *Patterns: Service-Oriented Architecture and Web Services*, SG24-6303
- ▶ *WebSphere Version 6 Web Services Handbook Development and Deployment*, SG24-6461
- ▶ *IMS Version 7 Java Update*, SG24-6536

Other publications

These publications are also relevant as further information sources:

- ▶ *DB2 Universal Database™ for z/OS: Application Programming Guide and Reference for Java*, SC18-7414-02
- ▶ *DB2 Universal Database for z/OS: Messages and Codes*, GC26-9011-02
- ▶ *IMS Version 9: Installation Volume 2: System Definition and Tailoring*, GC18-7823-00
- ▶ *IMS V9 Connect Guide and Reference*, SC18-9287-01
- ▶ *IMS Version 9: IMS Java Guide and Reference*, SC18-7821

Online resources

These Web sites and URLs are also relevant as further information sources:

- ▶ IBM Information Center - WebSphere Application Server Express Version 6.0.x
<http://publib.boulder.ibm.com/infocenter/wasinfo/v6r0/index.jsp?topic=/com.ibm>
- ▶ J2EE Connector Architecture
<http://java.sun.com/j2ee/connector>
- ▶ XML Protocol Working Group
<http://www.w3.org/2000/xp/Group>
- ▶ Web Services Description Language (WSDL) 1.1
<http://www.w3.org/TR/wsdl>
- ▶ Extensible Markup Language (XML)
<http://www.w3.org/XML>
- ▶ OASIS UDDI
<http://www.uddi.org>
<http://www.oasis-open.org/specs/index.php#wssv1.0>
- ▶ IBM developerWorks
<http://www.ibm.com/developerworks/views/webservices/standards.jsp>
- ▶ Web services transactions specifications
<http://www.ibm.com/developerworks/library/specification/ws-tx>
- ▶ OASIS Web Services Security (WSS) TC
http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss
- ▶ Conformance Claim Attachment Mechanisms Version 1.0
<http://ws-i.org/schemas/conformanceClaim>
- ▶ CICS Transaction Server for z/OS Information Center
<http://publib.boulder.ibm.com/infocenter/cicsts31/index.jsp>

How to get IBM Redbooks

You can search for, view, or download Redbooks, Redpapers, Hints and Tips, draft publications and Additional materials, as well as order hardcopy Redbooks or CD-ROMs, at this Web site:

ibm.com/redbooks

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services

Archived

Index

Numerics

-805 SQLCODE 61

A

A 34

activation specification, configuration 104

ANT

adding commons-net.jar to RAD 512

commons-net.jar 512

in Rational Application Developer 512

overview 512

Assured persistent

assured persistent mode 94

Authentication and Authorization Services (JAAS)

573

B

business processes 340

business rules 340

C

CA certificate, connecting to keyring 320

CA certificate, creating 319

CICS Transaction Gateway (CTG)

backward compatibility in WebSphere 185

common errors 291

common errors, CTG9630E 291

common errors, CTG9631E 292

local topology 184

remote topology 184

transaction management 292

client certificate, connecting to keyring 320

client certificate, creating 320

Command pattern 566

Common Secure Interoperability (CSI) 559

corbaname 554

corbaname value 556

Cursor stability 32

D

Database Resource Adapter (DRA) 64

DB2

CS isolation level 32

read stability 32

repeatable read 32

RR isolation level 32

RS isolation level 32

stability stability 32

Uncommitted read 32

UR isolation level 32

DB2 isolation levels 32

db2jcc.jar 47

db2jcc_license_cisuz.jar 47

DFSJMP 532

DFSJVMAP 531

DFSJVMEV 532

DFSJVMMS 530

DFSJVMWK 531

DFSPRP macro 64

-Djava.ext.dirs parameter 530

DRAName 64

Dynamic Virtual IP Address (DVIPA) 330

E

ejb-jar.xml, updating 206

Embedded Messaging (EM) 97

Enterprise Service Bus (ESB) concept 91

EXtended markup Language (XML) 343

G

Generic JMS provider 96

Generic JMS provider, pros and cons 99

Generic JMS provider, security administration 112

Generic JMS provider, security 112

global transaction 292

Group ID (GID) 495

I

ibm-web-bnd.xmi 544

If 335

IMS

calling Web services

classpath restriction in IMS 530

- import WSDL file 498
 - importing imsjava.rar 514
 - IMS Java classes 523
 - IMS Java transaction, example 525
 - IMS Java, memory 493
 - IMS requirements 494
 - imsjava.jar, location 514
 - JDK requirement 493
 - PROCLIB settings 530
 - shareable.application.classpath 530
 - STEPLIB requirements 530
 - testing the Java proxy 510
 - trusted.middleware.classpath 530
 - WebSphere requirements 493
 - z/OS requirements 493
 - Java transaction
 - class reloading 536
 - problem with authorization 533
 - testing, from 3270 terminal 533
 - IMS Connect
 - configuration details 298
 - configuration parameters for local usage 300
 - connection factory, custom properties 311
 - DataStoreName 309
 - GroupName 309
 - HostName 309
 - IMSConnectName 309
 - Password 309
 - PortNumber 309
 - TraceLevel 309
 - Userid 309
 - connection factory, DataStoreName 307
 - connection factory, defining 305
 - connection factory, IMSConnectName 307
 - connection factory, custom properties 307
 - DATASTORE GROUP 300
 - DATASTORE ID 300
 - DATASTORE MEMBER 300
 - high availability 330
 - HWS ID 300
 - imsico.rar 301
 - OTMA parameters 300
 - PORTID 300
 - Resource Adapter, installing 301
 - Resource Recovery Services (RRS) 300
 - startup procedure 299
 - sysplex 297
 - TCPIP PORTID 300
 - topologies 297
 - two-phase commit 335
 - IMS Java Database Connectivity (JDBC) 63
 - adding IMS load libraries to the servant region's STEPLIB 65
 - connection factory, configuring 69
 - creating a DRA module 64
 - custom service, adding 73
 - custom service, class name 73
 - custom service, classpath 73
 - Database Resource Adapter (DRA) 64
 - DFSPRP macro 64
 - DLIDatabaseView subclass 76
 - DRA Startup Table 64
 - DRAName 64
 - environment entries 74
 - environment variables, LIBPATH 75
 - imsjava91.rar 65
 - J2C connection factory 64
 - J2C connection factory, additional properties 72
 - J2C connection factory, custom properties 72
 - J2C connection factory, DatabaseViewName 72
 - J2C connection factory, IMS subsystem ID 72
 - J2C connection factory, JNDI name 70
 - J2C connection factory, DRAName 72
 - J2CA connection factory resource, configuring 70
 - ODBA runtime code 65
 - Open Database Access (ODBA) 64
 - Process Definition, defining 74
 - resource adapter, installing 65
 - resource adapter 65
 - SDFSJLIB 65
 - SDFSRESL 65
 - IMS Java Message Processing Region (JMP) 494
 - IMS Java Region (JMP) 532
 - imsico.rar 301
 - imsjava91.rar 65
- ## J
- J2EE Connector (J2C) Architecture
 - Application authentication 280
 - application managed security 284
 - application security 317
 - assignment of the current thread identifier 287
 - Authentication Alias 280

- authentication alias 286
- Authentication Alias, associating with J2C connector 282
- authentication by the application 284
- automating definitions 326
- CICS
 - high availability 289
 - problem determination 290
- common runtime errors 332
- Component-managed authentication alias 286
- component-managed Authentication Alias 307
- concept 175
- concepts
 - application 175
 - application contract 176
 - application server 175
 - Common Client Interface (CCI) 175–176
 - connection management contract 176
 - ConnectionFactory object 177
 - ConnectionFactory object 177
 - connections 177
 - ConnectionSpec object 177
 - connector 175
 - container contract 176
 - contract 176
 - DefaultConnectionFactory object 177
 - Enterprise Information System 175
 - interaction object 178
 - InteractionSpec object 178
 - managed connections 177
 - non-managed connections 177
 - resource adapter 175
 - security management contract 176
 - system contract 176
 - transaction management contract 176
- connection factory
 - cell scope 314
 - JNDI name 316
- connection factory, installing using script 328
- connection factory, properties 329
- connection properties, adding to application 318
- Container authentication 280
- Container managed security 282
- container-managed Authentication Alias 307
- developing
 - JNDI name, of resource adapter 192
- developing in RAD
 - Code page 197
 - Code page used by J2C Java Bean, finding out 204
 - connection factory, lookup 222
 - direct JNDI lookup, warning 206
 - EJB J2C Java Bean, adding external JARs 232
 - EJB J2C Java Bean, creating 227
 - EJB J2C Java Bean, sample class to test 230
 - EJB J2C Java Bean, testing 229
 - enabling J2C feature 180
 - input and output types for method, defining 196
 - J2C Java Bean, adding a method 194, 223
 - J2C Java Bean, adding input and output 224
 - J2C Java Bean, classes generated 226
 - J2C Java Bean, connection properties 220
 - J2C Java Bean, creating 218
 - J2C Java Bean, resource adapter selection 219
 - J2C Java Data Binding bean, COBOL to Java mapping 215
 - J2C Java Data Binding bean, creating 214
 - J2C Java Data Binding bean, target platform 216
 - J2C Web service, client 257
 - J2C Web service, creating 254
 - J2C Web service, creating proxy 261
 - J2C Web service, wsdl 256
 - meta-data 204
 - passing user ID and password 233
 - Platform 197
 - Platform used by J2C Java Bean, finding out 204
 - resource adapter, properties 200
 - selecting the CICS resource adapter 191
- infrastructure security 319
- interaction properties 319
- interaction properties, adding to application 318
- JAAS Logon Configuration 283
- passing user ID and password from the application 285, 318
- passing user ID and password to CICS 284
- passing user ID and password using custom properties 286
- passing user ID and password via Authentication Alias 286
- performance and availability 288

- res-auth parameter 317
- res-auth parameter, application authentication 317
- res-auth parameter, container authentication 317
- resource adapter, installing using script 326
- security decision tree for CICS 287
- specification, url 173
- tracing 331
- Java Database Connectivity (JDBC)
 - advantages of using a data source 28
 - connection methods 27
 - DataSource interface 27
 - developing in RAD
 - adding a data source to the application 18
 - adding a provider to the application 17
 - communication with z/OS 11
 - data source attributes 19
 - DB2 Connect, communication parameters 13
 - DB2 Connect, communication protocol 13
 - DB2 Connect, data source 16
 - DB2 Connect, database name 14
 - DB2 Connect, JNDI name 16
 - DB2 Connect, operating system 14
 - DB2 Connect, specifying the provider 16
 - DB2 Connect, testing the connection 15
 - DB2 Connect, user ID and password 15
 - deploying an application to a server 11
 - JNDI name of data source 18
 - providers and data sources inside application 16
 - setting up DB2 Connect 12
 - specifying JDBC properties for Type 4 driver 12
 - driver types 3
 - DriverManager interface 27
 - enabling on z/OS
 - BIND job 5
 - binding the JDBC packages 5
 - configuring a data source in WebSphere 9
 - db2.jcc.properties file, defining 6
 - db2.jcc.properties file, SSID 6
 - DSNHDECP load module 6
 - install path 4
 - license file 4
 - properties, driver-generic 7
 - README 5
 - setting up the JDBC provider in WebSphere 8
 - steps 4
 - variables in WebSphere 5
 - example of connection 29
 - legacy/CLI drivers 2
 - problem reporting 2
 - TraderDB sample application, deploying 20
 - TraderDB sample application, testing 23
 - Type 1 driver 3
 - Type 2 driver 3
 - Type 3 driver 3
 - Type 4 driver 4
 - types of drivers 2
 - Universal JDBC Drivers 2
- Java Message Service (JMS)
 - 1.1 specification 88
 - authorization checking 108
 - concepts 86
 - connection factory object 87
 - consumer 86
 - createConnection statement 86
 - destination object 87
 - domains 87
 - in WebSphere
 - default messaging 89
 - provider types 88
 - JMS to MQ message mapping 112
 - JMSXUserID property 111
 - point-to-point domain 87
 - producer 86
 - providers in WebSphere, pros and cons 98
 - publish/subscribe domain 88
 - unified connection factory 88, 96
- Java Message Service (JMS) 1.1
 - (unified) connection factory, defining 143
 - comparison with JMS 1.0 133
 - overview 133
 - unification of domains 133
- Java Message Service (JMS)
 - authentication in WebSphere 106
 - authentication in WebSphere, decision tree 106
- JDBC 5
- JNDI lookup 542
- JNDI lookup, binding 545
- JNDI lookup, logical 206
- JNDI name, logical 208
- JNDI name, of EJB 544
- JSSE keyrings 323
- Jython scripts 326

L

listener port, configuring 129–130

M

mediation 95

message broker 88

Message Drive Bean (MDB)

programming directives 105

Message Driven Bean (MDB) 100

activation specification 103

characteristics 101

identity 107

listener manager 101

listener port 101

listener port, configuration 103

message listener service 100

Message Exchange Pattern (MEP) 346

Message Queue Interface (MQI) 85

messaging

confidentiality and integrity 113

Messaging principles 85

messaging, security 106

Model-Volume-Controller (MVC) 566

MQ message, mapping to JMS header 111

N

Name space binding 555

Name space, bindings 556

namespaces

WSDL 357

O

ODBA runtime code 65

OMVS segment, creating 495

Open Database Access (ODBA) 64

P

PK07167 533

R

RACF keyring, creating 320

RACF keyring, creating for IMS Connect 320

RACF keyrings 320

Rational Application Developer (RAD)

meta-data 204

nature 41

Rational Application developer (RAD)

J2C, sample tutorials 185

Redbooks Web site 582

Contact us xviii

res-auth 280

RFH2 header 111

RMI over IIOP 537

rule engine 340

S

SDFSJLIB 65

SDFSRESL 65

service broker 339

service provider 338–339

service requestor 339

Service-Oriented Architecture

concepts 338

Service-Oriented Architecture (SOA) 337

Simple Object Access Protocol (SOAP) 343

Simple Open Access Protocol (SOAP) 345

Simple Open Access Protocol (SOAP)

actor 349

body 346, 350

Client 351

communication styles 351

convention 346

deserialization 352

detail 351

document communication style 351

encoding 352

encoding rules 346

envelope 346, 348

error handling 351

fault 351

fault element 351

faultactor 351

faultcode 351

faultstring 351

header 346, 349

header entries 349

literal encoding 352

marshalling 352

messaging mode 352

document/encoded 352

document/literal 352

RPC/encoded 352

RPC/literal 352

MustUnderstand 351

- mustUnderstand 349
- namespace 347
- namespace, example 347
- overview 345
- role 349
- RPC communication style 352
- serialization 352
- Server 351
- SOAP encoding 352
- SOAPException 351
- unmarshalling 352
- VersionMismatch 351
- SOAP 343
 - encoding 352
- SOAP, current standard level 343
- SQLJ
 - assignment statement 29
 - closing the context 31
 - coding syntax 28, 30
 - connection context 29
 - customization 27, 50
 - customization using RAD 51
 - default context 30
 - differences with JDBC 26
 - example of connection 29
 - executable statement 29
 - executable statement, syntax 30
 - executable statement, what to use for 30
 - files produced by translator 26
 - isolation levels 32
 - iterator declaration 30
 - mapping of SQLJ isolation levels to DB2 isolation levels 32
 - performance 61
 - Rational Application Developer (RAD)
 - .project file 41
 - adding SQLJ support to a project 38
 - connecting to the database 46
 - connection to DB2 48
 - customization 51
 - customization ant script 56
 - customization, class not found error 55
 - customization, command options 53
 - customization, database information 51
 - customization, editing the ant script 55
 - customization, packages 52
 - customization, running ant script 57
 - defining SQLJ translator 36
 - JDBC driver connection jars 46
 - Nature does not exist error 40
 - setup 36
 - SQL statement window 48
 - SQLJ Cached Template 44
 - SQLJ customization script folder 38
 - SQLJ Java source folder 38
 - SQLJ profile print command 38
 - SQLJ settings 37
 - SQLJ Template 45
 - SQLJ translator class name 37
 - SQLJ translator JAR file 37
 - wizard 42
 - READ COMMITTED 32
 - READ UNCOMMITTED 32
 - REPEATABLE READ 32
 - running the TraderDB 59
 - runtime problems 60
 - security 62
 - SELECT statement, example 31
 - SERIALIZABLE 32
 - set isolation level 32
 - SQLJ statements 29
 - translator 26
 - using dynamic SQL in SQLJ 62
- sqlj.zip 46
- SSLENVVAR 325
- SSLKeyStoreName 321
- SSLPORT 325
- SSLTrustStoreName 321
- sysplex distributor 330

T

- Trader
 - CICS
 - ECI J2C resource adapter 562
 - J2CA ECI connector 568
 - createTrader 567
 - data stores 563
 - DB2 table creation 576
 - DB2 table definitions 564, 575
 - dependencies 572
 - ear file contents 572
 - IMS
 - J2C resource adapter 562
 - J2CA connector 568
 - Java Database Connectivity (JDBC) 562, 568
 - packaging 570
 - running, URLs 575

- SQLJ 562
 - TRADER.COMPANY table 564
 - TRADER.CUSTOMER table 564
 - TraderCICS 563
 - TraderDB 563
 - TraderIMS 563
 - TraderLib.jar file 571
 - TraderMQ 563
 - TraderProcess 567
 - TraderProcessEJB 567
 - TraderServlet 566
 - TraderSuperServlet 567
 - VSAM file definitions 564
 - VSAM file, definition 577
 - Web application, architecture 566
 - Web front-end user interface 564
 - Web module 564
 - WebSphere MQ 569
 - connection factory 569
 - Message-Driven Bean (MDB) 569
 - TRADER.CICS.REPLYQ 569
 - TRADER.IMS.REPLYQ 569
 - TRADER.PROCESSQ 569
 - WebSphere MQ CICS bridge 562
 - WebSphere MQ IMS bridge 562
 - WebSphere MQ JMS Provider 562
 - TraderDB
 - changing the default user ID and password used 58
 - errors when importing into RAD 34
 - migrating from V5 to V6 34
 - versions supplied 33
 - TraderMQ
 - migrating from V5 to V6 in RAD 115
 - using V5 version in V6 114
 - TraderMQ, changes required to use JMS 1.1 134
 - TraderMQ, WebSphere MQ queue connection factory 124
 - TraderMQ, WebSphere MQ queue destinations, defining 127
- U**
- unified connection factory 96
 - Universal Description, Discovery, and Integration (UDDI) 344
- V**
- V5 default messaging JMS provider 97
 - V5 default messaging, security 112
- W**
- Web service
 - definition 341
 - properties 341
 - Web services 337
 - additional standards 344
 - core standards 343
 - WS-AtomicTransaction 345
 - WS-I BP 1.0 346
 - WS-I interoperability Basic Profile 1.0 346
 - WS-Security 345
 - Web Services Definition Language (WSDL) 341
 - document 353
 - elements 354
 - Binding 353
 - binding 354
 - data type 354
 - encoding 354
 - Message 353
 - message 354
 - messages binding 354
 - Operation 353
 - operation 354
 - operations binding 354
 - Port 353
 - port 354
 - port definition 354
 - Port type 353
 - port type 354
 - protocol extension 354
 - Service 353
 - Type 353
 - XML Schema 354
 - XSD type 354
 - example 355
 - namespaces 357
 - overview 353
 - Web Services Description Language (WSDL) 343
 - web.xml 543
 - WebSphere default messaging 89
 - access to the data store for a Messaging Engine 111
 - activation specification 100
 - activation specification, deployment descriptor 148
 - adding a bus member 151

- AllAuthenticated group 110
- authorization 108
- authorization, delegation 108
- best effort nonpersistent mode 94
- browser default role 109
- bus connector role 109
- bus destination 92
- bus members 91
- bus, types of destinations 92
- channels, defining in WebSphere MQ 157
- connection factory, defining 157
- connector default role 109
- Control Region Adjunct (CRA) 92
- creating tables for persistence 95
- creator default role 109
- data stores 94
- default data store 95
- default role 108
- default security roles 109
- destination alias 92
- destination role 109
- disabling non-secure inbound transport chains 113
- Enterprise Service Bus (ESB) concept 91
- Everyone group 110
- express nonpersistent mode 94
- foreign bus 93
- foreign bus role 109
- foreign bus, creating 153
- foreign destination 92
- IdentityAdopter default role 109
- InboundSecureMessaging 113
- indirect link 93
- Inter-engine authentication alias 110
- Inter-engine transport chain property 113
- local Service Integration Bus scope 93
- mediation 95
- mediation handler 95
- mediation handler list 95
- Mediations authentication alias 110
- messaging engine (ME) 92
- Messaging Engines (ME) 90
- MQ link, defining 154
- MQ link, inbound address 156
- multi server 90
- OutboundSecureMQLink 113
- persistence modes 94
- pros and cons 98
- Quality of Service settings 93

- Queue destination 92
- queues, defining 160
- receiver default role 109
- receiver, defining in WAS 155
- reliable nonpersistent mode 94
- reliable persistent mode 94
- remote queues in WebSphere MQ, defining 164
- scope of security settings 108
- scope of Service Integration Bus 90
- security administration 109
- security checking on the bus 108
- security roles 108
- sender default role 109
- sender, defining in WAS 155
- Service Integration Bus (SIB) 89
- Service Integration Bus link 93
- Service Integration Bus, setting up 150
- Service-Oriented Architecture (SOA) 91
- sibDDLGenerator command 95
- single server 90
- SSL secure transport connection 113
- Target inbound transport chain property 113
- topic role 109
- Topic space destination 92
- topic space root role 109
- topic, access to 110
- Transport chain property 113
- using Web services 93
- Version 5 JMS clients using 95
- WebSphere MQ link 93
- wsadmin command to list available commands 109
- wsadmin command to list command details 109
- WebSphere MQ
 - channel initiator 85
 - concepts 85
 - local queues 85
 - Message Channel Agent (MCA) 85
 - remote queues 85
 - transmission queue 85
 - types of access to a queue 85
- WebSphere MQ JMS provider
 - authorized and program controlled load modules 123
 - Bindings mode 96
 - Client mode 96
 - Direct mode 96
 - environment variables 123
 - JNDI name syntax 127

level required 96
libraries to add to STEPLIB 123
listener port, configuring 130
listener ports, configuring 129
listener ports, defining in RAD 142
MQ queue connection factory, defining 124
MQ_INSTALL_ROOT variable 95
MQJMS_LIB_ROOT variable 95
pros and cons 99
scope of definitions 124
transport types 96
WebSphere MQ queue destinations, defining
127
workflow 340
WS-AtomicTransaction (WS-AT) 345
WSDL 343
WSDL, current standard level 343
WS-Security 345

X

XML Protocol Working Group 345
XML Schema Definition (XSD) 353



Redbooks

WebSphere for z/OS V6 Connectivity Handbook



Redbooks

WebSphere for z/OS V6 Connectivity Handbook

**Learn how to use
Web services for
connectivity**

**Acquire the latest
technical details on
JCA connectors and
messaging**

**Follow sample code
to get started quickly**

This IBM Redbook (re)examines most connectivity scenarios between WebSphere Application Server for z/OS Version 6.01 and other application resources on z/OS and is an extension of and update on the *WebSphere for z/OS V5 Connectivity Handbook*, SG24-7064.

This book covers various connectivity scenarios from architecture, application development, and infrastructure points of view. New scenarios are introduced using WebSphere Default messaging, Web services, and SQLJ. We also updated all scenarios with the usage of Rational Application Developer Version 6.

A brief summary of connectivity technology explained in this book is:

- ▶ Database access using JDBC and SQLJ
- ▶ JMS 1.1 and WebSphere Default messaging
- ▶ Calling DB2 Stored Procedures using SOAP
- ▶ Bi-directional Web services between WebSphere and CICS
- ▶ J2C connectors to CICS and IMS
- ▶ RMI-IIOP

The technology is explained using sample code, available as additional material for this book.

**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